

Preface

This volume, “Petri Net Technology for Communication-Based Systems,” is a state-of-the-art report in the series *Advances in Petri Nets*. It shows how various well-established and new Petri net notions and techniques can be used for modeling communication-based systems, with special focus on workflow management and business processes.

In the last 6 years this topic has been studied by the DFG Forschergruppe Petri Net Technology in Berlin in close cooperation with the international community. The main results of this cooperation were presented at the 1st and 2nd International Colloquia on Petri Net Technologies for Modeling Communication-Based Systems, held in Berlin in 1999 and 2001, respectively. A careful selection of contributions by members of the DFG Forschergruppe and by international experts in this field are presented in this volume. Taking into account the fruitful discussions during the two colloquia and the cross-refereeing process for the accepted papers, a high degree of common understanding was achieved, leading to a highly comprehensive presentation in this volume.

The topics of the papers in this volume can be roughly classified into the following two areas:

- Petri net technology and
- application to communication-based systems.

Since most papers comprise aspects of both areas, we chose an alphabetic order. However, in the following we give a rough overview of the contributions in both areas according to the main focus of the corresponding papers.

Petri Net Technology

The main objective of the DFG Forschergruppe has been to develop an application-oriented Petri net technology. The main idea has been to develop a “Petri Net Baukasten” comprising three different views: the application developer view, expert view, and tool view. An overview of this “Petri Net Baukasten” is given in the paper by H. Weber, H. Ehrig, W. Reisig and the other main members of the DFG Forschergruppe. The application development assistance related to the application developer view is discussed in more detail in the paper by H. Weber. The concept of Petri net transformations and refinements is essential for the expert view. Transformations are studied in the papers by B. Braatz, H. Ehrig, M. Urbasek and F. Parisi-Presicce, with focuses on the conceptual and the formal framework. Rule-based refinement is studied by J. Padberg and M. Urbasek, and transition refinement for the verification of distributed algorithms by S. Peuker.

The tool view is supported by the two papers by M. Weber and E. Kindler on the Petri net kernel and the Petri net mark-up language, respectively. The generation of animation views in the paper by C. Ermel, R. Bardohl and H. Ehrig

is based on the tool GENGED. A more general view of the integration of tools is given by V. Braun, T. Margaria and B. Steffen using the Electronic Tool Integration (ETI) platform.

A comparison of Petri nets with activity diagrams is discussed by R. Eshuis and R. Wieringa, while Petri nets are proposed as a semantic model for message sequence charts by O. Kluge.

Applications to Communication-Based Systems

New concepts for business processing based on Petri nets are presented in the papers by W.M.P. van der Aalst and J. Dehnert, with focuses on inheritance and soundness respectively. Quantitative engineering, interorganizational management, and dynamic service outsourcing for business processes are considered by J. Desel, K. Lenz and A. Oberweis, and P. Grefen and S. Angelov. Interesting new theoretical frameworks are open Petri nets as semantic models for workflow integration and team automata for CSCW in the papers by R. Heckel and J. Kleijn, respectively. Moreover the new concept of algebraic higher-order nets is used by K. Hoffmann in order to achieve flexible modeling of business processes. Last, but not least, an integrated approach for automated control system development based on Petri nets is presented by E. Schnieder et al.

We are most grateful to the German Research Council (DFG) for support of the DFG Forschergruppe and the International Colloquia on Petri net technology in 1999 and 2001, to all authors and referees of this volume for their careful work, to Kathrin Hoffmann for her assistance in the editorial process, and, last, but not least, Springer-Verlag for a fruitful and smooth cooperation in publishing this volume in the well-established series *Advances in Petri Nets*.

August 2002

Hartmut Ehrig
Wolfgang Reisig
Grzegorz Rozenberg
Herbert Weber

Table of Contents

The Petri Net Baukasten of the DFG Forschergruppe PETRI NET TECHNOLOGY <i>H. Weber, H. Ehrig, W. Reisig, A. Borusan, S. Lembke, J. Dehnert, M. Weber, A. Martens, J. Padberg, C. Ermel, and A. Qemali</i>	1
Application Development Assistance with the Petri Net Baukasten <i>Herbert Weber</i>	22
Petri Net Transformations in the $\gg$ Petri Net Baukasten $\ll$ <i>B. Braatz, H. Ehrig, and M. Urbášek</i>	37
Four Steps Towards Sound Business Process Models <i>Juliane Dehnert</i>	66
Generation of Animation Views for Petri Nets in GenGED <i>Claudia Ermel, Roswitha Bardohl, and Hartmut Ehrig</i>	83
The Petri Net Kernel <i>Michael Weber and Ekkart Kindler</i>	109
The Petri Net Markup Language <i>Michael Weber and Ekkart Kindler</i>	124
Case Study Logistics: Flexible Modeling of Business Processes Using Algebraic Higher-Order Nets <i>Kathrin Hoffmann</i>	145
Rule-Based Refinement of Petri Nets: A Survey <i>J. Padberg and M. Urbášek</i>	161
Modelling a Railway Crossing with Message Sequence Charts and Petri Nets <i>Olaf Kluge</i>	197
Quantitative Engineering of Business Processes with VIPbusiness <i>Jörg Desel and Thomas Erwin</i>	219
Inter-organizational Business Process Management with XML Nets <i>Kirsten Lenz and Andreas Oberweis</i>	243
Three-Level Process Specification for Dynamic Service Outsourcing: From Petri Nets to ebXML and WFPDL <i>Paul Grefen and Samuël Angelov</i>	264
Open Petri Nets as Semantic Model for Workflow Integration <i>Reiko Heckel</i>	281

Team Automata for CSCW – A Survey <i>Jetty Kleijn</i>	295
Comparing Petri Net and Activity Diagram Variants for Workflow Modelling – A Quest for Reactive Petri Nets <i>Rik Eshuis and Roel Wieringa</i>	321
BASYSNET – An Integrated Approach for Automated Control System Development <i>Eckehard Schnieder, Mourad Chouikha, Stefan Einer, and Michael Meyer zu Hörste</i>	352
The Electronic Tool Integration Platform (ETI) and the Petri Net Technology <i>Volker Braun, Tiziana Margaria, and Bernhard Steffen</i>	363
Inheritance of Business Processes: A Journey Visiting Four Notorious Problems <i>W.M.P. van der Aalst</i>	383
A Formal Framework for Petri Net Class Transformations <i>Francesco Parisi-Presicce</i>	409
Concurrency Based Transition Refinement for the Verification of Distributed Algorithms <i>Sibylle Peuker</i>	430
Author Index	455

The Petri Net Baukasten of the DFG Forschergruppe PETRI NET TECHNOLOGY

H. Weber¹, H. Ehrig¹, W. Reisig², A. Borusan¹, S. Lembke¹, J. Dehnert¹,
M. Weber², A. Martens², J. Padberg¹, C. Ermel¹, and A. Qemali¹

¹ Technical University of Berlin

² Humboldt University of Berlin

Abstract. In the long history of Petri nets a universe of Petri nets has evolved consisting of an enormously rich theory, a wide variety of tools, and numerous successful applications and case studies in various application domains. This vast variety is not any more handable for anyone working with Petri nets, which results in the strong need of a structured access to Petri nets. This structured access has been the main aim of the DFG-Forschergruppe PETRI NET TECHNOLOGY, which has developed the so-called Petri Net Baukasten for this purpose. It is designed to support Petri net experts, application developers and tool developers alike in their specific work with Petri nets. This paper presents an overview of the concepts, initial and 2nd installment of the Petri Net Baukasten, which have been presented at the 1st and 2nd International Colloquium on Petri Net Technologies for Modelling Communication Based Systems in 1999 and 2001, respectively.

Keywords: Petri Nets, Petri Net Technology

1 Introduction

Petri Nets are a subject of interest to researchers since more than 30 years. They became popular since they allow the graphic representation of computational structures and have a formal underpinning that allows formal proofs of properties. Their use in engineering, however is limited to a rather small number of applications. Many of the graphic notions used in practice are similar to the graphic representation of Petri nets but do not make use of the rich body of theory that exists for Petri nets. This has frequently been attributed to the fact that Petri nets and the associated theory are hard to understand. Moreover, people applying Petri nets tend to require special brands of Petri nets which in turn led to a great variety of different net types. Finding the right Petri net types from the portfolio of exciting types, finding corresponding techniques and tools, selecting them and applying them successfully to the applications poses a problem and prevents their wide spread use. In order to improve this situation a Petri Net Baukasten has been developed by the DFG-Forschergruppe PETRI

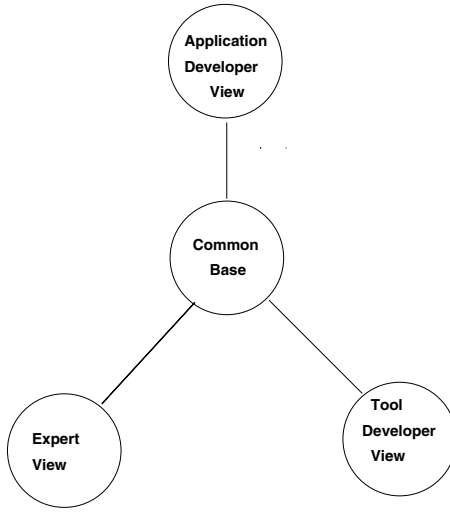


Fig. 1. The conceptual structure of the Petri net Baukasten

NET TECHNOLOGY in Berlin. It aims at the development of a support environment for the use of Petri nets in new applications including theoretical and tool's aspects. Figure 1 roughly sketches the conceptual structure of the Petri Net Baukasten.

The Petri net Baukasten is divided into different views, namely the Application Developer View, the Expert View, and the Tool Developer View, all linked via a Common Base. The Common Base consists of a classification of Petri nets and corresponding notions independent of their use in applications, their formalizations, and tool support. These Petri net notions are represented in the specific views. They concern the use of the Petri net notions within a certain application domain, their formalization in the theory of Petri nets, and their implementation as Petri net tools.

In this paper we present an overview of the Petri Net Baukasten, based on the initial installment, published as "Initial realization of the Petri Net Baukasten" in [18], and on the second installment in [19]. These installments have been presented at the 1st and 2nd International Colloquium on Petri Net Technologies for Modelling Communication Based Systems.

In Section 2 of this paper we review the concepts and the initial installment of the Petri Net Baukasten. The conceptual structure including the Common Base and three different views is shown in Figure 1. The initial installment comprises the contents of these views and the Common Base on a low technical level with direct access to the corresponding documents in a more or less static way. A more detailed presentation of the Petri Net Baukasten in the initial phase is given in [9].

In contrast to the initial installment, the second one presented in Section 3 of this paper provides an interactive access to the contents of the Common Base and the specific views including dynamic changes. A full implementation, however, is beyond the capacities of the research group PETRI NET TECHNOLOGY. The 2nd installment consists of a database, services operating on this database and explicit access for each user group (user interfaces). The database may be searched or updated by the services. Updating requires some mode of cooperation among the users in order to maintain the conceptual structure of the Petri Net Baukasten. More precisely, in order to maintain a correspondence between a Petri net notion, resp. Petri net type in the Common Base, its formalization in the Expert View, an application-oriented interpretation in the Application Developer View, and a tool in the Tool Developer View an inserted piece of data has to be related to corresponding data of the other views. As the knowledge of the corresponding data cannot be expected from a single user, the 2nd installment has to support the cooperation among users as well.

In Section 4 we discuss the database aspect in more detail and in Section 5 an architecture for the 2nd Installment. Finally in Section 6, we summarise the main ideas and discuss how other contributions of the PETRI NET TECHNOLOGY researcher groups in this volume are related to the Petri Net Baukasten.

2 Concepts and Initial Installment of the Petri Net Baukasten

2.1 Aims of the Petri Net Baukasten

The main aim of the Petri Net Baukasten is to provide support for various users of Petri nets.

This support is given with respect to specific application domains. As application domains we consider all areas where Petri nets have proven to successfully support the development of a system of that kind, e.g. automatic production, control systems, workflow management etc. The Petri net Baukasten enables an engineer who develops an application to use application-oriented interpretation of Petri net notions. The Petri Net Baukasten supports the application developer to find and use adequate Petri net based methods, Petri net techniques or just Petri net types which fit the developed application. The Petri Net Baukasten does not only provide the theoretical knowledge about the handling of Petri net notions, but also makes available practical examples and tool support.

The Petri net expert is supported in his work to elaborate and enhance existing formal techniques, to define new types and notions in a uniform way, to state properties of all variants of Petri nets in a formal and constructive manner, to transfer results between net types, and to make results and notions available for applications.

Last, but not least, the Petri Net Baukasten enables tool developers to distribute their tool, to find appropriate tools (also for tool development), to fit tools together in a prototyping way, to add and change tools and tool references.

2.2 The Concept of the Petri Net Baukasten

The central notion of the Petri Net Baukasten are Petri net techniques, illustrated in Figure 2. A Petri net technique is given as a consistent combination of a Petri net type, its formalization, a methodological procedure, and a corresponding tool.

A Petri net type is given by a semiformal description of its characteristics. The description uses Petri net notions concerning the basic constituents of a Petri net variant like place, transition, marking, etc. and concerning the available Petri net operations for analysis, or structuring techniques, etc. The Petri net type is represented in the Petri net classification in the Common Base and includes links to the other views.

The formal description of a Petri net type is given in the Expert View. For each aspect a mathematical formalization is given; basic constituents are captured in the definition of the Petri net class. Operational constituents correspond to definitions and results of Petri net operations.

In the Tool Developer View tools are provided, customized to the Petri net type. That is, they work on Petri nets comprising the basic constituents and implement the operational characteristics of the type.

A methodological procedure of how to work with a Petri net type in practical applications is given in the Application Developer View. It defines the sequence of development activities, including the available Petri net operations for verification, structuring, etc.

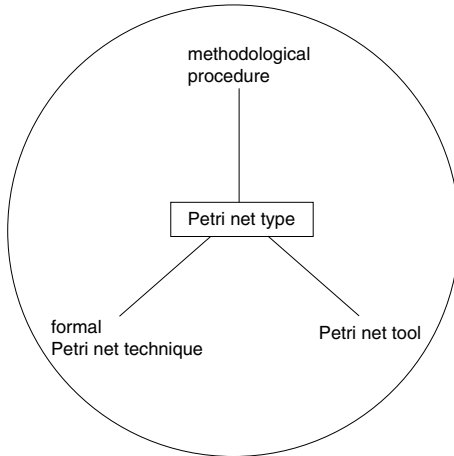


Fig. 2. The Petri net technique

2.3 The Common Base

The Common Base, see [4, 23], encompasses all Petri net types relevant for a specific application domain, it integrates all these types into a common scheme and defines the relationship between the Petri net types. It is intended that the Common Base is structured as simple as possible and understandable to application developers, tool developers and Petri net experts alike. The structuring concept that is considered to be simple and still powerful enough to capture all the properties of different Petri net types is a specialization/generalization relationship between the different Petri net types. The Petri net classification in the Common Base is a specialization hierarchy along distinguishing characteristics. This classification is represented in the Common Base using class diagrams of UML. The distinguishing characteristics are given in terms of attributes and attribute values. They describe Petri net notions on a conceptual level. The domain dependent interpretation of these concepts are part of the Application Developer View. The encoding into software belongs to the Tool Developer View and the mathematical formalization of these concepts is part of the Expert View.

2.4 The Application Developer View

The Application Developer View, see [15, 23], provides the prerequisites for an application-oriented assistance of Petri net based development projects. The planning of development activities in an application domain is supported in the following way: The Application Developer View provides information for the planning of development activities, i.e. methods of employing Petri net techniques. This information supports the application developer to choose the appropriate Petri net techniques, based on tools, methodological procedures and formal Petri net techniques for the development activities. Additionally, modelling activities are supported by sample and standard solutions, and methodological procedures for a chosen Petri net type. The Application Developer View includes information on Petri net notions and Petri net techniques in an application- and problem-oriented way. These so-called *assistance concepts* yield domain specific interpretations of Petri net notions of the Common Base.

2.5 The Expert View

The Expert View provides the formal foundation of Petri net types in the Petri Net Baukasten in terms of a mathematical presentation of the underlying notions and results. The Expert View is given in a structured way. It comprises formal Petri net techniques, abstract Petri net frames, actualizations, and transformations. Each of these notions includes a coherent and consistent piece of Petri net theory. Formal Petri net techniques, and transformations are directly related to the Common Base. Abstract Petri net frames, actualizations, and abstract transformations describe relation and dependencies of formal Petri net techniques and transformations on a more abstract level. This yields a uniform description of Petri nets as a foundation of the classification given in the Common Base.

2.6 The Tool Developer View

The main task of the Tool Developer View, [for details see [24, 12]], is to provide support for tool development. This comprises the management of existing tools, facilities for tool development as well as possibilities to extend tools. Petri net tools support Petri net based system development. They provide support for editing, simulating, structuring, and analyzing a Petri net variant. The management of existing tools is also important for the rest of the Petri Net Baukasten, since it offers tool support for the Petri net types of the Common Base. A Petri net tool corresponds to a Petri net type of the Common Base if it supports the notions comprised by that Petri net type. The Tool Developer View provides support for the development of tools in terms of object-oriented and parameterization concepts, a component-oriented approach for already existing tools and encoded algorithms. The Petri net type used in the encoded algorithm is given in the Common Base. The formal representation of the algorithm is given in the corresponding formal Petri net technique in the Expert View.

2.7 Relation of Common Base with Views

The Common Base is related to the views according to different representations of Petri net notions in the Common Base and the views.

The notions given in the Common Base in an informal way by attributes are represented in the Application Developer View by application-oriented notions within a methodological procedure, explaining the use of these notions in a specific application domain. Within the Tool Developer View these notions are represented either as algorithms or as tools. The Expert View provides consistent formal Petri net techniques for the Petri net type given in the Common Base.

As an example the notion *marking of a Petri net* may be considered

- in the Common Base to be a distribution of tokens over places indicating the state of the net,
- in the Application Developer View to be a representation of documents and business objects in different processes,
- in the Tool Developer View to be a record of places, and
- in the Expert View to be an element of the free commutative monoid over the set of places.

The specialization hierarchy of the Common Base can be used for the navigation with respect to all three views. It allows embedding of new tools or new theoretical results and making them available for practice.

2.8 Initial Installment of the Petri Net Baukasten

A detailed documentation of the initial installment is given in [3]. In the following we only summarise the main ideas.

Common Base

In the initial installment a class diagram representing the classification of Petri net types relevant for business processes has been established and is depicted in [7] using the UML tool Rational Rose. The classification comprises about 100 Petri net types and twelve different specialization paths, that can be partitioned into three categories: elementary, additional, and operational extensions.

Application Developer View

In the initial installment, an assistance system is provided on a conceptual level for the support of the application developer. There are two kinds of variants for the assistance that can be distinguished, namely assistance to find a suitable Petri net technique and assistance to find a suitable solution example. Both variants are supported by prescriptive, navigating, or descriptive assistance methods. The system architecture sketches the fundamental ideas about the assistance system in [7]. More details are given in [23].

Expert View

The structured representation of significant aspects of Petri net theory in the initial installment is based on schemes for abstract Petri net frames, formal Petri net techniques, actualizations, and transformations. Each of these schemes consists of a list of relevant keywords. The initial installment of the Expert View comprises instantiations of all these schemes, e. g. parameterized net classes as an actualization of abstract Petri net frames, or of algebraic high-level nets, coloured Petri nets, elementary nets, place/transition nets, and FunSoft nets as instantiations of formal Petri net techniques.

Petri Net Kernel in the Tool Developer View

The Petri Net Kernel in the initial installment is an object-oriented tool for the fast prototyping of simple Petri net tools. The basic idea of the Petri Net Kernel is the distinction between fixed and variable aspects of Petri nets. Variable aspects can be considered as parameters and allow the automatic generation of Petri net tools by actualization of these parameters. It implements the part of the Tool Developer View that affects tool development. The implementation of parameterized net classes with the Petri Net Kernel is discussed in [12] of this volume.

3 2nd Installment of the Petri Net Baukasten

3.1 Survey of the 2nd Installment

We are now going to give an overview over the 2nd installment of the Petri Net Baukasten. The 2nd installment roughly consists of three parts: The data which is to be provided, a user interface for communication between user and system, and thirdly an intermediate level which operates on the data. We call these parts *database*, *user interface* and *services*, respectively. Each of these parts is structured internally in order to increase clarity and handability. This internal structure is derived from the conceptual structure of the Petri Net Baukasten, as depicted in Figure 1. That is, each part is substructured into three views which

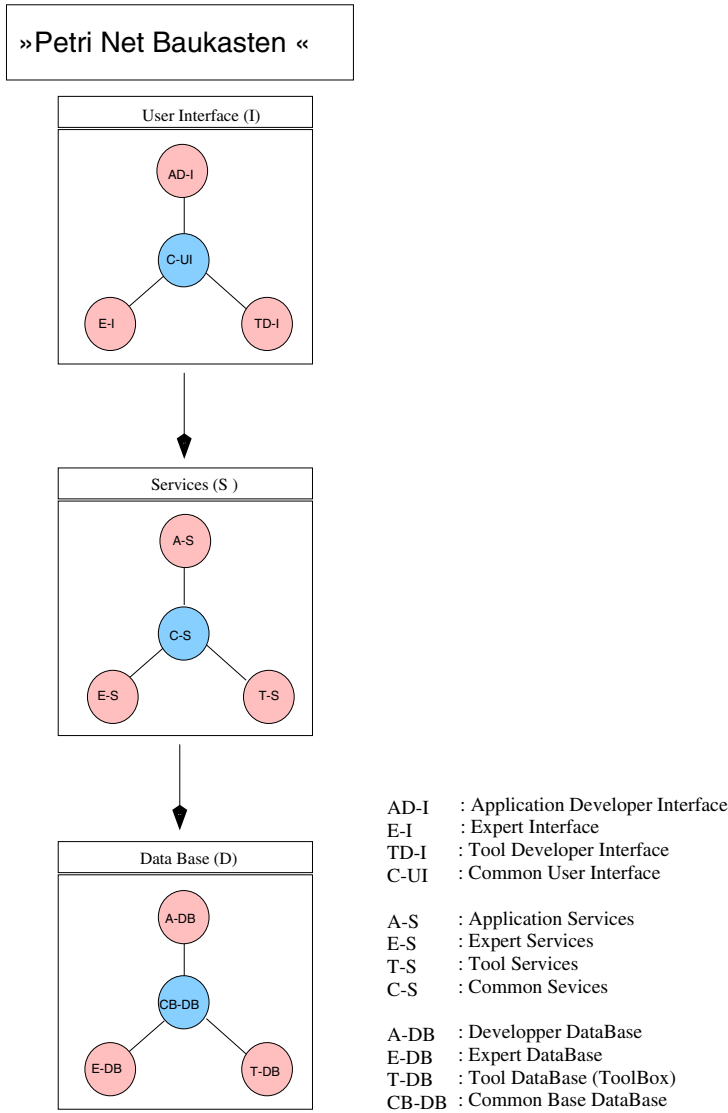


Fig. 3. 2nd Installment of the Petri Net Baukasten

correspond to one of the views (application developer view, tool developer view, and expert view). This structure is motivated by the general aim to support each of these groups. This view-oriented structure is shown in Figure 3. We provide an individual access to information for each group, because we can then focus on the relevant information, which naturally differs for each group.

Database The contents of the 2nd installment of the Petri Net Baukasten is stored in the database. In contrast to the initial installment it is not fixed, but may increase dynamically. According to the concept of the Petri Net Baukasten it is substructured into specific data for the various user groups and the Common Base. The Common Base contains the Petri net classification and Petri net notions and links to data of the specific views. In principle, we could start with an empty database, to be gradually enriched. For convenience we assume the data of the initial installment as its contents. More details about the database can be found in Section 4.

Services operate on the database. We distinguish between common-, application-, expert-, and tool-services, which are described in Section 3. Basically, there are search services which find data according to input parameters, and maintenance services. Maintenance services modify the database and trigger an asynchronous cooperation between users (of different groups). This cooperation is essential for maintaining the basic concept of the Petri Net Baukasten. For search services the set of admissible input parameters is derived from the contents of the database, and may thus also be changed by maintenance services.

User Interface The upper part in Figure 3 represents the user interface. We provide a general user interface from where a specialized interface, can be reached for any of the different user groups, application developers, experts, and tool developers. Each specialized interface offers the corresponding services together with an informal description. Search services are accompanied by a set of admissible input parameters. Moreover, the user is informed that a change of the database has been performed by another user. He may complete that piece of data by group-specific data in order to maintain the basic concepts of the Petri Net Baukasten. The interfaces are described in Section 2.

3.2 User Interface

We will describe the user interface of the Petri net Baukasten in more details here. We provide a general user interface as a basic for a specialized interface for any of the different user groups, application developers, experts, and tool developers. Each specific user interface offers an individual set of services enabling the access to information about Petri nets, which are relevant to the respective user group. As a result of this, a separation of concerns is achieved leading to a more requirement-oriented and easy use of the Petri Net Baukasten.

The belonging to a certain user group depends on the objectives of the particular users. If their objectives change over the time the belonging to a certain user group may also vary. For example, a user may be interested in the formal development of a specific Petri net type in one moment and in the implementation of an appropriate Petri net tool in the next moment. So, this person uses the Petri Net Baukasten as a Petri net expert first and as a tool developer afterwards. Accordingly, the person uses the Petri net expert user interface first and the tool developer interface afterwards. The specific sub-interfaces are reached

through the general user interface, which realizes the user choice according to their role. A second task of the general user interface is the information about changes in the database. In case new information has been added, all users are informed and asked to complete the data in order to maintain the basic concepts of the Petri Net Baukasten.

The specific user group interfaces support the input of parameters required for the execution of selected services and the representation of results of a service execution. The interfaces are quite diverse regarding their offered services but their appearances are uniform. The available services are listed in menus; forms allow to update parameters. Moreover, each interface offers a help function explaining its specific functionality.

3.3 Services

Assistance services are invoked in the user interface and operate on the database. Basically, there are two different kinds of assistance services: search and maintenance services.

Search services mainly realize the interactional aspect of the 2nd installment of the Petri Net Baukasten. They retrieve the requested data according to input parameters and present the search results in a manner which is adapted to the individual user preferences.

Maintenance services are essential for the dynamic aspect, as they perform modifications (addition or deletion) of the database. Moreover, they trigger a co-operation between users of different groups.

We distinguish between common-, application-, expert-, and tool-services, which are discussed below. These group-specific services use common services in order to fulfill their tasks.

3.4 Application Services

The Petri Net Baukasten supports application developers in the domain of Petri net based application management. Support can be given for the following tasks:

1. to find an adequate Petri net based method for developing the application,
2. to use Petri net techniques that serve best in the development of the application,
3. to find a Petri net type which fits the application,
4. to find a suitable tool,
5. provide the application developer with example solutions.

Accordingly, we define the following application services, which are in a one-to-one correspondence to the above listed tasks.

Petri Net Method Selection This service assists finding a suitable Petri net based development method. The service allows to search over objectives, intended application domains and covered development phases (analysis, design, etc.) of Petri net methods. After the search the application developers

can access information about each offered Petri net method, as e.g. intended objective, covered phases, underlying languages and techniques and supporting tools. Additionally, they may consult the method guide before selecting their suitable Petri net method.

Petri Net Technique Selection This service covers the assistance in finding a suitable Petri net technique. The application developer specifies required characteristics of a Petri net technique. This can be done, among others, with the help of keywords which outline the intended objective, application domain and underlying Petri net type. Petri net techniques fulfilling the specified characteristics are determined automatically and offered to the application developer.

Developers select their suitable Petri net technique from the offered set. They may use information about it, as e.g. application domain, objective, underlying Petri net types, or the essentials of its procedure. Moreover, they may use its informal description, the so-called *technique guide*, or its formal foundation. Last but not least, it is possible to search for a suitable Petri net tool supporting the preferred Petri net technique.

Petri Net Type Selection This service assists in finding a suitable Petri net type which fulfills the requirements of a specific application development process. There are several assistance methods for finding a suitable Petri net type, as for example, by an application aspect checklist or by navigation through the Petri net classification. Via the checklist it is possible to find a Petri net type which fits a combination of the most relevant aspects for their application purpose.

The service then automatically determines Petri net types which support corresponding language concepts. The found Petri net types are offered to the application developer. Together with the list of resulting Petri net types, some more information about the different results is provided, as for example intended application domains, possible tools, informal and formal description. These information facilitates the final Petri net type selection.

Petri Net Tool Selection This service supports application developers in selecting a suitable Petri net tool which enables the application of a Petri net technique, method, or type selected before. It provides the application developer as well with technical-functional criteria (as e.g. the required operating system, purpose, usability) and external criteria (as e.g. usage costs). This enables application developers to select the suitable tool or tool set for their development task. After the selection of a tool or tool set it can be downloaded or ordered from its supplier.

Select a Sample Solution This service is intended for finding solutions suitable for fulfilling the requirements of a certain application development. It enables application developers to find suitable sample solutions by their objectives, application domains, and their underlying Petri net types. Before application developers select a certain sample solution to integrate it within their own application development they can use information about it, for example about the solved problem or the underlying Petri net type and can consult the so-called *solution guide* for details.

Application Maintenance Via this service it is possible to add new information to the application developer database. This service can be used to enhance existing methods, to add new techniques or to make available examples for other users.

A more precise description of the application developer services, their realization and introduced terminology can be found in [22].

3.5 Expert Services

The Petri Net Baukasten supports Petri net experts doing research in the area of Petri nets. Support can be given for the following tasks:

1. to elaborate and enhance existing formal techniques,
2. to define new types and notions in a uniform way,
3. to state properties of all variants of Petri nets in a formal and constructive manner,
4. to transfer results between net types,
5. to make results and notions available for applications,
6. structured access to the theory of Petri nets.

Accordingly, we define the following expert services:

Search Formal Petri Net Technique This service finds all formal Petri net techniques comprising a given Petri net notion. It uses the common service **Common Base Parameter Search**. For choosing suitable Petri net notions the expert may consult the glossary of Petri net notions.

This service supports the tasks (1) and task (2) as the expert may use existing definitions for enhancement.

Add Formal Petri Net Technique This maintenance service adds a formal Petri net technique together with its semiformal description by a Petri net type to the database. The expert has to fill out the scheme for formal Petri net techniques. Moreover, he has to provide the semiformal description of the Petri net type using already existing Petri net notions and possibly defining new ones. Help is provided by the glossary of existing Petri net notions. The common services **Petri net type Insertion**, resp. **Link Construction** place the newly defined type in the Petri net classification and the service **Parameter maintenance** adds new Petri net notions.

Search Transformation This service is a special case of **Search Formal Petri Net Technique**, because transformations belong to a formal Petri net technique. Input parameters are restricted to Petri net notions concerning transformations.

Add Transformation Analogously to **Search Transformation**, this is a special case of **Add Formal Petri Net Technique**.

Search Scheme This service comprises search for abstract Petri net frames and instantiations, which are given by schemes in the Expert View. Optional input parameters specify the requested scheme. This service does not use common services, as these schemes are specific to the Expert View. Again, tasks (1) and (2) are supported as well as tasks (2), (3), and (4).

Add Scheme The expert fills out the corresponding scheme for abstract Petri net frames, or instantiations, which is added to the database.

Search Precondition This is a special case of searching for an abstract frame. It focusses on the preconditions of the formal parameters of that frame, which are presented as the result of the search. Consequently, there is no service “**Add Precondition**” as these are already added by adding an abstract frame.

Search Case Study The input of this service are Petri net notions or keywords concerning a certain problem. The service returns a set of case studies which are based on the Petri net notion or which solve the problem.

Add Case Study The expert may add a description of a case study, provided that the used formal Petri net technique is already stored in the database. Moreover, keywords describing the solved problem are required as input. The service adds the case study and creates a link from the formal Petri net technique. This service supports task 5.

3.6 Tool Services

The task of the tool developer is supported by the Tool Developer View, also see [24]. The support comprises various aspects of tool development:

1. management of existing tools,
2. facilities for tool development, and
3. support for extension of existing tools.

Accordingly, we define the following tool-services.

Petri Net Tool Selection This service supports tool developers in selecting a Petri net tool. It provides the tool developer with relevant information about the tool, such as functional range, implementation language, underlying programming concepts, operating system etc. This collection of information enables tool developers to compare existing implementations and hence to find tools or specific modules which can be enhanced or reused for their own tool implementation.

Search for Petri Net Input Formats Most of the existing Petri net tools use their own Petri net input format. This service provides a search function about existing formats and existing transformation scripts translating them mutually. It furthermore provides the tool developer with information about the current results concerning the development of a standard interchange format for Petri nets based on XML.

Facilities For building prototypes, this service uses the PETRI NET KERNEL(PNK). The PNK provides an infrastructure for building Petri net tools by offering standard functions and a graphical user interface. The PNK is not restricted to a particular Petri net type, but covers all relevant Petri net types due to parameterization. It covers the following sub-services:

- The PNK supports the implementation of new algorithms for analysis, simulation, or verification. Parsers, graphical interfaces etc. are provided by the PNK, such that the tool developer may concentrate on the new algorithm.
- The PNK comes with an XML based file format which respects the development of the standard interchange format for Petri nets.
- Information about the net is easily accessible via a simple interface, which reflects the typical mathematical notions on Petri nets such as pre- and postsets. Knowledge of a particular software technique is not required, such that an unexperienced programmer is able to efficiently use the interface within short time.
- The PNK supports the integration of several algorithms which have been developed independently into a single tool. This enables tailoring to specific application domains from a collection of available algorithms.
- The implementation of a tool for newly defined Petri net types is facilitated again without implementing additional parse operations or editor functions.

Tool Maintenance This service realizes a tool administration component which stores relevant information about tools in the tool database. The administration is based on a form containing all relevant information about a tool, as for example a general description, the underlying Petri net type(s), the implemented Petri net operations, some technical instructions, information about licences and some evaluation notes.

3.7 Common Services

Common services play a key role among the assistance services. They are not invoked directly by a user but indirectly by a user specific service. The common services integrate functionality such as:

1. search over the common base, and
2. care for a consistent and coherent update of the whole database, and provide a common platform for discussion.

The following common services have been defined:

Common Base-Navigation This service facilitates a search for a Petri net type by traversing the Petri net classification in the Common Base.

Common Base-Parameter Search This service searches for a Petri net type according to input parameters.

Petri Net Technique Completion This service cares for a consistent and, if available, complete descriptions of Petri net techniques. For the description of a Petri net technique data from all different views are combined. The Common Base contributes the Petri net type, the Application Developer View a Petri net methodological procedure, the Expert View contributes the formalization and the Tool Developer View contributes a corresponding tool or toolset. The service **Petri net technique Completion** is invoked, if one

of the maintenance services of the user groups adds something new to their database or the Common Base. This information is published to all users of the Petri Net Baukasten and is combined with the request to contribute to this piece of information in order to collect and establish links to the other views.

Petri Net Type Insertion This service updates the Common Base-database. For a new Petri net type a new class with corresponding attributes and functions is generated and ranged into the classification hierarchy. Finally it is linked to the user specific views.

Parameter Maintenance This service cares for a consistent update of all admissible input parameters. The parameter list can be changed explicitly through the access of users or automatically through a search of the database contents.

Link Construction In case new data have been added to the database this service is invoked to check for existing relations between this new piece of information and data belonging to other views. These relations are made explicit through the insertion of links.

4 Database

The database contains all data which are made available through the Petri Net Baukasten. According to the concept of the Petri Net Baukasten it is substructured into specific data for the different user groups and the Common Base.

4.1 Application Database

The application-oriented view upon Petri nets presented in the application database mainly comprises Petri net (development) methods, Petri net (development) techniques, and sample solutions. These are described by their names, their objectives, their application domains, and application-oriented keywords. Their informal application-oriented descriptions are given by documents, called *method guides*, *technique guides* and *solution guides* respectively. The application database manages several tool recommendations supporting the application of Petri net development methods and techniques, respectively. Moreover, application domains play a central role in the application data base. Application domains are characterized by a number of application-oriented aspects reflecting all relevant entities and questions of a particular application domain. Moreover, several suitable means of expression to specify a certain application-oriented aspect using Petri nets are managed in the database. Thereby, the database also contains an application-oriented interpretation of Petri net notions. Furthermore, the database contains application-oriented glossaries describing all used application-specific notions.

4.2 Expert Database

The theory of Petri nets presented in the expert database is based on the notions *abstract Petri net frames*, *formal Petri net techniques*, *actualizations*, and *transformations*. We use schemes for the structured representation of these notions in order to obtain a uniform structure. This database contains all the data of the initial realization, see [7]. Additionally, data for the service **Instantiation** are provided. This comprises definitions and conditions for all operations, which are based on general techniques. For example, the structuring technique of union is based on the categorical pushout, and correspondingly the definition is included in the expert database. Similarly, the definition of high-level replacement systems [8] is contained, because high-level replacement systems are the basis for the structuring technique of rule-based modification.

4.3 Tool Database

The tool database comprises all data which are especially relevant for developing a Petri net tool. This incorporates first of all tools which are provided either physically in form of executable program code or virtually in form of a link to the source. Furthermore the tool database stores all available information about the use and the construction of a tool. Information about use comprises manuals, user guidelines, tool documentations and information about the required operating system, about its licence, licence fee or its usability. The support for construction includes source codes, source code documentations, programming guidelines and information about existing input formats for Petri nets and input format transformations.

4.4 Common Database

The common database contains the current version of the classification diagram presented in the initial realization, see [7]. Every Petri net type and its properties are described within the class notation of UML. The relations between different Petri net types are denoted through the hierarchy relationships, such as specialization- and generalization, within the class diagram. The data stored in the common data base play a key rule, because they link the data of the three different views.

5 Architecture for the 2nd Installment

In this section we describe an architecture for the Petri Net Baukasten which satisfies the above requirements. We outline its use with the help of a scenario.

5.1 The Architecture

The architecture follows the classical trisection into graphical user interface (GUI), functionality layer and data repository. The architecture of the Petri

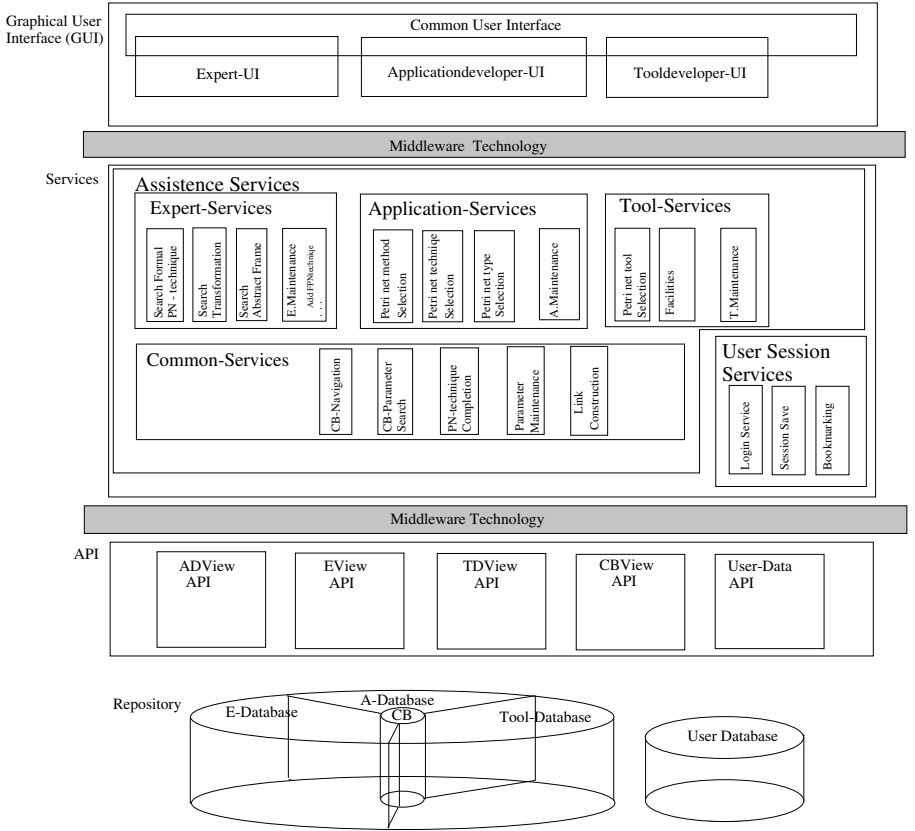


Fig. 4. Architecture of the Petri Net Baukasten

Net Baukasten is illustrated in Figure 4. It is based on middleware technologies in order to support distribution of the its parts (GUI, Services and Repository) on different servers.

The first part contains the user interface, which is subdivided into a general-, an expert-, an application developer-, and a tool developer-user interface. The specific sub-interfaces are achieved through the general user interface, which implements the choice of the user according to her role.

The user interface and the repository are related through services. The services are divided into *common services*, *application services*, *expert services* and *tool services*. These assistance services are split up into *search services* and *maintenance services*. The specific user group services, as *expert-*, *application-* and *tool services* are invoked through the corresponding user interfaces. The common services take over an integrating part. Their search services over the common base are used by every user group; hence are accessed via the corresponding user inter-

faces. Their maintenance services are invoked through the general user interface, hence addressing all users indepently from a chosen role.

The scope of the services and examples have been described in Section 3.

Apart from specific user group and common services we consider a third group of services: the *user session services*. This group is accessed via the general user interface and takes over tasks like administration and supervision of user sessions. It comprises storing and loading of login and session data as well as bookmarking.

Access to the repository is realized via an *application programming interface (API)*. The API is a convenient interface, allowing abstraction from the repository realization.

The last part of the described architecture is the repository. Here all the relevant data are mangaged. There is one repository which holds all the databases discussed in Section 4. Additionally, we consider a user database which holds session data, bookmarks, notes etc. These data elements are generated during a session and are stored and provided for further use.

5.2 Scenario

For better illustration we will explain the architecture with the help of a user scenario. This scenario concerns the use of the Petri Net Baukasten by an application developer and describes the search for a Petri net type suitable for the modeling of specific workflow aspects.

The *general user interface* is open and allows the user to login. With the help of the user identity the system loads the status of the last interaction of this user and asks if he/she wants to continue his/her activity as application developer. In this case the *Application developer interface* is opened and the old session data are loaded. The application developer may now continue her work. Lets assume she wants to select a certain Petri net type that fits certain workflow aspects. The user may want to select the Petri net type with the help of a checklist. Opening a modeling related checklist the user may take his/her choice between a combination out of functional, data/document, roles, resource and time aspects. Assume the user wants to model functional aspects combined with ressource and time aspects. The system determines Petri net types, which comprise language concepts that depict the selected choice of workflow aspects (e.g. [2, 1, 13, 6]). The resulting list is presented in a new window. To facilitate the choice between the variety of Petri net types, some more information (informal-, formal description of the used Petri net types, analysis methods, tool support, examples) about the results is provided. Assume our user wants to apply qualitative analysis methods and performance evaluation. This reduces the choice to the approaches based on stochastic Petri nets, whereby the approach in [6] provides tool support. After the selection of one Petri net type, the system provides the user with relevant data (full papers) and tool support.

The relevant assistance service in this scenario was **Petri net type selection**. The search was performed using a modeling related checklist of application

oriented aspects, invoking a search service provided through the *common services*. Accessed data have been the Petri net classification stored in the *Common Base database*, (meta-)information from the *application-database*, tools, or links to tools from the *tool-box* and user data from the *user database*.

6 Conclusion and Related Work

In the paper we have presented an overview of the concepts, initial and 2nd installment of the Petri Net Baukasten developed by DFG Forschergruppe PETRI NET TECHNOLOGY. For a more detailed presentation we refer to the proceedings of the 1st and 2nd International Colloquium on Petri Net Technologies for Modelling Communication of Based Systems[[25] [7]. One of the main aims of the Petri Net Baukasten is a unified presentation of the large variety of different Petri nets types and techniques. For this reason part I of the state of the art survey [9] on unifying Petri nets is mainly devoted to the Petri Net Baukasten, including an overview of the initial installment, assistance for the application development and an implementation of parameterized net classes with Petri Net Kernel. A main part of the expert view of the Petri Net Baukasten is based on parameterized net classes presented in part II of the same volume [10]. Other contributions closely related to the Expert View are the Petri net transformations [3], rule-based refinement [16], Petri net class transformations [17] and open Petri nets [14] presented in this volume. More details concerning the Application Developer View in general are given in [5], while specific case studies concerning logistics and train control systems are given in other contributions of this volume respectively. Last but not least the contributions [20, 21, 11] are most important for the Tool Developer View presenting the Petri Net Kernel, the Petri net markup language and tool support for animation of Petri nets respectively.

Finally we hope that the work on the Petri Net Baukasten will not only be continued by the Researcher group in Berlin, but also by other groups in different application domains. Moreover the general idea of the Petri Net Baukasten is not at all restricted to Petri nets, but can be extended to other kinds of specification and modelling techniques for communication based systems.

References

- [1] W. M. P. van der Aalst, K. M. van Hee, and G. J. Houben. Modelling workflow management systems with high-level Petri nets. In G. De Michelis, C. Ellis, and G. Memmi, editors, *Petri nets and related formalism*, second Workshop on Computer-Supported Cooperative Work, pages 31–50, 1994. 18
- [2] Nabil R. Adam, Vijayalakshmi Atluri, and Wei-Kuang Huang. Modeling and analysis of workflows using Petri nets. *Journal of Intelligent Information Systems*, 10:131–158, 1998. 18
- [3] B. Braatz, H. Ehrig, and M. Urbasek. Petri Net Transformation in the Petri Net Baukasten. In Ehrig et al. [10]. To Appear. 19
- [4] J. Dehnert. The Common Base of the Petri Net Baukasten. In Weber et al. [18], pages 211–229. 5

- [5] J. Dehnert. Four Steps Towards Sound Process Models. In Ehrig et al. [10]. To Appear. 19
- [6] J. Dehnert, A. Freiheit, and A. Zimmermann. Workflow Modeling and Performance Evaluation with Colored Stochastic Petri Nets. In *Bringing Knowledge to Business Processes, Workshop in the American Association for Artificial Intelligence (AAAI) Spring Symposium Series 2000*, 2000. accepted as poster. 18
- [7] DFG-Forschergruppe PETRI NET TECHNOLOGY. Initial realization of the $\gg$ Petri Net Baukasten $\ll$. Informatik-Berichte 129, Humboldt-Universität zu Berlin, October 1999. 7, 16, 19
- [8] H. Ehrig, M. Gajewsky, and F. Parisi-Presicce. *High-Level Replacement Systems with Applications to Algebraic Specifications and Petri Nets*, chapter 6, pages 341–400. Number 3: Concurrency, Parallelism, and Distribution in Handbook of Graph Grammars and Computing by Graph Transformations. World Scientific, 1999. 16
- [9] H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors. *Advances in Petri Nets: Unifying Petri Nets*, volume 2128 of LNCS. Springer, 2001. 2, 19, 21
- [10] H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors. *Advances in Petri Nets: Petri Net Technologies for Modeling Communication Based Systems*. LNCS. Springer, 2002. To Appear. 19, 20
- [11] C. Ermel, R. Bardohl, and H. Ehrig. Generation of Animation Views for Petri Nets in GENGED. In Ehrig et al. [10]. To Appear. 19
- [12] C. Ermel and M. Weber. Implementation of Parametrized Net Classes with the Petri Net Kernel of the Petri Net Baukasten. In H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors, *Advances in Petri Nets: Unifying Petri Nets*, LNCS. Springer, 2001. 6, 7
- [13] A. Ferscha. Qualitative and quantitative analysis of business workflows using generalized stochastic petri net. In G. Chroust and A. Benczu, editors, *CON'94: Workflow Management - Challenges, Paradigms and Products*, pages 222–234. Oldenbourg Verlag, 1994. 18
- [14] R. Heckel. Open Petri Nets as Semantic Model for Workflow Integration. In Ehrig et al. [10]. To Appear. 19
- [15] S. Lembke. The Application Developer View of the Petri Net Baukasten. In Weber et al. [18], pages 231–241. 5
- [16] J. Padberg and M. Urbasek. Rule-Based Refinement of Petri Nets: A Survey. In Ehrig et al. [10]. To Appear. 19
- [17] F. Parisi-Presicce. A Formal Framework for Petri Net Class Transformations. In Ehrig et al. [10]. To Appear. 19
- [18] H. Weber, H. Ehrig, and W. Reisig, editors. *Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems, Part II: The $\gg$ Petri Net Baukasten $\ll$* . Fraunhofer Gesellschaft ISST, October 1999. 2, 19, 20, 21
- [19] H. Weber, H. Ehrig, and W. Reisig, editors. *2nd Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, Berlin, Germany, 2001. Research Group $\gg$ Petri Net Technology $\ll$, Fraunhofer Gesellschaft ISST. 2
- [20] H. Weber and E. Kindler. The Petri Net Kernel. In Ehrig et al. [10]. To Appear. 19
- [21] H. Weber and E. Kindler. The Petri Net Markup Language. In Ehrig et al. [10]. To Appear. 19
- [22] H. Weber, S. Lembke, and A. Borusan. Improving the Usability of Petri Nets with the $\gg$ Petri Net Baukasten $\ll$. In G. Rozenberg, H. Ehrig, J. Padberg, and G. Juhás, editors, *Unifying Petri Nets*, Advances in Petri Nets. Springer, 2001. To appear. 12

- [23] H. Weber, S. Lembke, and A. Borusan. Improving the Usability of Petri Nets with the Petri Net Baukasten. In Ehrig et al. [9], pages 54–78. 5, 7
- [24] M. Weber. The Tool Developer View of the Petri Net Baukasten. In Weber et al. [18], pages 267–277. 6, 13
- [25] M. Wermelinger. *Specification of Software Architecture Reconfiguration*. PhD thesis, Universidade Nova de Lisboa, 1999. 19

Application Development Assistance with the Petri Net Baukasten

Herbert Weber

Technical University of Berlin
Einsteinufer 17, 10587 Berlin, Germany

Abstract. The use of Petri nets by non-specialists is intended to be simplified with the help of assisting tools and tool environments. They need to bridge the cognitive gap between application-specialists, engineers who develop applications, and Petri net specialists who provide the engineering techniques needed for the application development. Bridging that gap requires information about the three different realms of reasoning to be mapped onto each other. The information about the realms of reasoning is provided primarily in the form of natural language terms and nomenclatures of terms that form structured ontologies. The mapping of ontologies will be assisted by existing systems that provide for the automatic matching of semantic nets.

1 Application Development Assistance

It is assumed that many more applications of Petri nets in different applications would be possible if application developers knew Petri nets and their properties. In many engineering disciplines where the modeling and design of dynamic systems is demanded, Petri nets have not been made known or have only been explained inadequately for the engineers to enable them to take a decision on their usefulness and use. It is further assumed that proper assistance provided to engineers would help them to familiarize themselves with Petri nets and help them to decide on its use in their engineering work. Assistance, hence, refers to two tasks: Enabling engineers to understand Petri nets in their various variants and types and to understand the reasoning behind the development of the many variants, as well as to help engineers in their selection of the right Petri net variant for the particular engineering task. In order to develop that kind of assistance it is important to understand applications and engineering practices applied in the development of applications. It has been acknowledged in [1] that application development assistance is needed for non-Petri net-specialists to find their way into the usage potentials of Petri nets. It was especially recognized that application development assistance must follow a number of essential objectives.

1.1 Objectives

As described in [3] assisted application development pursues several objectives, which we will shortly explain again in the following.

Application Orientation. A main objective of assisted application development is application orientation. Application orientation is achieved by introducing an application domain specific interpretation of Petri nets. In our opinion, application developers can relate to the world of Petri nets more easily and quickly using familiar technical terms from their own application domain. Therefore, technical terms of particular application domains are mapped onto Petri nets. For example, the technical terms in the area of business process modeling are business process, process activity and roles which denote responsibilities. They may be mapped onto terms like transition, places, token flow etc. in Petri net terminology.

Problem Orientation. Each application development for a certain application domain is characterized by its specific problems. For example, business process modeling may concentrate on the modeling of existing business processes in one case. The modeled entities are then process activities, the logical order between these process activities, responsibilities etc. Business process modeling for other cases may emphasize the improvement of existing processes that relate then more to entities like time consumption and distribution. Assisted application development should therefore not only deal with several application domains but also with several typical instances of an application domain.

Method Orientation. Development methods explaining the use of Petri nets in application development processes have not been examined and documented enough. The usage of Petri nets in application development can become more effective if development methods are made available since these allow application developers to orientate themselves within their own development processes.

Solution Orientation. Assisted application development should make standard sample solutions available in order to reduce the time needed for orientation. The developer should be able to integrate such solutions into his work and to adapt them if necessary. Standard sample solutions may be independent of applications or application specific.

Tool Orientation. Modern application development doesn't work without suitable tool support during the phases of a development process. Therefore, assisted application development only makes sense if tools are provided for application developers which support Petri net techniques, methods, etc.

Need Orientation. Assisted application development shall be need-oriented in accordance with the requirements of a certain application development process. Especially the assistance (selection of methods, techniques, tools, sample solutions) should follow the principle of *you only get what you really need*.

Multiple-Way Orientation. Assisted application development should be provided by several assistance variants in order to find either the right development method, the right Petri net technique, the right Petri net type, the right tool or the right sample solution. All of these variants should be realized by several assistance methods, such as prescriptive, navigative or descriptive assistance methods. These allow application developers to choose the most suitable type of assistance.

1.2 Use Cases

This section describes several assisted application development scenarios as use cases. The use cases focus on selecting a suitable Petri net technique, Petri net method, Petri net type, Petri net tool, or sample solution.

Select a Petri Net Technique. The use case *select a Petri net technique* outlines the assistance variant for finding a suitable Petri net technique. A Petri net technique can be selected in different ways, for example by using a form-based or an overview-based search, compare Fig. 1.

If application developers use a form-based search then they can specify required characteristics of a Petri net technique in predefined boxes. This form enables application developers to find a suitable Petri net technique by keywords specifying its intended objective, application domain and its underlying Petri net type. Petri net techniques fulfilling the specified characteristics are determined automatically and offered to the application developers. Before application developers select their suitable Petri net technique from the offered set, they may

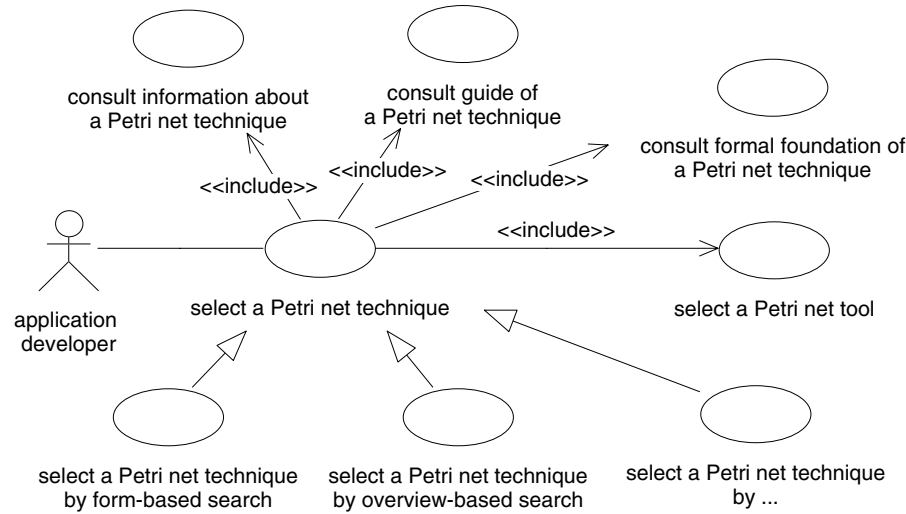


Fig. 1. Use case: select a Petri net technique

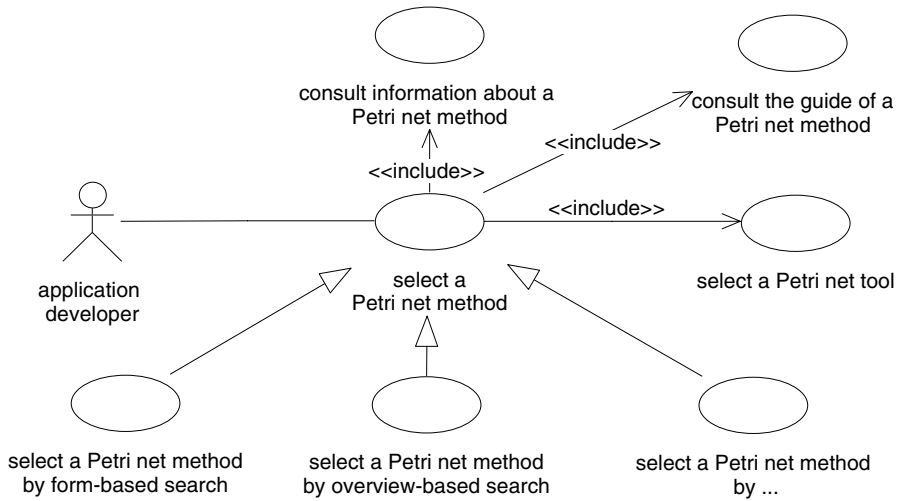


Fig. 2. Use case: select a Petri net method

use several information about it, like information about the application domain, its objective, the underlying Petri net type, or the essentials of its procedure. Moreover, they may use its informal description, the so-called technique guide, or its formal foundation. Last but not least, a suitable Petri net tool supporting the preferred Petri net technique is selected.

Select a Petri Net Method. The use case *select a Petri net method* outlines the assistance variant for finding a suitable Petri net-based development method. Similarly as described above, a Petri net method can also be selected in different ways.

It allows search over objectives, application domains and covered development phases (analysis, design, etc.) of Petri net methods. Here, application developers can select a preferred kind of an overview, for example a phase-related overview of Petri net methods. In this overview the existing Petri net methods are sorted according to their covered phases, for example pure analysis methods, pure design methods, etc. A further overview may sort Petri net methods according to their intended application domain.

Application developers can use different information about each offered Petri net method. For example, its intended objective, its covered phases, its underlying languages, and techniques. Additionally, they may also consult the method guide before selecting their suitable Petri net method. Finally, application developers select tools which enable the application of the selected method.

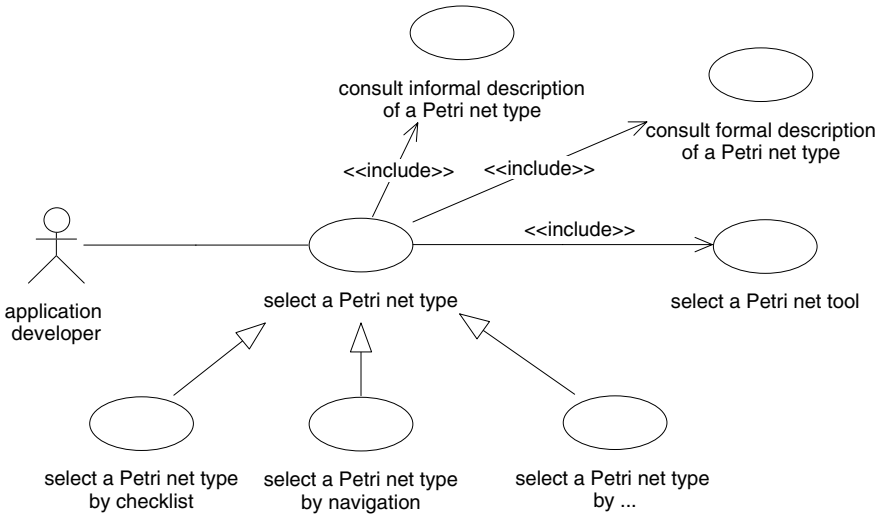


Fig. 3. Use case: select a Petri net type

Select a Petri Net Type. The use case *select a Petri net type* outlines the assistance variant for finding a suitable Petri net type which fulfills the requirements of a particular application development process. It is specialized by several use cases which reflect several assistance methods for finding a suitable Petri net type. These may be *select a Petri net type by checklist*, *select a Petri net type by navigation* etc., compare Fig. 3.

The navigation based use case enables the selection of Petri net types by navigation through the Petri net classification described in [4].

Select a Petri Net Tool. The use case *select a Petri net tool* has been already mentioned in all use cases described above. It supports application developers to select suitable Petri net tools which enable the application of a Petri net technique, method, or type selected before. The selection of a Petri net tool to enable the application of a Petri net technique and the selection to enable the application of a Petri net method differ a little from the selection to enable the application of a Petri net type. But in all cases, application developers may use information about a Petri net tool and download it, compare Fig. 4.

Let us have a closer look to select a Petri net tool to enable the application of a Petri net method or Petri net technique. The application of a Petri net method or Petri net technique can be enabled by a certain set of Petri net tools which are integrated to a workbench or by a certain Petri net-based development environment. In both cases, underlying Petri net types and several modeling, assess, or implementation functionalities required by a certain Petri net method or Petri net technique have to be supported.

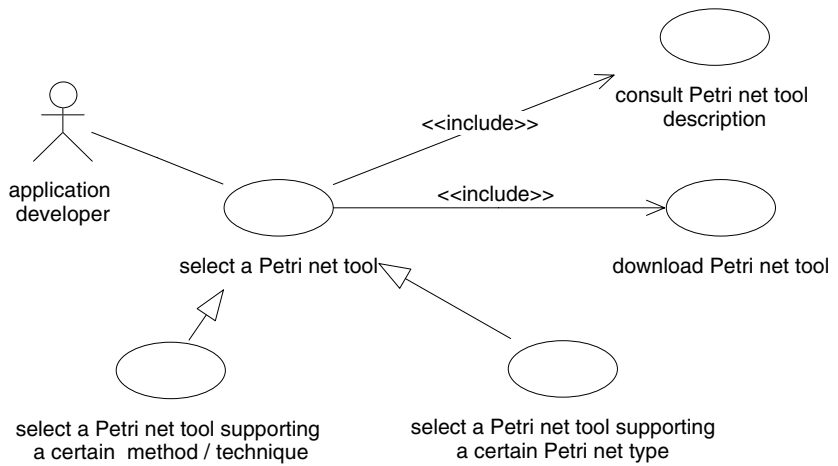


Fig. 4. Use case: select a Petri net tools

Application developers may use information about a selected tool recommendation, for example about the required operating system (e.g. UNIX) or the purpose of a single tool (e.g. supports modeling), and about a selected single tool of a tool recommendation, for example about its usage costs or its usability. Such information explained in [5] as technical-functional criteria and socially-assessable criteria enable application developers to select the suitable tool or tool set for their development task. Application developers can also define such criteria as further requirements for suitable tool or tool sets. If application developers have identified a suitable tool or tool set they can download/order it from its supplier.

The selection of a Petri net tool to enable the application of a certain Petri net type additionally comprises the specification of the required functionality. In the cases described above, the required functionality is already determined by the Petri net technique or method. In this case, application developers determine the required modeling, assessment, and implementation functionality to enable the application of Petri net type selected before yourself.

Select a Sample Solution. The final use case described here is intended for finding the proposed solutions suitable for fulfilling the requirements of a certain application development. It is called *select a sample solution* and is shown in Fig. 5.

Basically, a sample solution can be selected in different ways, for example by a form-based or an overview-based search. If a form-based search is used then application developers can specify required characteristics of suitable sample solutions similar to the form-based search described for the selection of a Petri net technique or the selection of a Petri net method. It enables application devel-

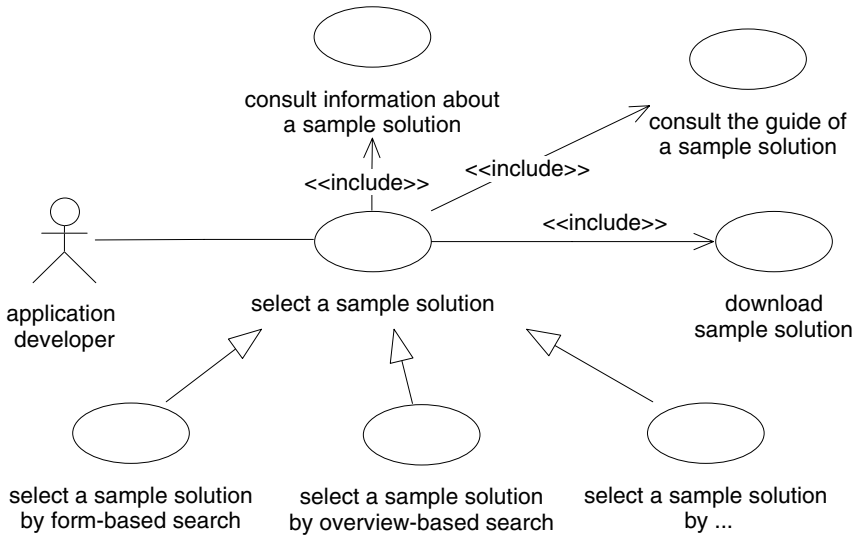


Fig. 5. Use case: select a sample solution

opers to find suitable sample solutions by their objectives, application domains, and their underlying Petri net types.

The overview-based search offers several kind of overviews of sample solutions, for example an application domain specific overview. Such an overview sorts sample solution according to their indented application domain.

Before application developers select a certain sample solution to integrate it within their own application development they can use information about it, for example about the solved problem or the underlying Petri net type, and can consult the so-called solution guide for details.

2 Structured Ontologies as Application Domain Models

It is important to understand that applications themselves and the engineering practices applied in these application developments adhere to different terminologies and nomenclatures specific to the respective application and engineering practices. This may be exemplified with the development of work processes that organize, support, govern and control collaborative work in large organizations conducted by large numbers of people. The nomenclature used in the development of work processes comprises terms like "division of labor", "business goal", "involved people", "skill requirements of involved people", "report structures", "organizational structures", "sequences of actions", etc. Engineers in development of these applications use terms like actions, activities, processes,

pre-conditions, post-conditions, ordering of actions, refinements of actions, enaction of processes, allocation of resources, etc. The two terminologies are sufficiently different to create confusion and misunderstandings between application specialists, defining the requirement for the application development, and engineers who develop the applications in accordance with the requirements provided by the application specialists. The reasoning introduced above and the example both indicate that application development assistance must provide for

- A recording of the terminologies used by application experts and engineers
- A proper explanation of the terms used in both by application domain experts and by engineers
- A mapping mechanism that allows the mapping of terms used in the application domain onto terms used by engineers.

Application development with Petri net-technologies hence requires two ontologies comprising the terms used in the application domain, and the terms used in Petri net-technologies.

The picture above represents terms in the two different ontologies and the mapping existing between the terms. Since the terms, used in both the application domain and in engineering, are usually not independent of each other but form a structured nomenclature the mapping between the two ontologies or subsets of these ontologies results in the mapping of term patterns.

This simple example already indicates that matching of term pattern may result into the following:

- A matching can be found for one or some terms of a pattern of one ontology with one or some terms of a pattern of another ontology but not for all terms of a pattern. This leads to a partial match.
- A matching can be found for all terms of a pattern of one ontology with all terms of a pattern of another ontology. This leads to a perfect match.
- No matching can be found for any terms of a pattern of one ontology with any pattern of another ontology. This leads to an incompatibility of the nomenclatures.

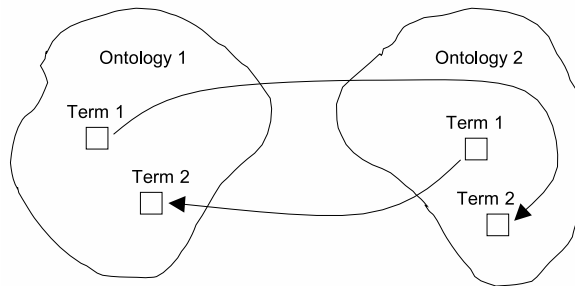


Fig. 6. Mapping of ontologies

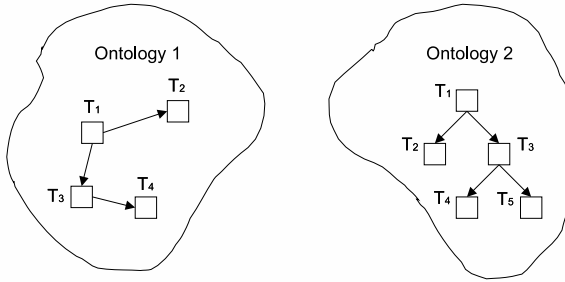


Fig. 7. Term patterns in ontologies

It is quite obvious that matches of whatever kind cannot be found on the syntactic form of terms used in the respective ontology but must be based on a common understanding of the terms. We hence call the possible matching between nomenclatures in an application domain and in the engineering domains semantic matches. Semantic matches can usually be found only in a clarification dialogue between the application expert and the engineer. Application development assistance hence refers to the following concepts:

1. Provision of an ontology for the terms of the nomenclature used in the application domain
2. Provision of an ontology for the terms of the nomenclature used in engineering
3. Support in the identification of matches between terms of the two ontologies

The development of application development assistance hence refers to the following tasks: the development of:

- Proper ontologies comprising the terms and explanations of the meaning of these terms (by use of other terms or phrases built of terms)
- Proper matching techniques such as identifications for syntactic matches, identification of synonyms and homonyms in ontologies, identification of patterns in the ontologies
- Decision support on the evaluation of the quality of matches - if they are only partial - as to whether they are still sufficient or not.

All of them are basically engineering tasks that require engineering concepts that guide engineers in their intellectual endeavor. As most of the application domains have established their own engineering concepts that suit the application domain requirements best, many of the problems related to the task above may seem to be easily solvable. But the engineering concepts, successfully applied in different application domains, may themselves differ significantly and are not necessarily close to engineering with the aid of Petri nets. As a consequence the picture presented above may be more complicated as the full matching process will have to take place in two steps where the

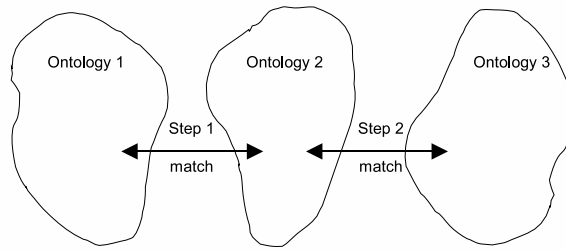


Fig. 8. Two step matching of ontologies

- First step provides for the match between application domain terms and nomenclatures and the application domain specific engineering terms and nomenclatures
- The second step provides for the match between the application domain specific engineering terms and nomenclatures and the Petri net technology specific terms and nomenclatures

Obviously this complicates the matching process significantly since it involves now three parties: the application domain specialist, the application domain specialized engineer and the Petri net technology competent engineer. Even though this two step approach should be avoided wherever possible, experience tells us that introducing Petri nets into new application domains usually requires the dialogue with the engineer, who's specialized in application domain, for the simple reason of a shorter cognitive distance. This may be explained with the following example that shows three different ontologies used in a development task.

Ontology 1 comprises terms such as "contracts", "customers", "deadlines", etc. Ontology 2 comprises terms such as "business processes", "orderings of activities", "enaction", etc. Ontology 3 comprises terms such as "program", "algorithm", "run time", "database" etc. It is very obvious that business professionals

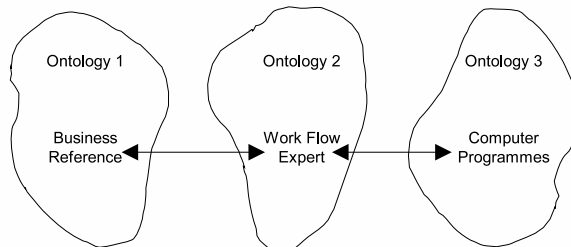


Fig. 9. Sample two step matching

will be able to communicate with workflow experts more easily as with computer programmers.

As indicated before the necessary matching is basically the result of a clarification dialogue between two partners. The depth in specificity in both application domain requirements and application domain engineering concepts require application domain assistance that is as specific as possible to both the application domain and the engineering practiced in the application domain. As a consequence application development assistance will be application domain specific.

3 Application Development Assistance with Structured Ontologies

As indicated in the previous chapter ontologies play a crucial role in application development assistance since both requirements and possible applicable engineering concepts are identified through their denotation by terms and nomenclatures. Communication between application domain experts and engineers in their clarification dialogue is bound to these terms and nomenclatures and need to be supported by proper ontologies. For a proper understanding of how ontologies may support the matching of terms and nomenclatures we need to specify their basic structure.

A first instance of such an ontology was the Petri net classification introduced in the common base of the Petri net Baukasten [4] Since we intend to support the matching process as much as possible through computer assistance we restrict ourselves in the structuring of the ontologies to those structures that can be treated automatically without increasing the complexity of the matching process beyond practical limits. The basic structure for ontologies chosen here are therefore directed acyclic graphs. The graphs are defined by a generalization/specialization relation: Nodes connected over the arc are hence in a superior/subordinate relationship. The following explains the relationship with an example. An ontology on the application domain "Business process management" may hence in part be depicted as follows:

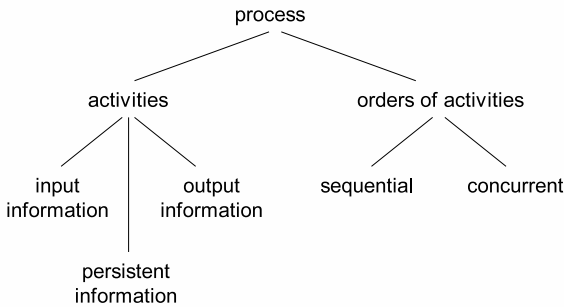


Fig. 10. Business process management ontology

An ontology on the engineering concepts may be depicted as follows:

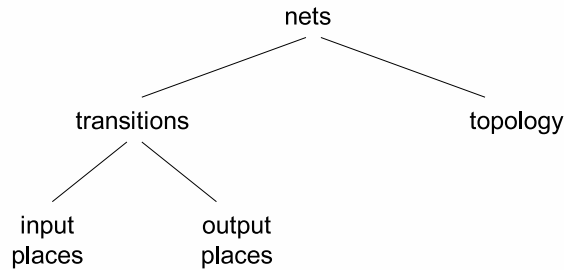


Fig. 11. Engineering concepts ontology

It is very obvious that a match between the two ontologies may exist for somebody who understands both business process management and Petri net technologies. Since in real practical situations these dual competencies cannot be assumed other means of matchfinding must be supported.

For their recording ontologies may be described in terms of UML class diagrams. They provide for the graphic depiction of generalization and specialization hierarchies between classes. They are also suited to depict terms and more complex forms of terms in which terms are put into proper syntactic structures that may be used as a whole in the characterization of the application domain. The particular virtue of using UML class diagrams as depiction of application domain characterization however, is that they obviously support the matching process directly since Petri net characteristics are depicted as UML classes as well. UML depiction will also have a role to play in another basic assistance process: In the navigation of application domain characteristics and Petri net characteristics. The development of matches with the aid of ontologies is assumed to take place as follows:

1. The application specialist is attempting a first match between the terms of concern in his nomenclature with terms used in the nomenclature of Petri nets or alternatively with terms used by the engineering specialists of the respective application domain. This first match may deliver an anchor point in the ontology of Petri net terms or of terms used by the engineers of the application domain. (Fig. 12)
2. Based on the finding of an anchor point the application development specialist inspects the neighborhood of the anchor point to understand the respective term of the anchor point and develops a first guess for the meaning of the term found in the anchor point. Based on the inspection of the neighborhood the application development specialist compares the/his findings in the engineering or Petri net ontology with the neighborhood of the initial term in his own ontology to develop a first matching of neighborhoods. (Fig. 13)

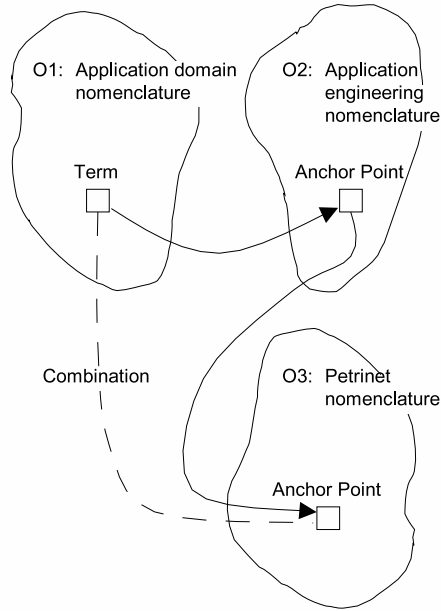


Fig. 12. Matching of ontologies: Finding of anchor points

This process can be repeated many times until the application development specialist is convinced of the found matching.

The two steps introduced above represent two basically different search strategies for terms and forms of terms. Whereas the first step refers to a key word search in the engineering or Petri net ontology based on the selection of a term from the application domain ontology, the second step refers to a form based search where a whole pattern of terms is compared to another pattern of terms for their match. Both of theses approaches can be supported by assistance systems that are available now. A system by the name BaSeWeP, which is capable to match forms of different ontologies that are structured as acyclic directed graphs, has been developed and can be made available [2]. The initial development of matches is unquestionable a highly demanding engineering task that can be performed by experienced application development experts and sometimes only with the help of Petri net-experts in a clarification dialogue. It is therefore important to record matches that have been developed in those cases and to make them available as sample matches for later use in other application developments with Petri nets. Matches that are of interest as samples for their later use are classified as those between

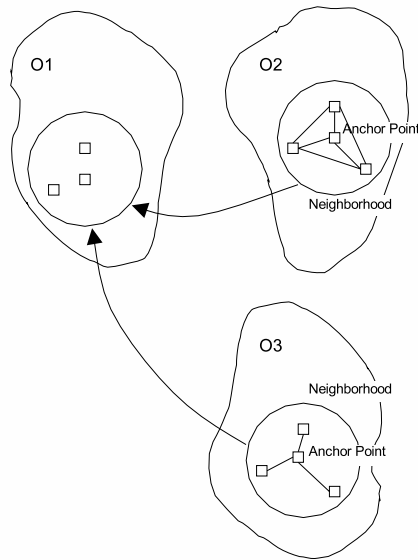


Fig. 13. Matches of ontologies: Neighborhood Inspection

engineering paradigms
 engineering principles
 engineering techniques
 etc.

that will then be matched properly with the

application domain paradigms
 application domain principles
 application domain techniques
 etc.

and will be kept as "stored matches" in the data base of the assistance system. Next to the storing of matches it will also be important to store meaningful navigation paths that one will have to follow in the inspection of an ontology. They altogether form the neighborhood of an anchor point and are subject of a forms match later on.

4 Conclusion

The paper outlines the assistance concept developed for the Petri net Baukasten to aid in the development of application of Petri nets. It conforms in many ways to similar concepts and systems that trade or broker information based on their syntactic structures and syntactic meta structures.

References

- [1] A. Battke, A. Borusan, J. Dehnert, H. Ehrig, C. Ermel, M. Gajewsky, K. Hoffmann, B. Hohberg, G. Juhás, S. Lembke, A. Martens, J. Padberg, W. Reisig, T. Vesper, H. Weber, and M. Weber. Initial realization of the $\gg$ Petri Net Baukasten $\ll$. Informatik-Bericht 129, Humboldt-Universität zu Berlin, Berlin, Germany, 1999. DFG-Forschergruppe PETRI NET TECHNOLOGY. 22
- [2] A. Billig, K. Sandkuhl, and A. Wendt. XML-based Content Management in Web-Portals: The BaseWeP Approach to Evolution. In B. Furht: Internet and Multimedia Systems and Applications Conference. Proceedings of the IASTED International Conference Las Vegas, Nevada, 2000. 34
- [3] C. Ermel and M. Weber. Implementation of Parameterized Net Classes with the Petri Net Kernel. In G. Rozenberg, H. Ehrig, J. Padberg, and G. Juhás, editors, *Unifying Petri Nets*, Advances in Petri Nets. Springer, LNCS, 2001. 22
- [4] M. Gajewsky and H. Ehrig. The $\gg$ Petri Net Baukasten $\ll$: An Overview. In G. Rozenberg, H. Ehrig, J. Padberg, and G. Juhás, editors, *Unifying Petri Nets*, Advances in Petri Nets. Springer, LNCS, 2001. 26, 32
- [5] Dietmar Wikarski. Petri net tools—a comparative study. Forschungsberichte des Fachbereichs Informatik 97-4, Technische Universität Berlin, 1997. 27

Petri Net Transformations in the $\gg$ Petri Net Baukasten $\ll$ *

B. Braatz, H. Ehrig, and M. Urbásek

Technical University Berlin, Germany
Institute for Software Technology and Theoretical Computer Science
{bbraatz,ehrig,urbasek}@cs.tu-berlin.de

Abstract. The purpose of this contribution is to give an overview of constructions and results for Petri net transformations in the $\gg$ Petri Net Baukasten $\ll$ developed by the “DFG-Forschergruppe PETRINETZ-TECHNOLOGIE”. The two main concepts of Petri net transformations considered in this context are net class and net model transformations. In both cases we present first the relevance of transformations in the application developer view and then the technical constructions and results in the expert view of the $\gg$ Petri Net Baukasten $\ll$. Net class transformations are transformations between different Petri net classes, like elementary nets, place/transition nets and algebraic high-level nets. Net model transformations on the other hand are transformations of the net structure for nets within one Petri net class, like place or transition refinement. The main technical results are concerning the preservation of safety resp. liveness properties of net model transformations and compatibility results between net class and net model transformations. The relevance of the constructions and results for the application developer view is demonstrated by a small case study modeling the interaction of a buffer, a printer and a communication unit consisting of secure and non-secure channels. Finally we give an overview of other concepts of transformations: On one hand, transformations between Petri nets and other system modeling techniques, and on the other hand transformations between different net representation formats using XML-schemes and DTD-standards.

1 Introduction

The $\gg$ Petri Net Baukasten $\ll$ presented in [WER99, DFG99, GE01] and in chapter 1 of this volume, provides a unified presentation with different views on theory, applications and tools of Petri nets: the Expert View, the Application Developer View and the Tool Developer View. In this paper we show that different kinds of Petri net transformations play an important role in these three views.

* This work is a part of the joint research project “DFG-Forschergruppe PETRINETZ-TECHNOLOGIE” between H. Weber (Coordinator), H. Ehrig (both from the Technical University Berlin) and W. Reisig (Humboldt University Berlin), supported by the German Research Council (DFG).

In the Application Developer View net transformations are essential for realizing stepwise system development within the given process models. These process models present a methodology in the way Petri nets are to be used in the different steps during software development. They typically start with an abstract model of the system which is refined in further development steps. Refinement here means integration of system aspects like time, roles, data, etc. as well as modification of these aspects for incorporating exception handling, business rules, etc.

Transformations in the Tool Developer View mainly concern the data structures used in Petri net tools. They yield an exchange format in order to couple different tools. Thus, Petri net models created with one tool can be transferred to another tool. Consequently, algorithms of the target tool can be employed which have not been available in the source tool. Thus, transformations in this sense are used to link tools together, which enables the development of tool environments.

In the Expert View transformations are used to perform modifications of a net. They are formalized on a rigorous mathematical foundation, see [GPP01b]. For a systematic study of such Petri net transformations we distinguish two levels of transformations, called *net model transformations* and *net class transformations*. We consider model transformations given by rule-based modifications in the context of High-Level Replacement (HLR) Systems [EHKP91, EGP99] and class transformations by functors in the sense of category theory [AHS90]. The purpose of the formal transformations is twofold:

- On one hand net class transformations extend the theory of a given net class in the following sense: Petri net operations in the target net class are made available also for the source net class by transforming the net class, performing the operation, and subsequently interpreting back the result of the operation in the source net class.
- On the other hand both kinds of transformations together allow arbitrary modifications of a (start) net. Therefore, they are suitable to support stepwise enhancement of nets in the context of system development. In this sense, they yield a formal support of the Petri net based process models in the Application Developer View.

1.1 Overview of the Results

The following results concerning transformations have been achieved in the “DFG-Forschergruppe PETRINETZ-TECHNOLOGIE”, where an overview of the main constructions and results are presented in more detail in sections 2 and 3 of this paper.

1. Classification of transformations w.r.t. preservation of system properties

Preservation of behavioral properties like liveness, deadlock freedom, or safety properties are important during the development of the system. In order to

ensure preservation of these properties by transformations, both net class transformations and net model transformations have to be classified according to the preservation of these properties.

In the case of net model transformations preservation of system properties leads to rule-based refinement introduced in [Pad96, Pad99]. Rule-based refinement has already been employed for the preservation of safety properties in [PGE98, PG98, PGH99, PHG00] and for liveness-preserving transformations in [GPU01]. Net class transformations have to be treated separately. There are already results in the literature concerning preservation of properties by functors, e. g. [Lil95], which however do not cover all system properties and are only formulated for particular functors.

2. Compatibility of structuring and transformation

For large systems adequate structuring techniques for composing subsystems and decomposing are indispensable. For transformation of subsystems and composition of transformed models, structuring is needed to be compatible with transformation.

On the level of model transformations this requirement is already given, see e. g. [PER95, EGP99]. On the level of net class transformations compatibility is granted if the net class functor preserves colimits, see [GPP00].

3. Consistency between models of different levels of abstraction

In Petri net based process models several models of the system are achieved on different levels of abstraction. Of course, all of these models should be consistent with each other, meaning that each model is a proper enhancement of the start model. This requirement can be achieved if model transformations are preserved by net class transformations between different levels of abstraction.

The corresponding results have been proved in [GPP00, GPP01b].

1.2 Introduction to the Case Study

To illustrate the relevance of Petri net transformations in the application developer view of the $\gg$ Petri Net Baukasten $\ll$ we will model a simple printing system and show how the development of this model can be structured with net model and net class transformations.

The system to be modelled consists of a buffer, a printer and a communication unit. The printer can only print one task at the same time. The communication unit has two channels, one secure channel and one channel for which the integrity of the printing task has to be verified after the transmission by comparing two copies of the task, which are sent independently.

In Sect. 2 two ways of obtaining a place/transition net model of this system using stepwise net model transformations are explored. First, the system is set up by introducing the components buffer, printer and communication unit separately and using gluing transformations to connect the components. Secondly, the system is introduced as a raw model and refinement rules are used to elaborate the printing and the transmission.

In Sect. 3 we show how the modelling of the system can be done using different Petri net classes on different layers of abstraction. First, a simple elementary net model is created and then transformed to place/transition nets, where the model of Sect. 2 is obtained. Then, the place/transition net is transformed to algebraic high level nets. This allows us to specify that the two tasks sent over the insecure channel can really be compared for equality.

1.3 Organization of this Paper

In section 2 we present net model transformations. Concerning the Application Developer View we show how to use transformations for stepwise development of communication based systems and discuss safety and liveness preserving transformations in our case study. The formal framework for net model transformations and main results concerning preservation of safety and liveness are contributions to the Expert View.

Net class transformations are presented in section 3. The case study is extended as a contribution to the Application Developer View. Net class transformations are formalized by functors between suitable net class categories in the Expert View.

A more detailed presentation of net class transformations can be found in the paper [PP02] in this volume.

The subsections of sections 2 and 3 concerning the Application Developer View can be read independently of those concerning the Expert View.

Finally in section 4 we give an overview of other transformations in the $\gg$ Petri Net Baukasten $\ll$, especially concerning the Tool Developer View, and a conclusion sketching further interesting lines of research concerning Petri net transformations.

2 Net Model Transformations

Net model transformations play an important role in the $\gg$ Petri Net Baukasten $\ll$. In the following subsections we discuss net model transformations from the Application Developer View and the Expert View.

2.1 Application Developer View

The main focus of an application developer is how net model transformations can be used to build up an application. The major applicability domain of net model transformations is stepwise development of communication based systems, which will be discussed next.

Stepwise Development of Communication Based-Systems. The main idea of stepwise development of systems is to offer to an application developer a number of transformation rules for different classes of Petri nets. These rules

describe how to change a net model in order to obtain another more elaborated, refined and expressive net model.

Each transformation step can be depicted as a transformation according to an appropriate rule. The sequence of transformations then provides a transformation from the initial net G to the final net H :

$$G = G_0 \xRightarrow{p_1} G_1 \xRightarrow{p_2} \dots \xRightarrow{p_n} G_n = H.$$

Each step $G_i \xRightarrow{p_{i+1}} G_{i+1}$ denotes a single transformation. The transformation process is rule-based and has a formal mathematical foundation, which is defined in the Expert View (see Section 2.2).

The preservation of suitable system properties during the transformation process is of interest in most applications, since the final model may be very large and hence difficult to check for certain system properties. It is of great importance for the application developer to check only the initial – usually quite small system – directly and to apply property preserving rules and transformations. For this reason structure and property preserving rules and transformations are supported in the Application Developer View of the $\gg$ Petri Net Baukasten $\ll$, especially horizontal structural techniques (union and fusion), safety-properties preserving rules for P/T, colored and AHL Petri nets, liveness preserving rules for P/T Petri nets.

A slightly different approach developer also in the “DFG- Forschergruppe PETRINETZ-TECHNOLOGIE” is the transition refinement technique for distributed algorithms in [Peu01]. On one hand transition refinement in [Peu01], where a specific class of high-level nets is considered and system properties can be expressed by general temporal logic formulas, is more general than rule based transformation, where only a specific class of formulas is considered. On the other hand rule based transformations are more general concerning the applicability to a large variety of low- and high-level nets classes, which can be considered as instantiations of high-level replacement systems (see [Pad96]). Safety property preserving transformations have been developed for low- and high-level nets [GHP99, PG00b, PGE98], liveness preserving transformations, however up to now for P/T nets only [GPU01].

Case Study: Simple Communication Based System. We illustrate the rule-based stepwise development of a simple communication based system constructed as interconnection of three components: a buffer with two tasks, a printer and a communication unit (network) between buffer and printer consisting of a secure and non-secure channel. The components are depicted on Figure 1.

The behavior of the printer and the buffer are obvious from the figure. The communication unit serves a secure or non-secure communication from the buffer to the printer.

This unit receives a task which has to be sent over a transition (**GT**). It can send the message through a secure channel (**SSC**) or via a non-secure one (**NSC₀**, **NSC₁**, **NSC₂**). On the non-secure channel the message may become

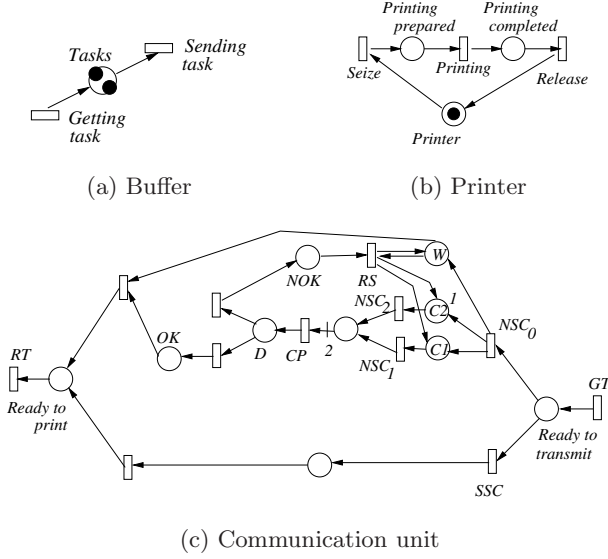


Fig. 1. Components of the system

corrupted. Therefore, when a non-secure channel is used, two copies (**C1**, **C2**) of the message are to be sent and a transmission subunit waits (**W**) for an acknowledgment. A receiving subunit on the other end receives both copies and compares them (**CP**). (We assume that a message cannot be lost during the transfer.) If both copies are the same, the **OK** acknowledgement is sent back and the communication is ended. If both copies differ, then **NOK** acknowledgement is sent back and the transmission subunit resends (**RS**) the message again (in two copies). The two possible results of the comparison are modeled by a nondeterministic choice (conflict) in place **D**. The communication ends when a successful transfer is performed (**RT**).

The three components of the system shown in Figure 1 are interconnected by the application of the rules shown in Figure 2. In a first step we apply the buffer-printer rule in Figure 2a to the buffer and printer components in Figure 1. The corresponding transformation is shown in Figure 2a.

The rule buffer-printer is shown in the top row of Figure 2a. It consists of a left-hand side net L , a right-hand side net R and an interface net I . The transformation from net N_1 to net N_2 via this rule is constructed in two steps. In the first step we apply L to N_1 and remove all items of L in N_1 which do not belong to the interface I . It leads to the intermediate net N_0 . In the second step we glue together nets N_0 and R via the interface I leading to the net N_2 . It is important to note that also N_1 can be considered as a result of a gluing construction, namely the gluing of N_0 and L along I such that the transformation in Figure 2a consists of two gluing constructions shown in diagrams (1) and (2)

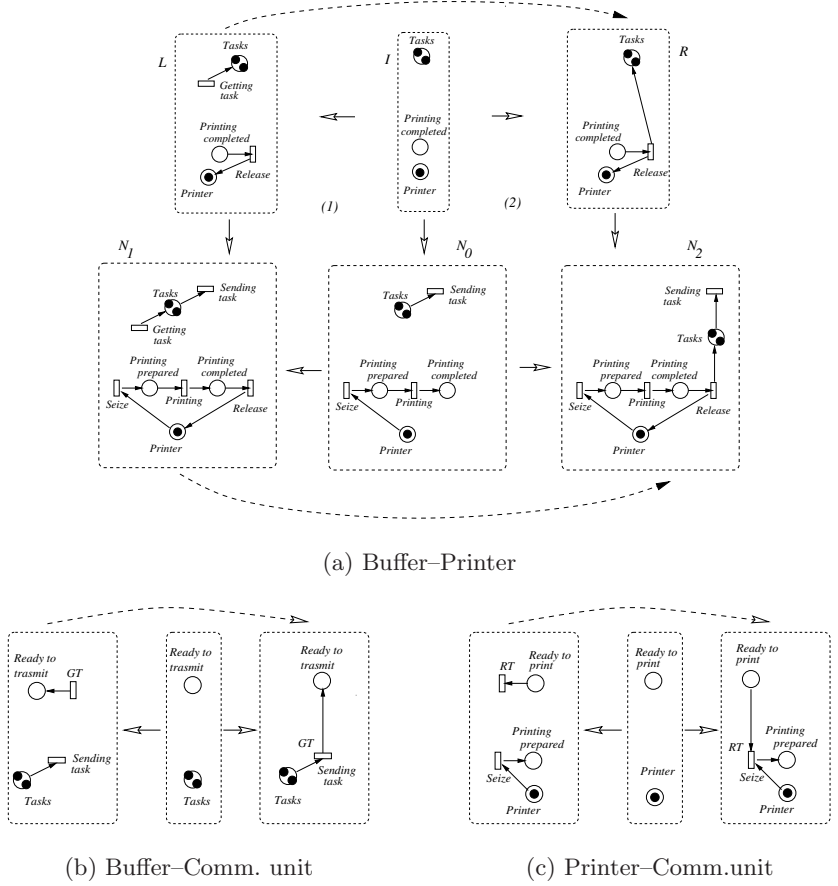


Fig. 2. Interconnection of components

respectively according to the general construction of net model transformations in Section 2.2.

In a similar way we can apply the two other rules in Figures 2b,c to net N_2 combined with the communication unit net in Figure 1 leading to the net in Figure 3.

Finally we can apply the rule in Figure 4 modeling mutual exclusion between the secure and the non-secure channel leading to the final model in Figure 5.

Transformations Preserving Safety Properties. Now we want to show suitable safety properties for the final model in Figure 5 from corresponding properties of the basic components in Figure 1 and the fact that the transformations from the basic components to the final model in Figure 5 considered above are safety properties preserving.

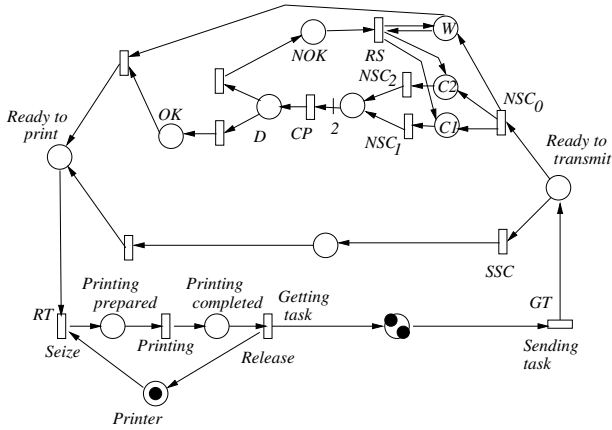


Fig. 3. Simple tasks' model

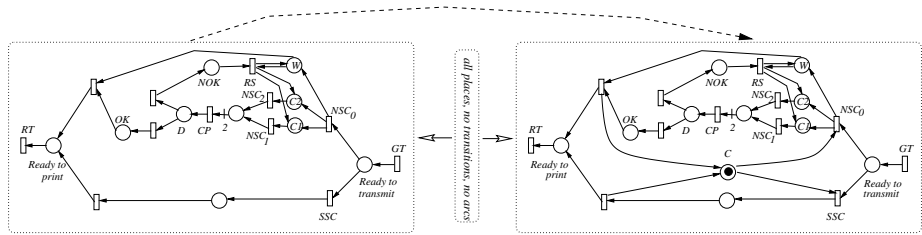


Fig. 4. Modeling exclusivity

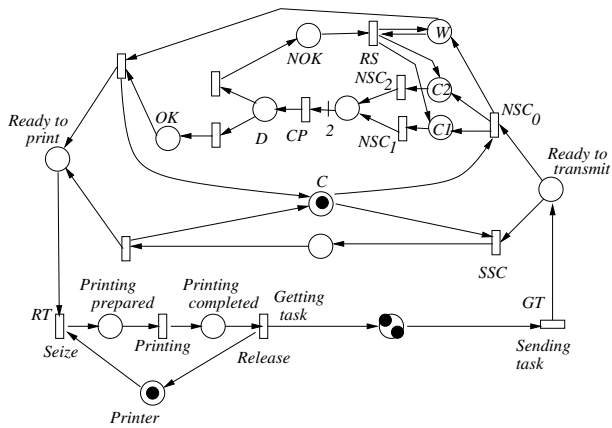


Fig. 5. Final model

Safety properties say in general that “nothing bad can happen” in a the system. Safety property is expressed by suitable temporal logic formulas. These formulas state a fact about the marking of the net and are given in terms of a number of formal expressions concerning tokens on places. The static formula $2\mathbf{d} \wedge 3\mathbf{a}$ is true for a marking M where at least 2 tokens are present on place $\mathbf{d}$ and at least 3 tokens on place $\mathbf{a}$. The always operator $\Box$ in a safety-property $\Box(2\mathbf{d} \wedge 3\mathbf{a})$ states that static formula $2\mathbf{d} \wedge 3\mathbf{a}$ is true for all reachable markings from M .

In our case study we have the safety property

$$\Box((\mathbf{NOK} \vee \mathbf{OK}) \implies \mathbf{W})$$

for the communication unit in the Figure 1c. This property expresses a fact that, independently of the result $\mathbf{NOK}$ or $\mathbf{OK}$ of the comparison of the two copies $\mathbf{C1}$ and $\mathbf{C2}$ modeled by nondeterministic choice in place $\mathbf{C}$, the transmission subunit waits for acknowledgement in place $\mathbf{W}$.

For the printer net in Figure 1 we have the safety property that at most one job is processed in each moment, i.e.

$$\Box((\mathbf{P} \text{ xor } \mathbf{PP} \text{ xor } \mathbf{PC}) \wedge \neg(\mathbf{P} \wedge \mathbf{PP} \wedge \mathbf{PC})),$$

where $\mathbf{P}$, $\mathbf{PP}$ and $\mathbf{PC}$ stand for *Printer*, *Printing prepared* and *Printing completed*, respectively and *xor* for exclusive or operator.

Surely, we would like to keep these safety properties valid during the transformation. There are two classes of rules available for the application developer which preserve safety properties for P/T Petri nets, namely transition gluing and place preserving rules, see [GHP99].

The main result concerning safety property preserving transformations reviewed in Section 2.2 states that for each net transformation sequence $N_1 \implies^* N_2$ via safety property preserving rules, where N_1 satisfies a safety property φ , we can conclude that net N_2 satisfies a corresponding translated safety property $\mathcal{T}(\varphi)$. In our small case study the rules shown in Figure 2 are *transition gluing* rules and the rule in Figure 4 is *place preserving*. This implies that the two safety properties for the communication unit and the printer considered above are also true in the final model in the Figure 5.

Liveness Preserving Transformations. Let us recall that a net is called live if for each reachable marking m and each transition t there is some other marking m' reachable from m such that t is enabled under m' . We want to show via liveness preserving transformations that the final model in Figure 5 is live (see main result concerning liveness preserving transformations in Section 2.2).

For this purpose we consider the simple model of printing tasks in Figure 6, where it is easy to check directly that this net is live. The printing-refinement and transmission-refinement rules in Figure 7 are liveness preserving rules and can be applied to the net in Figure 6 leading to the final model in Figure 5. (The meaning of marked subnets on the right-hand sides of the rules is explained in Section 2.2.) Since the net in Figure 6 is live and the rules liveness preserving also the final model is live.

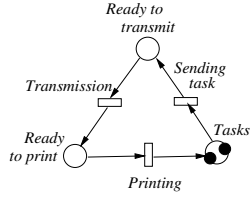


Fig. 6. Simple model of printing tasks

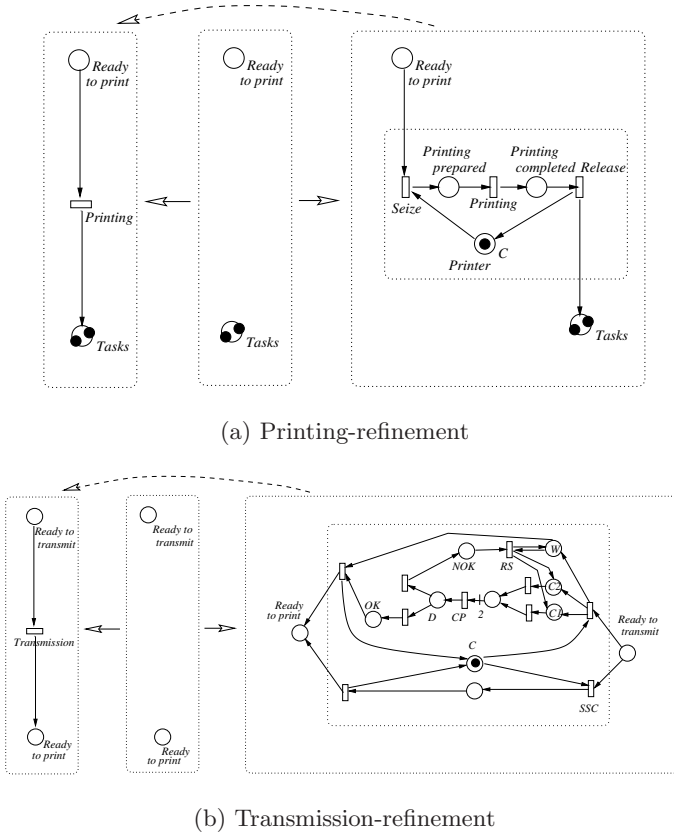


Fig. 7. Liveness preserving transition refinement

2.2 Expert View

Formal foundation and precise mathematical representation of net model transformations and corresponding results are provided in the Expert View of the $\gg$ Petri Net Baukasten $\ll$. This theory gives a formal categorical framework for stepwise development of systems based on nets, as demonstrated in section 2.1. In following we will present the formal framework of Petri net transformations and main results.

Formal Framework of Net Model Transformation. The idea of net transformation systems is one of the possible instantiations of the more general idea of high-level transformation systems (see [EHKP91]). In the next sections we summarize the main results. To present all the theoretical results in detail is beyond the scope of the paper. They can be found in the literature cited here.

The next definition introduces rules, transformations and net transformation systems formally for a given category **NET** of low or high level nets. More about the underlying theory can be found in [PER95] and in [Pad99]. We will assume to have a suitable category **NET** of nets and net morphisms, i. e. a category satisfying so called HLR-conditions stated in [EHKP91] and summarized in [PP02] in this volume.

Definition 1 (Rules, Transformations, Net Transformation Systems).

1. A rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ in a category **NET** consists of the nets L , K and R , called left-hand side, interface, and right-hand side, respectively, and two morphisms $K \xrightarrow{l} L$ and $K \xrightarrow{r} R$ with both morphisms $l, r \in \mathcal{M}$, a suitable class of injective morphisms in **NET**.
2. Given a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, a direct transformation $G \xRightarrow{p} H$ from a net G to a net H is given by two pushout diagrams (1) and (2) in the category **NET** as shown below.

$$\begin{array}{ccccc}
 L & \xleftarrow{l} & K & \xrightarrow{r} & R \\
 m \downarrow & (1) & \downarrow k & (2) & \downarrow n \\
 G & \xleftarrow{g} & C & \xrightarrow{h} & H
 \end{array}$$

The morphisms $L \xrightarrow{m} G$ and $R \xrightarrow{n} H$ are called occurrences of L in G and R in H , respectively. By an occurrence of rule $r = (L \xleftarrow{l} K \xrightarrow{r} R)$ in a net G we mean an occurrence of the left-hand side L in G .

In fact, the occurrence morphism m has to satisfy a specific condition, called gluing condition, in order to be able to apply the rule p to the net G .

3. Given a category of nets **NET** together with a suitable class of injective morphisms $\mathcal{M}$, a net transformation system $H = (\mathcal{S}, \mathcal{P})$ in $(\mathbf{NET}, \mathcal{M})$ is given by a start net $\mathcal{S} \in |\mathbf{NET}|$, and a set of rules $\mathcal{P}$.

Although the net transformation framework is a suitable concept for stepwise development of systems, very often there is a need to consider in addition more general morphisms for refinement or abstraction. The main idea is to enlarge the category of nets by $\mathcal{Q}$ -morphisms in the sense of [Pad96] in order to formulate refinement/abstraction morphisms.

More precisely, another category of nets **QNET** with a distinguished class of morphisms $\mathcal{Q}$, called $\mathcal{Q}$ -morphisms, is employed. The category **QNET** enriches the net transformation system defined in $(\mathbf{NET}, \mathcal{M})$ and yields the notion of $\mathcal{Q}$ -morphisms and $\mathcal{Q}$ -transformations. The class of $\mathcal{Q}$ -morphisms has to satisfy additional requirements called $\mathcal{Q}$ -conditions (see [Pad96]) to be adequate for refinement or abstraction. The formal definitions are given below.

Definition 2 (Class of $\mathcal{Q}$ -morphisms). Let **QNET** be a category, so that **NET** is a subcategory $\mathbf{NET} \subseteq \mathbf{QNET}$ and $\mathcal{Q}$ a class of morphisms in **QNET**.

1. The morphisms in $\mathcal{Q}$ are called $\mathcal{Q}$ -morphisms, or refinement/abstraction morphisms.
2. The $\mathcal{Q}$ -morphisms must satisfy so called $\mathcal{Q}$ -conditions, namely closedness of $\mathcal{Q}$, preservation of pushouts and inheritance of $\mathcal{Q}$ -morphisms under pushouts and coproducts.

Moreover, let $\mathcal{O} = (\mathcal{O}^q)_{q \in \mathcal{Q}}$ be an indexed class of adequate occurrence morphisms in **QNET** with respect to a refinement morphism $q \in \mathcal{Q}$. These occurrence morphisms restrict applicability of rules as defined in Theorem 1 below.

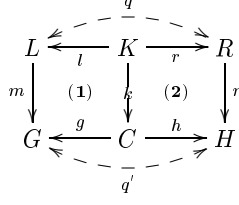
Definition 3 ($\mathcal{Q}$ -Rules and $\mathcal{Q}$ -Transformations).

1. A (preserving) $\mathcal{Q}$ -rule (p, q) is given by a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ in **NET** and a $\mathcal{Q}$ -morphism $q : L \rightarrow R$, so that $q \circ l = r$ in **QNET**.
2. A respecting $\mathcal{Q}$ -rule (p, q) is given by a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ in **NET** and a $\mathcal{Q}$ -morphism $q : R \rightarrow L$, so that $q \circ r = l$ in **QNET**.

The next fact states that $\mathcal{Q}$ -morphisms are preserved by transformations.

Theorem 1 (Induced Transformations and Pushouts in **QNET).** Let **QNET** be a supercategory of **NET** according to Definition 2.

1. Given a preserving $\mathcal{Q}$ -rule (p, q) and a transformation $G \xRightarrow{p} H$ in **NET** with an occurrence $m \in \mathcal{O}^q$ defined by the pushouts (1) and (2), there is a unique $q' \in \mathcal{Q}$, such that $q' \circ g = h$ and $q' \circ m = n \circ q$ in **QNET**. The transformation $(G \xRightarrow{p} H, q' : G \rightarrow H)$, or $G \xRightarrow{(p, q')} H$ for short, is called $\mathcal{Q}$ -transformation.



2. Given a respecting $\mathcal{Q}$ -rule (p, q) and a transformation $G \xRightarrow{p} H$ with an occurrence $n \in \mathcal{O}^q$ in **NET** defined by the pushouts (1) and (2), there is a unique $q' \in \mathcal{Q}$, such that $q' \circ h = g$ and $q' \circ n = m \circ q$ in **QNET**. The transformation $(G \xRightarrow{p} H, q' : H \rightarrow G)$, or $G \xRightarrow{(p, q')} H$ for short, is called $\mathcal{Q}$ -R-transformation.

The preserving and respecting transformations differ in the direction of $\mathcal{Q}$ -morphisms q, q' . The pushout squares (1) and (2) represent a transformation in the category **NET** (as in Definition 1). The $\mathcal{Q}$ -morphism q is a refinement/abstraction morphism in **QNET**. The morphism q' is the induced morphism (according to Theorem 1) which belongs to the class of $\mathcal{Q}$ -morphisms in **QNET** as well.

Compatibility with Fusion and Union. We now introduce two basic constructions for high-level structures, both generalizations of the notions union and fusion introduced by [Jen92] for coloured Petri nets. These notions have been reformulated and generalized in the frame of high-level replacement systems [PER95]. We show the compatibility of these notions with $\mathcal{Q}$ -transformations. Similarly to the previous section, we assume to have suitable categories **NET** and **QNET** with a class $\mathcal{Q}$.

Several results concerning horizontal structuring for net transformation systems have been adopted from HLR systems. The two basic structuring constructions for nets are union and fusion. Union allows a construction of larger nets from smaller ones with shared subpart, while fusion is a construction which allows to identify distinguished subnets.

Especially the concept of fusion is more general then the concept of fusion of places often cited in the Petri net literature. The fusion introduced in net transformation systems is not restricted to the fusion of places, but covers also fusion of subnets.

Definition 4 (Fusion and Union).

1. The fusion of two morphisms $f_1, f_2 : F \rightarrow G$ between nets F and G is a net G' together with a morphism $g : G \rightarrow G'$ defined by the coequalizer $(g : G \rightarrow G', G')$ of f_1 and f_2 . The fusion is denoted by $G \multimap G'$ via (F, f_1, f_2, g) , or $G \xrightarrow{F} G'$ for short.

2. The union of a pair of nets (G_1, G_2) via some interface I with the morphisms $i_1 : I \rightarrow G_1$ and $i_2 : I \rightarrow G_2$ is given by the pushout $G_1 \rightarrow G \leftarrow G_2$ of $G_1 \xleftarrow{i_1} I \xrightarrow{i_2} G_2$ and is denoted by $(G_1, G_2) \xRightarrow{I} G$ via (I, i_1, i_2) , or $(G_1, G_2) \xRightarrow{I} G$ for short.

In view of software development methodology it is important that horizontal structuring based on fusion and union is compatible with transformations and $\mathcal{Q}$ -transformations. This means that – under certain compatibility conditions – the result is the same whether we apply first union/fusion and then transformation or vice versa. For more details about horizontal structuring and transformations see e.g. [Pad96, Pad99].

Theorem 2. *Given a net transformation system $(\mathcal{S}, \mathcal{P})$ in $(\mathbf{NET}, \mathcal{M})$ such that HLR conditions and the compatibility conditions for fusion and union are satisfied.*

1. Given a fusion $G \circ^F G'$ and a compatible transformation $G \xRightarrow{P} H$, then there is a common net H' , obtained by fusion $H \circ^F H'$ and also by transformation $G' \xRightarrow{P} H'$. That means, we have

$$G \xRightarrow{P} H \circ^F H' = G \circ^F G' \xRightarrow{P} H.$$

2. Given a union $(G_1, G_2) \xRightarrow{I} G$ and a compatible transformations $G_i \xRightarrow{P_i} H_i$ then, there is a common net H obtained by the union $(H_1, H_2) \xRightarrow{I} H$ and by the transformation $G \xRightarrow{P_1 + P_2} H$ via the parallel rule $p_1 + p_2$, such that we have

$$(G_1, G_2) \xRightarrow{I} G \xRightarrow{P_1 + P_2} H = (G_1, G_2) \xRightarrow{(P_1, P_2)} (H_1, H_2) \xRightarrow{I} H$$

where $(G_1, G_2) \xRightarrow{(P_1, P_2)} (H_1, H_2)$ denotes the tupling of the separate transformations of $G_1 \xRightarrow{P_1} H_1$ and $G_2 \xRightarrow{P_2} H_2$.

3. If $\mathcal{Q}$ -conditions are satisfied then above results are valid also for $\mathcal{Q}$ -transformations and $\mathcal{Q}$ -R-transformations.

Main Results Concerning Preservation of Properties. As discussed in Section 2.1 the concept of $\mathcal{Q}$ -morphisms is important in order to study whether the transformation of nets is property preserving. The idea of property preserving transformations has been investigated in [GHP99, PG00a] and [PGE98] for safety properties and in [GPU01] for liveness. Safety-properties for Petri nets are stated as propositional logic formulas based on the actual marking of Petri nets. Morphisms preserving safety properties have been investigated for several types of Petri nets, see [GHP99] for P/T Petri nets, [PG00a] for colored Petri nets and [PGE98] for a class of algebraic-high level nets. This class of morphisms has been applied also in a case study of a medical information system in [Pad99] in order to prove relevant properties of the system.

Definition 5 (Formulas, Translations). Let $N = (P, T, \text{pre}, \text{post}, \hat{m})$ be a place/transition net, where P and T are sets of places and transitions respectively, pre and post are pre- and post-domain functions and $\hat{m}$ is initial marking.

1. A static formula λp is given for $\lambda \in \mathbb{N}$ and $p \in P$. The set of all static formulas over P is denoted by $\mathbb{F}$; static formulas are build up using the logical operators $\neg$ and $\wedge$:

$$\begin{aligned}\varphi_1 \in \mathbb{F} &\implies \neg \varphi_1 \in \mathbb{F}, \\ \varphi_1 \in \mathbb{F}, \varphi_2 \in \mathbb{F} &\implies \varphi_1 \wedge \varphi_2 \in \mathbb{F}\end{aligned}$$

The validity of formulas is given w. r. t. the marking of a net. Let $m \in P^\oplus$ be a marking of N then:

$$\begin{aligned}m \models_N \lambda p &\text{ iff } \lambda p \leq m \\ m \models_N \neg \varphi_1 &\text{ iff } \neg(m \models_N \varphi_1) \\ m \models_N \varphi_1 \wedge \varphi_2 &\text{ iff } (m \models_N \varphi_1) \wedge (m \models_N \varphi_2)\end{aligned}$$

2. Let φ be a static formula over N . Then $\Box\varphi$ is a safety property. The safety property $\Box\varphi$ holds in N under m iff φ holds in all markings m' reachable from m :

$$m \models_N \Box\varphi \iff \forall m' \in [m] : m' \models_N \varphi.$$

If m is the initial marking $\hat{m}$ we also write $N \models \Box\varphi$ instead of $\hat{m} \models_N \Box\varphi$.

3. The translation $\mathcal{T}_f$ of formulas over N_1 along a loose morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ (see below) to formulas over N_2 is given for atoms by

$$\mathcal{T}_f(\lambda p) = \lambda f_P(p).$$

The translation of formulas is given recursively by

$$\begin{aligned}\mathcal{T}_f(\neg\varphi) &= \neg\mathcal{T}_f(\varphi), \\ \mathcal{T}_f(\varphi_1 \wedge \varphi_2) &= \mathcal{T}_f(\varphi_1) \wedge \mathcal{T}_f(\varphi_2) \text{ and} \\ \mathcal{T}_f(\Box\varphi) &= \Box\mathcal{T}_f(\varphi).\end{aligned}$$

There have been two notions of safety properties preserving morphisms investigated, namely place preserving and transition gluing morphisms. The formal definition of these morphism is following.

Definition 6 (Place Preserving and Transition Gluing Morphisms). Given $N_i = (P_i, T_i, pre_i, post_i, \widehat{m}_i), i \in \{1, 2\}$ two place/transition nets. A morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ with functions $f_P : P_1 \rightarrow P_2$ and $f_T : T_1 \rightarrow T_2$ is called

loose if the following embedding conditions hold for all $t \in T_1$ and $p \in P_1$:

$$\begin{aligned}(a) \quad & f_P^\oplus(pre_1(t)) \leq pre_2(f_T(t)) \text{ and } f_P^\oplus(post_1(t)) \leq post_2(f_T(t)) \\ (b) \quad & f_P^\oplus(\widehat{m}_1|_p) \leq \widehat{m}_2|_{f_P(p)}\end{aligned}$$

place preserving if it is a loose morphism and the following place preserving conditions hold:

- (c) $\bullet(f_P(p)) = f_T^\oplus(\bullet p)$ and $(f_P(p))\bullet = f_T^\oplus(p\bullet)$ for all $p \in P_1$
 where $\bullet p = \sum_{t \in T} \text{post}(t)(p) \cdot t$ and $p\bullet = \sum_{t \in T} \text{pre}(t)(p) \cdot t$ define
 the pre and post sets of p
 and $\text{pre}(t), \text{post}(t) \in P_1^\oplus$ are considered as functions from P to $\mathbb{N}$.
- (d) f_T and f_P are injective
- (e) $\widehat{m}_2|_{f_P} = f_P^\oplus(\widehat{m}_1)$

transition gluing if it is a loose morphism and the following holds:

- (f) f_P is isomorphism
- (g) $f_P^\oplus(\widehat{m}_1) = \widehat{m}_2$
- (h) f_T is surjective s.t. $\text{pre}_2(t_2) = \sum_{t_1 \in f_T^{-1}(t_2)} \text{pre}_1(t_1)$
 with $t_1 \in T_1$ and $t_2 \in T_2$. *post* analogously.

Place preserving and transition gluing morphisms yield the safety property preserving transformations as in the next theorem, for which a proof can be found in [GHP99].

Theorem 3 (Safety Property Preserving Transformations). *Given a preserving $\mathcal{Q}$ -rule $(p = (L \leftarrow K \rightarrow R), q : L \rightarrow R)$ with q being*

- either a place preserving morphism
- or a transition gluing morphism

then we have for each $\mathcal{Q}$ -transformation step $N_1 \xrightarrow{(p,q)} N_2$ (under appropriate occurrence morphism from $\mathcal{O}^q$) holds:

$$N_1 \models \Box\varphi \implies N_2 \models \mathcal{T}_q(\Box\varphi).$$

In our small case study in Section 2.1 in the Figures 2 and 4, the rules and transformations are (according to Theorem 3) safety property preserving, actually the rules in Figure 2 are based on transition gluing morphisms, the rule in Figure 4 is based on a place preserving morphism.

Liveness preserving refinement is based on the standard notion of liveness as used in Petri net theory. We do not consider liveness in the sense of temporal logic formulas as it is often used in literature. Liveness in our approach means that no deadlock and even livelock can occur. The formal introduction of liveness follows.

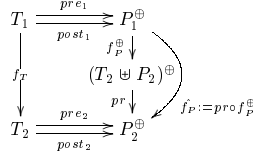
Definition 7 (Liveness). *A place/transition net $N = (P, T, \text{pre}, \text{post}, \widehat{m})$ is called live if for arbitrary $m_1 \in \langle \widehat{m} \rangle$ and arbitrary $t \in T$ there exists some $m'_1 \in \langle m_1 \rangle$ such that t is enabled under m'_1 .*

In [GPU01] it is shown that a special type of transition refinement preserves liveness in Petri nets. The idea is based on abstracting morphisms, which are closely related to vicinity respecting morphisms (introduced in [DM90]). Abstracting morphisms allow abstracting transitions and places to a single transition. A certain subclass of abstracting morphisms, called collapsing morphisms, allows the description of transition refinement as collapsing of a special subnet (called live in-out cycle) to one transition. The formal introduction of abstracting and collapsing morphisms follows.

Definition 8 (Abstracting Morphism).

Given two place/transition nets $N_i = (P_i, T_i, pre_i, post_i, \widehat{m}_i), i \in \{1, 2\}$. An abstracting morphism $f : N_1 \rightarrow N_2$ is given by $f = (f_T, f_P)$ with functions $f_T : T_1 \rightarrow T_2$ and $f_P : P_1 \rightarrow (T_2 \uplus P_2)$ such that the following conditions are satisfied:

1. for all $t \in T_1$:
 $f_P(pre_1(t)) = \{f_T(t)\}$ **or** $pr \circ f_P^\oplus \circ pre_1(t) = pre_2(f_T(t))$,
 where $pr : (T_2 \uplus P_2)^\oplus \rightarrow P_2^\oplus$ is the corresponding projection
analogously for the post function
2. for all $t_2 \in f_T(T_1)$:
 exists $t_{in} \in T_1$ with $f_T(t_{in}) = t_2$ and $pr \circ f_P^\oplus \circ pre_1(t_{in}) = pre_2(t_2)$
analogously for the post function
3. strict marking:
 for all $p \in P_1$ with $f_P(p) \in P_2$: $f_P^\oplus(\widehat{m}_1|_p) = \widehat{m}_2|_{f_P(p)}$
4. for all $p \in P_1$ with $f_P(p) \in T_2$: $f_T(\bullet p) = \{f_P(p)\}$
analogously for the post function



Example 1. Examples of abstracting morphisms are shown in Figures 7 a,b. The dashed arrow going from the right-hand side of each rule to the left-hand side represents an abstracting morphism between nets. The subnet in a dotted rectangle on the right-hand side of each rule is abstracted by the abstracting morphism to a single transition *Printing*, resp. *Transmission* on the left-hand side.

In order to capture what has been abstracted from a net N_1 by an abstracting morphism, we define the collapsing subnet.

Definition 9 (Collapsing Subnet). Given an abstracting morphism $f : N_1 \rightarrow N_2$. We have for all transitions $t \in T_2$ the collapsing subnet

$subst_f(t) = (\tilde{P}^t, \tilde{T}^t, \tilde{pre}^t, \tilde{post}^t, \widehat{m}_t) \subseteq N_1$ with

- $\tilde{P}^t = \{p_1 \in P_1 \mid f_P(p_1) = t\}$
these are all places mapped to t
- $\tilde{T}^t = \{t_1 \in T_1 \mid f_T(t_1) = t\}$
these are all transitions mapped to t
- $\tilde{pre}^t(t_1) = pre_1(t_1)|_{\tilde{P}^t}$ for all $t_1 \in \tilde{T}^t$
the pre- and post-domain restricted to collapsing places
- $\widehat{m}_t = \widehat{m}_1|_{\tilde{P}^t}$
 $\tilde{post}^t$ is defined analogously.

Example 2. In Example 1, the subnets in dashed rectangles on the right-hand sides of the rules are exactly the collapsing subnets $\text{subst}_f(\text{Printing})$, and $\text{subst}_f(\text{Transmission})$ respectively, with respect to the corresponding abstracting morphism f .

Live In-Out Cycles describe those subnets that are live, and are equipped with a guarding place. This guarding place ensures that each run within the subnet has to be completed before it may run again.

Definition 10 (Live In-Out Cycle).

Given a place/transition net $N = (P, T, \text{pre}, \text{post}, \hat{m})$, then N is called live in-out cycle if the following conditions hold:

1. N is live
2. There are two distinguished subsets T_{in} and T_{out} of T , called in-transitions T_{in} and out-transitions T_{out} of N , such that the following holds.
There is a safe (1-bounded) place $c \in P$ with $\hat{m}|_c = c$, called guarding place, which is in the predomain of all in-transitions $t_i \in T_{in}$, and in the post-domain of all out-transitions $t_o \in T_{out}$
with $\text{pre}(t)|_c = \begin{cases} c ; t \in T_{in} \\ \epsilon ; t \notin T_{in} \end{cases}$ and $\text{post}(t)|_c = \begin{cases} c ; t \in T_{out} \\ \epsilon ; t \notin T_{out} \end{cases}$

Remark 1. The role of the distinguished place c is twofold: on one hand it establishes the cycle, because of condition 2. In this sense it is crucial for liveness in condition 1. On the other hand it implements a mutual exclusion of the in-transitions including the prevention of parallel firing of one transition with itself.

Example 3. The subnets $\text{subst}_f(\text{Printing})$ and $\text{subst}_f(\text{Transmission})$ in Example 2 are live in-out cycles. The guarding place is named C in both subnets.

Definition 11 (Collapsing Morphisms). A collapsing morphism $f : N_1 \rightarrow N_2$ is an abstracting morphism which additionally satisfies the following conditions. Let

- $\mathbb{S} \subseteq T_2$ with $\mathbb{S} = \{t | \tilde{P}^t \neq \emptyset\}$
- $\mathbb{T} \subseteq T_1$ with $\mathbb{T} = \bigcup_{t \in \mathbb{S}} \tilde{T}^t$
- $\mathbb{P} \subseteq P_1$ with $\mathbb{P} = \bigcup_{t \in \mathbb{S}} \tilde{P}^t$

then we have

1. f_T is surjective, and f_P is surjective on $P_1 \setminus \mathbb{P}$
2. f_T and f_P are injective on $T_1 \setminus \mathbb{T}$, resp. $P_1 \setminus \mathbb{P}$
3. for all $t \in \mathbb{S}$ we have:

The collapsing subnet $\text{subst}_f(t)$ is a live in-out cycle which is only connected to the rest of N_1 via the in- and out-transitions.

Example 4. The abstracting morphisms in Example 1 are also collapsing morphisms due to the fact that the collapsing subnets are live in-out cycles.

Liveness is respected via collapsing morphisms as shown in [GPU01]. This fact yields liveness preserving transformations as stated below.

Theorem 4 (Liveness Preserving Transformations). *Given a respecting $\mathcal{Q}$ -rule $(p = (L \leftarrow K \rightarrow R), q : R \rightarrow L)$ with a collapsing morphism q , we have for each $\mathcal{Q}$ -R-transformation step $N_1 \xrightarrow{(p,q)} N_2$ (with appropriate occurrence morphism from $\mathcal{O}^q$):*

$$N_1 \text{ live} \implies N_2 \text{ live.}$$

The rules in our case study in Figure 7 of the Section 2.1 are based on collapsing morphisms. Therefore according to the foregoing theorem we obtain liveness preserving $\mathcal{Q}$ -R-transformations. Hence liveness of the simple model in Figure 6 implies liveness of the final model in Figure 5.

3 Net Class Transformations

Numerous kinds of Petri net classes have been introduced in the literature, which allow to apply different Petri net formalisms on different levels of abstraction. In the application developer view we show how net class transformations between different Petri net formalisms can be applied in the development process. In the expert view a formal foundation of net class transformations is given based on the notion of functors in the sense of category theory.

3.1 Application Developer View

Net class transformations can be used by an application developer to begin the modelling in an abstract class with simple nets and to switch to a more expressive class in later development steps.

Moreover, it is important for the development process, that net class transformations are compatible with net model transformations and with the horizontal structuring techniques union and fusion.

In the following we present a development process to derive an algebraic high-level net AHL_4 (Figure 10), via the place/transition net PT_3 (Figure 5 without marking) from the simple elementary net EN_1 (Figure 8) using net model transformations from Section 2.1 and net class transformations $Weight: \mathbf{EN} \rightarrow \mathbf{PT}$ and $Data: \mathbf{PT} \rightarrow \mathbf{AHL}$ as shown in Figure 11. In this section we only consider nets without initial marking, but sketch an extension including markings at the end of Sect. 3.2.

We can start the modelling of our running example with the elementary net EN_1 like in Figure 8. Then we can use an elementary net version $printref_{EN}$ of the printing-refinement used in Sect. 2 to elaborate the modelling of the printer. This results in the elementary net EN_2 in Figure 9.

To use the transmission-refinement $transref_{PT}$ from Sect. 2 as well, we need to transform the model to the class of place/transition nets, because the refinement adds an arc with weight 2. The net class transformation $Weight$ assigns

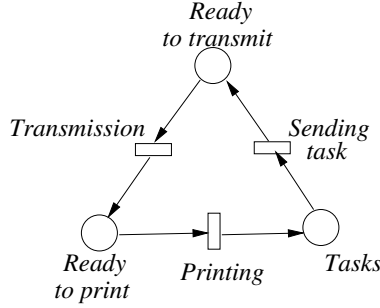


Fig. 8. Simple elementary net EN_1

the weight 1 to each arc leading to place/transition nets $PT_1 = Weight(EN_1)$ and $PT_2 = Weight(EN_2)$ and a net model transformation $printref_{PT}$ between them. After the application of the transmission-refinement $transref_{PT}$ we get the same model PT_3 as in Figure 5 of Sect. 2 without marking.

In order to be able to distinguish between different tasks we first transform our model to the class of algebraic high level nets. We obtain algebraic high level nets $AHL_1 = Data(PT_1)$, $AHL_2 = Data(PT_2)$ and $AHL_3 = Data(PT_3)$, where the net class transformation $Data: \mathbf{PT} \rightarrow \mathbf{AHL}$ adds a trivial data type specification and algebra (see Def. 14 below). The model transformations $printref_{PT}$ and $transref_{PT}$ can be transformed as well yielding transformations $printref_{AHL}$ and $transref_{AHL}$ in Figure 11.

Then we can further refine the net AHL_3 by an AHL net model transformation $specref_{AHL}$, which changes the arc inscriptions of AHL_3 to the ones of

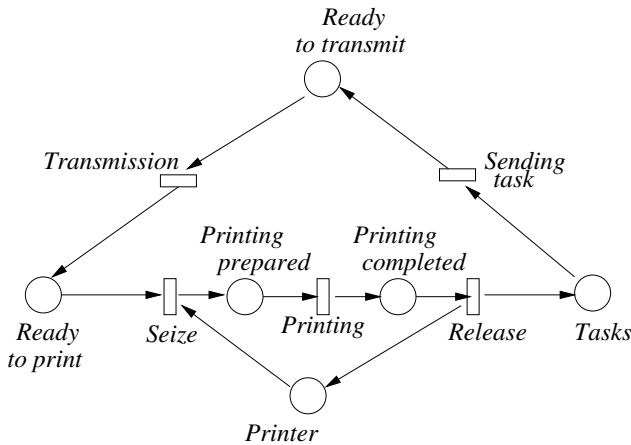


Fig. 9. Refinement of the printer EN_2

AHL_4 given in Figure 10 and adds the following specification with a suitable algebra:

BOOL+
sorts : *Printer, Task, Comm, Result*
opns : *printer*: $\rightarrow$ *Printer*
 com: $\rightarrow$ *Comm*
 *task*₁: $\rightarrow$ *Task*
 *task*₂: $\rightarrow$ *Task*
 equ: *Task Task* $\rightarrow$ *Bool*
 ok: *Task* $\rightarrow$ *Result*
 nok: $\rightarrow$ *Result*
 get: *Result* $\rightarrow$ *Task*
vars : *t*: *Task*, *r*: *Result*
eqns : *get(ok(t))* = *t*
 equ(t, t) = *true*

In the net AHL_4 we are able to model the comparison of the two tasks sent through the insecure channel by inscribing the transition CP with the equation

$$r = \text{if } equ(t', t'') \text{ then } ok(t') \text{ else } nok.$$

For simplicity we assume that our non-secure channel either transmits the duplicated task in a correct way (*ok*-case, $t' = t'' = t$) or t' and t'' are modifications of t with $t' \neq t''$ (*nok*-case). In the second case the transmission is repeated with the old value t of the task.

The whole development process is depicted in Figure 11, where horizontal steps are net model transformations and vertical steps are net class transformations.

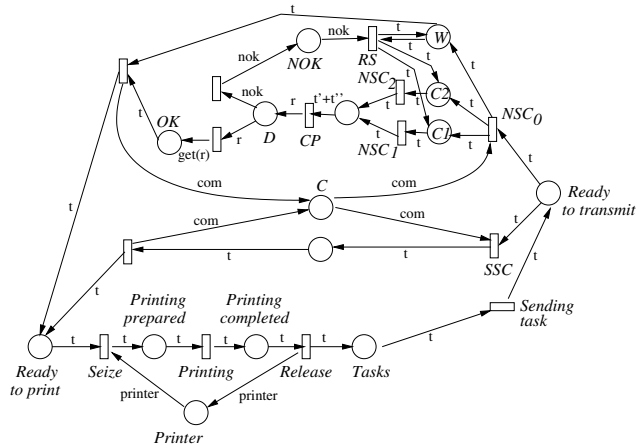


Fig. 10. Final algebraic high level net AHL_4

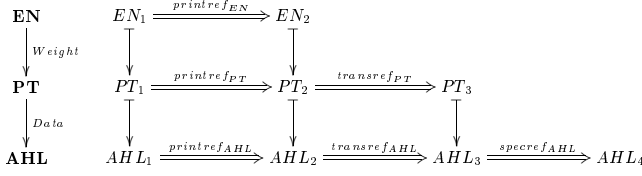


Fig. 11. Development of the printing system

3.2 Expert View

To describe net class transformations as functors we first have to define suitable net morphisms for each net class. In [GPP00, GPP01a] the categories of place/transition nets, elementary nets and algebraic high level nets are introduced in detail, where a net morphism $f = (f_P, f_T)$ is called transition preserving, if f_P and f_T are compatible with the pre- and post-domains of the transitions.

Definition 12 (Categories of Petri Nets).

- The class of all place/transition nets

$$PT = (P, T, pre, post) \text{ with } pre, post: T \rightarrow P^\oplus,$$

where $P^\oplus$ is the free commutative monoid over P , together with the transition preserving morphisms

$$f = (f_P, f_T): PT \rightarrow PT'$$

constitutes the category of place/transition nets **PT**.

- The class of all elementary nets

$$EN = (P, T, pre, post) \text{ with } pre, post: T \rightarrow \mathcal{P}(P),$$

where $\mathcal{P}(P)$ is the powerset of P , together with the transition preserving morphisms

$$f = (f_P, f_T): EN \rightarrow EN'$$

constitutes the category of elementary nets **EN**.

- The class of all marked place/transition nets

$$mPT = (P, T, pre, post, m_0) \text{ with } pre, post: T \rightarrow P^\oplus \\ \text{and initial marking } m_0 \in P^\oplus$$

together with the transition preserving morphisms

$$f = (f_P, f_T): mPT \rightarrow mPT' \text{ with } f_P^\oplus(m_0) \leq m'_0$$

constitutes the category of marked place/transition nets **mPT**.

- The class of all algebraic high level nets

$$\mathbf{AHL} = (\mathbf{SPEC}, X, A, P, T, \text{pre}, \text{post}, \text{cond}) ,$$

where $\mathbf{SPEC} = (\Sigma, E)$ is an algebraic specification, A a $\mathbf{SPEC}$ -algebra and

$$\text{pre}, \text{post}: T \rightarrow (T_\Sigma(X) \times P)^\oplus \text{ and } \text{cond}: T \rightarrow \mathcal{P}(\text{Eqns}(\Sigma))$$

are functions, together with transition and equation preserving morphisms

$$f = (f_{\mathbf{SPEC}}, f_X, f_A, f_P, f_T): \mathbf{AHL} \rightarrow \mathbf{AHL}'$$

constitutes the category of algebraic high level nets **AHL**.

Definition 13 (Net Class Transformations). Given categories **NET** and **NET'** of nets a net class transformation is a functor $T: \mathbf{NET} \rightarrow \mathbf{NET}'$.

We consider the following functors between the categories defined in Def. 12, where for simplicity we omit the transformation of net morphisms. More details are given in [GPP00] and in [PP02] in this volume.

Definition 14 (Functors between Petri Net Categories).

- Between **PT** and **EN** there is a functor

$$\text{Causality}: \mathbf{PT} \rightarrow \mathbf{EN}$$

forgetting all weights in the pre- and post-domains of transitions. Vice versa there is a functor

$$\text{Weight}: \mathbf{EN} \rightarrow \mathbf{PT}$$

assigning the weight 1 to all arcs.

- Between **PT** and **mPT** there is a functor

$$\text{Mark}: \mathbf{PT} \rightarrow \mathbf{mPT}$$

adding an empty initial marking to the net. Vice versa there is a functor

$$\text{Unmark}: \mathbf{mPT} \rightarrow \mathbf{PT}$$

forgetting the initial marking.

- Between **PT** and **AHL** there is a functor

$$\text{Data}: \mathbf{PT} \rightarrow \mathbf{AHL}$$

adding an empty specification, an empty algebra and a set of variables where each arc with weight n is inscribed with the sum of n distinct variables. Vice versa there is a functor

$$\text{Skeleton}: \mathbf{AHL} \rightarrow \mathbf{PT}$$

forgetting specification, variables and algebra and weighting each arc with the number of terms it is inscribed with.

Compatibility Results for Net Class Transformations. As pointed out in Sect. 3.1 it is important, that net class transformations are compatible with the horizontal structuring techniques union and fusion. Because union and fusion are finite colimits from the categorical point of view, this compatibility is achieved by a net class functor preserving colimits.

Definition 15 (Compatibility with Union and Fusion). *A net class transformation $T: \mathbf{NET} \rightarrow \mathbf{NET}'$ is compatible with union and fusion, if it transforms unions $(N_1, N_2) \succcurlyeq^I N$ in $\mathbf{NET}$ into unions $(T(N_1), T(N_2)) \succcurlyeq^{T(I)} T(N)$ in $\mathbf{NET}'$ and fusions $N \circ \xrightarrow{F} N'$ in $\mathbf{NET}$ into fusions $T(N) \circ \xrightarrow{T(F)} T(N')$ in $\mathbf{NET}'$, if T preserves colimits.*

Another desirable property is the compatibility of net class transformations with model transformations defined in the previous section. Transformations in one Petri net class shall be translated to transformations in another class by a net class transformation between these classes. This is achieved by net class functors being compatible with the classes $\mathcal{M}$ of morphisms used to define model transformations.

Definition 16 (Compatibility with Net Model Transformations). *A net class transformation $T: \mathbf{NET} \rightarrow \mathbf{NET}'$ is compatible with net model transformations in $(\mathbf{NET}, \mathcal{M})$ and $(\mathbf{NET}', \mathcal{M}')$, if it transforms net model transformations $N \xrightarrow{p} N'$ in $(\mathbf{NET}, \mathcal{M})$ to net model transformations $T(N) \xrightarrow{T(p)} T(N')$ in $(\mathbf{NET}', \mathcal{M}')$ with $T(p): T(L) \xleftarrow{T(l)} T(K) \xrightarrow{T(r)} T(R)$ for $p: L \xleftarrow{l} K \xrightarrow{r} R$.*

A proof of the following theorem is given in [GPP00] using the fact that all the mentioned functors preserve finite colimits.

Theorem 5. *The net class transformations given in Definition 14 are compatible with union, fusion and net model transformations in $(\mathbf{NET}, \mathcal{M})$ and $(\mathbf{NET}', \mathcal{M}')$ for suitable classes $\mathcal{M}$ and $\mathcal{M}'$ of injective net morphisms.*

Lifting of Net Class Transformations. In [Pad96] a theory of abstract Petri nets is introduced where different classes of Petri nets are described uniformly by net structure functors $Net: \mathbf{Set} \rightarrow \mathbf{Set}$ leading to classes $\mathbf{PNC}$ of nets with $pre, post: T \rightarrow Net(P)$. For example, the class of place/transition nets $\mathbf{PT}$ is obtained by the construction of the free commutative monoid $\_^\oplus: \mathbf{Set} \rightarrow \mathbf{Set}$ and the class of elementary nets by the powerset functor $\mathcal{P}: \mathbf{Set} \rightarrow \mathbf{Set}$.

It can be shown that a natural transformation $v: Net \Rightarrow Net'$ between two net structure functors yields a net class functor $F_v: \mathbf{PNC} \rightarrow \mathbf{PNC}'$. With this fact the functors *Causality* and *Weight* can be reconstructed by giving natural transformations $c: \_^\oplus \Rightarrow \mathcal{P}$ and $w: \mathcal{P} \Rightarrow \_^\oplus$ between the free commutative monoid functor $\_^\oplus$ and the powerset functor $\mathcal{P}$.

To integrate algebraic high level nets into this approach pre- and post-domains are replaced by $pre, post: T \rightarrow Net(T_\Sigma(X) \times P)$. The usual algebraic

high level nets **AHL** are obtained by using the net structure functor $_{-}^{\oplus}: \mathbf{Set} \rightarrow \mathbf{Set}$. By using $\mathcal{P}: \mathbf{Set} \rightarrow \mathbf{Set}$ instead we get the new class **EAHL** of elementary high level nets.

The concept of marked nets can be abstracted by taking into account initial markings $m_0 \in \text{Net}(P)$ resp. $m_0 \in \text{Net}(T_{\Sigma}(X) \times P)$. This leads to the usual marked place/transition nets **mPT** with $m_0 \in P^{\oplus}$ as well as new classes of marked elementary nets **mEN** with $m_0 \in \mathcal{P}(P)$, marked algebraic high level nets **mAHL** with $m_0 \in (T_{\Sigma}(X) \times P)^{\oplus}$ and marked elementary high level nets **mEAHL** with $m_0 \in \mathcal{P}(T_{\Sigma}(X) \times P)$.

By combining these different abstract notions we obtain the cube of Petri net classes and net class transformations given in Fig. 12. More details about the Petri net cube can be found in [GPP00] and in [PP02] in this volume.

4 Overview of Other Transformations and Conclusion

In this final section we give an overview of other transformations in the $\gg$ Petri Net Baukasten $\ll$ and discuss open problems.

4.1 Other Transformations in the $\gg$ Petri Net Baukasten $\ll$

In addition to net class transformations, there are several transformations between net classes and other modeling techniques which are important for the Application Developer View. One important aspect is to bridge the gap between semiformal techniques in engineering and formal techniques in computer science, which is one of the main aims of the “DFG-Schwerpunktprogramm: Integration von Techniken der Softwarespezifikation für Ingenieurwissenschaftliche Anwendungen”. A typical example is the transformation of the MFERT modeling technique for production automation into timed hierarchical Predicate/Transition nets. Another important aspect is to bridge the gap between semiformal and formal techniques in computer science, where visual modeling techniques including message sequence charts (MSC) and different kinds of UML diagrams are considered to be semiformal because no widely accepted formal semantics is available.

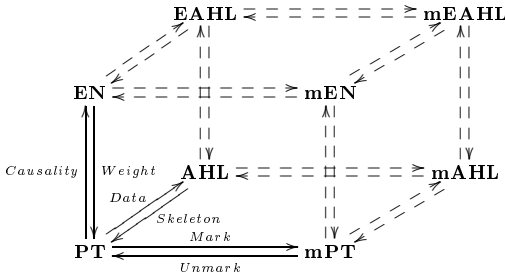


Fig. 12. Cube of Petri net categories

On the other hand, these visual techniques are very important especially in the phase of requirement analysis. A typical example is the transformation from MSC-scenarios to Petri nets, which has been used in the DFG-SPP reference case study “Railway Traffic Crossing” in order to transform different railway crossing scenarios presented by MSC-diagrams into Petri nets (see [KPE00]).

Finally let us discuss some other kinds of transformations which are important in the Tool Developer View of the $\gg$ Petri Net Baukasten $\ll$. One of the main problems for the integration of different tools has been the different net representation formats for each of these tools. The “Petri Net Markup Language” developed by the “DFG-Forschergruppe PETRINETZ-TECHNOLOGIE” provides XML-schemes resp. DTD-standards for different net representation formats. For n different individual formats we only have to provide $2n$ explicit transformations from each individual format to the DTD-standard and vice versa in order to provide $n(n-1)$ transformations between each pair of different individual formats:

$$IndividualFormat_i \leftrightarrow DTD-Standard \leftrightarrow IndividualFormat_j$$

For similar reasons a graphical exchange language GXL is going to be developed in the graph transformation community (see [Tae01]). In order to allow integration of Petri and graph transformation tools it is important to have transformations between the corresponding representation formats PN-DTD for Petri nets and GRA-DTD for graphs. A transformation from PN-DTD to GRA-DTD has been implemented already in [Ehr01], which allows to use graph transformation tools for Petri nets. A typical application is the use of the GENGED-approach & environment [Bar99] for Visual modeling languages, behavior & animation in order to allow the animation of Petri nets using domain specific visual modeling (see [BEE00]) and the contribution in this volume for more details.

4.2 Conclusion and Open Problems

In this paper we have given an overview of different kinds of Petri net transformations and their relevance for the $\gg$ Petri Net Baukasten $\ll$. Especially we have discussed the use of net model transformations for stepwise development of communication based systems and that of net class transformations for Petri net based software development models. These aspects are important for the Application Developer View of the $\gg$ Petri Net Baukasten $\ll$. The formal framework for net model transformations are double pushout transformations in a suitable category **NET** of Petri nets and that for net class transformations are functors between different categories of Petri nets. These concepts and the corresponding theory are important contributions for the Expert View of the $\gg$ Petri Net Baukasten $\ll$. Finally we have discussed transformations between individual formats of Petri net representations for tools and suitable DTD-standards which are important in the Tool Developer View.

It remains open to integrate the concepts and instances of all these different kinds of transformations into the 2nd Installment of the $\gg$ Petri Net Baukasten $\ll$ concerning user interfaces, services and data bases. Moreover, in each

of the areas discussed above there are still several interesting open problems. One typical problem is to study liveness preserving transformations also for algebraic high level nets in section 3.1 similar to those for place/transition nets in section 2.1.

References

- [AHS90] J. Adamek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. Series in Pure and Applied Mathematics. John Wiley and Sons, 1990. 38
- [Bar99] R. Bardohl. *GENGED – Visual Definition of Visual Languages Based on Algebraic Graph Transformation*. PhD Thesis, TU Berlin, Verlag Dr. Kovac, Germany, 1999. 62
- [BEE00] R. Bardohl, H. Ehrig, and C. Ermel. Generic Description, Behaviour and Animation of Visual Modeling Languages. In *Proc. Integrated Design and Process Technology*. Dallas, USA, 2000. 62
- [DFG99] DFG-Forschergruppe PETRI NET TECHNOLOGY. *Initial Realization of the $\gg$ Petri Net Baukasten $\ll$* . Informatik-Berichte 129, Humboldt-Universität zu Berlin, October 1999. 37
- [DM90] J. Desel and A. Merceron. Vicinity Respecting Net Morphisms. In *Advances in Petri Nets*, pages 165–185. Lecture Notes in Computer Science 483. Springer Verlag, 1990. 52
- [EGP99] H. Ehrig, M. Gajewski, and F. Parisi-Presicce. High-Level Replacement Systems with Applications to Algebraic Specifications and Petri Nets. In *Handbook of Graph Grammars and Computing by Graph Transformation*, Volume 3: Concurrency, Parallelism, and Distribution, pages 341–400. World Scientific, 1999. 38, 39
- [EHKP91] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. From Graph Grammars to high level replacement systems. In *4th Int. Workshop on Graph Grammars and their Application to Computer Science*, pages 269–291. Lecture Notes in Computer Science 532. Springer Verlag, 1991. 38, 47
- [Ehr01] K. Ehrig. Converting XML Files with XSLT and XPATH. <http://tfs.cs.tu-berlin.de/lehre/SS01/gragra.html>, 2001. Student’s Project Status Report. 62
- [Gaj00] M. Gajewsky. Concepts and Requirements for Transformations within Petri Net Based Process Models. In *5th World Conference on Integrated Design and Process Technology*, Special Session on Model Integration. CD-ROM, 8 pages, 2000.
- [GE01] M. Gajewsky, and H. Ehrig. The $\gg$ Petri Net Baukasten $\ll$. In *Unifying Petri Nets*, Advances in Petri Nets, Lecture Notes in Computer Science 2128. Springer Verlag, 2001. 37
- [GHP99] M. Gajewsky, K. Hoffmann, and J. Padberg. *Place Preserving and Transition Gluing Morphisms in Rule-Based Refinement of Place/Transition Systems*. Technical Report 1999-14, Technical University Berlin, 1999. 41, 45, 50, 52
- [GPP00] M. Gajewsky, and F. Parisi-Presicce. *Formal Transformations of Petri Nets*. Technical Report 2000-12, Technical University Berlin, 2000. 39, 58, 59, 60, 61

- [GPP01a] M. Gajewsky, and F. Parisi-Presicce. Transformations between Petri Net Classes with Application to Software Development. In *2nd Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems*. Berlin, 2001. 58
- [GPP01b] M. Gajewsky, and F. Parisi-Presicce. On Compatibility of Model and Class Transformations. In *Recent Trends in Algebraic Development Techniques*, 15th International Workshop WADT 2001, pages 24–25, Lecture Notes in Computer Science 2267. Springer Verlag, 2001. 38, 39
- [GPU01] M. Gajewsky, J. Padberg, and M. Urbásek. *Rule-Based Refinement for Place/Transition Systems: Preserving Liveness-Properties*. Technical Report 2001-8, Technical University of Berlin, 2001. 39, 41, 50, 52, 55
- [Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, Volume 1: Basic Concepts. EATCS Monographs in Theoretical Computer Science 28. Springer Verlag, 1992. 49
- [KPE00] O. Kluge, J. Padberg, and H. Ehrig. Modeling Train Control Systems: From Message Sequence Charts to Petri Nets. In *Proc. Formale Techniken für die Eisenbahnsicherung (FORMS)*, pages 25–42. Fortschritt-Berichte VDI, 2000. 62
- [Lil95] J. Lilius. *On the Structure of High-Level Nets*. PhD thesis, Helsinki University of Technology, 1995. Digital Systems Laboratory, Research Report 33. 39
- [Pad96] J. Padberg. *Abstract Petri Nets: A Uniform Approach and Rule-Based Refinement*. PhD thesis, Technical University Berlin, 1996. Shaker Verlag. 39, 41, 48, 50, 60
- [Pad99] J. Padberg. Categorical Approach to Horizontal Structuring and Refinement of High-Level Replacement Systems. In *Applied Categorical Structures* 7(4), pages 371–403, 1999. 39, 47, 50
- [PER95] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic High-Level Net Transformation Systems. In *Math. Struct. in Comp. Science*, Volume 5, pages 217–256, 1995. 39, 47, 49
- [Peu01] S. Peuker. *Halbordnungsbasierte Verfeinerung zur Verifikation verteilter Algorithmen*. PhD thesis, Humboldt University Berlin, 2001. 41
- [PG98] J. Padberg, and M. Gajewsky. Using High-Level Replacement Systems to Preserve Safety Properties in Place/Transition Net Transformations. In *Sixth Int. Workshop on Theory and Application of Graph Transformation*, pages 356–365. Universität-Gesamthochschule Paderborn, Fachbereich Mathematik-Informatik, 1998. 39
- [PG00a] J. Padberg, and M. Gajewsky. *Safety Preserving Transformations of Coloured Petri Nets*. Technical Report 2000-13, Technical University Berlin, 2000. 50
- [PG00b] J. Padberg, and M. Gajewsky. Rule-Based Refinement of Petri Nets For Modeling Train Control Systems. In *Petri Nets in Design, Modelling and Simulation of Control Systems*, Special Session at the IFAC Conference on Control Systems Design, pages 299–304, 2000. 41
- [PGE98] J. Padberg, M. Gajewsky, and C. Ermel. Rule-Based Refinement of High-Level Nets Preserving Safety Properties. In *Fundamental Approaches to Software Engineering*, pages 221–238. Lecture Notes in Computer Science 1382. Springer Verlag, 1998. 39, 41, 50
- [PGH99] J. Padberg, M. Gajewsky, and K. Hoffmann. Incremental Development of Safety Properties in Petri Net Transformations. In *Theory and Applica-*

- tion of Graph Transformation*, pages 410–425. Lecture Notes in Computer Science 1764. Springer Verlag, 1999. 39
- [PHG00] J. Padberg, K. Hoffmann, and M. Gajewsky. Stepwise Introduction and Preservation of Safety Properties in Algebraic High-Level Net Systems. In *Fundamental Approaches to Software Engineering*, pages 249–265. Lecture Notes in Computer Science 1783. Springer Verlag, 2000. 39
- [PP02] F. Parisi-Presicce. A Formal Framework for Petri Net Class Transformations. In *Petri Net Technology for Communication Based Systems*, Lecture Notes in Computer Science. Springer Verlag, 2002. 40, 47, 59, 61
- [Tae01] G. Taentzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. In *Int. Workshop on Uniform Approaches to Graphical Process Specification Techniques (UNIGRA'01)*, Sattelite Event of ETAPS'01, 2001. 62
- [WER99] H. Weber, H. Ehrig, and W. Reisig, editors. *Int. Colloquium on Petri Net Technology for Modelling Communication Based Systems*, Part II: The $\gg$ Petri Net Baukasten $\ll$. Fraunhofer Gesellschaft ISST, October 1999. 37

Four Steps Towards Sound Business Process Models^{*}

Juliane Dehnert

Technical University Berlin
dehnert@cs.tu-berlin.de

Abstract. Supporting the everyday operation of business processes with the help of workflow management systems is a necessary prerequisite for many companies in order to stay competitive. Managing business processes comprises their modeling, analysis and execution support. Although investigated since the beginning of the nineties there is still a gap between the possibilities of the provided modeling languages, the expertise of the modeler and the requirements posed through the different steps within the management of business processes.

In this paper we propose an iterative procedure model for the management of business processes which starts on an informal and more abstract level and leads the modeler to a formal description containing details of the implementation. We start the procedure using event driven process chains an informal but in practice widely accepted modeling language. Throughout the procedure the description is transformed in a formal specification based on Petri nets. The Petri net model is then used for analysis purposes and finally refined to serve as base for the coordination of the process at run time. The innovation of the proposed procedure is based on a pragmatic correctness interpretation of business process descriptions.

Keywords: Business Process Management, Event driven Process Chains, Petri nets

1 Introduction

The Petri Net research group aims at the development of a support environment for the use of Petri Nets in new applications including theoretical and tool aspects [6]. The work presented in this paper is part of the work of the project team "Net based Software development" (NSE). Within this group a procedure model is striven for supporting the application developer in developing information systems. As application domain we focus on the management of workflow processes.

^{*} This work is part of the joint research project "DFG-Forschergruppe PETRI NET TECHNOLOGY" between H. Weber (Coordinator), H.Ehrig and W. Reisig supported by the Deutsche Forschungsgemeinschaft

Throughout the procedure the systematical use of Petri nets is aspired whereby other formal or semi-formal description techniques should be integrated if this advances acceptance and usability of the proposed approach.

As part of this objective we will in the following suggest a procedure model for the management of business processes. In doing so we will focus on the control flow aspects, thus the concerned tasks and their order relation (including no order).

The management of business processes comprises modeling, analysis and execution support of business processes. The challenge of our objective is hidden in the different requirements posed to the modeling technique throughout the three phases: The technique used for the modeling should be easy to understand and to use. This is required as the modelers are often experts in their domain but do not necessarily have high modeling knowledge. The technique should have a simple graphical appearance with an intuitive and not too strict semantic. This is also necessary as the model serves as basis for communication between participants with totally different background and “knowledge cultures”, as CEOs, heads of department, department staff, IT experts and so on. The technique should furthermore be widely accepted and possibly tool supported.

Almost completely different requirements hold for the analysis of processes. Here the underlying modeling technique should be formally founded. Preferably analysis techniques should exist and be supported by tools. Simple graphical appearance and an intuitive understanding play an inferior role.

A further requirement is added to the underlying modeling technique if we want to utilize the model to support the process execution at runtime. The technique should then additionally provide an execution semantic.

It is clear that these are requirements one single modeling technique can not accomplish. Using only one technique throughout the whole procedure there will be deficiencies in some of the mentioned parts. Possibilities to cope with the variety of requirements are the use of different techniques for different purpose or the refinement of a technique to fit the various requirements posed throughout the three phases.

In this paper we propose a combined approach. The idea is to first embed different techniques and to later on refine the resulting model to fit the posed requirements. The gap between informal and formal semantic will be bridged with the help of a pragmatic correctness interpretation.

The proposed approach suggests an iterative procedure consisting out of four steps. Support for modeling and analysis is provided through the steps one to three. The execution of the process then gets supported within step four.

Figure 1 shows the general procedure model. The four steps are: “Modeling with EPC”, “Translation into WF nets”, “Correctness Check and Feedback” and finally the “Control Model Construction”.

In the first step of the procedure, Event driven Process Chains (EPC’s) cf. [16] are used for the modeling. EPC’s are widely accepted in practice due to its comprehensive tool support. They provide an intuitive graphical notation which is proven to meet the intuition of the domain experts. The drawback of EPC’s

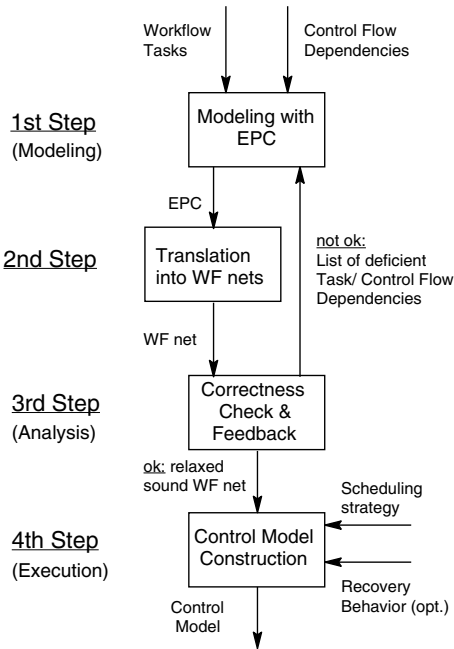


Fig. 1. A Procedure Model for Business Process Modeling

is that syntax and semantics have been defined only in a semiformal manner. As a consequence specifications may be interpreted differently by the designer. When this misunderstanding is discovered by the domain expert it is often too late to correct the design accordingly.

In order to establish a better understanding between the participants in the software development process we provide the modeler with a correctness interpretation for EPC's. As correctness criterion we use relaxed soundness as introduced in [8]. This criterion only checks for some minimum requirements the process specification should satisfy. We will argue that for EPC's the criterion is nevertheless adequate.

The correctness check is part of one of the next steps of the procedure. To apply the correctness criterion we transform the semi-formal model into a formal specification based on Petri nets. As suitable Petri net type we chose Workflow nets (WF nets), cf.[2]. In contrast to previous approaches this transformation does not resolve ambiguities but makes them explicit. The transformation is well defined hence enables the transfer from properties of the WF net to the original EPC. If the WF net is deficient, a precise failure detection is given. The failure disclosure facilitates the improvement of the original specification. This step is repeated until the WF net accomplishes the desired property. As a result of this

step we obtain an EPC describing reasonable behavior as well as a corresponding relaxed sound WF net.

On the basis of the WF net the analysis can be extended in any desired way applying a variety of results of the Petri net theory. Furthermore the possibility to execute the WF net can be utilized for simulation of the process in order to detect optimization potential.

At this point of the procedure the specification developed so far only needs to satisfy some minimum requirements. Relaxed soundness states that at least all intended behavior has been described correctly, but it does not guarantee that malicious executions do not exist. To guarantee a reliable execution of the process a workflow controller is needed supervising and coordinating the execution at run time. The last step of the procedure therefore concerns the construction of a control model. This model is an extension of the relaxed sound WF net meeting stricter requirements. Information that are incorporated pertain a chosen scheduling strategy, thus incorporate efficiency aspects. Operating on the control model the workflow controller will guarantee a reliable execution of the process at run-time.

The benefit of the proposed approach is obvious. It supports the design of business processes starting from an intuitive and semiformal description and conducts the modeler up to a formal specification including details of the implementation. It furthermore relieves the modeler from thinking about efficiency aspects during the first phases of the development process. To decide for an efficient execution strategy is often only possible at run time having further information about the execution context. Such information are e.g. the occurrence probability of a certain failure, costs of failure compensation or changing priorities. The procedure model proposed here postpones the incorporation of efficiency aspects into the last step.

According to the proposed steps the paper is organized as follows: In Section 2 we outline the appearance and the use of EPC's. In Section 3 we introduce WF nets and sketch the transformation from EPC's into WF nets. In Section 4 we describe the application of the correctness criterion "Relaxed Soundness", discuss its appropriateness and outline the state of tool support. In Section 5 we illustrate the construction of a control model incorporating different scheduling strategies and enumerate propositions that are necessary to generate such control models automatically. In Section 6 we finally summarize the work that has been done so far and discuss future work.

2 EPC's for Modeling Business Processes

In an early design phase the processes are described typically with the help of a semiformal, graphical modeling language providing the application developer with some basic modeling patterns [4], such as sequences, synchronization and parallel or alternative splits. As an example of such a graphical modeling language we chose the semiformal Event-driven Process Chains (EPC's) of the Architecture of integrated Information Systems (ARIS) described in [16, 9].

EPC's are widely used (especially in Germany) due to its comprehensive tool support through the commercial ARIS tool-set. In addition EPC's have also been investigated quite thoroughly in research [15, 10, 3, 11]. The language of EPC's provides the modeler with a set of graphical elements for the representation of (business) functions, events and routing constructs to describe the control flow. Functions are used to model the dynamic part of the process. Typical functions are procurement, quality assurance or processing an invoice.

Another constructive element is the event. An event either triggers a function or marks the termination of it. For example (cf. Figure 2), the event *not_ok* triggers the function *notify_cancel* whereas the event *order finished* marks the termination of the function *archive*. Furthermore, to describe more complex behavior such as sequential, conditional, parallel, and iterative routing, connectors are introduced. These fall into two categories: splits and joins. In both we have AND, XOR and OR connectors.

EPC's just describe a set of accepted executions. Deficient executions are only described implicitly, as the set of executions which does not fit the described pattern.

Fig. 2 shows an EPC modeling the process "Handling of incoming order". It represents a reduced version of a real life process of a telephone company. The

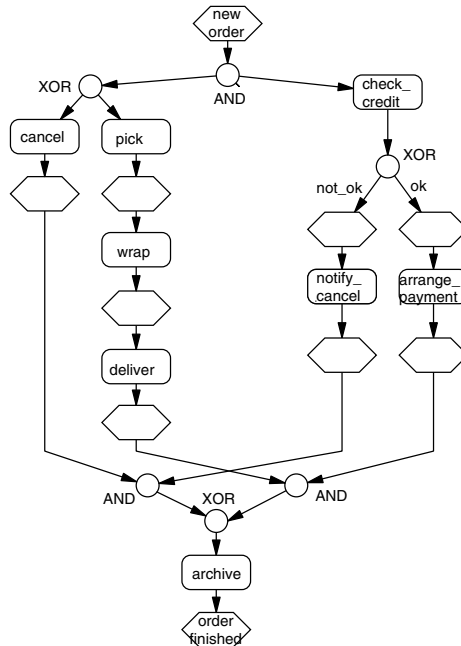


Fig. 2. Handling of incoming order

process models the ordering of a mobile phone which involves two departments: the accountancy handling the payment and the sales handling the distribution.

The process starts with the event *new order*. After that the execution is split into two parallel threads (AND split), the right one models the accounting, whereas the left one models the sales. In the accounting the customer standing is checked first (*check_credit*). The result of this task is either *ok* or *not_ok*. In case the result is positive the payment is arranged (*arrange_payment*), in the latter case the instance is canceled.

The left path models the tasks on the sales side. Here the order is either handled executing tasks *pick*, *wrap* and *deliver*, or *cancel*. The two AND-connectors at the end make sure that only executions are accepted where both sides, the accountancy and the sales department, either cancel the instance or proceed the order. The process "Handling of incoming order" is finished by archiving information on that instance (*archive*).

EPC's are easy to learn and their semantic fits an intuitive modeling understanding. Modeling business processes the modeler normally starts by describing "how the execution should be", hence describes a set of good executions. EPC's are intuitive in its usage because they support exactly this *modus operandi*. They do not expect the modeler to think about all possible executions but just specify the set of accepted executions.

These advantages are compensated by significant disadvantages: first of all ambiguity and vagueness. In the original publications concerning EPC's neither a comprehensive and consistent syntax nor a corresponding semantics has been defined. So EPC's leave room for interpretation and hence ambiguities which makes them less suitable as a basis for the design of information systems. It cannot be in the interest of the domain experts if the processes described in the specification are interpreted differently by the designer. When this misunderstanding is discovered by the domain experts it is often too late to correct the design accordingly. Hence, we suggest to transform the EPC's into a formal representation preserving the ambiguities, i.e. all possibly intended behavior. On the basis of the resulting model we provide a means to check the quality of the underlying EPC and to help with the revision if necessary.

3 Translation into WF Nets

In this section we will transform the semiformal model into a formal specification based on Petri nets. As suitable Petri net type we chose Workflow nets (WF nets). They are introduced in the first parts. In the second part of this section the translation rules are sketched.

3.1 Workflow Nets

Workflow nets (WF nets) are a special kind of P/T nets which have been introduced by Van der Aalst, see [1]. Petri nets are highly suitable for the modeling of control flow aspects, e.g. [2],[5]. They provide an intuitive manner for the

control flow description, and are supported by a rich variety of existing analysis techniques and tools.

A WF net is a Petri net, which has a unique source place (i) and a unique sink place (o). In such a net a task is modeled by a transition and intermediate states are modeled by places. A token in the source place (i) corresponds to a case which needs to be handled, a token in the sink place (o) corresponds to a case that has been handled. The process state is defined by a marking. In addition, a WF net requires all nodes (i.e. transitions and places) to be on some path from (i) to (o). This ensures that every task (transition) and every condition (place) contributes to the processing of cases. In this paper we take the definitions of a Petri net and WF net from [1]:

Definition 1 (Petri net). A Petri net is a triple (P, T, F) :

P is a finite set of places,

T is a finite set of transitions ($P \cap T = \emptyset$),

$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation).

$a\bullet = F(a) = \{b | (a, b) \in F\}$ is called post-set of a .

$\bullet a = F^{-1}(a) = \{b | (b, a) \in F\}$ is called pre-set of a

Definition 2 (strongly connected). A Petri net is strongly connected, if and only if, for every pair of nodes (i.e. places and transitions) x and y , there is a directed path leading from x to y .

Definition 3 (WF net). A Petri net $PN = (P, T, F)$ is a WF net iff:

(i) PN has two special places: i and o . Place i is a source place: $\bullet i = \emptyset$ and place o is a sink place: $o\bullet = \emptyset$.

(ii) If we add a transition t^* to PN which connects place o with i , then the resulting Petri net $\overline{PN}$ is strongly connected.

The global state of a Petri net, called a *marking*, is the distribution of tokens over places $M : P \longrightarrow N$.

A marking M changes by firing a transition t . A transition t may fire only if it is enabled: all its input places are filled with a token. When t fires, from every input place a token is removed, and a token is added to every output place. A transition $t \in T$ is said to be enabled by the marking M iff $\forall p \in \bullet t, p \in P : M(p) \geq 1$. In this case t can fire. Firing of transition t leads from marking M to marking M' , where

$$M'(p) = \begin{cases} M(p) - 1 & \text{if } p \in \bullet t - t\bullet, \\ M(p) + 1 & \text{if } p \in t\bullet - \bullet t, \\ M(p) & \text{otherwise} \end{cases}$$

written as $M \xrightarrow{t} M'$. We write $M_1 \xrightarrow{*} M_n$ if there exist a firing sequence $t_1 t_2 t_3 \dots t_{n-1}$ which leads from state M_1 to state M_n , i.e., $M_1 \xrightarrow{t_1} M_2 \xrightarrow{t_2} \dots \xrightarrow{t_{n-1}} M_n$.

A pair (PN, M_i) is called a system and denotes a Petri Net PN with an initial marking M_i .

3.2 The Translation of EPC's into WF Nets

There have been some attempts to define the semantics of EPC's by mapping it onto Petri nets [15, 10, 3, 11]. Petri nets are used because they have a clear and precise semantics [12] and a similar graphical notation to that of EPC's. In addition they give us access to many existing analysis techniques and tools.

The main difference of EPC's in contrast to Petri nets is that EPC's do not have an operational semantic. An EPC only describes a set of accepted executions, whereas a Petri net describes all possible behavior. Petri nets do have a formal execution semantic specifying *how* an execution is reached. This difference has been neglected in previous attempts of mapping EPC's to Petri nets. As a result EPC's have been considered malicious if the corresponding Petri nets were deficient. As a way out the modeling facilities of EPC's have often become restricted such that only correct Petri nets have been gained through the transformation. In the proposed procedure we support the whole range of the EPC modeling elements (particularly the ambiguous OR connector) and appreciate the provided abstraction level, as it relieves the designer to think about efficiency aspects of the execution in an early design phase.

The EPC in Figure 2 describes the set of executions, where either both departments proceed or both departments cancel the order. An accepted execution of the EPC from Figure 2 is *check_credit, arrange_payment, pick, wrap, deliver, archive* but also *cancel, check_credit, notify_cancel, archive*. The execution *check_credit, notify_cancel, pick, wrap, deliver, archive* is not accepted, hence not described by the EPC. Everybody will affirm that this EPC describes reasonable behavior. It should therefore be considered correct. Previous approaches would reject this process specification, because of deficiencies of the corresponding Petri net. In the following we will see that the Petri net is vulnerable to deadlocks.

Despite of this it should be possible to provide the modeler with a means to check whether the EPC is reasonable, hence whether the described set of executions is a correct one. We think the answer lies in a pragmatic interpretation of the correctness of the corresponding Petri net. We will ensure that the model covers reasonable behavior that can be used as a basis for either the revision of the EPC or as basis for the design of the information system.

The translation of EPC's into WF nets has been introduced in [7]. The transformation takes place in several steps. First, elements of the EPC are mapped onto Petri net-modules. Second, the different modules are combined to form a complex process model. If necessary the resulting Petri net is enhanced finally, such that a WF net is obtained.

The transformation is well-defined. Applying the proposed rules each EPC is assigned to exactly one WF net. This relation will be exploited later transferring WF net properties to EPC's. Figure 3 shows the resulting WF net, applying the translation rules from [7] to the EPC from Figure 2.

In the following we will introduce the properties soundness and relaxed soundness as a means both to check the quality of the underlying EPC and to help with the revision of the business process if necessary.

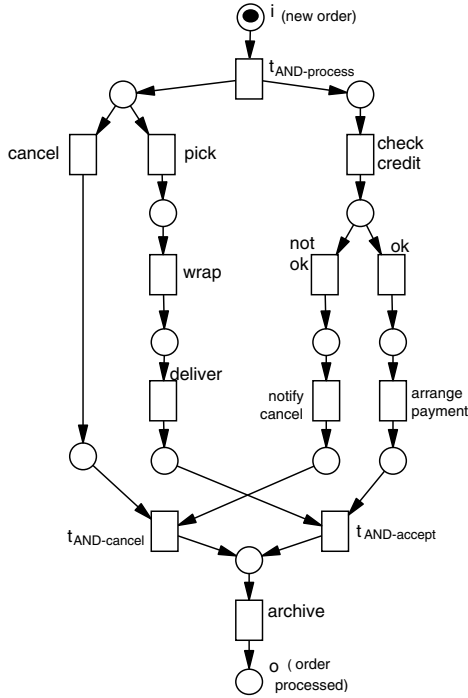


Fig. 3. WF net: "Handling of incoming order"

4 Relaxed Soundness Check

In this section we provide a means to check the EPC for some reasonable behavior. As an adequate means for correctness we refer to *relaxed soundness* [8]. After introducing the new criterion we will describe its application, discuss its appropriateness and outline the state of tool support. In the end of this part we show how the results can be used for revision of the primary specification.

4.1 Correctness Criteria for WF Nets

As WF nets are a special type of Petri nets there exist a lot of analysis techniques that can be applied. The standard property every workflow process should satisfy is soundness as introduced in [2]. Soundness ensures that the process can *always* terminate with a single token in place *o* and all the other places are empty. In addition it requires that there are no dead tasks, i.e. any task can be executed. Next we show soundness as defined in [2].

Definition 4 (Sound Process). *A process specified by a WF net $PN = (P, T, F)$ is sound if and only if:*

(i) For every state M reachable from state i , there exists a firing sequence leading from state M to state o .

Formally: $\forall M (i \xrightarrow{*} M) \Rightarrow (M \xrightarrow{*} o)$.

(ii) State o is the only state reachable from state i with at least one token in place o (proper termination).

Formally: $\forall M (i \xrightarrow{*} M \wedge M \geq o) \Rightarrow (M = o)$

(iii) There are no dead transitions in PN with initial marking i .

Formally: $\forall t \in T \exists M, M' (i \xrightarrow{*} M \xrightarrow{t} M')$

For the correctness check of EPC's the soundness property is not adequate. For this purpose soundness is too strict. If we consider an EPC to be correct only if the corresponding Petri net is sound, we would reject EPC's that do describe reasonable behavior as e.g. the EPC in Figure 2. The corresponding WF net (cf. Figure 3) is not sound because it may deadlock, cf. firing sequence:

– $t_{AND-process}, pick, wrap, check\_credit, not\_ok, deliver, notify\_cancel$.

We therefore use the weaker soundness version: *relaxed soundness* as introduced in [8]. *Relaxed soundness* only ensures that there *exist* a reasonable amount of executions which terminate properly.

We will define the term *sound firing sequence* to explain the differences between the criteria soundness and relaxed soundness formally. In the following we use i and o as identifier for start- and end place as well as to depict the markings where these places contain a token. The respective meaning can be derived from the context.

Definition 5 (Sound Firing Sequence). Let $S = (PN, i)$ be a system. A firing sequence $t_1 t_2 \dots t_k$ is called *sound* iff there exist transitions $t_1 \dots t_n$ and markings $M_1, \dots, M_{n-1}$ that $(i \xrightarrow{t_1} M_1 \dots \xrightarrow{t_k} M_k \xrightarrow{t_l} M_l \dots \xrightarrow{t_{n-1}} M_{n-1} \xrightarrow{t_n} o)$.

A sound firing sequence can be extended such that state o is reached. Relaxed soundness requires that there *exist* a reasonable amount of sound firing sequences. In fact we require at least so many sound firing sequences that each transition is contained in one of them. Note that soundness implies relaxed soundness.

Definition 6 (Relaxed Soundness). A process specified by a WF net $PN = (P, T, F)$ is *relaxed sound* iff every transition is element of some sound firing sequence. For all $t \in T$ there exist M, M' such that $(i \xrightarrow{*} M \xrightarrow{t} M' \xrightarrow{*} o)$.

Intuitively relaxed soundness means that there exist enough executions which terminate properly. Enough means at least so many that every transition is covered. So a relaxed sound WF net may have firing sequences which do not terminate properly, but deadlock before or leave token in the net. In spite of that relaxed soundness is still reasonable because it requires that at least all intended behavior has been described correctly.

The WF net from Figure 3 is relaxed sound. A set of sound firing sequences which contains all transitions is:

1. $t_{AND-process}, pick, wrap, check\_credit, ok, deliver, arrange\_payment,$
 $t_{AND-accept}, archive$ and
2. $t_{AND-process}, check\_credit, not\_ok, notify\_cancel, cancel, t_{AND-cancel},$
 $archive.$

To check for relaxed soundness of a WF net we have to check whether every transition is part of a firing sequence that starts in state i and ends in state o . As there are many transitions, this should be automated by a tool.

4.2 Tool Support for the Relaxed Soundness Check

The relaxed soundness check for bounded systems has been automated within the model checking tool LoLA [17]. LoLA (a Low Level Petri net Analyzer) has been implemented at the Humboldt University of Berlin.

LoLA has been extended to prove for eCTL-formulas. Within extended Computation Tree Logic (eCTL, cf. [14] or [13]), an extension of CTL, it is possible to quantify not only over states, but also over actions. So reachability of a certain state (o) can be required on a certain path which includes the execution of a transition t . To check for relaxed soundness, this has to be done, for every transition of the WF-net.

The approach LoLA takes to verify relaxed soundness is to find sound firing sequences and to mark the contained transitions. For each transition that is not marked, it then tries to find another sound firing sequence. The algorithm stops, if a "witness path" for every transition has been found. If there is a transition, for which no witness path can be found, the Petri net is not relaxed sound. In this case the whole reachability graph has been computed. As result of the relaxed soundness check LoLA outputs either TRUE or FALSE. For every transition that is part of a sound firing sequence a witness path is indicated.

In the positive case (test result: relaxed sound) the provided algorithm will be in general more efficient than the test for soundness, where the whole reachability graph has to be computed always.

During these days the test for relaxed soundness is also added to the functionality of the verification tool Woflan [19]. It will be available in the next release Woflan 3.0

4.3 Transferring the Results to EPC's

We consider an EPC to be relaxed sound if the corresponding WF net is relaxed sound. In terms of the EPC, relaxed soundness means that every function can be executed reaching a desired set of end events.

The results from the correctness check of the WF net can directly be transferred to the primary model. If the result of the relaxed soundness check is TRUE we can conclude that the EPC represents reasonable behavior and can hence be used as a basis for software design. If the result is negative then we have transitions that are not contained in any sound firing sequence. Hence the corresponding part in the EPC needs improvement. Put in other words: as a general

rule we have to consider transitions that are not contained in some sound firing sequence when we are looking for parts of the process that need revision.

As a result of this phase we obtain an EPC describing reasonable behavior and a corresponding relaxed sound WF net. The relaxed sound Petri net is then starting point for a further refinement step. In the last phase of the proposed procedure we will incorporate information about a chosen scheduling strategy.

5 Control Model Construction

Given a relaxed sound process description we know, that “correct (sound) execution is possible”. Relaxed soundness does not imply that the process always terminates properly.

For reliable execution there should be no execution of the processes that deadlocks or leaves tokens in the net. If there is indeed no malicious execution (the process is sound), nothing has to be done.

If the process model is only relaxed sound, the execution of the process should be supported by a workflow controller at run time. The workflow controller supervises the execution and takes care that only sound firing sequences are chosen.

We consider two approaches to implement a workflow controller. The first approach is based on the adaption and customization of existing transaction models (e.g. SAGA's, nested transactions) and to delegate the reliable execution to supporting transaction management systems. The application of transaction management to the workflow domain is investigated since the early nineties. In [18] Sheth and Rusinkiewicz introduced *transactional Workflow* as workflows with transaction support. Since then a lot of work has been done to integrate workflows and transaction models. For a detailed consideration of related work we refer the reader to [8]. In our work we concentrate on a second approach. We will construct a control model starting from the relaxed sound specification.

5.1 Transforming Relaxed Sound WF Nets

Within the second approach a control model is constructed. Operating on the control model the workflow controller will guarantee that only sound executions are chosen. It is clear that the control model must be sound. In the last step of the procedure we therefore concern the construction of a sound model starting from the relaxed sound specification that has been reached so far.

A relaxed sound process may have malicious executions, whereas in sound processes all firing sequences of the corresponding WF net are sound. To generate a sound process model means therefore to restrict the possible firing sequences of the relaxed sound WF net to a subset of only sound firing sequences. Within this restriction not only unsound firing sequences are eliminated but often also sound firing sequences. Which firing sequences are eliminated depends on a chosen scheduling strategy. Possible are optimistic and pessimistic strategies.

Following a pessimistic scheduling strategy the workflow controller would avoid deadlocks by waiting for decisions to be taken in advance. This would be implemented synchronizing both processes at the decision points. Figure 4 shows the control model implementing a pessimistic scheduling strategy for the running example. The primary process model has been enhanced by two places $S1$, $S2$ and corresponding arcs. Through the introduced pattern the two transition ok and $pick$, and transition not_ok and $cancel$ become synchronized. Using this control model as input for the workflow controller the execution of the process becomes serialized. The customer check would always been executed before the ordering. Here all sound but parallel firing sequences of the relaxed sound WF net have been eliminated.

In case the delivery process takes a long time and the customer check takes a long time, this would be very inefficient. This is especially undesirable if it is very rare that a customer check results in a not_ok . Then this way of pessimistic scheduling would be annoying. It would be more efficient to just start the delivery of the order to the customer hoping the customer check will be ok , i.e. following an optimistic approach. Only in the rare case that the decision not_ok was taken, the order should be returned to stock and should be canceled after all.

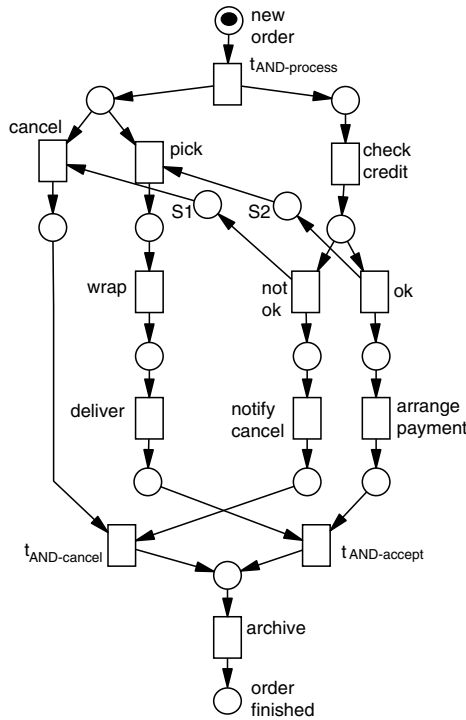


Fig. 4. An implementation of a pessimistic scheduling strategy

To build an optimistic control model some additional information are needed. The application developer has to determine: from which states recovery is possible and not too expensive, what has to be done for the recovery, which are the points, where the process can start again and which event triggers the recovery. These information can be gained from long-term observation of the process or by simulation of the relaxed sound process model. In the latter case the model should be enhanced by some probability distributions at external decision points¹ and appraisals of activity duration and -costs. As results of this investigation we require:

- a list of states from which recovery of a valid state is possible respectively a list of tasks which can be reset or compensated
- a list of compensation tasks
- possible re-entry points
- the trigger information

With these information it is possible to build a control model implementing an optimistic strategy. For the running example we assume that:

- tasks *pick* and *wrap* can be reset without extraordinary charges, whereas task *deliver* is considered to be nonreversible
- the compensation tasks are *recover I* and *recover II*. The first one returns the item to the stock whereas the second additionally unwraps the item
- re-entry point for both compensation tasks is the “canceled-state” on the sales side
- the recovery is triggered if the customer check turned out to be *not_ok*.

An implementation of the optimistic scheduling strategy under these assumptions is shown in Figure 5. Using this optimistic control model the two departments can operate in parallel. The customer check of the accountancy can be executed concurrently to preparative tasks of the sales handling. Here all sound firing sequences of the relaxed sound WF net are maintained. As consequence some additional and not so effective executions are accepted as well.

Both control models, the pessimistic as well as the optimistic one, are sound. There exist no firing sequences that deadlock or do not terminate properly. Furthermore there exists no dead task. As these models are sound they can be used as input for a WF controller. Operating on either of them a reliable execution at runtime can be guaranteed.

6 Conclusion and Future Work

In this paper we proposed a procedure model for the management of business processes. The benefit of the proposed approach is obvious. It supports the

¹ decisions that rely on data or on input from the environment as e.g. interaction of partners or clients

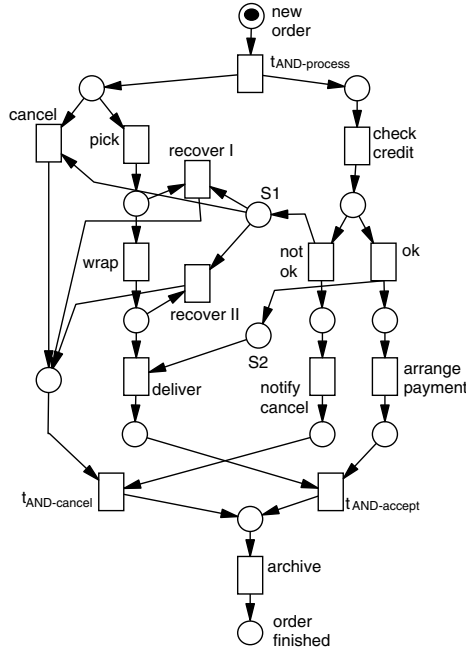


Fig. 5. An implementation of an optimistic scheduling strategy

management of business processes through all its phases, as modeling, analysis and execution, bridging the gap between low modeling expertise on the expert-domain side and high requirements on the system design side.

Starting with an intuitive and semiformal specification we provide rules for the stepwise refinement and end up with a formal and sound business process model.

The innovation of the procedure is based on a pragmatic interpretation of the correctness of WF nets.

In the first phase the application developer uses a semiformal modeling language, namely EPC's which provide an intuitive graphical notation for the specification. At this point of the procedure we do not require high modeling expertise. The modeler is only asked to specify accepted executions but does not need to care about all possible executions. To check the model for correctness we transform the resulting model into a formal specification based on Petri nets. This is done in the second step of the procedure. For this step a set of translation rules is provided. The transformation is well-defined and allows to draw conclusions from the properties of the resulting WF net back to the primary model. In the third step the WF net is checked for *relaxed soundness*. If the net is not relaxed sound a detailed failure disclosure is provided which helps the application developer to revise the primary model. This step is repeated until a relaxed sound specifica-

tion is obtained. Relaxed soundness guarantees that some reasonable behavior is covered. It does not exclude the existence of deficient executions. The malicious executions are eliminated in the last step of the proposed procedure. Here the relaxed sound model is transformed into a sound WF net. Restricting the behavior of the described process to only sound executions a certain strategy becomes implemented. The choice for a strategy (either optimistic or pessimistic) determines the efficiency of the process execution. This late determination has two advantages: It facilitates an intuitive modeling relieving the modeler to think about efficiency aspects already at design time and extends the possibility to reuse modeling results under changing priorities.

The last phase has been illustrated constructing a pessimistic as well as an optimistic control model for the running example. In future we will elaborate on automatic transformations between relaxed sound WF nets and sound WF nets. We aim to provide rules for the generation of a control model for pessimistic as well as optimistic scheduling policies. Another objective of future work is the implementation of an environment supporting the proposed procedure. As one of the next tasks here we concern the construction of an interface between the ARIS tool-set and the model checking tool Woflan implementing the transformation rules from EPC's into WF nets.

References

- [1] W. M. P. van der Aalst. Verification of Workflow Nets. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *LNCS*, pages 407–426. Springer-Verlag, Berlin, 1997. 71, 72
- [2] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998. 68, 71, 74
- [3] W. M. P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999. 70, 73
- [4] W. M. P. van der Aalst, B. Kiepuszewski, A. ter Hofstede, A. Barros, and B. Kiepuszewski. Advanced Workflow Patterns. In O. Etzion and P. Scheuremann, editors, *Proceedings Seventh IFCIS International Conference on Cooperative Information System (CoopIS 2000)*, volume 1901 of *LNCS*, pages 18–29. Springer-Verlag, September 2000. 69
- [5] N. R. Adam, V. Atluri, and W. K. Huang. Modeling and Analysis of Workflows Using Petri Net. *Journal of Intelligent Information System, Special Issue on Workflow and Process Management*, 10(2):131–158, 1998. 71
- [6] J. Dehnert, M. Gajewsky, S. Lembke, and H. Weber. The petri net baukasten: 2nd installment. In *Uniform Approaches to Graphical Process Specification Techniques (UniGra)*, *Satellite workshop to European Joint Conferences on Theory and Practice of Software (ETAPS)*, Genova, Italy, 2001. Springer. 66
- [7] J. Dehnert and P. Rittgen. Relaxed Soundness of Business Processes. In K. L. Dittrich, A. Geppert, and M. C. Norrie, editors, *Advanced Information System Engineering, CAISE 2001*, volume 2068 of *LNCS*, pages 157–170. Springer Verlag, 2001. 73

- [8] W. Derks, J. Dehnert, P. Grefen, and W. Jonker. Customized atomicity specification for transactional workflow. In H. Lu and S. Spaccapietra, editors, *Proceedings of the Third International Symposium on Cooperative Database Systems and Applications (CODAS'01)*, pages 155–164. IEEE Computer Society, April 2001. 68, 74, 75, 77
- [9] G. Keller, M. Nüttgens, and A. W. Scheer. Semantische Prozessmodellierung auf der Grundlage Ereignisgesteuerter Prozessketten (EPK). Veröffentlichungen des Instituts für Wirtschaftsinformatik, Heft 89 (in German), University of Saarland, Saarbrücken, 1992. 69
- [10] P. Langner, C. Schneider, and J. Wehler. Petri net based certification of event driven process chains. In J. Desel and M. Silva, editors, *Application and Theory of Petri nets*, volume 1420 of *LNCS*, pages 286–305. Springer Verlag, Berlin, 1998. 70, 73
- [11] D. Moldt and J. Rodenhagen. Ereignisgesteuerte Prozessketten und Petrinetze zur Modellierung von Workflows. In H. Giese and S. Philippi, editors, *Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme, Workshop des Arbeitskreises GROOM der GI Fachgruppe 2.1.9 Objektorientierte SW-Entwicklung (in German)*, volume 24/00-I of *Fachbericht*, pages 57–63, 2000. 70, 73
- [12] T. Murata. Petri nets: Properties, analysis, and applications. *Proc. of the IEEE*, 77(4):541–580, April 1989. 73
- [13] R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pages 407–419. Springer Verlag, 1990. 76
- [14] S. Roch. extended Computation Tree Logic. In H.D. Burkhard, L. Czaja, A. Skowron, and P. Starke, editors, *Workshop Concurrency, Specification & Programming*, number 140 in *Informatik-Bericht*, pages 225–234, Humboldt-Universität zu Berlin, 2000. 76
- [15] F.J. Rump. *Geschäftsprozessmanagement auf der Basis ereignisgesteuerter Prozessketten. Formalisierung, Analyse und Ausführung von EPKs (in German)*. Teubner, Stuttgart, 1999. 70, 73
- [16] A. W. Scheer. *Business Process Engineering, ARIS-Navigator for Reference Models for Industrial Enterprises*. Springer-Verlag, Berlin, 1994. 67, 69
- [17] K. Schmidt. Lola: A low level analyser. In *Proc. Int. Conf. Application and Theory of Petri net*, volume 1825 of *LNCS*, pages 465–474, 1999. 76
- [18] A. P. Sheth and M. Rusinkiewicz. On transactional workflows. *Data Engineering Bulletin*, 16(2):34–40, 1993. 77
- [19] H.M.W. Verbeek and W.M.P. van der Aalst. Woflan 2.0: A petri-net-based workflow diagnosis tool. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets*, volume 1825 of *LNCS*, pages 475–484. Springer, 2000. 76

Generation of Animation Views for Petri Nets in GenGED^{*}

Claudia Ermel, Roswitha Bardohl, and Hartmut Ehrig

Institut für Theoretische Informatik und Softwaretechnik
Technische Universität Berlin
{lieske,rosi,ehrig}@cs.tu-berlin.de
<http://tfs.cs.tu-berlin.de>

Abstract. Formal specification techniques like Petri nets allow for the formal description and analysis of systems. Tool support exists for many different Petri net classes for editing, simulating and analyzing formal models. A domain-specific animation of net behavior going beyond the well-known token game, however, is not yet supported in most cases. In this paper, we present a formal approach for the generic specification of animation views for different Petri net classes based on GENGED and graph transformation.

The GENGED approach, developed at the TU Berlin, allows for the generic description of visual modeling languages including different Petri net classes. In our framework, the animation view of a system modeled as a Petri net consists of a domain-specific visual layout and an animation view according to the firing behavior of the Petri net class. The basic idea is to generate visual animation rules based on visual syntax rules defining the corresponding Petri net language. We propose a view transformation from the classical Petri net layout to the animation layout. The well-known producer/consumer system modeled as an elementary Petri net serves as running example. We provide an animation view for the application domain *kitchen* where producing and consuming is visualized by icons for *baking* and *eating* cakes, respectively.

1 Introduction

The use of visual modeling techniques today is indispensable in software system specification and development. Especially, specification techniques for communication-based systems must provide means for modeling distributed systems and concurrent behavior. Petri nets allow already the formal specification and analysis of concurrent or distributed systems and the visual description of net models by the graphical notation of nets. This kind of graphical visualization of nets, however, is not always sufficient.

^{*} This work is supported by the joint research project “DFG-Forschergruppe PETRI NET TECHNOLOGY” (H. Weber, Coordinator) at TU Berlin and HU Berlin and by the project GRAPHIT (DLR, Germany/CNPq, Brazil).

In order to support an intuitive understanding of Petri net behavior, especially for non-Petri net experts, it is desirable to have a layout of the model in the application domain. However, there are no specific tools to support an application-specific layout of system states and transformations modeled by Petri nets. Moreover, a formal relationship between the system model based on Petri nets and a corresponding layout of the model as icons from the application domain is missing. Such a support, called animation view for Petri nets in our framework, is presented in this paper based on our general framework for animation of visual languages in [2, 4] and in [13]. The animation view shows directly the states and dynamic changes of a system. In our sample application domain of a kitchen, the animation view of the well-known producer/consumer system visualizes producing as baking and consuming as eating cakes. Our paper is based on a generic approach how to visualize behavior and animation of a system given as a diagram of a specific visual language (VL) which defines e.g. a Petri net class. For this purpose, we use GENGED [1] as generic description of VLs, especially we consider different types of Petri nets (low-level and high-level Petri nets).

The GENGED approach is based on algebraic graph transformation and graphical constraint solving techniques and tools [1, 15, 16] and has been successfully applied to a variety of VLs, including different versions of UML class diagrams, Nassi-Shneiderman diagrams, statecharts as well as low-level and high-level Petri nets [8, 1, 3, 6, 2]. The corresponding visual environment [1, 7] supports the generic description of VLs and the generation of language-specific graphical editors.

Our approach to define the animation of Petri nets relies on the formal basis of GENGED for specifying VLs, including visual modeling techniques like Petri nets. In order to clarify the necessities for such descriptions let us take a closer look on diagrams like those shown in Figure 1. On the one hand, we have some graphical *symbols* like classes and associations in a UML class diagram or like places, transitions and arcs in a Petri net. On the other hand, there are spatial *relations* between symbols, e.g. an association arrow must start at the border of a class symbol, or an arc in a Petri net always connects a place and a transition.

In analogy to formal textual languages, for a specific VL an *alphabet* is defined over which sentences (diagrams), can be constructed. This visual alphabet captures all information about symbols, their relations and layout conditions. Yet, it is not sufficient to construct diagrams over an alphabet. In this case, there would be diagrams with illegal syntactical constellations (like multiple tokens on one place in a diagram over an alphabet for elementary Petri nets). Thus, like in formal textual languages, we have to give some *rules* defining the generation of syntactically legal diagrams over an alphabet. Together with a *start diagram*, these rules form a *syntax grammar*. The alphabet and the grammar of a VL establish a VL specification over which diagrams can be generated. Using graph grammars as underlying formal basis, we have a natural visual formalism for the definition of VLs. Moreover, the well-defined concepts of VL specifications in GENGED offer the basis for the specification of animation.

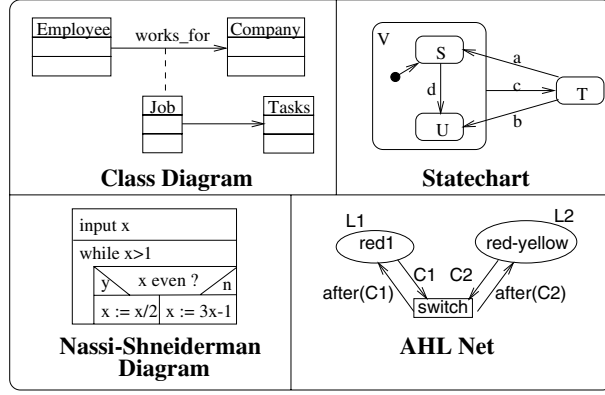


Fig. 1. Diagrams from Visual Languages

The behavior of a Petri net is given in our proposed GENGED framework by a set of behavior rules which correspond to the firing behavior of the transitions ([5, 9, 17]). An animation view for a Petri net basically is defined by a new layout of the Petri net tokens in each marking, i.e. a combination of suitable icons according to the specific application domain of the model. Additionally, the behavior rules modeling the firing behavior are transformed into animation rules for the animation view defining the state transitions of the system in the new animation layout. In order to have compatibility with the behavior rules, we define view transformation rules leading from the behavior rules to the animation rules of the animation view. In the case of our simple producer/consumer example, the animation rules directly define the production and consumption of cakes in the kitchen.

The paper is organized as follows: In Section 2 we introduce our running example, a producer/consumer system specified as elementary Petri net. We present an animation view of the system in the domain of a kitchen where two persons are baking (producing) and eating (consuming) cakes, respectively. Section 3 at first reviews the formal specification technique of algebraic graph grammars. Then we go into more detail and explain the GENGED approach applying its concepts to the specification of the visual syntax of a Petri net language. The GENGED framework is extended to incorporate the specification of behavior and animation of Petri nets in our main section 4. The current state of the implementation of the concepts in the GENGED environment is sketched in Section 5.

2 Example: A Producer/Consumer System

As running example for the formal specification of a system model and its animation view, we use the well-known specification of a producer/consumer system as elementary Petri net. This example is (like the reader/writer protocol) one of

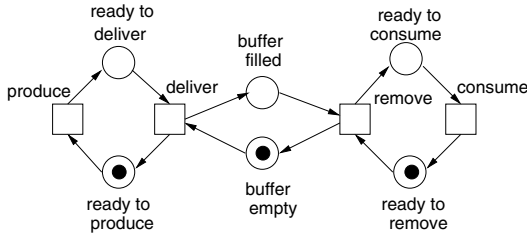


Fig. 2. Elementary Net *Producer/Consumer*

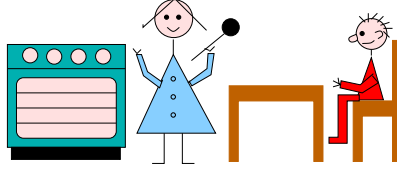


Fig. 3. Animation View for *Producer/Consumer*

the basic models for communication-based systems: two independent agents (the *producer* and the *consumer*) communicate via a channel (the *buffer*). The producer sends messages (writes) to the channel, and the consumer receives (reads) them from the channel. Rather than visualizing the flow (reading/writing) of messages, we suggest an animation view where real goods are produced and consumed. Thus, the underlying idea is visualized in a more concrete way.

The visual language (VL) we specify by using the GENGED approach defines the common graphical representation of elementary nets.

Fig. 2 shows the Petri net modeling the behavior of a producer who is producing and delivering goods, and a client (consumer) who is removing the goods from a buffer and consuming them. The places modeling the different buffer states on the one hand ensure that goods are delivered only if the consumer needs them (the buffer is empty). On the other hand, the consumer can consume a new good only if the buffer was filled by the producer in advance. The left subnet consisting of the places *ready to deliver* and *ready to produce* and the transitions *produce* and *deliver* is the specification part corresponding to the producer. Analogously, the right subnet (the places *ready to consume* and *ready to remove* and the transitions *remove* and *consume*) corresponds to the consumer. The places *buffer filled* and *buffer empty* and the arcs to the producer and consumer subnets model the buffer, i.e. the interface between producer and consumer.

A possible animation view of the net is illustrated in Fig. 3. The producer and the consumer are visualized as symbols representing a mother and her child in a kitchen. The mother is producing cakes and the child is consuming them.

The producer subnet corresponds to the mother standing near the stove baking (producing) cakes, and putting (delivering) them onto the table (the buffer), whereas the consumer subnet is visualized as a child taking (removing)

the cakes from the table and eating (consuming) them. As the Petri net marking models a state of the system where both producer and consumer are ready (waiting) to produce respectively consume, and the buffer is empty, there is no cake to be seen in our animation view, but the mother is ready to produce one, and the child is waiting for it.

The transitions of the Petri net in Fig. 2 are not visualized in the animation view. In the user interface of the GENGED environment they correspond to animation rules visualized in an extra menu (see Section 5). The animation then is triggered by the user who selects an animation rule whereupon an action is performed (if the respective transition is enabled in the corresponding Petri net marking). Thus the behavior of the Petri net (the token game) can be simulated in the animation view.

Of course, different animations for the same system model are feasible. For example, communication between two partners in general might be visualized by special symbols for requests and answers, or by animating the contents of a message appropriately.

3 Defining Petri Net Languages within GenGED

In this section we review the basic concepts used for generic description of syntax, behavior and animation of Petri nets using GENGED. In general these basic concepts are given by algebraic graph transformation which is briefly introduced in Section 3.1. In the GENGED approach graph transformation is applied to the generic description of visual languages (VLs) consisting of a visual alphabet and a visual grammar. We review the GENGED concepts with the focus on the specification of a place/transition Petri net language: in Section 3.2 we give a visual alphabet of Petri nets, and in Section 3.3 we propose a visual syntax grammar which is based on the visual alphabet.

3.1 Review of Graph Transformation

In GENGED, diagrams as the Petri net in Fig. 2 are visual sentences of a VL, i.e. they consist of an abstract syntax level (the symbols and links) and a concrete syntax level (their layout). Diagrams are formalized as attributed graph structures, a generalization of attributed graphs. Attributed graph structures allow to define arbitrary graphical symbols as sorts and their connections as operations in a corresponding attributed graph structure signature. The formalization of rule applications as categorical pushout construction in the category of attributed graph structures is slightly different to the construction for graphs and allows a cleaner separation of operations on graphs and data type attributes. In this paper, it is sufficient to keep in mind that all attributed graphs are attributed graph structures and that the transformation of visual sentences by rules of a visual grammar works in a way similar to attributed graph transformation. Hence, in this section we review the main concepts of attributed graph transformation [19] as they are used within GENGED. A detailed discourse on the formal backgrounds can be found in [1].

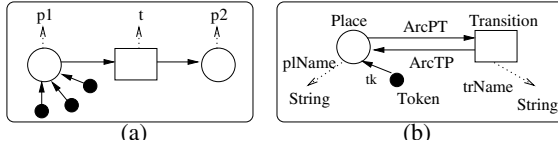


Fig. 4. A Graph (a) typed over the Type Graph (b)

We illustrate the use of attributed graph grammars in GENGED by specifying simple place/transition nets (P/T nets) as graphs and sketching their manipulation as graph rules. Our P/T nets allow multiple black tokens for each place, but restrict the arc weight to one for all arcs, therefore we have no arc inscriptions in a net. The sample elementary net in Fig. 2 then can be expressed as a sentence of our specified P/T net language.

In the theory of algebraic graph transformation, a graph is given by two disjoint sets (graph objects), called *nodes* (vertices) and *edges* (directed arcs) from a source node to a target node. Every graph object is typed over a *type graph*. Fig. 4 (a) represents a graph with six nodes and five arcs (between them). The nodes are of type *Place* (white circles), *Transition* (rectangles) and *Token* (black dots). The arcs representing Petri net arcs are of type *ArcPT* and *ArcTP* (solid lines), whereas tokens belonging to a place are represented by *Token* nodes connected to a *Place* node by arcs of type *tk*. The corresponding type graph is shown in Fig. 4 (b). Here, the nodes and arcs represent the types themselves, whereas the graph objects in Fig. 4 (a) can be seen as instances of these types. Note that the type graph poses some restrictions on possible instances as, e.g. instances must not have arcs connecting two places or two transitions.

Nodes and arcs may be additionally labeled by *attributes* which are used to store data together with the graph objects. In this paper we will only use attributes for nodes. In the type graph attributes are denoted by an edge carrying an attribute type name connecting a node to its attribute type (a set). In the instance graphs an attribute edge will connect a node with the current value of that attribute. In Fig. 4, the type graph (b) specifies that a *Place* node contains an attribute named *plName* of data type *String* denoting the name of a place. In the instance graph (a) the value of this attribute is a concrete name for each place. We allow abstract data types for attributes, i.e. we consider not only the sets of types, but also operations on these types. In particular, the use of abstract data types allows us to use variables and terms as attributes (by choosing a term algebra as attribute algebra). As we will see later on, this is useful for a specification of behavior as graph grammar.

A relationship between two graphs can be expressed by a *graph morphism* g which maps the nodes and arcs of the first graph G to nodes and arcs of the second graph H , denoted by $g : G \rightarrow H$. The graph objects in G are called *origins* and in H *images*. The mappings have to be *type compatible* (nodes and arcs are mapped to nodes and arcs of the same type) and *compatible with structure* (the

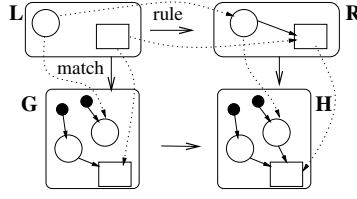


Fig. 5. Application of a Graph Rule

source/target node of an arc is mapped to the source/target node of the arc's image). Attribute values (if any) also have to coincide.

Graph transformation defines a rule-based manipulation of graphs¹. Graph grammars (consisting of a start graph and a set of graph rules) generalize Chomsky grammars from strings to graphs. The start graph represents the initial state of the system, whereas the set of rules describes the possible state changes that can occur in the system. A rule comprises two graphs: a left-hand side L (or LHS) and a right-hand side R (or RHS), and a graph morphism $r : L \rightarrow R$ between the graph objects of L and R . Graph objects in L which do not have an image via r in R are deleted; graph objects in R without original in L are created, and graph objects in L which are mapped to R by r are preserved by the rule.

The application of a rule to a graph G (derivation) requires a mapping from the rule's left-hand side L to this graph G . This mapping, called *match*, is a graph morphism $m : L \rightarrow G$. A match marks the graph objects in the working graph that participate in the rule application, namely the graph objects in the image of m . The rule application itself consists of three steps. First, the graph objects marked in the rule for deletion are deleted. Thereafter, the new graph objects are appended to the graph. As a last step, all dangling arcs are deleted from the graph. The graph transformation results in a transformed graph H . Fig. 5 shows the application of a rule inserting an arc between a place and a transition to a graph G representing a Petri net. The resulting graph H contains the Petri net after the application of the rule, i.e. the arc has been inserted.

Rules may contain variables and terms as attributes. Using attributed graphs, the attribute values or variables of the rule's left-hand side have to match as well. An attribute variable is bound to an attribute value in the mapped graph object by the match. In the transformed graph, the attribute values are evaluated depending on the rule's right-hand side and result in a constant value.

3.2 The Visual Alphabet

In general, a diagram consists of a set of *symbol graphics* that are spatially related. We offer graphical constraints for these spatial relationships, called *link constraints*. Symbol graphics and link constraints concern the layout of diagrams,

¹ We here follow the Algebraic Single-Pushout approach to graph grammars [19, 10].

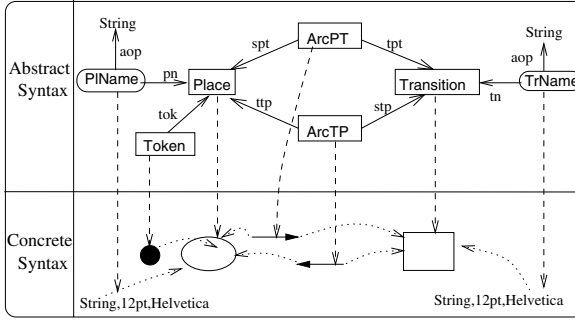


Fig. 6. Abstract and Concrete Syntax of the P/T Net Alphabet

called *concrete syntax*. The logical part of diagrams is called *abstract syntax*. The combination of both syntactical levels is called *visual syntax* level and can be represented by attributed graphs.

A *visual alphabet* establishes a type system for *symbols* and *links*, i.e. it defines the vocabulary of a VL. Note that in an alphabet, the symbol and link types have to be unique as well as the link arcs have to be acyclic.

Symbol graphics and graphical constraints specify layout conditions. In addition to logical (node) attributes as considered in Section 3.1, symbol graphics define a further kind of attributes for all abstract symbol nodes. Graphical constraints specify layout conditions. They are given by (in-)equations over constraint variables denoting the positions and sizes of graphical objects. For example, the constraint "point p_1 lies always to the left of point p_2 " can be expressed as in-equation $p_1.x \leq p_2.x$ over the x coordinates of both points. The set of all constraint variables and constraints define a constraint satisfaction problem (CSP) that has to be solved by an adequate variable binding in a diagram over the alphabet.

Definition 1 (Visual P/T Net Alphabet).

The visual alphabet of the P/T net language, called P/T net alphabet, is briefly illustrated as a graph in Fig. 6. We use rectangles for the abstract syntax of lexical symbols and rounded rectangles for the abstract syntax of attribute symbols. The dashed arrows mark the connections of the abstract syntax and the concrete syntax level.

In the P/T net alphabet, the attribute symbol for the place name is called `PIName` and linked by `pn` to the lexical symbol `Place`. The attribute symbol for the transition name is called `TrName` and linked by `tn` to the lexical symbol `Transition`. Each name is given by a `String` data type that is to be written in a certain text font and text size. We distinguish arcs that run from places to transitions (`ArcPT`) and arcs that run from transitions to places (`ArcTP`). Both kinds of arcs have a certain source and target symbol where they are linked to (depicted by the edges `spt`, `tpt`, `stp`, `tsp`, short for source/target of place-transition arc resp. transition-place arc).

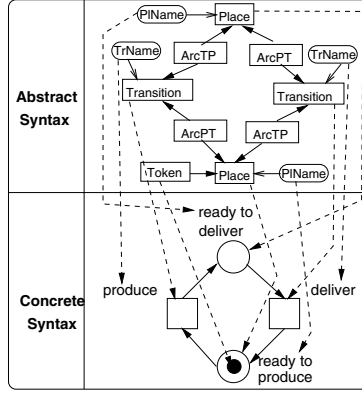


Fig. 7. Visual Sentence over the P/T Net Alphabet

Some link constraints are illustrated by dotted arrows between the symbol layouts. The constraints force a specific layout of diagrams typed over the visual alphabet. For example, one constraint ensures that the place name is always “near” the ellipse. Another constraint on the connection of a token symbol and its place states that a token is always visualized “inside” the corresponding place symbol. $\triangle$

Note that the visual alphabet of the AHL net language [4] is similar to the visual alphabet of P/T nets presented above. The difference is given by arc inscriptions and the modeling of tokens. In the AHL net alphabet, tokens are modeled as string data types instead of pictorial objects as in the P/T net alphabet. Arc inscriptions are modeled as string data types, too. Obviously, the definition of alphabets for different Petri net classes in GENGED is straightforward. Especially for the design of visualizations of new Petri net features, GENGED offers a simple way to generate a prototype editor. For an example, see [14] where a visual language for model evolution is defined consisting of a combination of AHL nets and class diagrams.

Let us proceed with the P/T net language. An example for a visual sentence over our P/T net alphabet is our producer/consumer system in Fig. 2. Here, only symbols and links from our alphabet are used and connected according to the alphabet (the type graph). All graphical constraints are satisfied. For illustration we show the visual syntax of a subnet of this system in Fig. 7.

As already mentioned, the visual alphabet establishes a type system for all possible instances. Such instances also occur in visual syntax grammars we consider in the following section.

3.3 The Visual Syntax Grammar

The visual alphabet is the basis to define the syntax grammar for our P/T net language. The syntax grammar is represented by a graph grammar: it consists of

a *start diagram* and a finite set of *graph rules*. The start diagram and both sides of a rule are diagrams typed over a specific alphabet, as well as the diagrams which can be derived by applying grammar rules. In our graph grammar rules, we use in addition to the left-hand rule sides so-called negative application conditions (NACs) which restrict the application of a rule. An NAC is a graph containing a forbidden graph pattern. The rule must not be applied to a graph if there is a match from the NAC to the graph, i.e. the forbidden pattern is found in the graph.

In Def. 2 we define the syntax grammar for the P/T net language. This definition has to be fixed once and allows the generation of a syntax-directed editor for arbitrary P/T nets (sentences corresponding to the defined syntax of the language). We define the insertion of the symbols *Place*, *Transition*, *ArcPT*, *ArcTP* and *Token*, as well as their graphical relation.

Definition 2 (P/T Net Syntax Grammar).

Fig. 8 illustrates a syntax grammar for our P/T net language which is based on the visual alphabet in Def. 1. In our P/T net grammar, the start sentence is empty. The first rule supports the insertion of a place together with a place name; the NAC requires that the place with the user-defined name is not already in the sentence where the rule is going to be applied to. The second rule analogously supports the insertion of a transition symbol. The next two rules allow for the insertion of arcs, either running from a place to a transition (*insArcPT*) or running from a transition to a place (*insArcTP*). The NACs forbid the application if there is already an arc. Last but not least, tokens can be inserted by applying the rule *addToken*. $\triangle$

The application of a rule to a diagram G is obtained via a match morphism on the abstract syntax level as explained in Section 3.1. The derivation of the abstract syntax of diagram H has to be extended by its concrete syntax according to the alphabet. The diagram-specific CSP of H is derived from the CSP of the alphabet, and a CSP satisfaction (a solution) is computed. Fig. 9 illustrates the application of the rule *InsArcPT*. This rule allows the insertion of an arc between a place and a transition in our visual P/T net language. The match and rule morphisms are indicated by the numbers of corresponding nodes.

A VL is generated by applying syntax grammar rules. The sentences occurring in the grammar as well as those which are derived by applying grammar rules are typed over the corresponding visual alphabet. Hence, a VL is generated by a VL specification.

Definition 3 (Visual P/T Net Language).

The visual P/T net language is given by the VL specification

$$P/T \text{ Net Specification} = (P/T \text{ Net Alphabet}, P/T \text{ Net Syntax Grammar})$$

where the P/T Net Alphabet has been defined in Def. 1, and the P/T Net Syntax Grammar is given in Def. 2. $\triangle$

The visual P/T net language as regarded so far is the basis to define behavior rules and domain-specific animation views as explained in the following section.

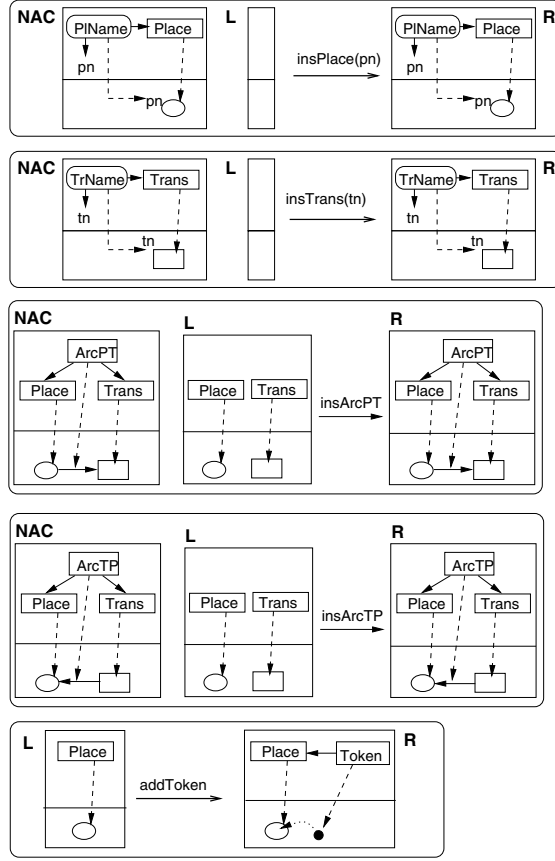


Fig. 8. Visual Syntax Grammar for the P/T Net Language

4 Animation of Petri Nets within GenGED

In order to bridge the gap between the underlying formal, descriptive specification of a process model (i.e. a Petri net) and a natural dynamic visual representation of processes being simulated, we suggest the definition of an animation view for a visual process model. On the one hand, this animation view has to be readily comprehensible; people who are non-specialists in the modeling technique of Petri nets should be able to observe functional behavior of the model. On the other hand, the behavior shown in the animation view has to correspond to the behavior defined in the original process model. In Section 4.1, we consider the graph grammar based concepts for the behavior specification of Petri nets. These concepts are explained along our Producer/Consumer net. Thereafter, in Section 4.2 we give guidelines for the specification of an animation view for a specific Petri net and present an application-specific animation grammar for our Producer/Consumer net. To ensure a consistent mapping of the Petri net

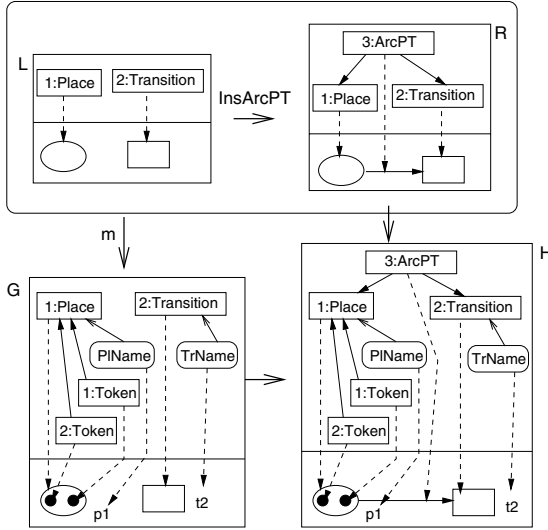


Fig. 9. Application of the Rule *InsArcPT*

behavior, we propose in Section 4.3 a graph grammar based view transformation from the Petri net to its animation view.

4.1 Behavior of Petri Nets

In Petri nets and similar visual process models, two aspects can be considered. The first aspect concerns the topological structure of the model, i.e., which visual elements exist and how are they linked to each other. The second aspect concerns the behavior of the modeled system, i.e., the flow of control within processes and the flow of communication between processes. In the previous section, we dealt with the visual specification of the Petri net language, i.e. we specified what a Petri net *is*, but not what it *does*. Therefore, in this section we focus on simulating the dynamic behavior of P/T nets by a visual grammar approach based on our P/T net language.

In the literature one can find several graph grammar based approaches for the specification of Petri net behavior: Schneider [21] summarizes different approaches to define the behavior of process systems (e.g. Petri nets, event structures and actors) by graph grammars. He states that graph grammars are well suited to describe in a uniform way not only the syntactical structure of visual process models but also their semantic properties. In Petri nets, the basic form of a state transformation is the firing of a transition. The straightforward technique for behavior simulation therefore is playing the token game.

One of the first who discussed the relationship between graph grammars and Petri nets is Kreowski [18]. He associates a graph rule to each transition, with the rule being applicable if and only if the transition is allowed to fire. Tokens within

a place are modeled as a bundle of new nodes connected to the node representing the place. This approach is able to handle both places with bounded capacity and places with unbounded capacity and can easily be extended to individual tokens.

Parisi-Presicce et al. [20] use a structured alphabet for labeling places with tokens. This alphabet has to allow changes of node labels in rule morphisms. In the case of high-level Petri nets, multiple and individual tokens can be represented using multisets as labels and the multiset inclusion as the structure of the alphabet. A further step is to allow arbitrary categories to label the places. In [2], we modeled tokens in AHL nets as arbitrary algebraic data types which are attributes of *Token* nodes.

We consider a Petri net as the set of all sentences over the Petri net language with the same net structure whose markings are given by an initial marking and all possible successor markings that can be reached by arbitrary transition firing steps. The behavior of a P/T net is defined by a visual grammar, called *behavior grammar*: the start sentence corresponds to the initial system state (the initial marking), and the behavior rules capture all possible transition firing steps in the net. When constructing a behavior grammar whose rules correspond to firing of the transitions of the net we have to ensure that

- a transition in the net is enabled if and only if the corresponding rule is applicable to the net,
- firing a transition in the net corresponds to a derivation step in the grammar and vice versa.

The token game then can be simulated by applying the rules of the grammar to a Petri net. The left-hand side defines the applicability condition, i.e. the marking corresponding to the transition's pre-domain. The right-hand side describes the effect of the transition, i.e. the marking of the pre domain places is removed, and the required tokens are added to the post domain places. This approach to Petri net simulation can be applied to various types of low-level and high-level Petri nets (see e.g. [4]). For each Petri net type, therefore, a specific net's behavior grammar can be generated automatically according to the Petri net type's firing behavior.

Example 1 (Behavior Grammar for Producer/Consumer System).

Fig. 10 illustrates the behavior grammar for the Producer/Consumer System. The start graph is the initially marked net as depicted in Fig. 2. Note that the P/T net behavior grammar is defined on top of the visual P/T net grammar of our P/T net language using the same visual alphabet.

◇

4.2 Animation View

In our approach, both the Petri net and its animation view consist of visual sentences based on the same abstract visual alphabet. They differ in the concrete syntax, i.e. we define different layouts for the same underlying process model. We

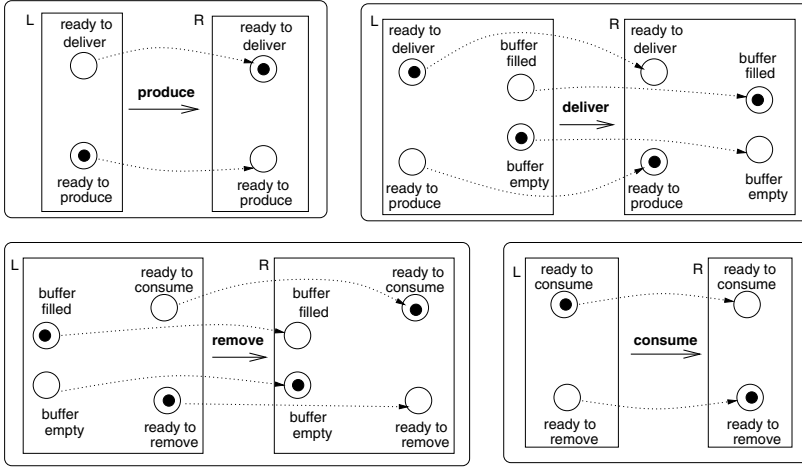


Fig. 10. Behavior Grammar for the *Producer/Consumer* Net

suggest the following guidelines for the definition of an animation view layout for all symbols and links from a Petri net alphabet (like the alphabet from Fig. 6):

- Places in a Petri net give meanings to tokens by defining their properties. In a net-independent animation view, places are not needed any more because properties of tokens now are incorporated in the concrete layout of the tokens themselves. Therefore, we visualize places as symbols of the “*fixed part*” of the animation view, i.e. the part of the view which is not changed by animation.
- The *animated part* consists of the symbols which are changed during animation and corresponds to the token game of the Petri net.
- Transitions are replaced by rule names in the animation view which are the user interface to trigger a state transformation step corresponding to a firing step of the transition in the Petri net view.
- Arcs in the Petri net have the function to define the firing behavior in a static way. They are not needed in the animation view as the behavior now is defined by the animation rules and visualized by their application (the animation). Thus, arcs are not visualized in the animation view.

Example 2 (Animation View).

The *Producer/Consumer* net as illustrated in Fig. 2, is one sentence of our P/T net language we defined in Def. 3. The animation view of this sentence has been already motivated by Fig. 3 where two people in a kitchen are visualized. The abstract syntax of the alphabet of the animation view is equal to that of Fig. 6 but the concrete layout differs.

The fixed part of the animation view consists of the symbols for the mother standing besides the stove (corresponding to the producer places *ready to produce* and *ready to deliver*), the table (the buffer places *buffer filled* and *buffer empty*) and

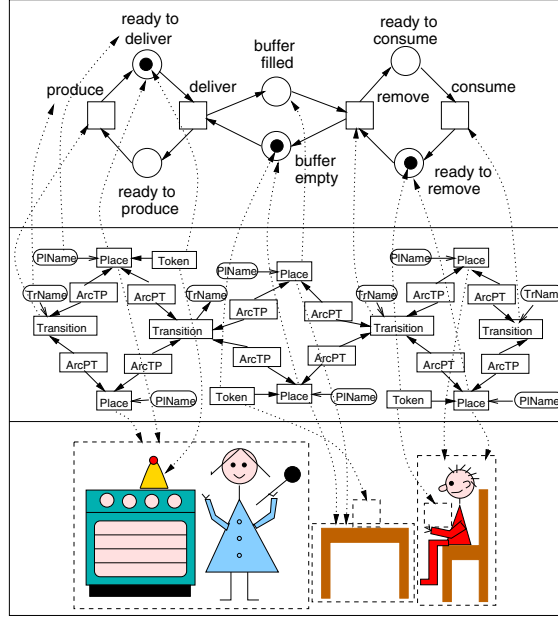


Fig. 11. Visual Syntax of the Animation View of the Petri Net *Producer/Consumer*

the child sitting on a chair (the consumer places ready to remove and ready to consume). The tokens – corresponding to the animated part of the animation view – model the different locations of a cake. A token on place ready to produce means that a cake may be taken out of the stove but is not yet to be seen. In the concrete syntax of the animation view, the token is marked as invisible. The same holds for tokens on the places buffer empty (no cake on the table) and ready to remove (the child has got no cake). Therefore, in the animation view of the initially marked net (Fig. 3), no cake is to be seen.

Fig. 11 shows our net with the only possible successor marking of the initial marking. Here, a token lies on place ready to deliver: the cake has been taken out of the stove and is visible in the animation view on top of the stove. Successor markings would be visualized by the cake put onto the table (token on place buffer filled) and the cake on the lap of the child (token on place ready to consume). The abstract syntax underlying both views (the Petri net and the animation view) also is depicted in Fig. 11, together with some of the respective connections to the different layouts.

◇

Based on the abstract syntax, we define the generation of an animation view by grammar rules that transform all possible states of the Petri net into an appropriate state of the animation view (view transformation rules). The generation of the animation view in Example 2 is described in Section 4.3.

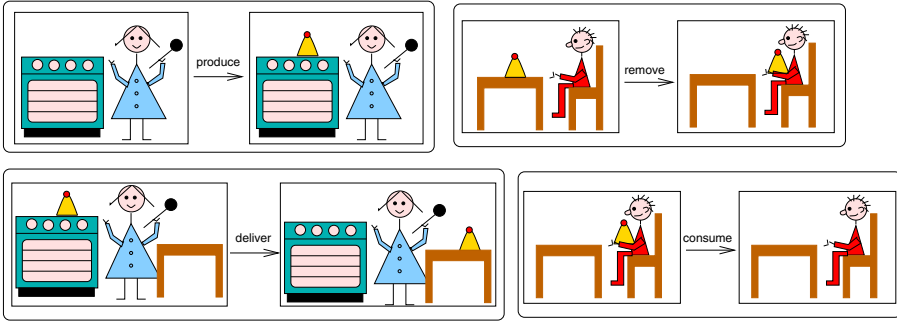


Fig. 12. Animation Grammar for the *Producer/Consumer System*

The behavior of the system in the animation view is given by a set of animation rules on the VL sentences of the animation view. The abstract syntax of the animation rules equals the abstract syntax of the behavior rules for the Petri net (see e.g. Example 1 for a behavior grammar of a specific Petri net). We call the grammar containing the animation rules, the *animation grammar*.

Example 3 (Animation Grammar).

Fig. 12 shows the animation grammar for our producer/consumer system. Each animation rule corresponds to the behavior rule of the same name from the behavior grammar in Fig. 10.

◇

4.3 Generating Animation Views

The aim, of course, is to construct the animation grammar in a way that the animation is consistent to the behavior specification. Therefore, we now define the generation of an animation view by grammar rules that transform all possible states of the system model into an appropriate state of the animation view (view transformation rules). The view transformation rules must be defined by the animation view designer. On the basis of these rules it is possible to enforce coherence between the behavior grammar of the original visual process model (the Petri net) and the animation grammar of the animation view. The view transformation rules allow to transform the VL sentences from the old layout to the new layout and the behavior rules into the animation rules.

In general, each of the view transformation rules transforms a part of the Petri net to a part of the animation view by combining elements of the abstract syntax with new concrete syntax elements (i.e. by giving them a different layout). The concrete syntax of the Petri net transitions, arcs and the attributes is invisible in the animation view. After application of all view transformation rules in a suitable way, the VL sentence denoting the initially marked Petri net is transformed into a corresponding animation view.

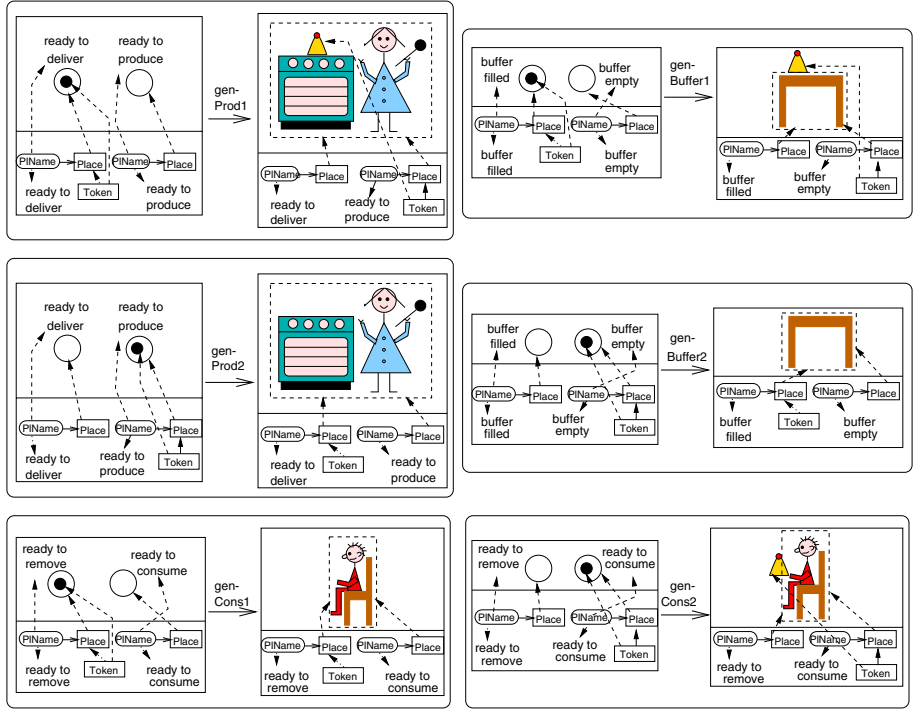


Fig. 13. View Transformation Rules for the Producer/Consumer System

Formally, view transformation rules operate on the union of the visual syntax of the Petri net and the visual syntax of its animation view because both of them contain different graphics. In the following example, therefore, graphics from both concrete syntax definitions (net layout and animation view layout) are shown in the same rules.

Example 4 (View Transformation Rules).

Fig. 13 shows the view transformation rules needed for the generation of the animation view depicted in Fig. 3 from the Petri net depicted in Fig. 2. We define a view transformation rule for each part of the animation view that has a symbol for an underlying fixed part, i.e. for the producer, the buffer and the consumer. Note that the layout of a token in the animation view depends on the place it belongs to. For example, a token on place *buffer filled* is shown as a cake on the table, whereas a token on place *buffer empty* is invisible in the animation view. In the view transformation rules in Fig. 13, the abstract syntax remains, but the symbols are re-linked to the new animation view graphics as they are introduced in the right-hand sides of the rules.

◇

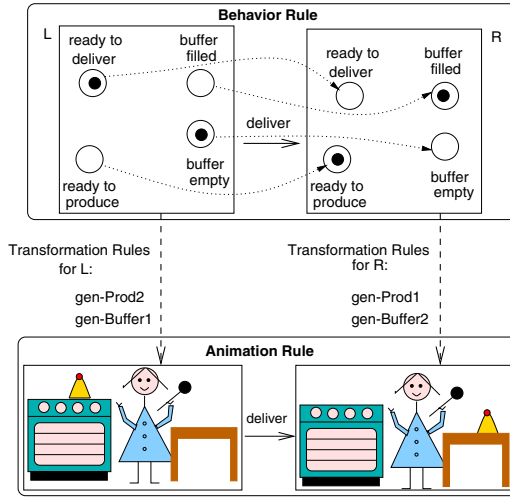


Fig. 14. Derivation of an Animation Rule with View Transformation Rules

By applying the view transformation rules from the view transformation grammar described in Example 4 to the Petri net in Fig. 2, we generate the animation view depicted in Fig. 3. The behavior of a Petri net now can be transferred consistently to the animation view by applying the view transformation rules to the LHSs and RHSs of the behavior rules.

Example 5 (Generating Animation Rules).

Fig. 14 illustrates the derivation of an animation rule by applying rules from the transformation grammar in Fig. 13 to a behavior rule in Fig. 10.

◇

A first rough animation of the system can be performed by applying the animation rules in the animation view of the system model. A nice extension of the approach towards a more sophisticated animation would be the presentation of system behavior not as discrete steps but as a movie ("smooth" animation), i.e. showing a series of intermediate states for the firing of one transition. With this aim in mind, an animation framework as proposed in [22] could be combined with the GENGED environment.

5 Implementation

According to the constituents of a VL-specification, the GENGED environment as sketched in Fig. 15 comprises two major components: the *Alphabet Editor* and the *Grammar Editor* for the visual definition of VLs. From the VL definition using these editors, a VL specification is generated which is the input of the *Graphical Editor* for syntax-directed diagram drawing. This means that the

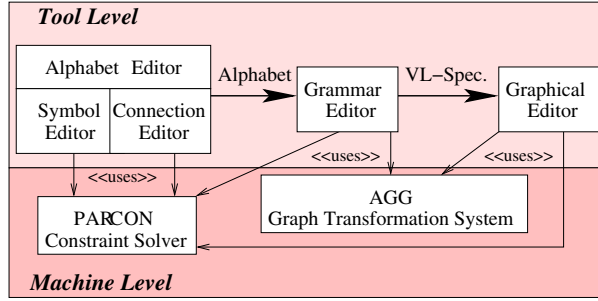


Fig. 15. Overview about the GENGED environment

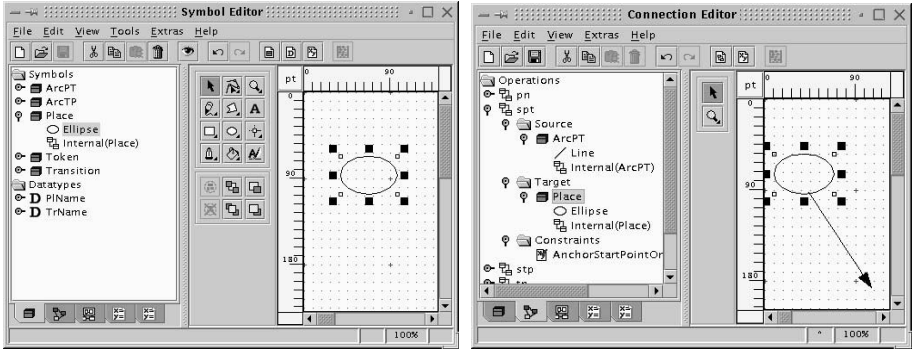


Fig. 16. The Symbol Editor and the Connection Editor of the Alphabet Editor

language-specific editing commands of the Graphical Editor are given by the grammar rules of the visual grammar. Hence, not only a VL is specified but the VL-specific Graphical Editor also. Note that we distinguish two kinds of users, namely users defining a VL (*language-designer*), and those who use a Graphical Editor.

To assure the graphically correct drawing, all GENGED editors use the constraint solver PARCON [16]. The transformation of diagrams via rule application in the Grammar Editor and the Graphical Editor is done by the graph transformation system AGG [15]. The GENGED environment is implemented in Java, so is the AGG system. The PARCON constraint solver – implemented in C – is available for Linux and Solaris, thus GENGED runs on these two platforms.

5.1 The Visual Syntax

The specification of a Visual Alphabet is implemented as *Alphabet Editor* which is a bundle of two sub-editors – the *Symbol Editor* and the *Connection Editor*. A snapshot of the both the Symbol Editor and the Connection Editor is shown in Figure 16.

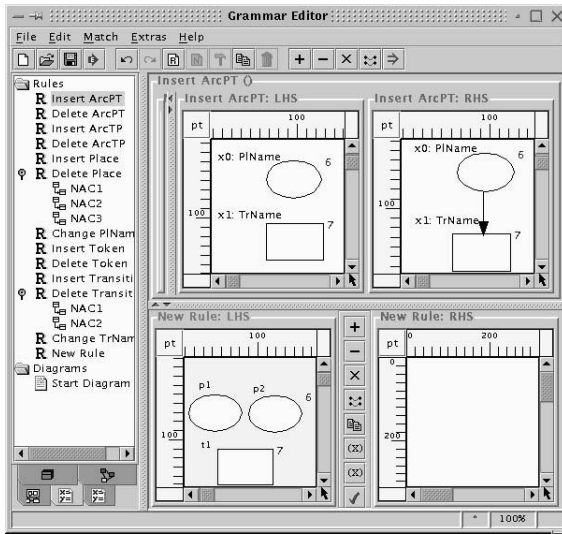


Fig. 17. Grammar Editor

In the Symbol editor, the language elements are defined: For each symbol type in the abstract syntax the user gives a unique symbol name (e.g. *Place*) and a symbol graphic (e.g. an ellipse) and possibly some symbol constraints. For the definition of the symbol graphics, the Symbol Editor works similar to well-known vector editors except that the grouping of symbols is handled using graphical constraints to connect the primitives in a symbol graphic. Available primitives are e.g. lines, poly-lines, rectangles, ellipses, images (GIF/JPEG) and text. The primitives' properties like color, line width or text properties can be edited, too. Attribute symbols appear as independent graphical objects in the Symbol Editor.

The Connection Editor supports the definition of links between symbols. In order to define a link, the user can select any two symbols as source and target of the link in the abstract syntax (e.g. *Place* and *ArcPT*). A constraint dialog supports the definition of link constraints in the concrete syntax.

The definition of a visual syntax grammar is supported by the Grammar Editor available in the GENGED environment. The Grammar Editor gets an alphabet as input. From this input, so-called *alphabet rules* are generated defining the editing commands of the Grammar Editor. The set of alphabet rules comprises rules for the insertion and deletion of symbols. In the snapshot of the Grammar Editor shown in Fig. 17 one can see the alphabet rule for the insertion of a transition name in the upper part.

The lower part of the Grammar Editor denotes the *working areas*: here we build the start diagram, and the LHS and RHS (or LHS and NACs, respectively) of a VL-rule, add mappings between the two rule sides and edit the rule parameters. Applying a rule with rule parameter to a diagram in one of the working

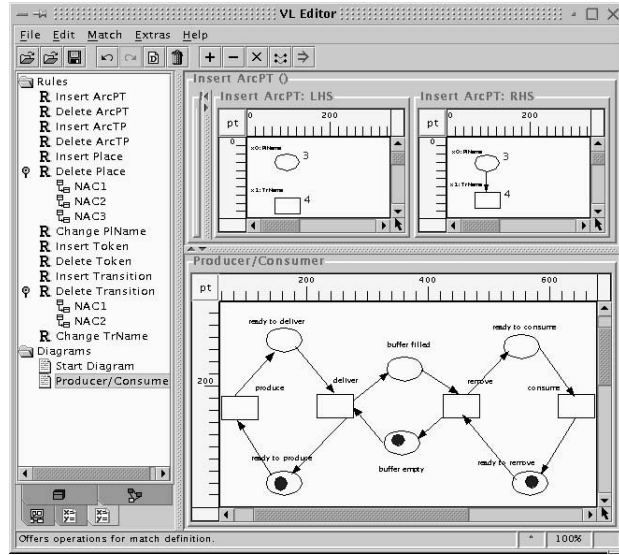


Fig. 18. Graphic Editor for Petri nets

areas, the user is first asked to define the match morphism, i.e., to map the symbols of the rule's LHS to type-consistent symbols in the diagram. Then, the user has to give a value (or a variable) for the parameter such that the expressions in the RHS can be evaluated during transformation.

The next step is to export the set of VL-rules and the start diagram into a visual language grammar². Then, the Graphic Editor uses the grammar rules to provide the language-specific editing commands. Fig. 18 shows the generated Graphic Editor where our Producer/Consumer Petri net is drawn in the edit panel using the visual grammar rules of the syntax grammar. The syntax grammar rules can be selected in the tree view at the left-hand side in order to edit symbols.

5.2 Behavior and Animation

It is possible to generate behavior rules for arbitrary Petri nets automatically according to the general definition of firing transitions in nets of the specific Petri net class. (For the class of AHL nets see [5]). The algorithm for generating behavior rules for Elementary nets is sketched in Def. 4. It will be used for the implementation of our simulation and animation concepts in the GENGED environment. Then, for the well-known Petri net types, it is not necessary for the user to specify the behavior for each specific net. Instead, the behavior rules for the transitions of a net can be generated by the tool.

² The alphabet is added automatically, so in fact we export a VL-specification.

Definition 4 (Translation of Elementary Nets to Graph Grammars).

Each transition $t \in T$ is translated to an attributed graph grammar rule $r_t : L_t \rightarrow R_t$. The attributed graphs in the LHS L_t and the RHS R_t of such a rule both contain nodes for all places in the pre and post domain of t . In L_t , the places in the post domain are not marked. The marking of the pre domain places p_i is computed as follows:

for each arc $a : p_i \rightarrow t$
 generate a **Token** vertex;
 connect the **Token** vertex by an edge of type **tok** to place p_i ;

Analogously, in R_t , only places in the post domain p_j become marked:

for each arc $a : t \rightarrow p_j$
 generate a **Token** vertex;
 connect the **Token** vertex by an edge of type **tok** to place p_j ;

Moreover, an NAC is added to the rule $r_t : L_t \rightarrow R_t$ containing the places from the post domain $post(t)$ marked by one token each. This NAC ensures that the rule is applied only if the places in the post domain of the transition are unmarked. $\triangle$

As the automatic generation of behavior rules is not yet implemented in the GENGED environment, we define them in the same way as the syntax rules: For the definition of Petri net behavior, again the Grammar Editor is started. After editing the behavior rules the Petri net simulator (similar to the Graphic Editor but with behavior rules instead of syntax rules) is generated. Fig. 19 illustrates the simulator for our Producer/Consumer net. Applying a rule this time means to simulate the firing of a transition. In the screenshot, rule **produce** is selected and the match is indicated by equal numbers for corresponding objects. The application of the rule removes the token from the place **ready to produce** and adds a new token to the place **ready to deliver**.

In the corresponding animation grammar the animation rules have the layout of the animation view. The implementation of support for defining the view transformation grammar is still work in progress. Here, we need a user interface allowing the animation view designer to define the view transformation rules. These rules are to be applied suitably (in a user-controlled order) to the LHSs and RHSs of each behavior rule. Thus, the view transformation interface has to support the application of rules to rules in order to generate animation rules from behavior rules. Moreover, we need to define rule constraints to ensure specific layouts for symbol connections which are valid only if a specific rule is applied. For instance, in the Producer/Consumer animation view, the cake symbol is placed on top of the table symbol when applying rule **deliver**, but it is placed on top of the stove symbol when applying rule **produce**. Fig. 20 illustrates the desired animation view interface.

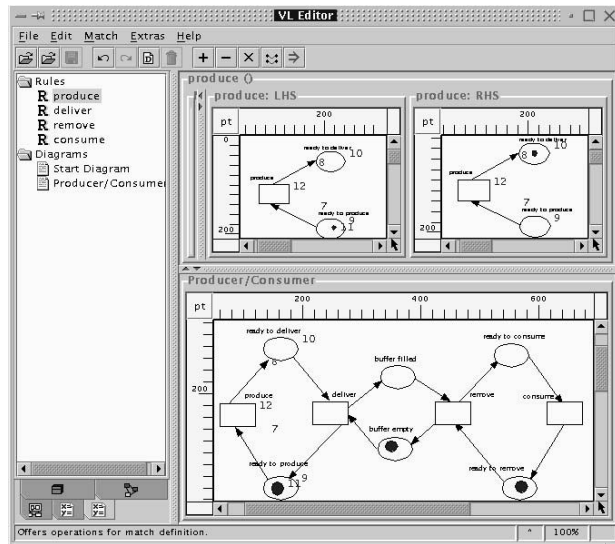


Fig. 19. Simulator for the Producer/Consumer Net

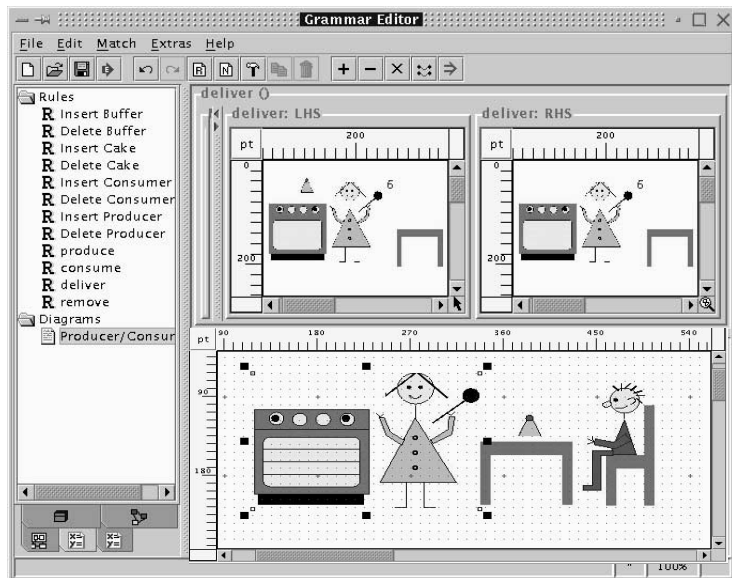


Fig. 20. Animation View for the Producer/Consumer Net

6 Conclusion

We have presented the GENGED approach supporting the generic description of visual languages and provided concepts for an extension of GENGED in order to allow the description and implementation of application-specific animation views for communication-based systems modeled as Petri nets. An animation of the Petri net behavior then shows directly the state transitions in the layout of the application domain. An animation view is realized by graphical icons corresponding to parts of the system. During animation, some icons are changed according to the selected state transition. GENGED is based on graph transformation. This provides a natural formal basis to express Petri nets and their animation views as graphs, whereas the behavior of the model (i.e. transition firing steps resp. state transitions in the animation view) can be formalized as graph grammars.

A specification of a specific Petri net type has to be provided once as visual language specification (graphical symbols and syntax rules) and allows the generation of a graphical editor for arbitrary nets of the specified Petri net type. The specifications of different Petri net types all have the same basis (places, transitions, arcs and tokens as symbols) and only differ in the representation of tokens (black dots or strings) and in their firing rule.

The behavior grammar for a specific Petri net can be generated automatically according to the firing rule of the corresponding net type: Each transition is converted to a graph grammar rule whose left/right-hand side corresponds to the transition's pre/post domain.

The designer of an animation view therefore only has to specify the correspondence of tokens to icons from the animation view representing the application domain. This relation of a Petri net and its animation view is defined as view transformation graph grammar and allows to map the behavior of the Petri net consistently to the animation view. Due to the generic and modular definition of syntax, behavior and animation for different Petri net types, the presented framework reduces considerably the amount of work to realize an application-domain animation of a system modeled as a specific Petri net.

It remains to develop a formal theory to handle behavior and animation of visual process models in general (i.e. covering other Petri net classes and modeling techniques like statecharts or message sequence charts). The theory should include formal definitions for an automatic generation of behavior rules for well-known Petri net classes and a formal transformation of the states and behavior rules of a general visual process model into its animation view. Implementational work is still in progress, i.e., concerning the user interface for defining view transformation rules, rule constraints and the mapping of rules to rules.

Within the Petri Net Baukasten [11] the proposed animation framework will be an extension of the functionality provided by the Petri net tool infrastructure PNK and by the external tools integrated over the PNK. In order to offer the features of the extended GENGED tool environment to PNK users, an XML conversion between the XML file interchange formats of the PNK and GENGED has been implemented [13, 12]. Thus, it becomes possible on the one hand to

use the editing, simulation and analysis features provided by the PNK and on the other hand to have a visual environment for the definition of domain-specific animation views for Petri nets provided by GENGED.

References

- [1] R. Bardohl. *GENGED – Visual Definition of Visual Languages based on Algebraic Graph Transformation*. Verlag Dr. Kovac, 2000. PhD thesis, Technical University of Berlin, Dept. of Computer Science, 1999. 84, 87
- [2] R. Bardohl, H. Ehrig, and C. Ermel. Generic Description, Behaviour and Animation of Visual Modeling Languages. In *Proc. Integrated Design and Process Technology (IDPT 2000)*, Dallas (Texas), USA, June 2000. 84, 95
- [3] R. Bardohl and C. Ermel. Visual Specification and Parsing of a Statechart Variant using GENGED. In *Proc. Symposium on Visual Languages and Formal Methods (VLFM'01)*, Stresa, Italy, September 5–7 2001. 84
- [4] R. Bardohl, C. Ermel, and H. Ehrig. Generic Description of Syntax, Behavior and Animation of Visual Models. TR 2001/19, TU Berlin, 2001. ISSN 1436-9915. 84, 91, 95
- [5] R. Bardohl, C. Ermel, and J. Padberg. Formal Relationship between Petri Nets and Graph Grammars as Basis for Animation Views in GenGED. In *Proc. IDPT 2002: Sixth World Conference on Integrated Design and Process Technology*, 2002. To appear. 85, 103
- [6] R. Bardohl, C. Ermel, and L. Ribeiro. Towards Visual Specification and Animation of Petri Net Based Models. In *Proc. GRATRA 2000 - Joint APPLI-GRAPH and GETGRATS Workshop on Graph Transformation Systems*, pages 22–31. Technische Universität Berlin, March 2000. 84
- [7] R. Bardohl, M. Niemann, and M. Schwarze. GENGED – A Development Environment for Visual Languages. In *Int. Workshop on Applications of Graph Transformations with Industrial Relevance (ACTIVE'99)*, LNCS 1779, pages 233–240. Springer, 2000. 84
- [8] R. Bardohl, T. Schultze, and G. Taentzer. Visual Language Parsing in GENGED. *Electronic Notes of Theoretical Computer Science*, Vol. 50, No. 3, June 12–13 2001. 84
- [9] A. Corradini and U. Montanari. Specification of Concurrent Systems: From Petri Nets to Graph Grammars. In G. Hommel, editor, *Proc. Workshop on Quality of Communication-Based Systems, Berlin, Germany*. Kluwer, 1995. 85
- [10] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic Approaches to Graph Transformation II: Single Pushout Approach and Comparison with Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*, chapter 4, pages 247–312. World Scientific, 1997. 89
- [11] H. Ehrig, W. Reisig, and H. Weber et al. The Petri Net Baukasten of the DFG-Forschergruppe PETRI NET TECHNOLOGY. In this Volume. 106
- [12] K. Ehrig. Converting XML Files with XSLT and XPath, <http://tfs.cs.tu-berlin.de/lehre/SS01/gragra.html>, 2001. Student's Project Status Report. 106
- [13] C. Ermel, R. Bardohl, and H. Ehrig. Specification and Implementation of Animation Views for Petri Nets. In Weber et al., editors. *2nd Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, Berlin, Germany, Sept. 2001. Fraunhofer Gesellschaft ISST, pages 75–92. 84, 106

- [14] C. Ermel, R. Bardohl, and J. Padberg. Visual Design of Software Architecture and Evolution based on Graph Transformation. In *Int. Workshop on Uniform Approaches to Graphical Process Specification Techniques (UNIGRA'01)*, at ETAPS'01, 2001. Electronic Notes in Theoretical Computer Science, Vol. 44, No. 4. 91
- [15] C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999. 84, 101
- [16] P. Griebel. *Paralleles Lösen von grafischen Constraints*. PhD thesis, University of Paderborn, Germany, February 1996. 84, 101
- [17] M. Korff and L. Ribeiro. Formal Relationship between Graph Grammars and Petri nets. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, pages 288 – 303. Springer, 1995. 85
- [18] H.-J. Kreowski. A Comparison between Petri-nets and Graph Grammars. In *LNCS 100*, pages 1–19. Springer Verlag, 1981. 94
- [19] M. Löwe. Algebraic Approach to Single-Pushout Graph Transformation. *TCS*, 109:181–224, 1993. 87, 89
- [20] F. Parisi-Presicce, H. Ehrig, and U. Montanari. Graph Rewriting with Unification and Composition. In *3rd Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 291*, Berlin, 1987. Springer Verlag. 95
- [21] H.J. Schneider. Graph Grammars as a Tool to Define the Behaviour of Process Systems: From Petri Nets to Linda. In *Proc. Fifth International Workshop on Graph Grammars and their Application to Computer Science*, pages 7–12, Williamsburg, Va., USA, 1994. 94
- [22] C. Weidauer. Animations-Framework in Java. Systematische Animationsentwicklung mit Mehrschichtenarchitektur. *Informatik - Forschung und Entwicklung, Band 15, Heft 2*, pages 83 –91, June 2000. 100

The Petri Net Kernel

Michael Weber^{1,*} and Ekkart Kindler²

¹ Humboldt-Universität zu Berlin, Institut für Informatik
`mweber@informatik.hu-berlin.de`

² Technische Universität München, Fakultät für Informatik
`kindler@informatik.hu-berlin.de`

Abstract. The *Petri Net Kernel (PNK)* is an infrastructure for building Petri net tools. It relieves the programmer of a Petri net tool of implementing standard functionality on Petri nets. Moreover, it allows users to customize and to extend a PNK based tool according to their needs. In this paper, we discuss the goals, the concepts, and the realization of the Petri Net Kernel.

1 Introduction

Implementing a Petri net tool is a time consuming and expensive venture. In most cases, it is much more expensive and takes much longer than planned. Most of the implementation effort for different Petri net tools, however, was spent for almost the same functionality: a graphical editor, a visualization device, parse functions, etc. The effort for implementing this ‘standard functionality’ is spent over and over again when developing new Petri net tools. It is even spent again, when revising a tool after some time of successful use. Is this waste of manpower really necessary? Wouldn’t it be nice if this standard functionality could be shared among different tools? And wouldn’t it be nice if you could easily include a nice analysis function from another tool in your favorite tool? Wouldn’t it be nice if other Petri net tools with a less sophisticated graphical user interface could simply use the new ergonomic graphical user interface of the new CPN/Tools [5]? Wouldn’t it be nice if we could easily customize or extend an existing tool according to our specific needs. We are sure that it would!

The problem, however, is that in most tools the graphical user interface and the Petri net functionality itself are not clearly separated. Therefore, it is quite hard – or even impossible – to share standard functionality among different tools. Sharing and exchanging functionality among different tools requires a well-defined interface for *modules* implementing such functionality, and it requires a clear definition of how these modules interact with the Petri net and with each other. Moreover, it requires an infrastructure¹ for building a tool from modules implementing such functionality.

* Supported by the Deutsche Forschungsgemeinschaft within the project ‘Petri Net Technology’.

¹ For now, an infrastructure can be considered to be a framework. But, actually by using the term infrastructure rather than the term framework, we stress some differences, which will be discussed later in this paper.

One major problem in developing such an infrastructure is the variety of different Petri net types. Almost every Petri net tool supports a slightly different version of Petri nets, where the differences are both, in syntax and in semantics. In order not to exclude any Petri net type, there must be a way to freely define new Petri net types and to use the infrastructure with different net types without knowing them beforehand.

In 1994, we started dreaming on such an infrastructure[10]. Later on, we started working on an infrastructure for building Petri net tools: the *Petri Net Kernel* [11]. The target was the definition of a clear interface for the different types of functionalities of a Petri net tool such that it would be easy to build a Petri net tool by simply combining some predefined modules. Moreover, it should be easy to extend the functionality of an existing Petri net tool by simply adding a new module. In order to validate and to improve the definition of the interface, we implemented a prototype, which we called Petri Net Kernel, too. So, the term Petri Net Kernel stands for the concepts of the infrastructure as well as for its implementation. Where necessary, we will refer to the implementation by the acronym *PNK*.

The Petri Net Kernel was developed within the German research project ‘Petri Net Technology’. The purpose of this project was to make the various techniques from Petri net theory applicable for systems and software development. The essence of this project is captured in the so-called ‘Petri Net Baukasten’, which is discussed in another contribution to this volume [16]. The implementation of the Petri Net Kernel is the materialization of the ‘Petri Net Baukasten’ in terms of software. On the one hand, the ‘Petri Net Baukasten’ provides the formal underpinning of the Petri Net Kernel; on the other hand, the Petri Net Kernel is the validation of the concepts of the ‘Petri Net Baukasten’.

In this paper, we discuss the goals, the design principles, and the concepts of the Petri Net Kernel. Moreover, we discuss some aspects of its implementation and its use for building Petri net tools. For full details, however, we refer to the reference manuals of the PNK [14].

2 Goals

As mentioned above, the main target of the Petri Net Kernel was to build an infrastructure for building Petri net tools. But, this is a quite general and vague goal, which is pursued by many other projects from quite different research areas. Therefore, we give a brief overview on a variety of related projects and discuss the relation to our goals:

Software Engineering. There are different tools and techniques from software engineering that allow us to build software from components and to share and to re-use components among different projects. *Frameworks*, *component based software*, *meta computing*, and *interoperability*, are only a few keywords. These techniques apply to all kinds of software and are not restricted to Petri net tools.

From a software engineering point of view, the Petri Net Kernel can be considered as a framework for building Petri net tools. But, the Petri Net Kernel has a different target group. Whereas frameworks are designed to be used by software engineers, the Petri Net Kernel is designed to be used by Petri net experts, who are not (necessarily) experts in software engineering. Therefore, the Petri Net Kernel should be usable without too much knowledge of software engineering techniques. In the best case, it should be possible to build a Petri net tool without doing any programming at all by selecting functionality from a list of predefined modules. Therefore, the Petri Net Kernel must provide some way to relieve the Petri net expert from programming (as much as possible). Due to this additional requirements, we call the Petri Net Kernel an *infrastructure* rather than a framework.

Translation between Verification Tools. Currently, there are many different tools that support different kinds of formal methods for the automated verification of systems (or at least of parts of a system). *VeriTech* [9] is an ambitious project, which aims at exchanging system models and analysis results among possibly all existing tools. The ETI project pursues a similar goal [15]. ETI is a platform for the evaluation of formal methods or the tools available for formal methods. Currently, the focus of ETI is on model checking tools. This platform should help non-experts in formal methods to choose the most appropriate tool for their specific needs. A similar, but less ambitious project, is the Model-Checking Kit [13], which concentrates on model checking tools, too.

On the one hand, the Petri Net Kernel is much less ambitious than the VeriTech project, because it is restricted to Petri net tools. Moreover, there is no intention to integrate all available tools². Rather, it provides an infrastructure for those who want to join. On the other hand, the Petri Net Kernel is more ambitious than VeriTech in another aspect: It provides a much tighter coupling of components into a single tool with one graphical user interface.

Integration. The extreme counter part to the VeriTech and the ETI projects are tools that integrate as much functionality as possible within a single tool. PEP [7] and Design/CPN [5] are quite successful examples for such Petri net tools.

The Petri Net Kernel does not compete with these tools. If all desired Petri net types and all analysis techniques are available in an existing tool, then we should better use this tool. The Petri Net Kernel comes in, when a Petri net type is not supported by an existing tool or when a new analysis technique is not available. Moreover, experiments and experience with the Petri Net Kernel could help to design an architecture for integrate functionality into future versions of monolithic tools.

Unification of Petri Nets. In Petri net theory, there are different approaches in unifying Petri nets (see [6] for a collection of articles devoted to this subject). These approaches try to capture the essence of Petri nets in some

² The *Petri Net Markup Language*, which is discussed in another contribution to this volume [17] tackles the problem of exchanging Petri nets between different tools.

kind of mathematics without fixing a particular version of Petri net. This way, they give a mathematical answer to the question³: What is a Petri net? The Petri Net Kernel answers this question too. But, it does not give a mathematical answer; it gives a software technological answer by interfaces for accessing Petri nets and for defining Petri net types.

Altogether, the goal of the Petri Net Kernel was to design and to validate an infrastructure for building Petri net tools. The APNN-toolbox [1] and CPN-AMI [3] are projects which come quite close to this target. CPN-AMI is a Petri net based CASE environment which supports the integration of different Petri net tools. The underlying FrameKit supports the definition of new formalisms; the definition of these new formalisms, however, is not restricted and tuned to Petri nets. The APNN-toolbox is an open toolset based on the interchange format APNN [2]. Its focus is on efficient analysis methods exploring the complete state space.

The focus of the Petri Net Kernel is on the following issues:

- On the one hand, it should be easily usable for people who are not experts in software engineering. In particular, it should be possible to build a tool from standard modules or from application modules without doing any programming. In a way, the infrastructure should provide a ‘plug and play’ feature for application modules for Petri nets.
- On the other hand, the infrastructure should provide enough flexibility for implementing completely new functionalities and to define completely new Petri net types. In this case, some programming will be necessary. The way in which new application modules are implemented should reflect the mathematics of Petri nets as much as possible such that it is easy to implement mathematical concepts as an application module for the Petri Net Kernel.
- Moreover, the interfaces should be governed by Petri nets: All objects and functions should have a clear interpretation in the field of Petri nets. So the interfaces should capture the essence of all current and future versions of Petri nets from a software technological point of view.

3 Principles

In the previous section, we have discussed the overall goal of the Petri Net Kernel. Before presenting the design and the implementation of the Petri Net Kernel, let us briefly discuss the principles that we had in the back of our minds when designing the Petri Net Kernel.

Simplicity. Above all, it should be easy to use the Petri Net Kernel for building a Petri Net Tool. In particular, the Petri Net Kernel should be usable

³ We borrow this question from the title of a paper of Desel and Juhás [4] who give an informal resp. philosophical answer to this question.

without much programming experience – at least in cases when it is used for tailoring a new tool from existing modules. Beyond that, implementing new application modules or new Petri net types should in no case be more difficult than without using the Petri Net Kernel; there should be a straightforward transformation from the conceptual design of a new functionality or a new Petri net type to a Petri Net Kernel implementation.

Extensibility. There are two different aspects of extensibility. One aspect is covered already by the goals: the possibility of defining new Petri net types. But, there is another aspect of extensibility: The implementation of the Petri Net Kernel provides standard modules for editing Petri Nets and for visualizing analysis results. Since the functionality of these modules is quite restricted, it should be easily possible to extend it or to completely exchange it by a more sophisticated editor.

Modularity and Re-usability. Again, one aspect of modularity and re-usability is covered already by the goal of the Petri Net Kernel. Beyond that, the interfaces for defining new Petri net types and for defining new application modules should support the re-use of parts of already existing application modules and the re-use of parts of already existing Petri net types. In many cases, the definition of a new type should be possible by choosing from a list of predefined features.

Separation of Concerns. The interfaces of the Petri Net Kernel should provide a clear separation between a net's syntax, its semantics and its graphical representation. In particular, it should be possible to completely ignore graphical aspects of the Petri net in an application module. On the other hand, application modules should be able to change the graphical appearance of a net.

Consistency. The Petri Net Kernel should take care that the maintained net is always consistent. Actually, there are several levels of consistency, and it should be up to the tool builder to decide which level of consistency should be maintained. The lowest level, however, should always be maintained: the graph structure of the net should always be consistent, e. g. there should be no dangling arcs.

4 Concepts

In this section, we discuss the concepts of the Petri Net Kernel from a users point of view. Note that there are two kinds of users: the *PNK user*, who uses the PNK for building a Petri net tool, and the *end user*, who uses a tool built by the PNK user. The following discussion is addressed to the PNK user only. Information for the end user, e. g. on the use of the PNK editor and the use of the graphical user interface can be found in the PNK manual [14].

The main concepts of the Petri Net Kernel are Petri nets, Petri net types, application modules, and the built tool itself.

4.1 Petri Nets and Petri Net Types

As mentioned earlier, the Petri Net Kernel supports any version of Petri nets. This requires a precise definition of Petri nets. This definition must be general enough to capture all current and future versions of Petri nets, on the one hand. But, it must also rule out formalisms that are clearly not Petri nets. This is a subtle issue because there are formalisms for which the question whether they are Petri nets cannot clearly be answered. For example, what about a formalisms that allows us to have arcs from transitions to transitions⁴?

In order to be as general as possible, we consider each formalism that consists of a set of *places*, a set of *transitions*, and a set of *arcs* as a Petri net. We call the places and transitions the *nodes* of the Petri net. In general, an arc may run between any two nodes of the net (even from a place to a place or from a transition to a transition). The nodes and the arcs of the net are called the *elements* of the net.

More information on a net element can be attached to it by so-called *labels*. For example, the marking of a Petri net can be represented by a corresponding label for each places. In addition to the marking, there can be other labels. For example, there could be a label for the capacity of a place, there could be a label for the firing time of a transition, or there could be a label for an arc inscription. A priori, we do neither restrict the number of labels for each element, nor do we restrict their possible values. It is up to a particular *Petri net type*, to define the legal labels for the net's elements, their possible values, and the legal combination of values in a concrete Petri net. So, all information beyond the net structure is in the labels. Note that the net in a whole may have labels, too. For example, there could be labels for global declarations, the net's name, etc.

The concept of Petri net types introduced so far is quite general. It is the concept underlying also the Petri Net Markup Language (see [17] in this volume). But, the format is purely syntactical: It basically says which labels and which combination of values are allowed for a particular version of Petri net. It does not say anything about the semantics. In order to provide some semantics for a Petri net type, the elements and labels of the Petri net type can be equipped with methods. For example, the labels corresponding to a marking are equipped with plus and minus operations. From these operations, the Petri Net Kernel can infer a firing rule, or the Petri net type comes with its own definition of a firing rule.

In order to allow the Petri Net Kernel to maintain a Petri net of a particular type, the Petri Net Kernel provides a *net type interface* for defining new Petri net types. Basically, there must be a class for each legal label, which implements a certain interface. Then, a Petri net type consists of a collection of such classes defining the labels for all its elements.

⁴ On a first glance, most people from the Petri net community would rule out such formalisms. But, there are formalisms called Signal/Event systems [12] that, in spirit, are very much alike Petri nets, but have arcs from transitions to transitions.

4.2 Application Modules and I/O-Modules

The Petri net type interface allows us to define different versions of Petri nets. Once a net type is defined, the Petri Net Kernel is able to maintain a Petri net of this type. An application can access the net maintained by the PNK. To this end, the PNK provides an interface that allows an application to create, to delete and to update the net structure and its labels: the *net interface*.

An *application module* may use this interface for accessing the net structure, for doing some analysis on the net, and for visualizing the analysis results. The *application module* may even change the net if this is what the application should do. Basically, a Petri net tool consists of a collection of such application modules. Even the editor or a visualization module are (special) application modules. This makes it easy to exchange the editor of a tool or to equip the tool with a fancy animation of the net behaviour. If we like, we can build a tool with several editors working on the same net all of which will be kept consistent by the PNK.

Likewise, we can easily exchange the modules for loading and saving a net: the so-called *I/O-modules*. This way, we can easily equip a tool with new file formats. For example, we can easily implement a converter between different file formats by adding one I/O-module for each file format.

Of course, the editor and the I/O-modules need not be implemented by the PNK user himself. There is a standard editor, and there are I/O-modules for different file formats.

4.3 Petri Net Tool

As mentioned above, a Petri net tool consists of a collection of application modules that work on one or more Petri nets and are maintained by the Petri Net Kernel. The application modules of a Petri net tool and the different nets are coordinated by the *application control*. The application control knows all active Petri nets along with the associated application modules. In particular, the application control takes care that modifications of one application module are propagated to all other application modules that are registered for the same net. This way, changes made by one application become visible in the editor or to another visualization module.

Moreover, the application control provides a graphical user interface for creating new nets and for starting new application modules on a net. This way, it is easy to integrate new application modules into an existing Petri net tool.

Figure 1(a) shows the rough architecture of a Petri net tool: It consists of some Petri net types, some application modules, and some I/O-modules, which are coordinated by the application control. Each net maintained by the PNK has a specific type, and there are application modules running on that net. Typically, there is one special application module: the editor. The access of the application modules to the net is controlled by the application control. This is shown in Fig. 1(b).

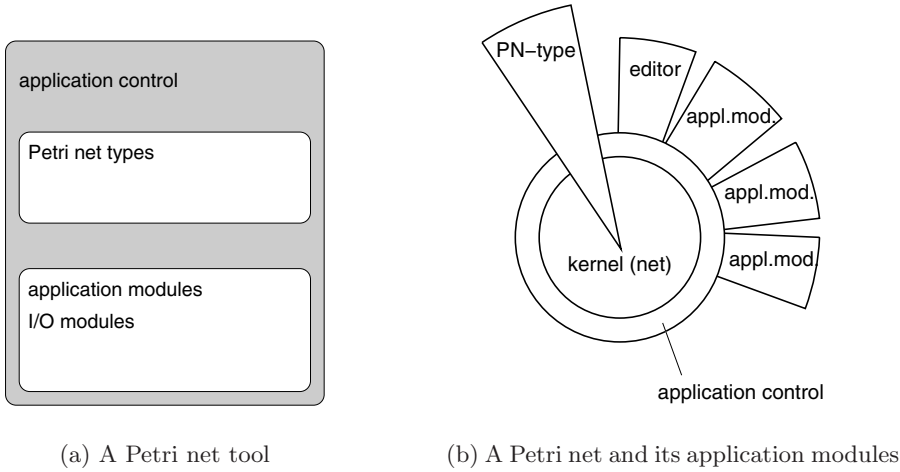


Fig. 1. Architecture of the PNK

5 Realization and Use

In this section, we discuss some more details on the realization of the PNK and on how to use it for building a Petri net tool. Realizing the concepts of the Petri Net Kernel in an object oriented programming language suggests itself. A first version of the PNK was implemented in the programming language Python. In principle, the Petri Net Kernel could be implemented in any object oriented programming language. For the new version of the Petri Net Kernel, we use Java for the following reasons:

- Java is a strongly typed programming language. Therefore, it enforces programming discipline syntactically, which is important for projects with more than one involved programmer.
- Java is a widely used programming language, which is available on almost all platforms and is known to almost all computer scientists.
- Java is a well-designed programming language with a clear semantics; in particular it avoids misleading or possibly ambiguous concepts.
- Java comes with a bunch of standard packages; in particular, it has its own platform independent packages for graphical user interfaces.
- Java has a clear concept for defining interfaces.

The PNK has different interfaces corresponding to the different tasks for building a Petri net tool:

- The *net interface* provides methods to access and to modify a net maintained by the PNK. This interface will be used for programming an application module with its functionality.

- The *net dialog interface* provides methods to visualize information within a net and to interact with the end user. This interface will be used for programming application modules with direct user interaction.
- The *net type interface* says how to define the labels and the firing rule of a Petri net type, and it says how to define a net type.
- The *application interface* says how to define an application module and the special I/O-modules.
- The *tool definition interface* says how to define a tool.

These interfaces will be discussed in some more detail below.

5.1 Net Interface

The *net interface* of the PNK consists of several Java classes that represent a Petri net graph along with its labels. The basic classes are `Net`, `Place`, `Transition`, `Arc`, and `Extension`, where the class `Extension` represents the labels.

Each class provides methods for accessing and for modifying the corresponding net or its elements. For example, for an object `net` of class `Net`, the method call `net.getPlaces()` returns a list of all its places. Likewise, there are methods for returning the in-going arcs of a transition or of a place; or there are methods to access the label of an element. For example, for an object `place` of class `Place`, the method call `place.getExtension("marking")` returns the label that represents the marking of that place. As usual, we can obtain the external representation of a `label` by the method call `label.toString()`. These examples should be enough to provide a flavour of the net interface. Therefore, we do not discuss all methods of the net interface. For a complete list of all access and modification methods, we refer to the reference manual [14].

For all these methods, the PNK will take care of maintaining the net consistent. Note that any change of the net or its elements will be updated in the editor or the visualization applications of the tool. This is achieved by the so-called observer concept, which is in the responsibility of the application control.

5.2 Dialog Interface

In order to present analysis results to the user, the PNK provides a *dialog interface*. The dialog interface provides methods for highlighting a set of net elements, for assigning temporary textual information to some net elements that is displayed to the end user, and for interacting with the end user. For example, the dialog interface provides a method that presents a set of net elements to the end user and asks him to select an element from this set; the result of such a method call will be the element chosen by the end user. This way, it is possible to implement interactive application modules.

Note that the dialogs are not implemented within the PNK. Rather, they are implemented in special application modules, which are called *dialog modules* (see Sect. 5.4 for more details). The application control will dispatch a dialog request to an appropriate dialog module. Again, the PNK user need not implement such

dialog modules himself since the standard editor of the PNK is a dialog module. So, by default, dialog requests are dispatched to the PNK editor. But, the PNK user can implement and use his own dialogs, if he wants to.

5.3 Petri Net Types

As mentioned above, a Petri net type is a collection of labels and a specific firing rule. Technically, the definition of a Petri net type consists of three tasks:

- For each label of the Petri net type, we must implement a class that represents the label.
- We must define the firing rule by implementing a certain interface.
- At last, we must define the Petri net type by listing all its labels.

Here, we will discuss only the first and the last task. For the definition of a firing rule, we refer to the reference manual.

Technically, the class defining a label must be derived from class `Extension`, which is already known from the net interface. So, by definition, it has all methods of the net interface. In order to define the specific aspects of this particular label, the class must implement the two methods `localParse()` and `defaultToString()`. The method `localParse()` is used to parse a string and to convert it to an internal representation. The method `defaultToString()` defines the external representation of the default value for this label.

Note that the class cannot directly define the method `toString()` and the method `valueOf()`, because these methods are final methods of class `Extension`. This way, modifications made by calling methods of the net interface are handled by methods of the PNK; thus, the PNK can keep track of all accesses and modifications on a net and can inform all observers of a net, if necessary.

Some labels of net elements have particular meaning for the semantics of a net. The most important example is markings. At the net interface, these methods have additional methods for addition and subtraction of two markings: `add()` and `sub()`. A marking is a special label derived from class `Marking` with these and some more methods for operations on markings. A class derived from class `Marking` defines the meaning of addition and subtraction by implementing two methods⁵ `localAdd()` and `localSub()`.

There are some more of these special labels (e.g. for arc inscriptions). But we will not discuss them here.

The Petri net type itself is specified in an XML file. Basically the file lists all possible labels for each kind of Petri net element. Listing 1 shows an example of a Petri net type specification. It starts with XML specific stuff, which is not important for understanding the concepts. The net type specification starts with the name of the net type. In our example, we use `PTNet`, which stands for

⁵ Again, the methods `add()` and `sub()` cannot be implemented directly, because they are final methods of class `Marking`. Thus, the PNK can keep track of modifications by calling these methods.

Listing 1. An example Petri net type

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE netTypeSpecification SYSTEM "netTypeSpecification.dtd">
<netTypeSpecification name="PTNet">
  <extendable class="pnk.kernel.Place">
5    <extension name="marking"
      class="pnk.netElementExtensions.llNet.NaturalNumber"/>
    <extension name="initialMarking"
      class="pnk.netElementExtensions.llNet.NaturalNumber"/>
  </extendable>
10 <extendable class="pnk.kernel.Arc">
    <extension name="inscription"
      class="pnk.netElementExtensions.llNet.NaturalNumber1"/>
  </extendable>
</netTypeSpecification>

```

Place/Transition-nets. Then, there is a sequence of XML elements `extendable`, which define the labels for each kind of Petri net element. In our example, we define labels for places and arcs. First, we define the labels for places: marking and initial marking. For each label, we define its name and the class implementing it within the XML element `extension`. In our example, we use the same built-in class `NaturalNumber` for implementing the marking and for implementing the initial marking. As expected, this class implements the natural numbers with the usual addition and subtraction operations and with 0 as the default value. For arcs, we define one label only: the arc inscription. The implementation class for the arc inscription is the built-in class `NaturalNumber1`, which is identical to the class `NaturalNumber`, except for the default value, which is 1.

Note that the XML file in Listing 1 completely defines a Petri net type. It is not necessary to mention transitions because they do not have labels in Place/Transition-nets. In the net type definition, we must mention only those elements with additional labels.

Note that the elements of the XML file defining the labels refers to the implementation of the Petri net element. In our example, we refer to the standard implementation of places (`pnk.kernel.Place`) and to the standard implementation of arcs (`pnk.kernel.Arc`). But, we could implement our own classes for Petri net elements, too. We could have even two different implementations for transitions. This way, it is possible to define more exotic Petri net types. For example, we could implement signal arcs between transitions. For more conventional Petri net types, however, the implementation of new Petri net elements should not be necessary. Therefore, we do not discuss the details on implementing own classes for Petri net elements here.

5.4 Application Interface

Basically, an application module consists of a method or a collection of related methods that implement some functionality on Petri nets by using the net interface and the dialog interface. Here, we will discuss how to make an application module, resp. its methods, available to the PNK for building a tool.

Technically, an application module is a class derived from the PNK class `MetaApplication`. In the simple case, an application module implements one single method `run()`. This method must implement the desired functionality, where the net corresponding to the application can be accessed by the method `getNet()`. In a more complex application, there can be arbitrarily many methods. In that case, however, the application must implement a method `getMenus()` which provides the PNK with the necessary information on the available methods, such that the PNK can provide corresponding menus for the end user and start a method on user request.

As mentioned before, the PNK user can implement some more special application modules: *net observers* and *dialog applications*. An observer must implement the interface `NetObserver`. This interface requires methods for informing the observer on changes of the net. This way, the observer is able to keep track of all modifications. A dialog application must implement the interface `ApplicationNetDialog`, which requires some dialog methods. For example, there must be a method `selectObject()` that presents a list of objects to the end user for selection; upon selection it must return the selected object.

Note that an application can be both, a net observer and a dialog application (technically, it is a class implementing both interfaces). An example for such an application is the standard editor of the PNK.

Similar to applications, I/O-modules are classes derived from the class `InOut`. They must implement the two methods `load()` and `save()`, which load resp. save a net to or from a given URL. This way, it is easy to define one's own file format or to export nets in a different file format. For example, the standard file format of the PNK is PNML, which works for any Petri net type. This is implemented in the class `PnmlInOut`.

5.5 Tool Definition

When we have all the Petri net types, all the application modules, and all the I/O-modules, it remains to define the tool. The tool is defined by listing all Petri net types and all its methods in an XML file: the *tool definition*.

Listing 2 shows an example of the definition of a simple tool. The tool is an editor for Place/Transition-nets which reads and writes both, the file format of a previous version of the PNK and the Petri Net Markup Language (PNML) [8]. The file starts with XML specific stuff. Then, it lists the Petri net types used by this tool. In our example, we use the Petri net type defined in Sect. 5.3. In general, a tool can use arbitrarily many Petri net types. After the net types,

Listing 2. An example tool

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE toolSpecification SYSTEM "toolSpecification.dtd">
<toolSpecification>
  <!-- net types -->
5  <nettype id="n1" typeSpecification="file:PTNet.xml"/>
  <!-- application modules -->
  <application id="a1" mainClass="pnk.editor.Editor" maxinstances="inf">
    <allowedNettypes>
      <ntref ref="n1"/>
10  </allowedNettypes>
  </application>
  <!-- Input/Output -->
  <format id="pnml" ioClass="pnk.appControl.PnmlInOut">
    <allowedNettypes>
15    <ntref ref="n1"/>
    </allowedNettypes>
  </format>
  <format id="net" ioClass="pnk.appControl.NetInOut">
    <allowedNettypes>
20    <ntref ref="n1"/>
    </allowedNettypes>
  </format>
  <!-- default settings -->
  <standardNettype ref="n1"/>
25 <standardApplication ref="a1"/>
  <standardFormat ref="pnml"/>
</toolSpecification>

```

there is a list of application modules of the tool. In our example, there is only one application module – the editor of the PNK which is implemented in a Java class in the given path (`mainClass`). Note that we need not state that the application module is a net observer or a dialog application. This can be inferred from the type of the application module by the Java type system. The XML attribute `maxinstances` specifies the maximal number of instances of this application that can be invoked by the end user. In our example, there is no bound. Moreover, the XML file defines the net types for which an application module works (`allowedNettypes`). In our example, the XML element `ntref` refers to the net type via the net type's identifier. After the application modules, the XML file defines the used file formats by naming the corresponding I/O-modules along with the net types the format can deal with.

Finally, the XML file specifies the standard behaviour of the tool by defining some defaults: a default application module, a default net type, and a default file format.

5.6 Application Control

When the PNK is started with a tool definition, the application control builds the corresponding net types and loads the corresponding application modules and I/O-modules. After this initialization, the application control coordinates the user dialogs and the different application modules. In particular, it will instantiate and start an application module on request by an end user (by pressing the corresponding button). For short, the application control keeps things running in a tool.

Though the application control is the heart of the PNK, there is only little need for application modules to interact with the application control directly (except for dialogs which can be considered as a part of the application control).

In some cases, however, an application module might want to start another application module, it might want to control the interaction, or it might want to lock a net for its exclusive use during some critical operation. The application control provides some methods for these purposes. But, we do not discuss these methods here.

6 Conclusion

In this paper, we have presented the main concepts of the Petri Net Kernel and the underlying design principles. Recently, we have released the Java version of the Petri Net Kernel, which can be freely obtained from

<http://www.informatik.hu-berlin.de/top/pnk/>.

Up to now, we have implemented only a few widely used Petri net types and we have implemented only a few application modules in order to validate the usability of the Petri Net Kernel. The next steps will be the development of a library of Petri Net Types or its constituents as well as a library of application modules for analysis, simulation, and visualization. Then a tool can be built by simply selecting the needed Petri net types and application modules from this library.

One of the main features of the Petri Net Kernel is the possibility to define arbitrary net types and arbitrary application modules. Therefore, the data types and the object model is not tuned to the specific needs of a particular application or a particular Petri net type. In particular, it is not tuned to the efficiency requirements of a specific application function. When necessary, the application module could define its own internal data structure and translate the net maintained by the Petri Net Kernel to its own format, do the analysis and, in the end, use the Petri Net Kernel for visualizing its results. This way, universality and efficiency can be reconciled. Maybe, future versions of the Petri Net Kernel will provide a collection of some more efficient data structures for Petri nets.

References

- [1] APNN-Toolbox. <http://ls4-www.cs.uni-dortmund.de/APNN-TOLBOX/>. 2002/03/18. 112
- [2] Falko Bause, Peter Kemper, and Pieter Kritzinger. Abstract Petri net notation. *Petri Net Newsletter*, 49:9–27, October 1995. 112
- [3] CPN-AMI. <http://www-src.lip6.fr/logiciels/mars/CPNAMI/>. 2002/03/18. 112
- [4] Jörg Desel and G. Juhás. “What is a Petri net?” – Informal answers to the informed reader. In *Unifying Petri Nets*, volume 2128 of *LNCS, Advances in Petri Nets*, pages 1–25. Springer, 2001. 112
- [5] Design/CPN. <http://www.daimi.au.dk/designCPN/>. 2001/09/21. 109, 111
- [6] H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors. *Unifying Petri Nets*, volume 2128 of *LNCS, Advances in Petri Nets*. Springer, 2001. 111
- [7] Bernd Grahlmann and Eike Best. PEP — more than a Petri net tool. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 1055 of *LNCS*, pages 397–401. Springer-Verlag, March 1996. 111
- [8] Matthias Jünger, Ekkart Kindler, and Michael Weber. The Petri net markup language. In Stephan Philippi, editor, *7. Workshop Algorithmen und Werkzeuge für Petrinetze*, pages 47–52, Universität Koblenz-Landau, Germany, June 2000. <http://www.informatik.hu-berlin.de/top/pnml/>. 120
- [9] Shmuel Katz and Orna Grumberg. VeriTech: Translating among specifications and verification tools. Technical report, The Technion, Haifa, Israel, March 1999. 111
- [10] Ekkart Kindler and Jörg Desel. Der Traum von einem universellen Petrinetz-Werkzeug — Der Petrinetz-Kern. In J. Desel, A. Oberweis, and E. Kindler, editors, *3. Workshop Algorithmen und Werkzeuge für Petrinetze*, number 341 in *Forschungsberichte. Institut AIFB, Universität Karlsruhe*, October 1996. 110
- [11] Ekkart Kindler and Michael Weber. The Petri Net Kernel – An infrastructure for building Petri net tools. *Software Tools for Technology Transfer (STTT)*, 3(4):486–497, 2001. 110
- [12] Arndt Lüder and Hans-Michael Hanisch. A signal extension for Petri nets and its use in controller design. *Fundamenta Informaticae*, 41(4):415–431, 2000. 114
- [13] Model-Checking Kit. <http://wwwbrauer.in.tum.de/gruppen/theorie/KIT/>. 2001/12/19. 111
- [14] Petri Net Kernel Team. *Documentation of the Java Version of the PNK*, August 2001. <http://www.informatik.hu-berlin.de/top/pnk/dokujav.html>. 110, 113, 117
- [15] Bernhard Steffen, Tiziana Margaria, and Volker Braun. The electronic tool integration platform: concepts and design. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1+2):9–30, 1997. <http://www.eti-service.org/>. 111
- [16] Herbert Weber, Hartmut Ehrig, and Wolfgang Reisig. The Petri Net Baukasten of the DFG-Forscherguppe PETRI NET TECHNOLOGY. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication Based Systems*, LNCS, Advances in Petri Nets. Springer, 2002. In this volume. 110

- [17] Michael Weber and Ekkart Kindler. The Petri Net Markup Language. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors, *Petri Net Technology for Communication Based Systems*, LNCS, Advances in Petri Nets. Springer, 2002. In this volume. [111](#), [114](#)

The Petri Net Markup Language

Michael Weber^{1,*} and Ekkart Kindler²

¹ Humboldt-Universität zu Berlin, Institut für Informatik
`mweber@informatik.hu-berlin.de`

² Technische Universität München, Fakultät für Informatik
`kindler@informatik.hu-berlin.de`

Abstract. The *Petri Net Markup Language (PNML)* is an XML-based interchange format for Petri nets. PNML supports any version of Petri net since new Petri net types can be defined by so-called *Petri Net Type Definitions (PNTD)*.

In this paper, we present the syntax and the semantics of PNML as well as the principles underlying its design. Moreover, we present an extension called *modular PNML*, which is a type independent module concept for Petri nets.

1 Introduction

One of the most required features of Petri net tools are functions for exporting Petri nets to other tools and for importing nets from other tools. The problem with this apparently simple and purely technical feature is the multitude of different *Petri net types* and the multitude of different tools and file formats for these different net types. This makes it impossible to provide all desired import and export functions (with reasonable expenses). This situation was the starting point of a standardization effort launched during the International Conference on Application and Theory of Petri Nets 2000 with several proposals for XML-based interchange formats [1].

The *Petri Net Markup Language (PNML)* was one of these proposals, which focussed on the problem of the different Petri net types. Since that time, we have worked out the details and have implemented PNML as the file format for the Petri Net Kernel [5, 10]. In this paper, we discuss the concepts of PNML and present its syntax and semantics.

The design of PNML was governed by the following principles:

Readability. The format should be human readable and editable with a conventional text editor.

Universality. The format should not exclude any version of Petri nets. Rather, it should be possible to represent any version of Petri nets with any kind of extensions.

* Supported by the Deutsche Forschungsgemeinschaft within the project ‘Petri Net Technology’.

Mutuality. The format should allow us to extract as much information as possible from a Petri net – even if the Petri net type is unknown. Therefore, the format must extract the common principles and the common notations of Petri nets.

Clearly, the use of XML guarantees the readability of the format¹. Universality can be guaranteed by attaching all additional information of a particular Petri net type to the *objects* of the net. This is achieved by *labelling* net objects and the net itself. The legal labels, their possible values, and the possible combination of values are defined by a *Petri Net Type Definition (PNTD)*. Mutuality can be guaranteed by *conventions*, which are a set of standardized labels. Technically, the conventions are an extensible collection of possible labels along with a description of their semantics and their typical use. Then, a new PNTD can be built from these labels and, possibly, some new ones.

Another important issue for interchange formats is the size of real world systems. Typically, real world systems are too large to be drawn on a single page – even if the page is unbounded in principle. Therefore, tools have mechanisms for editing large systems. But, each tool provides a different mechanism. To cope with this problem, PNML provides a net type independent mechanism for editing and structuring large nets. Actually, there are two mechanisms, a simple one and a more flexible one:

Pages and References. *Pages* and *references* allow the user to draw a net on different pages and to relate these nets by merging some nodes via so-called *references*. Due to its simplicity and because most tools support similar mechanisms, it should be easy to export and import those nets by any tool. Therefore, this mechanism is part of PNML. The problem with this simple concept, however, is that it does not support abstraction. It basically allows us to draw large nets without any structure.

Modules. In many cases, the use of *modules* is more convenient because the same module can be used several times, once defined. Thus, a system can be built recursively from *module instances*, which reflects the way engineers build systems. Therefore, modules support abstraction much better than the simple page concept. Since the module concept is more evolved, it might not be supported by most tools. But, we provide a semantics in terms of pages and references, which could be run as a preprocessor for those tools not supporting *modular PNML*.

Pages and references² are a widely used concept for drawing large systems. Many of today's tools support this concept with a similar semantics. Maybe, the use of pages has become popular by Design/CPN [4]. Modules, however, are not widely used, and there is no unique semantics for modules. The reason is that, in

¹ To be honest, the true argument for using XML is its popularity, which is a good sales argument.

² References are sometimes called merge nodes.

most cases, the module concept exploits the special features of a particular Petri net type. In particular, this applies to module concepts that support dynamic creation of module instances at runtime such as in object nets of Renew [11] or in higher-order nets of the Moses project [9]. Here, we present a module concept that works for any Petri net type. Therefore, we restrict ourselves to static instances (module instances are created at the buildtime of a system only). This concept is similar to the module concept used for Signal/Event-systems [7]; it is more general, however, because it supports modules with parameters and because it is not restricted to a particular Petri net type.

2 Concepts

The concepts of PNML are independent from its syntactic representation. In particular, it is independent from XML. Therefore, we discuss its concepts and its terminology, first. Its XML syntax will be discussed in Sect. 3.

Remember that universality is one of the main principles of PNML. Therefore, PNML must be sufficiently general to represent all versions of Petri nets, on the one hand. On the other hand, mutuality requires to capture the essence of Petri nets and to exclude all kinds of nonsense, which does not represent a Petri net at all. This is achieved by providing a general format, which is restricted to the specific needs of a particular version of Petri net by defining a Petri net type.

2.1 General Format

We start with the general format. Basically, the general format of PNML is a labelled graph with two kinds of nodes: places and transitions. But, there are many more concepts, which will be explained in the following paragraphs. Figure 1 gives an overview on all concepts, which can serve as a road-map while reading the following paragraphs.

Petri Nets and Objects. A file that meets the requirements of the interchange format is called a *Petri net file*; it may contain several *Petri nets*. Each Petri net consists of *objects*, where the objects, basically, represent the graph structure of the Petri net. Thus, an object is a *place*, a *transition*, or an *arc*. For structuring a Petri net, there are three other kinds of objects, which will be explained later in this section: *pages*, *reference places*, and *reference transitions*. Each object within a Petri net file has a unique *identifier*, which can be used to refer to this object.

For convenience, we call a place, a transition, a reference place, or a reference transition a *node*, and we call a reference place or a reference transition a *reference node*.

Labels. In order to assign further meaning to an object, each object may have some *labels*. Typically, a label represents the name of a node, the initial marking of a place, the guard of a transition, or the inscription of an arc. In addition, the

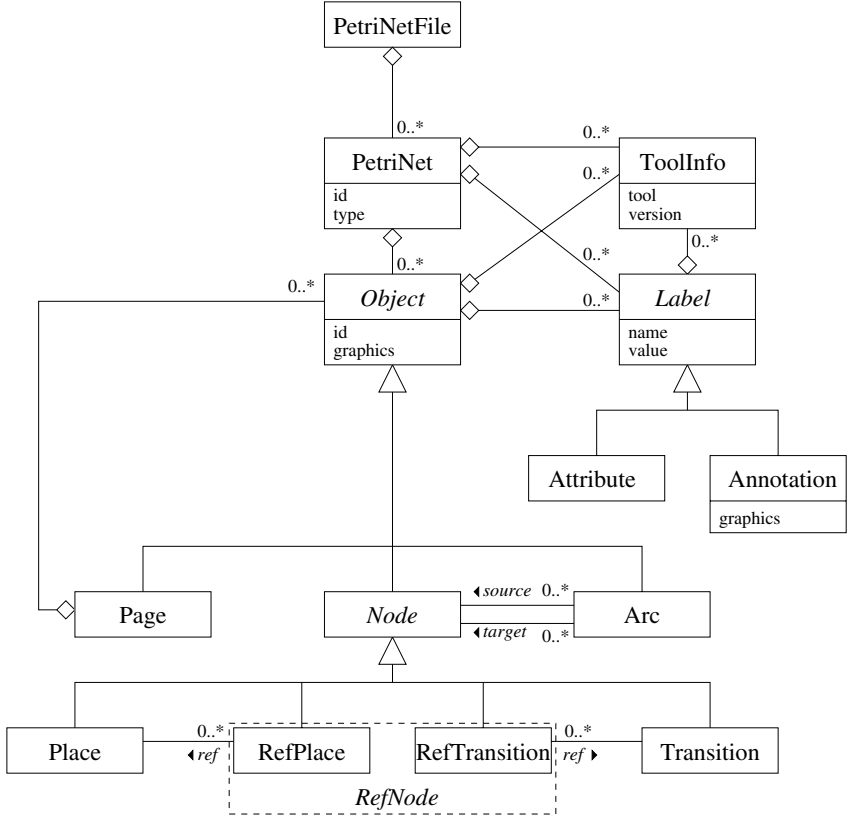


Fig. 1. Pure PNML: An overview

Petri net itself may have some labels. For example, the declarations of functions and variables that are used in the arc inscriptions could be labels of a Petri net. The legal labels and the legal combinations of labels are defined by the *type* of the Petri net, which will be discussed in Sect. 2.3.

We distinguish between two kinds of labels – *annotations* and *attributes*. An annotation is a label with an infinite domain of legal values. Typically, a label will be displayed as text near the corresponding object. For example, names, initial markings, arc inscriptions, and transition guards are annotations. An attribute is a label with a finite (and small) domain of legal values. Typically, the value of an attribute is not display textually, but is represented in the form, style, or colour of the object itself. For example, the arc type could be an attribute of an arc with domain: **normal**, **read**, **inhibitor**, **reset** (and maybe some more). The shape of the arc will depend of this attribute. Another example is an attribute for classifying the nodes of a net as proposed by Mailund and Mortensen [8]. So, the basic difference between an annotation and an attribute of an object is that an annotation is displayed as a separate text near the object, whereas an

attribute has impact on the shape of the object itself. Therefore, an annotation needs some information on the (relative) position of the displayed text, whereas attributes do not need this information. Note, however, that PNML does not define the impact of an attribute on the shape of the corresponding object. This is left to the implementation of the tool³.

Graphical Information. Each object and each annotation is equipped with some graphical information. For a node, this information is its position; for an arc, it is a list of positions that define intermediate points of the arc. For an annotation, the graphical information is its relative position with respect to the corresponding object⁴. Absolute as well as relative positions refer to the *reference point* of an object or of an annotation respectively. By default, the reference point is the middle of the graphical representation for an object; it is the lower left point of the graphical representation for an annotation. For an arc, the reference point is the middle of the first segment of the arc. Future extensions might allow us to define the position of a reference point of an object or an annotation explicitly. All positions refer to Cartesian coordinates (x, y) . As for many graphical tools, the x -axis runs from left to right and the y -axis runs from top to bottom; but, we do not fix a cunit⁵.

Tool Specific Information. For some tools, it might be necessary to store some internal information, which is not supposed to be used by other tools. In order to store internal information, each object and each label may be equipped with *tool specific information*. The internal format of the tool specific information is up to the tool. But, tool specific information is clearly marked and is assigned the name of the specific tool. Therefore, other tools can easily ignore this information. In general, we discourage the use of tool specific information. In some cases, however, tool specific information might be unavoidable.

Pages and Reference Nodes. A Petri net can be structured by the help of *pages* as known from several Petri net tools (e. g. Design/CPN [4]). A page is an object that may consist of other objects – it may consist even of further pages. An arc, however, may connect nodes on the same page only. In order to connect Petri net nodes on different pages, we can use *reference nodes*: A reference node may refer to any node of the Petri net – located on any page of the net. We require only that there are no cyclic references; this guarantees that, in the end, each reference node refers to exactly one place or exactly one transition of the Petri net. A reference node is only a representative for this node. Reference nodes may have labels, too. But, these labels do not have any meaning. Concerning

³ Of course, the conventions for the use of some labels can recommend a graphical representation of objects with a particular attribute. But, this is not part of PNML.

⁴ For an annotation of the net itself, the position is absolute.

⁵ The size of objects and of labels as well as units are not part of PNML; but they can be easily included in a future version.

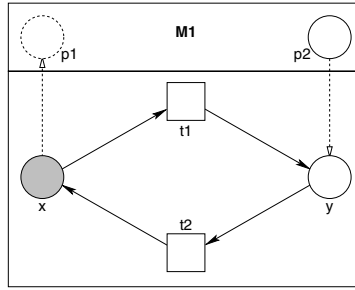


Fig. 2. A module M1

the semantics of the net, the reference node inherits the labels from the node it refers to. This way, it is always possible to flatten the corresponding net without knowing the meaning of the labels and without knowing the semantics of the particular Petri net type (see Sect. 4 for details).

2.2 Modules

Up to now, we have introduced the concepts of pure PNML, i.e. PNML without modules. In this section, we introduce the additional concepts of modular PNML, which are basically *module definitions* and *modules instances*. In order to illustrate these concepts, we start with an example.

Example. For simplicity, we use P/T-systems as the Petri net type in our example. Modular PNML, however, works for any Petri net type.

First, we consider the *definition of a module*. Figure 2 shows the definition of a module M1. This module consists of two places⁶ x and y , two transitions $t1$ and $t2$ and some arcs. This internal *implementation*, however, is not accessible from outside the module. In order to give the environment access to some internal elements of the module, the module defines an *interface*. In our example, the interface consists of a place $p1$, which is *imported* from the environment of the module, and a place $p2$, which is *exported* to the environment of the module. The import place $p1$ is a formal parameter, which is supplied when instantiating the module (see below for details); it is represented by a dashed circle. The export place can be used in the environment of an instance of the module; it is represented by a solid circle. The interface and the implementation of a module are related by references. In our example, reference place x , which is represented as a shaded circle, refers to import place $p1$. The reference is represented graphically by a dashed arrow and should not be confused with a Petri net arc, which would be represented by a solid arrow. So, the reference place x is a representative for a place that will be provided as a parameter when the module is instantiated.

⁶ Actually, x is a reference place, which is indicated by the shading.

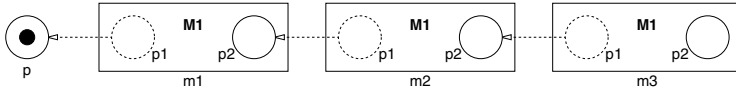


Fig. 3. A net $n1$ built from three instances of module $M1$

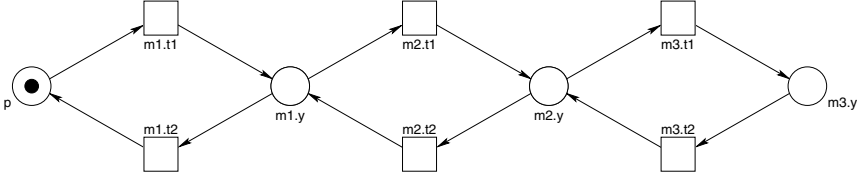


Fig. 4. The semantics of $n1$

Likewise, the export place $p2$ refers to place y . So, a reference to export place $p2$ of an instance of the module will actually refer to place y (see below).

Next, we build a net from several instances of module $M1$. Figure 3 shows a graphical representation of a net $n1$ with three instances of module $M1$, which are named $m1$, $m2$, and $m3$, respectively. In the definition of the net, we define a place p with initial marking 1 and three instances of module $M1$. The first instance $m1$ takes the place p as the actual parameter for import place $p1$, which is represented graphically by a reference from the import place $p1$ of instance $m1$ to place p . The second instance $m2$ takes the export place $p2$ of $m1$ (denoted by $m1.p2$) as the actual parameter for $p1$. Likewise, the third instance $m3$ takes $m2.p2$ as the actual parameter for $p1$. Altogether, the net gives us the P/T-system shown in Fig. 4. We call this P/T-system the *semantics* of the net $n1$. This semantics will be defined, by recursively *inlining* the modules for the corresponding instances (see Sect. 4 for details). As expected, the three instances form a line, which starts with place p and which merges $p2$ of an instance with $p1$ of the next instance. The names of the places and transitions in the different instances are qualified by the name of corresponding instance in order to avoid name clashes. Note that the import places and the export places have completely vanished; they are only representatives for the actual parameters.

Of course, the number of instances and their arrangement may be different in other nets built from module $M1$. For example, we could arrange the three instances of module $M1$ as a ring by passing $m3.p2$ as a parameter for $p1$ of the first instance. We will see in the definition of the semantics, that such cyclic use of export objects as parameters for import objects does not cause any problems.

This finishes our example, and we start with a more detailed discussion of the concepts of PNML.

Symbols. Sometimes, it is necessary to pass other arguments than nodes to a module. For example, a module could implement a channel for some type of

messages, where the particular type is a sort provided by the environment when instantiating the module. This is known from templates in C++ or from parameterized data types, in general. Since modular PNML should be independent from a concrete Petri net type, we cannot fix a syntax for legal parameters for a module. But, we permit the definition of *symbols* – without knowing their meaning. Then, these symbols may be imported and exported in the same way as discussed for nodes in our example. Thus, symbols are objects, too. In particular, there are also *reference symbols*, which refer to other symbols. A symbol may occur within any label and must have a unique identifier.

In high-level Petri nets, the symbols could be sort symbols, operation symbols, and variable symbols, which define the legal inscriptions of places and arcs. By allowing to export and to import these symbols, it is possible to define such a symbol once and to use it in other modules.

Identifiers. In modular PNML, we restrict the values of identifiers to strings not containing a dot character (.). The reason is, that we need the dot for qualifying the name of an object of a particular instance of a module, as known from object oriented programming.

Module Definitions. Simply spoken, a *module definition* is a net with an *interface*. For an instance of a module, only the objects of the interface are accessible from outside the instance. The rest of the module definition is its *implementation*.

Import and Export. The interface of a module contains objects, which can be accessed from outside the module. Remember that an object can be either a node or a symbol. We distinguish two kinds of interface objects: *import* objects and *export* objects. Import objects are representatives of objects that are provided as parameters upon instantiation of the module. The implementation may refer to these objects by a reference to the corresponding import object (cf. the reference from `x` to import node `p1` in Fig. 2). Export objects are defined inside the implementation of the module. Actually, an export object is just a reference object that refers to an object of the implementation of the module. This way, an export object allows the environment to refer to some object in the implementation of the module without knowing implementational details. For semantical reasons, however, modular PNML does not allow direct or indirect references from an export object to an import object of the module itself (see Sect. 4 for details).

Global Nodes, Symbols, and References. Sometimes, we would like to have access to an object from all modules. We could define this object in the outermost net and pass it as a parameter to all other modules. This, however, is quite inconvenient. Therefore, modular PNML supports the definition of *global objects*, which can be referred to from any module without explicitly passing the

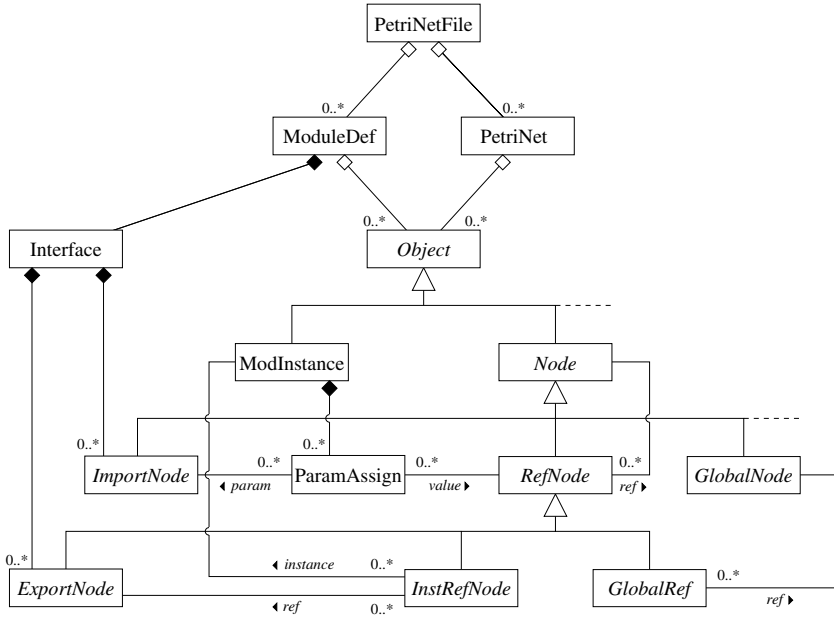


Fig. 5. Modular PNML: An overview

global object as a parameter to that module. A reference to a global object is called a *global reference*. Note that nodes as well as symbols can be global.

Module Instances. A module can be used in a net (or in another module) by instantiating the module. This means that an instance defines a unique identifier within the net and assigns actual objects of the net to the parameters (i. e. to the import objects) of the module. Graphically, these assignments are represented by references from the import objects to its actual parameter. For example, import place *p1* of instance *m1* in Fig. 3 is assigned the actual parameter *p*. Export objects of a module instance are regarded as reference objects. This means that export objects can be used like reference objects. In particular, other reference objects can refer to them or they can be a parameter for an import object of a module instance.

Overview. Figure 5 gives an overview on the basic concepts of modular PNML. But, we leave out some details. For example, we do not distinguish places and transitions; we consider nodes only. Moreover, we omit symbols, since they exhibit the same structure as nodes.

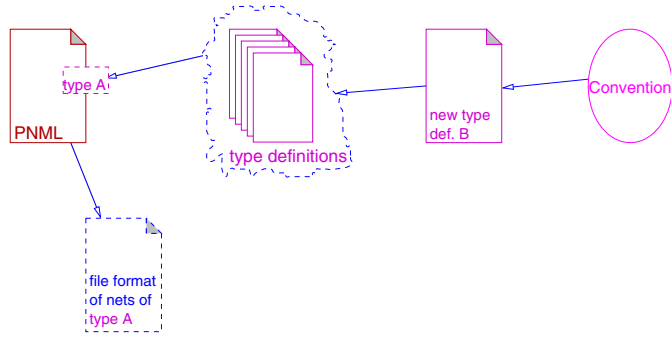


Fig. 6. Interplay of PNML, PNTD and the conventions document

2.3 Type Definition

Up to now, we have discussed the general structure of a Petri net file. The available labels and the legal combinations of labels for a particular object are defined by a *Petri net type*. Technically, a Petri net type is a document that defines the XML syntax of labels; e.g. a Document Type Definition (DTD) file or a schema defined with an XML schema language such as XML Schema [13] or TREX [2]. Conceptually, a Petri net type is a specialization of the general format presented in the previous sections. It adds the definitions of the labels to the objects and to the net respectively.

Figure 6 illustrates the relation between PNML and a Petri net type definition (PNTD). A PNTD which is taken from a pool of various type definitions parameterizes the basic form of PNML. Thus, we get the PNML based file format for Petri nets of that type described by the PNTD. The right part of the illustration will be explained below.

2.4 Conventions

In principle, a Petri net type can be freely defined. In practice, however, a Petri net type chooses the labels from a collection of predefined labels that are provided in a separate document: the *conventions*. The conventions guarantee that the same label has the same meaning in all Petri net types. This allows us to exchange nets among tools with a different, but similar Petri net type.

The conventions are a collection of predefined labels. This collection, however, is not part of PNML. PNML provides only the mechanism for defining the conventions and for including parts of the conventions into a Petri net type. Defining and maintaining the conventions document is an on-going process⁷. In Sect. 3.4, we discuss a small example, which illustrates how such a conventions

⁷ In fact, this process is strongly connected with the work of a standardization group that was founded at the 22nd Conference on Petri Nets (ICATPN) 2001 in Newcastle, U. K.

document could look like. It may serve as a starting point to develop the conventions document. This development will converge in a document containing the most relevant labels from all kinds of Petri nets.

Figure 6 illustrates the relation between the conventions, the Petri net types, and the PNML. The right side shows the definition of a Petri net type based on the conventions, i. e. the labels are chosen from the conventions document. The new Petri net type definition is added to the pool of Petri net types and can be used as a net type in a PNML file.

3 Realization

In this section, we briefly present the PNML⁸ syntax by discussing some examples. PNML is based on the Extensible Markup Language (XML) [12]. The Petri net, the objects, and the labels are represented as XML *elements*. An XML element is included in a pair of a start tag `<element>` and an end tag `</element>`. An XML element may have XML *attributes*⁹ that equip the element with additional information. An XML attribute of an XML element is represented by an assignment of a value to a key (the attribute's name) in the start tag of the XML element `<element key=value>`. An XML element may contain text or further XML elements. An XML element without text or sub-elements is denoted by a single tag `<element/>`. In our examples, we sometimes omit some XML elements. We denote this by an ellipsis (...). All parts of PNML (PNML itself, the different PNTDs and the conventions) were implemented with the XML schema language TREX¹⁰ [2].

The tags of the XML elements defined in PNML are named after the concepts (e. g. `<place>`, `<transition>` or `<page>`) given in Sect. 2. These tags of the concept are the keywords of PNML; they are called *PNML elements*. Labels, however are named after their meaning. Thus, any unknown XML element appearing in a Petri net or in an object can be clearly identified as a label of the net or the object. In our examples, the PNML keywords and the label for the name of an object are underlined. The tags of the other labels, however, are not underlined because they are keywords of a certain Petri net type definition not of the PNML itself.

3.1 The Petri Net Markup Language

Here, we discuss the syntax of the PNML concepts presented in Sect. 2.1. The first examples (List. 1-3) refer to the example net in Fig. 7.

⁸ Please refer to <http://www.informatik.hu-berlin.de/top/pnml/> for a full definition of PNML version 1.1 [6] and some examples.

⁹ Do not confuse XML attributes with attributes of Petri net objects.

¹⁰ TREX has been merged with RELAX to create RELAX NG [3]. TREX and RELAX NG are very similar such that the TREX implementation of PNML is its RELAX NG implementation, too. RELAX NG is specified by a committee of the XML standardization organization OASIS (<http://www.oasis-open.org>).

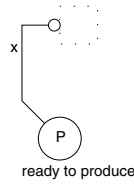


Fig. 7. An example net

The unique identifier of a Petri net or an object of a Petri net is given by an XML attribute `id` of the corresponding PNML element. The value of this attribute must meet the requirements for the attribute type ID of XML (cf. [12]); i.e. it must be a string starting with a letter or the underscore character, followed by letters, digits or several other characters except ‘.’.

Listing 1. The PNML code of a place

```

<place id="p1">
  <graphics>
    <position x="20" y="40"/>
  </graphics>
5  <name>
    <value>ready to produce</value>
    <graphics>
      <offset x="-10" y="10"/>
    </graphics>
10 </name>
    <initialMarking>
      <value>P</value>
      <graphics>
        <offset x="-1" y="-1"/>
15 </graphics>
    </initialMarking>
  </place>

```

Listing 1 shows the representation of a place with the identifier `p1`. The place has two labels; to be more precise, it has two annotations. The first one represents the name of the place `<name>`, whereas the second one represents its initial marking `<initialMarking>`. An annotation consists of its value `<value>` and, possibly, of some graphical information. In our example, both annotations as well as the place itself have graphical information, which are represented by XML elements `<graphics>`. The concrete definition of the XML element `<graphics>`

depends on the context in which it appears. A place has a position, whereas an annotation has an offset position.

Listing 2 shows the representation of a transition, which is similar to the representation of a place. Transition `t1` contains tool specific information,

Listing 2. The PNML code of a transition

```

<transition id="t1">
  ...
  <toolspecific tool="PN4all" version="0.1">
    <hidden/>
5  </toolspecific>
</transition>

```

which makes it a hidden transition of some imaginary tool PN4ALL version 0.1. Syntactically, `toolspecific` information is represented by `<toolspecific>`; this XML element must have at least the shown XML attributes and may contain further XML elements defined by the tool.

Listing 3 shows the representation of an arc: The source and the target

Listing 3. The PNML code of an arc

```

<arc id="a1" source="p1" target="t1">
  <graphics>
    <position x="10" y="30"/>
    <position x="10" y="10"/>
5  </graphics>
  <inscription>
    <value>x</value>
    <graphics>
      <offset x="-6" y="-16"/>
10  </graphics>
    </inscription>
    <type value="inhibitor"/>
  </arc>

```

node are given as XML attributes of the corresponding element `<arc>`. PNML requires that each arc has a unique identifier, which allows us to have two or more arcs between the same nodes. The graphical information of the arc contains a list of points. These points represent intermediate points of the arc. The offset in the graphical information of the `<inscription>` defines the position of the

label relative to the reference point of the arc. Arc **a1** has an additional attribute called **<type>**; it indicates that it is an inhibitor arc. Note, that the shape of the inhibitor arc is tool dependent; other tools could use a different shape.

Listing 4 shows the representation of a page and of reference nodes of a Petri net. A page may have the same objects as the net itself – even pages and

Listing 4. The PNML code of a page

```

<page id="pg1">
  <name>
    <value>Example page of the net</value>
  </name>
5  <referencePlace id="rp1" ref="p1">
    <name>...</name>
    <graphics>
      <position x="20" y="20"/>
    </graphics>
10 </referencePlace>
    <referenceTransition id="rt1" ref="t1">
      ...
    </referenceTransition>
    <place id="p2">...</place>
15 <transition id="t2">...</transition>
    <arc id="a2" source="rp1" target="t2">
      ...
    </arc>
    ...
20 </page>

```

reference nodes. A reference node (indicated by tags **<referencePlace>** or **<referenceTransition>**) refers to a node of the net via the XML attribute **ref**. Its value refers to the identifier of a node of this net. Furthermore, a reference node may have its own graphical information, tool specific information, and labels. Remember that these labels have no real meaning, since they are ignored in the underlying flattened net. But, they allow reference nodes to carry their own name or other informal information. Note that the source and the target of an arc must be nodes on the same page.

Listing 5 shows the representation of a Petri net. A net consists of pages, modules instances (see below) and other objects. In our example, there is an annotation defining the net's name. The type of the net is given in the XML attribute **type**.

Listing 5. The PNML code of a net

```

<net id="n1" type="HLnet">
  <name>
    <value>Example high-level net</value>
  </name>
5  <place id="p1">
    ...
  </place>
  <page id="pg1">
    ...
10 </page>
    ...
  </net>

```

3.2 Modular PNML

Now, we discuss the syntax of modular PNML. To this end, we come back to the examples of Sect. 2.2. In order to keep the examples small, we omit graphical information from the PNML code.

A module is defined by the PNML element `<module>`. This element contains both, the interface of the module and its implementation. The interface is tagged by `<interface>` and contains import and export objects of the module. The nodes of the interface may have graphical information as described above. The rest of the module contains the implementation of the module, which is the same as for nets. In addition, an implementation of a module may use instances of any other module. The only restriction is that there is no cyclic dependency between the modules. Moreover, reference objects of the implementation may refer to the import nodes of the module's interface.

Listing 6 shows the PNML code of module M1 in Fig. 2. There is a module with its interface and its implementation.

The interface of a module contains nodes and symbols with their identifiers to be imported or exported. In our example (List. 6, cf. Fig. 2), there is one import place p1 and there is one export place p2. The implementation part in our example does not use other modules. Note that reference objects of the implementation may refer to import objects but not to export objects of the interface of the module. Export objects refer to objects of the implementation. But, they are not allowed to transitively refer to import objects.

If a module M_1 (or a net) contains an instance of a module M_2 , we say M_1 *uses* M_2 . The use of a module is tagged by the PNML element `<instance>`. This element refers to the Uniform Resource Identifier (URI) of the corresponding module with the instance's XML attribute `ref`. As mentioned above, the uses relation must not have cycles. The PNML element `<instance>` contains references to nodes and symbols which serve as actual parameters for the import

Listing 6. The PNML Code of the module in Fig. 2

```

<module name="M1">
  <interface>
    <importPlace id="p1"/>
    <exportPlace id="p2" ref="y"/>
5  </interface>
    <referencePlace id="x" ref="p1"/>
    <transition id="t1"/>
    <transition id="t2"/>
    <place id="y"/>
10  <arc source="x" target="t1"/>
    <arc source="t1" target="y"/>
    <arc source="y" target="t2"/>
    <arc source="t2" target="x"/>
  </module>

```

objects of the module. Such a reference names the parameter that is instantiated and refers to a ‘real’ object occurring in the instantiating net or module.

References to export objects of an instance are composed of a reference to that module instance (the XML attribute `instance`) and a reference to an export object of that instance (the XML attribute `ref`).

Listing 7 shows the PNML code of the net in Fig. 3. Net `n1` contains a place `p` with an initial marking of one token and three instances of the module `M1`. Place `p` serves as the actual parameter for `p1` in instance `m1`. The module instance `m2` gets the export place `p2` of the instance `m1` as its actual parameter `p1` and so on.

Furthermore, PNML allows us to define global nodes and global symbols. They are tagged by `<globalPlace>`, `<globalTransition>`, and `<globalSymbol>` respectively. Similarly, we add the XML attribute `gref` to reference objects and import parameters of interfaces for references to global objects. This XML attribute is alternative to both the XML attribute `instance` and `ref`. The value of the XML attribute `gref` refers to a globally defined object.

3.3 Petri Net Type Definition

Next, we discuss the syntax of Petri net type definitions (PNTD). As mentioned above, we use the XML schema language TREX [2] for defining PNML and the particular PNTDs, as well. Listing 8 shows the PNTD for P/T-systems `ptNet.pntd`. For P/T-systems, we need two additional labels: one label for places, which represents the initial marking, and one label for arcs, which represents the arc inscription. Listing 8 shows the TREX file for the definition of P/T-systems. It starts with some TREX specific stuff for defining a *grammar*. Then it includes the definition of PNML and of the conventions document, from

Listing 7. The PNML Code of a net using modules (cf. Fig. 3)

```

<net id="n1">
  <place id="p">
    <initialMarking>
      <value>1</value>
5    </initialMarking>
  </place>
  <instance id="m1" ref=URI#M1>
    <importPlace parameter="p1" ref="p"/>
  </instance>
10 <instance id="m2" ref=URI#M1>
    <importPlace parameter="p1" instance="m1" ref="p2"/>
  </instance>
  <instance id="m3" ref=URI#M1>
    <importPlace parameter="p1" instance="m2" ref="p2"/>
15 </instance>
</net>

```

Listing 8. The PNTD for P/T-systems ptNet.pntd

```

<grammar ns="http://www.informatik.hu-berlin.de/top/pnml"
  xmlns="http://www.thaiopensource.com/trex">
  <include href="pnml.trex"/>
  <include href="conv.trex"/>
5  <define name="NetType" combine="replace">
    <string>ptNet</string>
  </define>
  <define name="Place" combine="interleave">
    <optional>
10    <ref name="InitialMarking"/>
    </optional>
  </define>
  <define name="Arc" combine="interleave">
    <optional>
15    <ref name="Inscription"/>
    </optional>
  </define>
</grammar>

```

which we choose the definition of the labels. The conventions document itself, will be discussed in Sect. 3.4. Both, the PNML and the conventions document are TREX grammars, too.

Then, the file gives a name to the defined Petri net type (`ptNet`). Next, the file lists the objects that are equipped with additional labels. In our example, these objects are places and arcs (`"Place"` and `"Arc"`). The definition of a label can be either given explicitly, or can be taken from the conventions document. In our example, the labels are taken from the conventions document by referring to their names in that document (`"InitialMarking"` and `"Inscription"`). Technically, this is achieved by the TREX element `<ref>`. The `<optional>` elements indicate that the labels are optional in this Petri Net type.

3.4 Conventions

At last, we give an example of a very simple conventions document, which defines just those two labels that were used in the example above. It should illustrate the mechanism of PNML for including definitions from a conventions document. A real conventions document would contain many more label definitions and would require a more involved taxonomy for labels. A standardized conventions document is subject to future research and to many discussions.

Listing 9 shows an example of entries in a conventions document. First,

Listing 9. An example conventions document

```

<grammar ns="http://www.informatik.hu-berlin.de/top/pnml"
  xmlns="http://www.thaiopensource.com/trex">
  <include href="pnml.trex"/>
  <define name="InitialMarking">
5    <element name="initialMarking">
      <ref name="Annotation"/>
    </element>
  </define>
  <define name="Inscription">
10   <element name="inscription">
      <ref name="Annotation"/>
    </element>
  </define>
</grammar>

```

it includes the definition of labels (annotations in our case) from the general PNML definition `pnml.trex`. Then, it defines the two labels, which were used in the example above (`"initialMarking"` and `"inscription"`). Both of them are normal PNML annotations, which means that their value can be an arbitrary string; but we could use the full power of TREX for defining other domains.

4 Semantics

In this section, we will briefly discuss the semantics of PNML. Of course, it is impossible to define the semantics of the Petri net represented by a PNML file because this strongly depends on the Petri net type. Though there is some theoretical work on a unified framework for defining Petri nets, there is no formalism yet that deals with all types of nets definable by a PNTD. Therefore, we assume that along with a new PNTD, there will be a definition of the semantics of this Petri net type for nets without pages, references and modules.

In the following, we will give a precise semantics to pages, references, and modules by translating them to a net without these concepts. We proceed in two steps. First, we define a semantics for pure PNML by eliminating pages and references; then, we give a semantics for modular PNML by eliminating modules (resp. their instances) by translating them to pure PNML. For a more detailed discussion of this semantics, we refer to [6].

4.1 Pages and References

The semantics of pages and references is quite simple: We omit the pages from the PNML file and we resolve all references. Resolving references means that each reference object is replaced by the object it refers to (directly or indirectly via other reference objects). Actually, we delete the reference object, and we redirect all its attached arcs to the object it refers to. Note that by deleting the reference object, all labels of the reference object are lost, but this is no problem because labels of reference objects do not have any meaning.

Resolving all references is always possible, because PNML requires that there are no cyclic references between reference objects. This way, it is always possible to obtain an equivalent PNML file without pages and reference nodes. We have lost only graphical information and structuring information. We call this transformation *flattening* of a PNML net.

4.2 Modules

For modular PNML as defined in Sect. 2.2, we define the semantics by translating it to pure PNML. To this end, we basically replace each instance of a module by a copy of its implementation. In order to avoid name clashes, the identifiers of the implementation of a module will be preceded by the instance's name, where the names are separated by a dot¹¹. In order to keep the module structure of the original model, the implementation of each module instance is defined on a separate page, which receives the name of the module instance. We call this replacement the *inlining* of a module instance.

The inlining process can be applied recursively for modules that use module instances in their implementation. In the end, we obtain a net in pure PNML.

¹¹ By forbidding the dot in identifiers in modular PNML, this simple naming scheme gives us unique names for all instances, even when applied recursively.

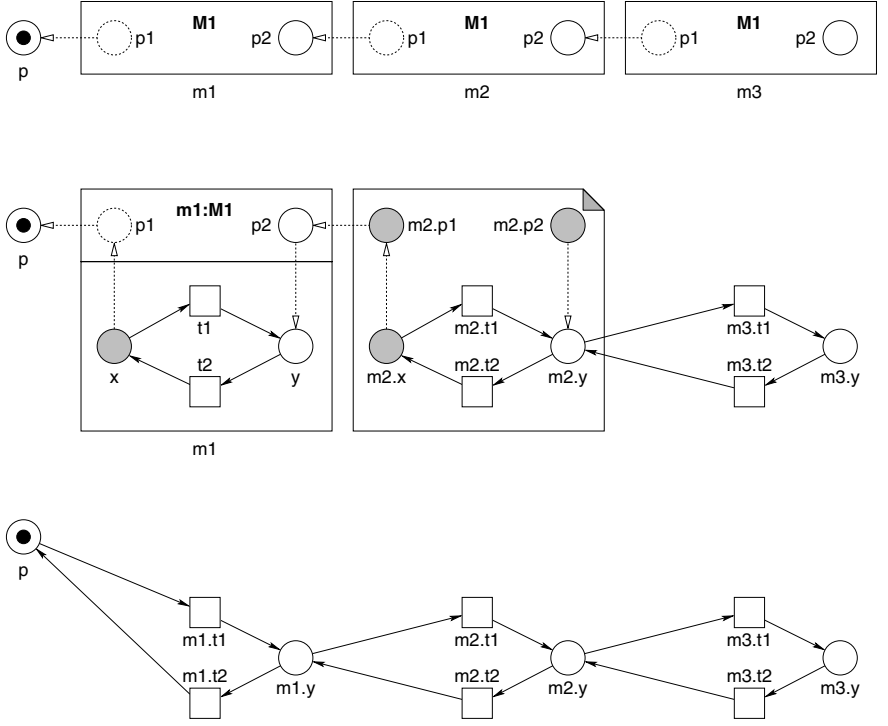


Fig. 8. Illustration of inlining and flattening

Note that inlining of module instances will never give us cyclic references, provided that there are no cyclic references in the module definition itself and no references from an export node to an import node. Thus, we can flatten the obtained net after the inlining process.

Figure 8 shows three stages of the inlining and flattening process for the net from Fig. 3. The first line shows the original net with three instances of module $M1$. The second line shows three different stages of the inlining process for the three different instances $m1$, $m2$, and $m3$: for $m1$, it shows the replacement by the module definition; for $m2$, it shows the replacement after renaming all objects (by preceding each name with the instance's name) on a separate page; for $m3$, the reference nodes and the page have already been eliminated (flattening). The bottom line shows the overall semantics of the three module instances after inlining and flattening.

5 Conclusion

In this paper, we have presented PNML and its modular extension as an interchange format for Petri nets. The main feature of PNML is its universality, which is achieved by representing a net as a labelled graph along with a Petri Net Type

Definition. The Petri Net Type Definition defines the legal labels for a particular Petri net type. The definition of modular PNML shows that universality carries over to a module concept, which can be used with any Petri net type. This way, PNML provides a way to exchange Petri nets between different tools without loosing too much information.

PNML itself is now in a stable state. Currently, it is supported by the PNK [10] and Renew [11]. For a broader acceptance, we need a standardized set of Petri net types and a standardized set of labels, which covers most of the existing versions of Petri nets. PNML provides a mechanism for defining Petri net types and for using labels from a conventions document. The Petri net types and the conventions document, however, are not part of PNML. The development and the standardization of the conventions document is an on-going process, which requires further research and interaction among researchers. More information on this process can be found on the web: <http://www.informatik.hu-berlin.de/top/pnml/>.

References

- [1] Rémi Bastide, Jonathan Billington, Ekkart Kindler, Fabrice Kordon, and Kjeld H. Mortensen, editors. *Meeting on XML/SGML based Interchange Formats for Petri Nets*, Århus, Denmark, June 2000. 21st ICATPN. 124, 144
- [2] James Clark. TREX – tree regular expressions for XML. <http://www.thaiopensource.com/trex/>. 2001/01/20. 133, 134, 139
- [3] James Clark and Makoto Murata (eds.). RELAX NG specification. <http://www.oasis-open.org/committees/relax-ng/>. 2001/12/03. 134
- [4] Design/CPN. <http://www.daimi.au.dk/designCPN/>. 2001/09/21. 125, 128
- [5] Ekkart Kindler and Michael Weber. The Petri Net Kernel – an infrastructure for building Petri net tools. *Software Tools for Technology Transfer (STTT)*, 3(4):486–497, 2001. 124
- [6] Ekkart Kindler and Michael Weber. A universal module concept for Petri nets. an implementation-oriented approach. Informatik-Berichte 150, Humboldt-Universität zu Berlin, June 2001. 134, 142
- [7] Arndt Lüder and Hans-Michael Hanisch. A signal extension for Petri nets and its use in controller design. *Fundamenta Informaticae*, 41(4):415–431, 2000. 126
- [8] Thomas Mailund and Kjeld H. Mortensen. Separation of style and content with XML in an interchange format for high-level Petri nets. In Bastide et al. [1], pages 7–11. 127
- [9] The Moses Project. <http://www.tik.ee.ethz.ch/~moses>. 2002/03/04. 126
- [10] Petri Net Kernel. <http://www.informatik.hu-berlin.de/top/pnk/>. 2001/11/09. 124, 144
- [11] Renew: The Reference Net Workshop. <http://www.renew.de>. 2002/03/04. 126, 144
- [12] World Wide Web Consortium (W3C) (ed.). Extensible Markup Language (XML). <http://www.w3.org/XML/>. 2000/10/06. 134, 135
- [13] World Wide Web Consortium (W3C) (ed.). XML Schema. <http://www.w3.org/XML/Schema>. 2001/05/02. 133

Case Study Logistics: Flexible Modeling of Business Processes using Algebraic Higher-Order Nets

Kathrin Hoffmann

Institute for Software Engineering and Theoretical Computer Science
Technical University Berlin, Germany
email : hoffmann@cs.tu-berlin.de

Abstract. In this paper we present a case study logistics using Algebraic Higher-Order Nets. Algebraic Higher-Order Nets are an extension of the well defined formalism of Algebraic High-Level Nets by higher-order functions leading to a more flexible modeling technique. After an introduction of Algebraic Higher-Order Nets at an informal level we present the case study logistics and demonstrate the advantage of our approach which allows a flexible modeling of business processes including exceptions and roles without changing the net structure of our nets.

1 Introduction

In the context of parallel and concurrent systems Petri nets represent a well-known and widely used formalism and have been employed in practical applications in different areas. Their graphical representation and formal semantics excellently support the modeling and simulation of such systems. Among a large variety of different high level net classes (see e.g. [Gen91,Jen92]) Algebraic High-Level Nets [EPR94] give rise to a formal and well defined description of the dynamic behaviour of concurrent and distributed systems due to their combination of algebraic specification and Petri nets. This formalism is adequate in application domains, where the context is known from the very beginning such that the system can be modeled by a Petri net with a fixed structure. In other application domains like business process modeling it is also desirable to support the fact that an organization of a system is not fixed once and for all, that is, changes of the environment can only be modeled by changing the structure of Petri nets.

In the area of Petri nets higher-order structures as tokens is a topic of research (see e.g. [Han97,Val98,Hof00]) leading to a more flexible modeling of business processes. In this paper we show that our concept of Algebraic Higher-Order Nets [Hof00] including algebraic higher-order signatures as data type part [MHKB99] is suitable for flexible modeling of business processes.

The paper is organized in the following way: In Section 2 we illustrate the main idea of Algebraic Higher-Order Nets with a small example using a simplified notation of algebraic higher-order signatures, specifications and algebras.

Subsequently we present our case study logistics using Algebraic Higher-Order Nets (Section 3), which allows a flexible modeling of business processes without changing the net structure (Section 4), and a short conclusion (Section 5).

2 Motivation and Informal Introduction

In this section we motivate the notions of Algebraic High-Level Nets and Algebraic Higher-Order Nets in terms of a small example in order to demonstrate in the following section how Algebraic Higher-Order Nets are used for flexible modeling of business processes without changing the net structure.

In the application domain of business processes Petri nets are a widely used formalism and have been successfully employed in practical applications. For the integration of data elements into business processes it is adequate to use at least High-Level Petri Nets. In this way we obtain a more compact model, because a particular part of the net structure is reduced to data elements.

Let us first consider Algebraic High-Level Nets [PER95], an integration of classical Petri nets and algebraic specifications [EM85]. Technically, the algebraic specification is used to define net inscriptions by terms over the specification. Furthermore firing conditions defined by algebraic equations guarantee, that certain constraints are respected. The marking of Algebraic High-Level Nets consists not only of black tokens, but also of data tokens which are elements from a given algebra. In contrast to black tokens of low level Petri nets data tokens can be modified during the firing of transitions.

Figure 1 shows a very simple Algebraic High-Level Net. The idea of the net is to compute alternatively the successor, the square or the cube of natural numbers. The net is inscribed with terms over the signature NAT of natural numbers given below.

$$\begin{aligned} \underline{\text{NAT}} = \\ \underline{\text{sorts}} : \quad & \text{nat} \\ \underline{\text{opns}} : \quad & \text{zero} : \rightarrow \text{nat} \\ & \text{succ} : \text{nat} \rightarrow \text{nat} \\ & \text{square} : \text{nat} \rightarrow \text{nat} \\ & \text{cube} : \text{nat} \rightarrow \text{nat} \end{aligned}$$

We consider the NAT-algebra A consisting of natural numbers with constant $\text{zero}_A = 0$ and the well-known functions succ_A , square_A and cube_A , respectively. In Figure 1 the type of place **p1** is given by the sort *nat* and the initial marking of place *p1* consists of natural numbers $n_1, \dots, n_k \in \mathbb{N}$.

To demonstrate the firing behaviour of the transition **Compute succ** (and similar for **Compute square** and **Compute cube**) we first assign a natural number n to the variable x . The follower marking is computed as follows: the data element n is consumed from place **p1** and the result $\text{succ}_A(n)$ is added to the place **p2**.

The Algebraic High-Level Net **Computation** in Figure 1 can be represented in a more compact way by an Algebraic Higher-Order Net **Computation'** in

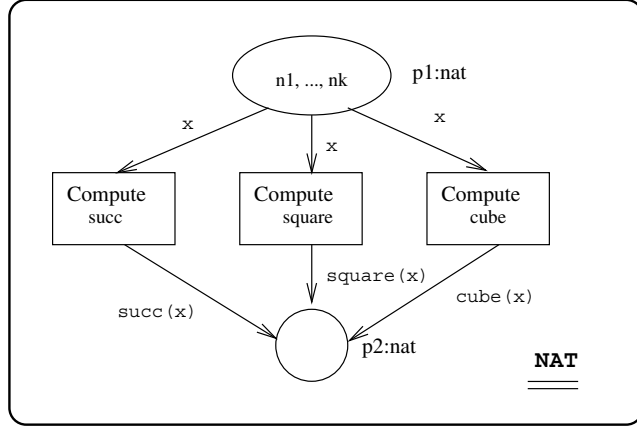


Fig.1: Algebraic High-Level Net Computation

Figure 2, where the three different transition in Figure 1 are presented by one transition together with a higher-order place p and three different function type tokens $succ$, $square$ and $cube$. The function type in this case is $nat \rightarrow nat$ which means that $succ$, $square$ and $cube$ can be interpreted as functions from natural numbers to natural numbers. In order to fire transition **Compute** in Figure 2 we have to give not only an assignment to variable x of type nat (as in Fig. 1), but also an assignment to variable f of type $nat \rightarrow nat$. Assigning n to x and $succ$ to f we can fire transition **Compute**, where token n is removed from place p_1 , token $succ(n)$ is added to place p_2 , but token $succ$ remain on place p as indicated by the horizontal line between transition **Compute** and place p . In fact, place p can be considered as a context place in the sence of contextual nets (see [MR95]), where, however, p is now a place of higher-order type.

In order to allow places and tokens of function type we have to replace the algebraic signature NAT by an algebraic higher-order signature HO-NAT and the NAT-algebra A by a HO-NAT-algebra $HO-A$. This is possible in the framework of Algebraic Higher-Order Nets, which is formally introduced in our paper [Hof00] bases on algebraic higher-order specification (see [MHKB99]). For the presentation in this paper, however, we use a most simplified notation for higher-order signatures and algebras as follows:

$$\begin{aligned}
 &\underline{HO - NAT} = \\
 &\underline{basic \ sorts} : \quad nat \\
 &\underline{opns} : \quad \begin{aligned} &zero : \rightarrow nat \\ &succ : nat \rightarrow nat \\ &square : nat \rightarrow nat \\ &cube : nat \rightarrow nat \end{aligned}
 \end{aligned}$$

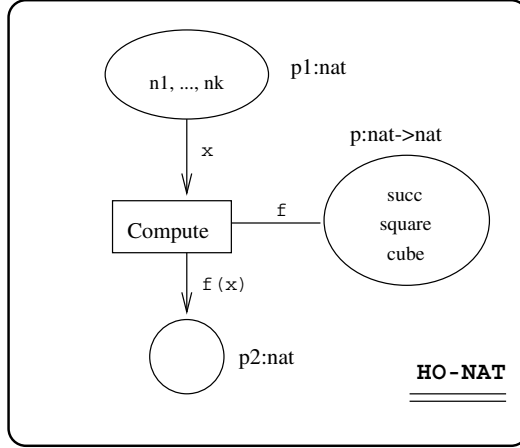


Fig.2: Algebraic Higher-Order Net Computation'

where *succ*, *square* and *cube* - written as operation symbols from *nat* to *nat* - are in fact higher-order function symbols of type $nat \rightarrow nat$. This notation of operation symbols allows to build up terms like *succ*(*x*), *square*(*x*) and *cube*(*x*) with the usual interpretation, or more general $f(x)$, where f is a variable of type $nat \rightarrow nat$. In addition to the basic sort *nat* HO-NAT has also product and function types like $nat \times nat$, $nat \rightarrow nat$ and $(nat \rightarrow nat) \times nat$, where in Fig. 2 we only need the basic sort *nat* and the function type $nat \rightarrow nat$. This allows to consider *succ*, *square* and *cube* as higher-order constants of type $nat \rightarrow nat$, which are used as tokens in place *p* in Fig. 2.

For a precise notation of higher-order signatures, algebras and Algebraic Higher-Order Nets we refer to [MHKB99, Hof00]. In [Hof00] we have also introduced two types of transformations between specific Algebraic High-Level Nets and Algebraic Higher-Order Nets, called folding and unfolding. In our case the Algebraic Higher-Order Net in Fig. 2 is the folding of the Algebraic High-Level Net in Fig. 1, and vice versa the Algebraic High-Level Net in Fig. 1 is the unfolding of the Algebraic Higher-Order Nets in Fig. 2.

The advantage of Algebraic Higher-Order Nets is the reusability of the fixed net structure. To include further computations of function type $nat \rightarrow nat$ into the Algebraic Higher-Order Net Computation', we extend the marking of place *p*, but can keep the same net structure. In the Algebraic High-Level Net Computation the net structure would have to be extended by one transitions for each further computation.

In Figure 2 we have only used the basic sort **nat** and the function type $nat \rightarrow nat$. As explained above we can also use product types and combined function and product types.

In Figure 3 the type of place *p* is given by the product type $(nat \rightarrow nat) \times person$. The functions as given by the marking of place *p* are extended by roles

$r1$ and $r2$, respectively. This means, that e.g. the functions *succ* can only be applied by a person in role $r1$, while the firing behaviour is equivalent to the Algebraic Higher-Order Net Computation’.

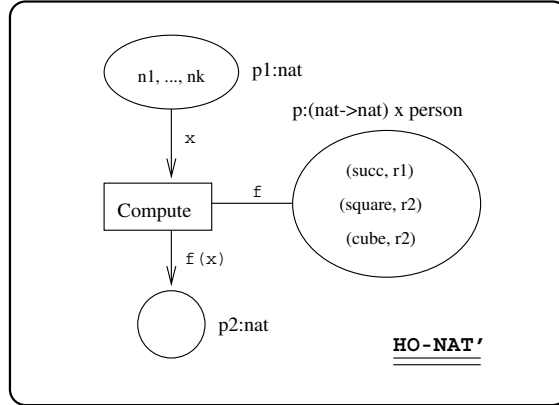


Fig. 3: Place with Product Type

Summarizing Algebraic Higher-Order Nets give rise to a more compact model by using functions as tokens. The advantage is the flexible modeling of business processes, that means that changes of the environment are modeled by changing the corresponding marking. Furthermore we are able to introduce a role model into Algebraic Higher-Order Nets.

3 Case Study Logistics

The business process logistics consists of the planning and scheduling functions concerned with the distribution of products to customers. In view of departments the business process can be divided into different parts, each of them with specific documents and activities. After receiving an order from a customer in the order department the availability of articles is checked. In the delivery department the corresponding delivery note is generated and articles are sent to the customer. Finally in the accounts department invoices are captured and cleared with incoming payments.

Our case study logistics is based on data structures defined in [Sch94] by entity/relationship-diagrams and processes modeled by event driven process chains. In more detail our case study logistics consists of three Algebraic Higher-Order Nets **Order Department**, **Delivery department** and **Account department** corresponding to the departments described above. To obtain the overall system of the business process logistics we first observe, that there are places with the same name which are present in different Algebraic Higher-Order Nets. Our Algebraic Higher-Order Nets are sequentially composed by

merging places `checked order` and `stock` of the net `Order Department` and `Delivery department` and places `receipted delivery note` and `delivered order` of the net `Delivery Department` and `Account department`. The resulting net `Logistics` is depicted in Fig. 4 (without net inscriptions).

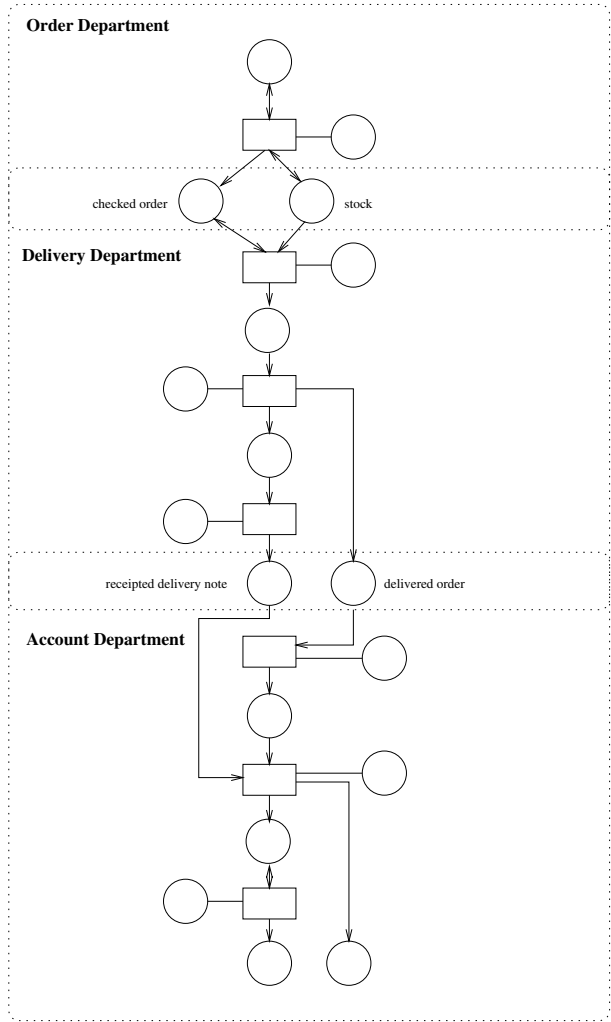


Fig. 4: Logistics

Let us point out that in this section we only make limited use of the advantage of Algebraic Higher-Order nets compared with Algebraic High-Level Nets, because we use only one or two token on the higher-order places. But the notation as Algebraic Higher-Order nets will allow in Section 4 the flexible modeling

of business processes by adding additional token on higher-order places in Fig. 8 or changing the type of higher-order places in Fig. 9, without changing the net structure of the nets.

In the following we describe the three different Algebraic Higher-Order Nets **Order**, **Department**, **Delivery** department and **Account** department depicted in Fig. 4 in more detail.

In Figure 5 the department of dealing with an order of a customer is depicted. Here the availability of ordered articles is checked by comparing the list of articles, which are available at the moment, with the list of ordered articles. We split the order into two parts, the current order including all articles, which are available at the moment, and a new order including all remaining articles to be carried out later as soon as they are available.

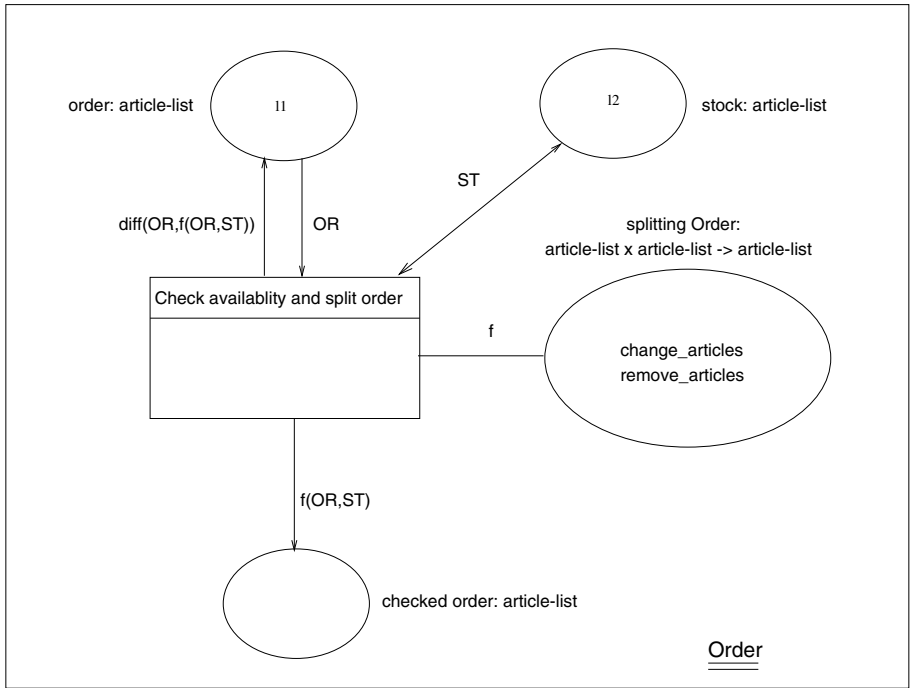


Fig.5: Order Department

The net **Order** department consists of three places called **stock**, **order** and **checked order**, where the type is given by the basic sort *article-list*. The initial marking l_1 of place **order** means that there is a list of articles ordered by a customer. Similar the marking l_2 of place **stock** represents a list of articles, which are available at the moment. Furthermore there is a higher-order place **splitting order** with function type $\text{article-list} \times \text{article-list} \rightarrow \text{article-list}$ and a marking consisting of functions *change_articles* and *remove_articles* to split

an order. These higher-order functions *change_articles* and *remove_articles* and also the function *diff* are declared in the higher-order signature Order not shown explicitly. The function *remove_articles* can be interpreted by removing articles, which are not available at the moment, from the current order. These articles are postponed to the new order. In contrast the function *change_articles* means that the number of pieces of an ordered article is splitted into two parts. The first is the number of pieces available in the stock, i.e. l_2 . The second is the difference of those in l_1 and l_2 . So the article remain a part of the current order with quantities available in l_2 and becomes also a part of the new order with remaining quantities.

During the firing of transition **Check availability and split order** one of the functions *change_order* and *remove_order*, respectively, is selected by the assignment of the variable f . Furthermore the variable OR is assigned to the order l_1 and the variable ST is assigned to the stock l_2 . Then we compute the current order $f(l_1, l_2)$, which is added to place **checked order**, while the new order $diff(l_1, f(l_1, l_2))$ is added to place **order** and consists of all remaining articles given by the difference of the original order l_1 and the current order $f(l_1, l_2)$.

In Figure 6 the Algebraic Higher-Order Net of the delivery department is depicted. In a first step articles of an order are removed from the stock and provided for loading up to trucks. Furthermore a delivery note is generated to inform the customer about articles to be received.

In the Algebraic Higher-Order Net **Delivery department** we have a closer look at the corresponding higher-order signature Delivery in this case, which is depicted below. It is used first of all to specify on one hand the construction of a list of articles and delivery notes. Due to the operation *make_article* articles consist of a pair of numbers of pieces *volume* and a specific article number *article-nr*. The list of articles is generated by the operation *no_list* for the empty list and the operation *add_article* to add recursively articles to an existing list. On the other hand the higher-order signature Delivery declares activities *remove_article*, *make_delivery_note* and *receipted_delivery_note*, which are carried out during the process in the order department. The meaning of these function is given in a corresponding higher-order algebra A of signature Delivery, which is not given explicitly in this paper.

$$\begin{aligned}
 \underline{\underline{Delivery}} &= \underline{\underline{Bool}} + \\
 \underline{\underline{basic\_sorts}} : & \text{article, volume, article} - \text{nr, article} - \text{list, delivery} - \text{notes} \\
 \underline{\underline{opns}} : & \text{make\_article} : \text{volume} \times \text{article} - \text{nr} \rightarrow \text{article} \\
 & \text{no\_list} : \rightarrow \text{article} - \text{list} \\
 & \text{add\_article} : \text{article} - \text{list} \times \text{article} \rightarrow \text{article} - \text{list} \\
 & \text{remove\_article} : \text{article} - \text{list} \times \text{article} - \text{list} \rightarrow \text{article} - \text{list} \\
 & \text{make\_delivery\_note} : \text{article} - \text{list} \times \text{bool} \rightarrow \text{delivery} - \text{note} \\
 & \text{receipted\_delivery\_note} : \text{delivery} - \text{note} \rightarrow \text{delivery} - \text{note} \\
 & \dots
 \end{aligned}$$

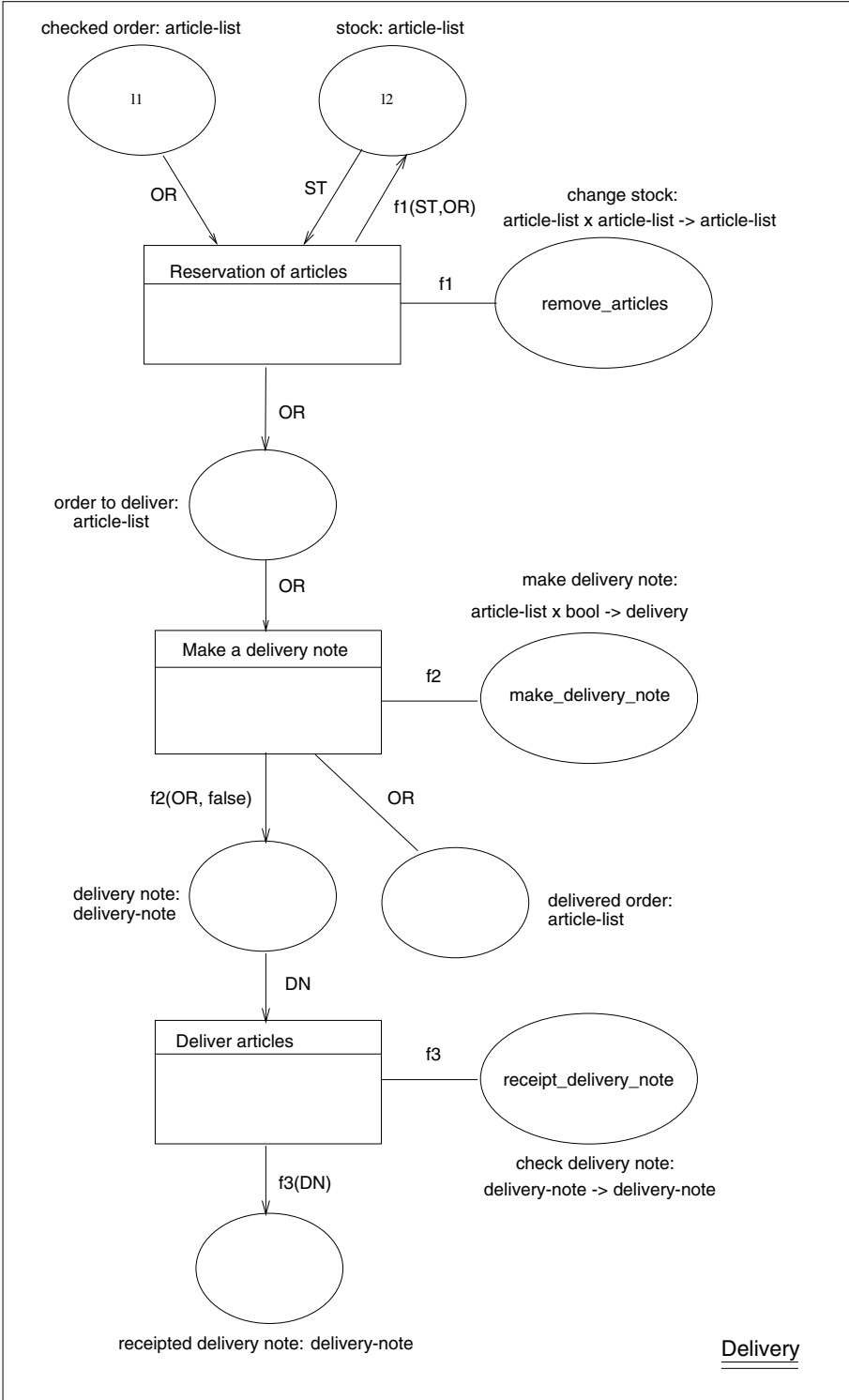


Fig.6: Delivery Department

The Algebraic Higher-Order Net **Delivery department** (see Fig. 6) consists of four places **stock**, **checked order**, **order to deliver** and **delivered order**, where the type is given by the basic sort *article-list*. Similar to the net **Order department** (see Fig. 5) the marking of the place **checked order** consists of a list of articles l_1 to define an order of a customer. Note, that the availability of ordered articles is already checked in the net **Order department** (see Fig. 5). Analogously the marking of the place **stock** consists of a list of articles l_2 , which are available at the moment.

The type of places **delivery note** and **receipted delivery note** is given by the basic sort *delivery_note*, that is a lists of articles together with a boolean value. The boolean value *true* indicates that the delivery note is checked by customers. The type of the higher-order places **change stock**, **make delivery note** and **check delivery note** are given by corresponding higher-order function types with markings consisting of different activities carried out in the delivery department.

In the following we explain the firing behaviour of the Algebraic Higher-Order Net **Delivery department** in more detail. In order to fire transition **Reservation of articles** in Fig. 6 we have to give not only an assignment to variables *ST* and *OR* of type *article-list*, but also of variable f_1 of type $\text{article-list} \times \text{article-list} \rightarrow \text{article-list}$. Assigning l_1 to *OR*, l_2 to *ST* and f_1 to *remove_articles* we can fire transition **Reservation of articles**. The function *remove_articles* means, that ordered articles are removed from the current stock by changing the number of pieces. Then tokens l_1 and l_2 are consumed from place **checked order** and **stock**, respectively. On one hand token l_1 is added to the place **order to deliver**. On the other hand token *remove_articles*(l_2, l_1) is added to place **stock** and can be considered as the new stock, that is the original list of articles l_2 without ordered articles l_1 . The marking of the higher-order place **change stock** remains to be unchanged as indicated by the horizontal line between the transition **Reservation of articles** and the higher-order place **change stock**.

In a next step a delivery note is generated by the application of the function *make_delivery_note* given as marking of place **make delivery note** to an order. In detail, we assign the order $l_3 = l_1$ to the variable *OR* and the function *make_delivery_note* to the variable f_2 . During the firing of transition **Make a delivery note** we remove token l_3 from place **order to deliver** and add a token $l_4 = \text{make\_delivery\_note}(l_3, \text{false})$ to place **delivery note** and a token l_3 to place **delivered order**. The token *make_delivery_note*(l_3, false) can be interpreted as a list of ordered articles, which have to be checked by the customer.

Finally, during the firing of transition **Deliver articles** we assign the delivery note l_4 to the variable *DN* and the function *receipt_delivery_note* to the variable f_3 . Then the delivery note l_4 is removed from place **delivery note** and is checked by the customer by changing the boolean value. This means that a token *receipt_delivery_note*(l_4) is added to place **receipted delivery note**, while the marking of the higher-order place **check delivery note** left to be unchanged.

In Figure 7 we model the process of the account department. After articles are delivered an invoice is generated. In a next step the invoice is compared with the receipted delivery note. Once a week outstanding payments are checked and the account department reminds related customers.

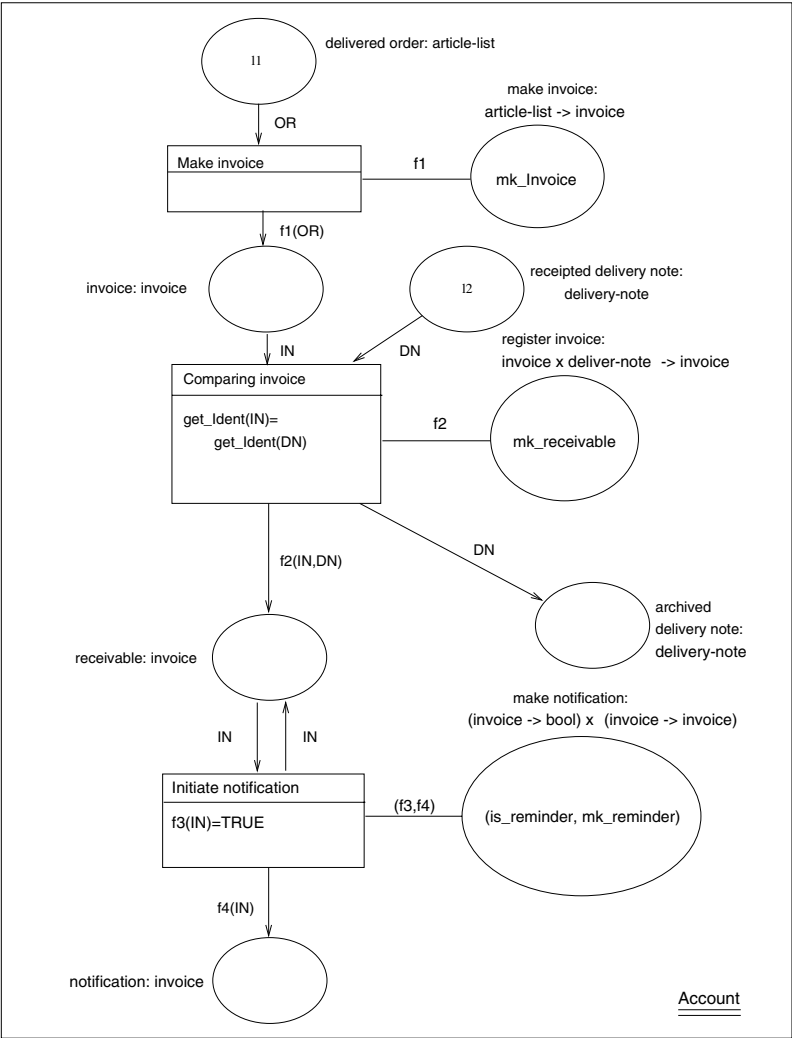


Fig.7: Account Department

4 Flexible Modeling of Business Processes

In this section we demonstrate the flexible modeling of business processes by introducing on one hand alternative activities and on the other hand a role model into the Algebraic Higher-Order Nets given in Section 3.

In Figure 7 we have modeled the usual process of the account department dealing with invoices. But in practice there are a lot of exceptions in the business process logistics, e.g. the delivery can be incomplete or not perfect with respect to their conditions. Here the customer can act in the following two ways: the order is canceled or he demands a credit note. The exceptions are introduced into the business process of the account department by an extension of the corresponding higher-order signature Account with activities concerning the cancelation and the credit note. We use the same net structure as given in Figure 7, but add tokens *mk_cancelation* and *mk_change* to place **register invoice**. Furthermore tokens (*is_cancelation*, *cancelation*) and (*is_credit*, *credit_note*) are added to place **make notification** and indicate, that in the account department cancelations and credit notes are generated in addition to the usual reminder (see Fig. 8).

The role model encompasses informations about persons, roles and corresponding activities. In [DG94] the role model is specified by a set of tables, showing roles allocated to activities and to persons, respectively. Table 1 and Table 2 give the role model for the business process **Order department** (see Figure 5). Here two different activities of splitting an order can be carried out by different persons: Maria, Marta and Martin can change the number of pieces of articles, while Maria and Markus can remove articles of an order.

Role	Activity
role 1	change articles and remove articles
role 2	
role 3	

Table 1: Roles allocated to activities

Role	Person
role 1	maria
role 2	marta
role 2	martin
role 3	markus

Table2: Roles allocated to persons

To introduce the role model into our case study logistics, we extend the higher-order signature Order by a new basic sort *person* corresponding to person working in the company. The marking of the place **splitting order** is changed

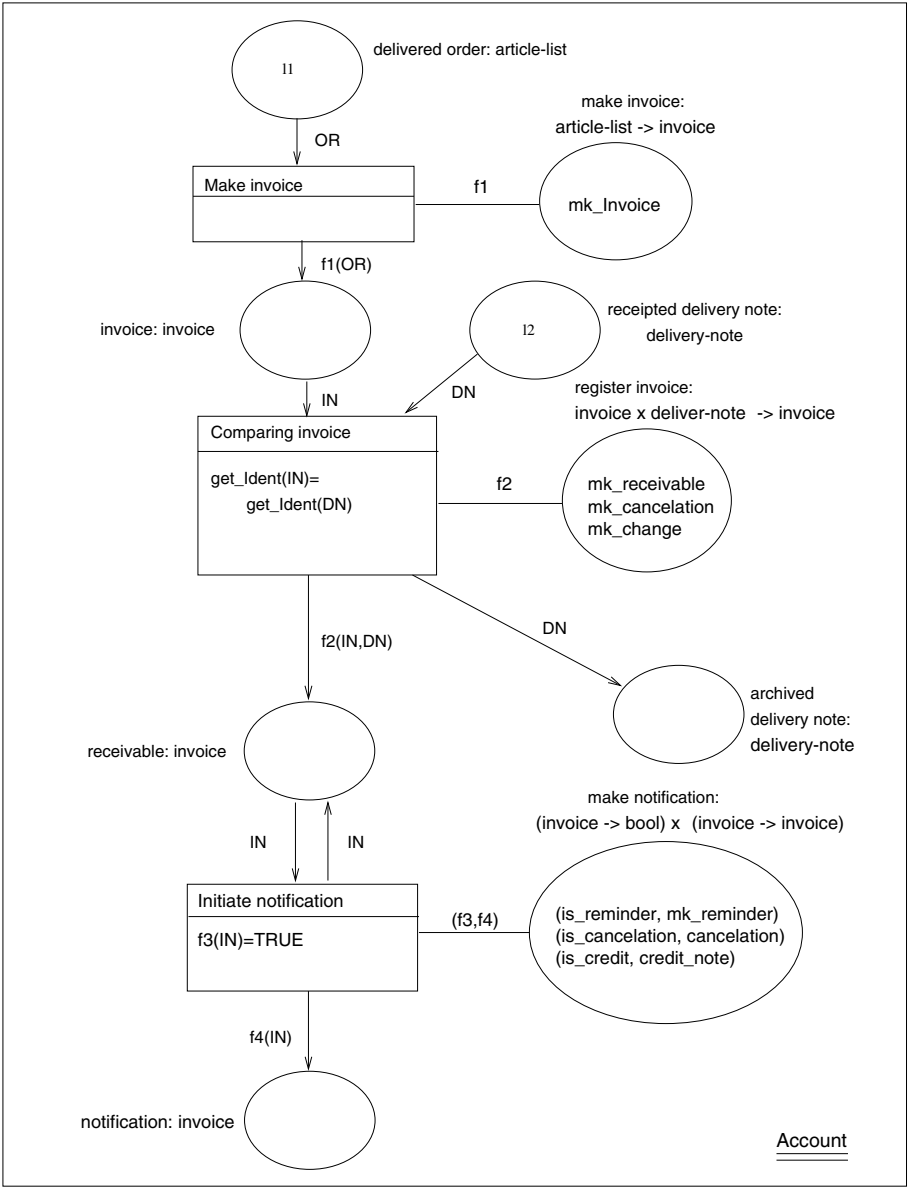


Fig. 8: Account Department with Alternative Activities

into a marking consisting of the product type $(article-list \times article-list \rightarrow article-list) \times person$ (see Figure 9) in contrast to the function type $article-list \times article-list \rightarrow article-list$ in Figure 7.

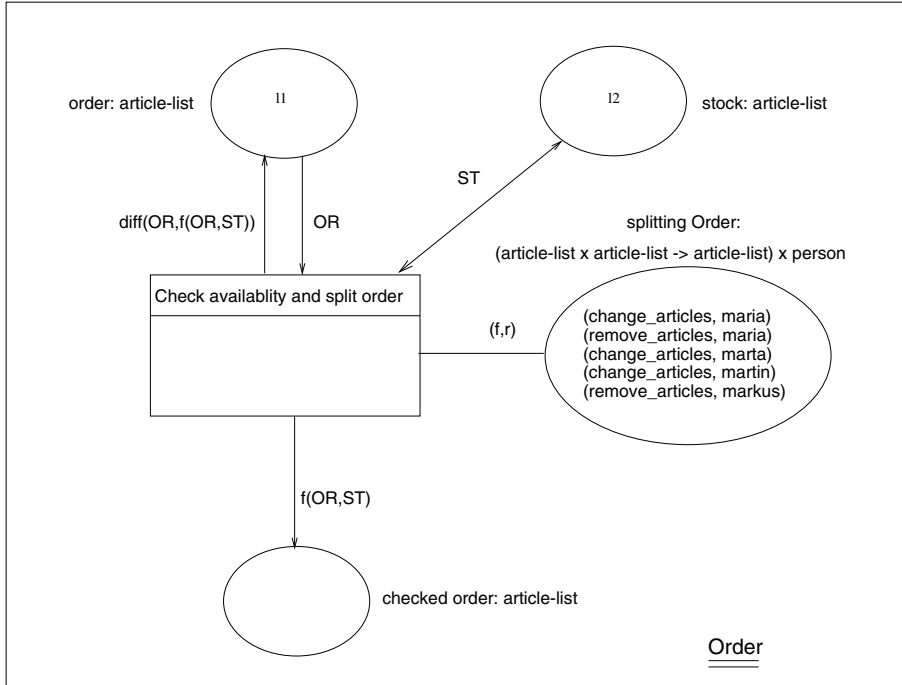


Fig.9: Order Department with Role Model

Another promising topic is the introduction of ownerships for documents to specify activity allocation constraints in the role model. These constraints can, for example, express that a person who has executed one activity to a certain document must also execute another specific activity. Similar to the procedure described above we change the basic sort of places with marking consisting of documents into a product type of persons and documents. For example we replace the basic sort of the place *order* by the product type $article-list \times person$. So the marking of the place *order* consists not only of orders given by a list of articles, but also of persons which have the ownership for modification of an order. Then in the equations of transition **Check availability and split order** we make sure that a list of articles can only be modified by a that person, which is addressed to the order.

5 Conclusion

We have presented Algebraic Higher-Order Nets leading to a more flexible and powerful modeling technique for processes. For software engineering purposes, especially in the area of business process engineering, this is an interesting approach, because Algebraic Higher-Order Nets support a flexible development of business process models without changing the net structure. We have demonstrated the use of Algebraic Higher-Order Nets in our case study logistics on one hand by the abstraction of different activities (see Section 3) and on the other hand by the extension to exceptions and roles (see Section 4). The abstract view of the overall system modeling the cooperation of users, documents and data bases and the modeling of further departments are presented in our technical report [HS02].

Another promising topic is the extension not only by function types and product types, but also by predicates, partiality and sub sorting (according to higher-order specifications in [MHKB99]). But this remains for further research.

References

- [DG94] W. Deiters and V. Gruhn. The FUNSOFT Net Approach to Software Process Management. *International Journal on Software Engineering and Knowledge Engineering*, 4(2):229–256, June 1994.
- [EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1985.
- [EPR94] H. Ehrig, J. Padberg, and L. Ribeiro. Algebraic High-Level Nets: Petri Nets Revisited. In *Recent Trends in Data Type Specification*, pages 188–206. Springer Verlag, 1994. Lecture Notes in Computer Science 785.
- [Gen91] H.J. Genrich. Predicate/Transition Nets. In *High-Level Petri Nets: Theory and Application*, pages 3–43. Springer Verlag, 1991.
- [Han97] Yanbo Han. *Software Infrastructure for Configurable Workflow System - A Model-Driven Approach Based on Higher-Order Nets and CORBA*. PhD thesis, Technical University of Berlin, 1997.
- [Hof00] K. Hoffmann. Runtime Modifikation between Algebraic High Level Nets and Algebraic Higher Order Nets using Folding and Unfolding Construction. In G. Hommel, editor, *Communication-Based Systems, Proceedings of the 3rd International Workshop*, pages 55–72. TU Berlin, Kluwer Academic Publishers, 2000.
- [HS02] K. Hoffmann and T. Schreiter. Case study logistics using algebraic higher-order nets. Technical report, Technical University Berlin, 2002.
- [Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts. Springer Verlag, EATCS Monographs in Theoretical Computer Science edition, 1992.
- [MHKB99] T. Mossakowski, A. Haxthausen, and B. Krieg-Brückner. Subsorted Partial Higher-Order Logic as an Extension of CASL. In *Recent Trends in Algebraic Development Techniques-14th International Workshop WADT'99*, pages 126–145, 1999.
- [MR95] U. Montanari and F. Rossi. Contextual nets. *Acta Informatica*, 32, 1995.

- [PER95] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic High-Level Net Transformation Systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995.
- [Sch94] A.-W. Scheer. *Business Process Engineering, Reference Models for Industrial Enterprises*. Springer-Verlag, Berlin, 1994.
- [Val98] Rüdiger Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. *Proc. of the International Conference on Application and Theory of Petri Nets*, 1998.

Rule-Based Refinement of Petri Nets: A Survey^{*}

J. Padberg and M. Urbášek

Technical University Berlin, Germany
Institute for Software Technology and Theoretical Computer Science
`{padberg,urbasek}@cs.tu-berlin.de`

Abstract. This contribution provides a thorough survey of our work on rule-based refinement. Rule-based refinement comprises the transformation of Petri nets using rules while preserving certain system properties. Petri net rules and transformations are expressed by morphisms and pushouts. This allows an abstract formulation of our notions independent of a specific Petri net class, as place/transition nets, elementary nets, predicate/transition nets etc. Hence, it is adequate to consider our approach as rule-based refinement of Petri nets in general. We have presented various results in recent years at different conferences. So this contribution gives an overview of our work in a compact form leaving out the technical details.

1 Introduction

Our work on rule-based refinement of Petri nets has been part of the research project “DFG- Forschergruppe PETRINETZ-TECHNOLOGIE”. Here we present our results concerning Petri net transformations that preserve system properties. These properties comprise safety properties as well as liveness of Petri nets. The formal foundation is expressed in a categorical way in order to achieve an approach that is valid for different net classes. We illustrate the concepts at length with several examples in different net classes. Here we concentrate on the intuitive notions and the meaning of the main results. As the underlying category theory is based on morphisms we introduce the basic definitions of the involved morphisms and of the rules as well as the transformations.

First in Subsection 1.1 a motivation for Petri net transformations is given stating the advantages of this specific approach as well as relating it to the main work of the “DFG- Forschergruppe PETRINETZ-TECHNOLOGIE”. In Subsection 1.2 we discuss the basic concepts of transformation of Petri nets and the way properties of nets can be preserved. As the last part of the introduction we relate our work to literature. Subsequently we present two extensive examples

^{*} This work is part of the joint research project “DFG- Forschergruppe PETRINETZ-TECHNOLOGIE” between H. Weber (Coordinator), H. Ehrig (both from the Technical University Berlin) and W. Reisig (Humboldt-University Berlin), supported by the German Research Council (DFG).

in Section 2. The one for place/transition (in Subsection 2.1) nets shows how a token ring is refined preserving certain safety properties as well as liveness. The second example (in Subsection 2.2) concerns the refinement of coloured Petri nets using a simplified example from the context of ETCS (European Train Control Systems). Since the main technical contents is based on morphisms, in Section 3 we give a brief outline of the involved morphism classes both for place/transition nets (in Subsection 3.1) and coloured Petri nets (in Subsection 3.2). Then in Section 4 we state the results. These results are independent of a specific net class, so we need not to differentiate between place/transition nets and coloured Petri nets. We give a short summary and discuss future work in Section 5.

1.1 Motivation

We focus on the specification process of a system in contrast to approaches concerning the verification of (distributed) algorithms as for example in [Peu02] in this book.

Stepwise modification of Petri nets has been an issue as long as they have been used for specification purposes. The need to develop a model in several steps arises as soon as large models are needed for the specification of a system. Our approach - as it is called - is based on rules and provides a visual way of expressing the development steps of a net. The transformation of a net is achieved by applying a rule to the given net. Representing changes as rules in a visual form is very intuitive and does not require a deep understanding of the mathematical theory behind it. In particular we claim that the examples in Section 2 can be followed and even worked out without the knowledge of the underlying theory. But modification alone is often not sufficient. The model describes some desired system properties that need to be guaranteed after each development step. Verification of each intermediate model requires a lot of effort and hence is cost intensive. Obviously the idea of refinement concerns the modification of nets so that system properties are preserved. Hence the verification of those properties needs only to be done once when introduced. Rule-based refinement modifies Petri nets using rules so that specific system properties are preserved. Preservation of system properties by a transformation is to be understood in the following way: If a Petri net has a certain system property (e.g. safety properties, liveness, etc.) then the transformed net has the corresponding property as well. Preservation of system properties is of interest in many applications as it allows omitting the tedious verification of system properties at different stages of the development. Our transformations are based on the double-pushout approach as first introduced for graphs. Rules are given as a span of two Petri net morphisms and its application is achieved by two pushouts (see the following Subsection 1.2); hence the name double-pushout approach.

The notion of Petri net techniques has been developed in the research project “DFG- Forschergruppe PETRINETZ-TECHNOLOGIE” and is inspired by the need to have a more application-oriented presentation of Petri nets. The definition of Petri net techniques has been one of the main research areas in this project (see the corresponding papers in these proceedings [WER99, WER01] and [ERW02]

in this book). The concept of Petri net techniques focuses on Petri nets as a specification technique, hence it comprises more than the Petri net model. It may include structuring and refinement of nets, tool support or exchange formats, or process model and exemplary methodology, so that adequate and scalable use of nets is provided for specific application domains (for more details see [ERW02]). The formal Petri net technique is the formal description of Petri net technique. Within a formal Petri net technique one important way to manipulate nets is rule-based refinement. One motivation for Petri net techniques has been the need for generalizing various notions of Petri nets. Many concepts of Petri nets are given for specific net classes, e.g. place/transition nets, elementary nets, algebraic high-level nets, predicate/transition nets, etc. only. Nevertheless, in principle they often are independent of the specific net class. So we have chosen a generic description method. Our approach to rule-based refinement is independent of specific net classes as it is formulated in a more abstract way. We employ category theory that allows coping with objects (i.e. nets) and morphisms. We characterize specific properties a category (i.e. a net class) has to provide and achieve our results on this abstract basis. In fact, the results do not hold for Petri nets only but can be transferred to other specification techniques as state charts, algebraic specifications, graphs etc. We have ensured that rule-based refinement fits into the concept of a Petri net technique. One of the main features of a technique is that the offered possibilities to cope with nets of that specific class are compatible with each other. Rule-based refinement is compatible with various structuring techniques as union and fusion (see [9]). Furthermore process models [7, 5] based on rule-based refinement are provided.

1.2 Basic Ideas

Rules and transformations of Petri nets are given by an instantiation of high-level replacement systems. These can be considered as general description of replacement systems, where a left-hand side of the rule is replaced by a right-hand side. Generally, rules and transformations fully capture the replacement and thus can define any kind of system development or modification. High-level replacement systems have been introduced in [3] as a categorical generalization of graph transformations. The application of high-level replacement systems to different domains as place/transition nets, algebraic specification [1] etc. requires a suitable category. High-level replacement systems are formulated for an arbitrary category $\mathbf{CAT}$ with a distinguished class $\mathcal{M}$ of morphisms called $\mathcal{M}$ -morphisms. Here we give the corresponding notions of replacement systems in terms of Petri nets and not on the abstract level as in [9]. In this paper we have focused on place/transition and coloured Petri nets. The same approach can be applied to other net classes, e.g. algebraic high-level nets [13].

We now explain the used notions rather informally to show that the complex category theory behind it can be omitted in applications. The reader interested in details of the theory is referred to the corresponding papers. The most important notion from category theory used almost throughout whole theory of high-level replacement systems is the notion of a pushout. In the next paragraph we

describe the pushout construction. For illustration see also the Figure 1. Next we briefly address other necessary notions as rules, transformations, property preserving rules, structuring techniques and proof rules.

Pushouts of Petri nets. Pushouts are a categorical construction that requires a commutative square and has some universal properties. Informally, a pushout can be characterized as the “largest” object, that yields a commutative square for two given morphisms without “new” nodes. A pushouts of Petri nets can be considered as a union of nets with respect to a common interface. That is, for injective morphisms we glue the nets together as illustrated in Figure 1. For coloured Petri nets the construction of pushouts has to take additional components like e.g. the data, arc inscription, transition guards, etc. into account. Basically the construction of these components is analogous to the presented construction of the net structure.

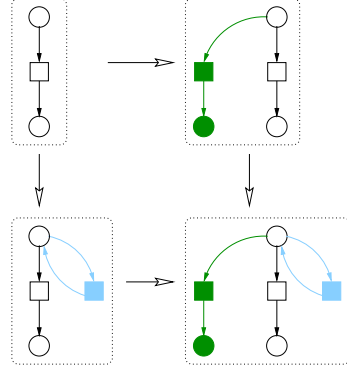


Fig. 1. An example of a pushout

Rules. A rule $r = (L \xleftarrow{k_1} K \xrightarrow{k_2} R)$ consists of the Petri nets L , K and R , called left-hand side, interface and right-hand side respectively, and two $\mathcal{M}$ -morphisms $K \xrightarrow{k_1} L$ and $K \xrightarrow{k_2} R$.

Transformations. Given a rule $r = (L \xleftarrow{k_1} K \xrightarrow{k_2} R)$ a direct transformation $N_1 \xRightarrow{r} N_2$, from N_1 to N_2 is given by the following two pushout diagrams (1) and (2). The morphisms $L \rightarrow N_1$ and $R \rightarrow N_2$ are called occurrences. The net C is called pushout complement.

$$\begin{array}{ccccc} L & \xleftarrow{k_1} & K & \xrightarrow{k_2} & R \\ \downarrow & & \downarrow & & \downarrow \\ N_1 & \xleftarrow{(1)} & C & \xrightarrow{(2)} & N_2 \end{array}$$

Informally, a rule $r = (L \xleftarrow{k_1} K \xrightarrow{k_2} R)$ is given by three nets L , K , and R . Moreover, K is a subnet of both L and R expressed by the morphisms k_1 and k_2 . Application of a rule to net N_1 is a transformation of N_1 . The transformation means replacing a subnet specified by the left-hand side of the rule with the net specified by the right-hand side. More precisely, we first identify the subnet L in N_1 . Then we delete those parts of the subnet L which are not subnets of the interface net K as well. This results in an intermediate net C , where in a further step we add the difference of R and K to the preserved subnet C to obtain the transformed net N_2 . In case the left-hand side is empty, we simply add the right-hand side to the first net. A transformation sequence $N_1 \xRightarrow{*} N_{n+1}$ between nets N_1 and N_{n+1} means that there is a sequence of $n \geq 1$ direct transformations: $N_1 \xRightarrow{r_1} N_2 \xRightarrow{r_2} \dots \xRightarrow{r_n} N_{n+1}$. In this case we also denote the sequence as $N_1 \xRightarrow{r_1, r_2, \dots, r_n} N_{n+1}$.

Structuring. There are two abstract structuring constructions in the theory of high-level replacement systems, namely union and fusion. Generally, they combine two subnets or two different nets into one. The union of two Petri nets is given with respect to a defined subnet. Union is defined as the pushout of two nets and is given by a span of morphisms. The resulting net preserves the common subnet, i.e. the source of both morphisms and keeps the rest of the two nets distinct, e.g. see Figure 1. The fusion is the gluing of two subnets within one Petri net.

Refinement. Based on the notion of rules and transformations, the general theory of high-level replacement systems has been enriched by the $\mathcal{Q}$ -theory in order to formulate abstraction/refinement morphisms of structures. These morphisms are more suitable for the stepwise development of systems. The main idea is to add an abstraction/refinement morphism to a rule going from left-hand side of a rule to the right-hand side or vice versa (see the drawing in the next paragraph). The main advantage of this approach is the fact that the additional abstraction/refinement morphisms can be defined as preserving or reflecting certain properties. This means that certain important system properties may be preserved by transformations as defined below. The general theory of rules and transformations with additional refinement morphisms has been introduced in [9] in the general framework of high-level replacement systems.

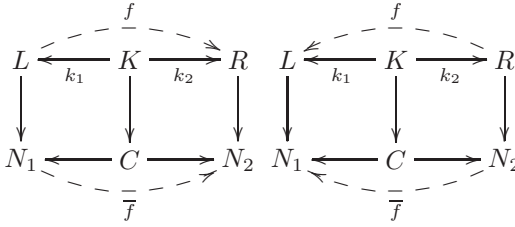
System Properties. Petri nets are an adequate specification technique for behavioral aspects of a system. So, the desired properties of the system to be specified usually concern the behavior of the model. These properties can be expressed in various ways, e.g. in terms of Petri nets (as liveness, boundedness etc.), in terms of logic (e.g. temporal logic, logic of actions etc.) in terms of relation to other models (e.g. bisimulation, correctness etc.) and so on. Up to now we have focused on liveness of Petri nets and on safety properties in the sense of temporal logic. Liveness of nets means that no deadlock and even livelock of a net can occur, i.e. there always exists a firing sequence which enables any chosen transition from any reachable marking. A safety property is expressed by a logic formula stating facts about markings of a net. A formula is given in terms of numbers of tokens on places. For a place/transition net the static formula $2\mathbf{d} \wedge 3\mathbf{a}$ is true for a marking m where at least 2 tokens are present on the place $\mathbf{d}$ and at least 3 tokens on the place $\mathbf{a}$. The always operator $\Box$ in a safety property $\Box(2\mathbf{d} \wedge 3\mathbf{a})$ requires that the static formula $(2\mathbf{d} \wedge 3\mathbf{a})$ is true for all reachable markings from m . The adaptation of safety properties to coloured Petri nets has to be made due to the different structure of the marking.

Property Preserving Rules. A pair (r, f) is a property preserving rule, if $r = (L \xleftarrow{k_1} K \xrightarrow{k_2} R)$ is a rule with morphisms $k_1, k_2 \in \mathcal{M}$ and with

- either a property preserving morphism $f : L \dashrightarrow R$ s.t. $f \circ k_1 = k_2$.
- or a property respecting morphism $f : R \dashrightarrow L$ s.t. $f \circ k_2 = k_1$.

According to the notion of property preserving morphisms and rules, we can now define property preserving transformations. The general idea is that the application of a rule that preserves properties leads to a net transformation that also preserves these properties.

Property Preserving Transformations. Given a property preserving rule $(r = (L \leftarrow K \rightarrow R), f)$ with $f : L \rightarrow R$ being a property preserving morphism ($f : R \rightarrow L$ being property respecting, respectively). Then the direct transformation $N_1 \xrightarrow{(r,f)} N_2$ is a property preserving transformation¹ with a property preserving (respecting, respectively) morphism $\bar{f} : N_1 \dashrightarrow N_2$ ($\bar{f} : N_2 \dashrightarrow N_1$, respectively). The graphical representation of such transformations is depicted below.



Property preserving transformations represent a strong theoretical result with an obvious impact to the applications. The interpretation of the results is usually stated in form of so-

called proof rules. The proof rule indicates the property (stated under the line) which can be derived for a system when certain assumptions (stated above the line) are fulfilled. For the property preserving transformations $N_1 \Rightarrow N_2$ we have the following proof rule:

(r, f) is a property preserving rule; N_1 satisfies the corresponding property N_2 satisfies the property too

The following property introducing transformations are a special kind of general transformations with the empty set on the left-hand side of the rule.

Property Introducing Rules and Transformations. Given a rule $r = (\emptyset \leftarrow \emptyset \rightarrow R)$ such that R satisfies certain property, then the rule is called property introducing rule yielding property introducing transformations $N_1 \Rightarrow N_2$ and the following proof rule.

$r = (\emptyset \leftarrow \emptyset \rightarrow R)$ is a rule; R, N_1 satisfy a certain property ² N_2 satisfies the property too

¹ provided the morphisms satisfy certain assumptions

² sometimes this condition can be relaxed in such a way that there is no assumption on N_1 made

The proof rules are a powerful expression of the properties of transformations. They are very useful for designers as they abstract from this theory and focus on applications only. Therefore the complicated theory of high-level replacement systems is applicable in practice.

1.3 Related Work

As the reader may already have noted we have in this paper two ways of referring to literature. We have split the references, by having listed our papers on rule-based refinement separately and by referring to other publications as usual. References of our papers on rule-based refinement are given by numbers, the other as an abbreviation of the authors names.

Intuitively, a safety property expressed by a temporal logic formula in the sense of [MP92] means that “nothing bad” can happen in the system, and is thus important for correctness. The combination of transformations with the preservation of safety properties is relevant in software engineering as the verification of such properties in large and complex systems is often necessary, but very complicated (if possible at all) and thus expensive. Thus, it is desirable to state and prove safety properties at an early state and to preserve them throughout the subsequent development. In the area of low-level Petri nets there are many contributions concerning verification with temporal logic [DDGJ90, BS90, HRH91] and refinement [BGV91], [DM90, GG90, BDH92]. In the area of high-level nets, verification [Jen94, Sch96] is much more difficult and even more the compatibility of system properties with refinement. One main problem of verification in formal software engineering can be described by the following demand: Rigorous software development requires continuous verification during all phases of the software development process. Nevertheless, resources are restricted and an entirely new verification at each step is usually considered to be too expensive and time consuming. Thus, vertical structuring techniques should preserve verified properties. Refinement of nets in a way that liveness is preserved is another thoroughly investigated area of Petri nets. Liveness for Petri nets states the fact that in a net all transitions can fire from all reachable markings. Nevertheless, the combination with stepwise modification of nets based on replacement rules has been investigated in [8, 16] only. This approach is in line with previous publications concerning rule-based modification preserving safety properties of nets, see e.g. [13, 15, 14]. A related idea has been pursued in [DA92], where various reduction methods for ordinary nets have been proposed. These methods operate on the net structure and also preserve liveness, by basically shortening paths in the net. The analysis in this case is based on a Petri net reduction. One disadvantage is that the whole Petri net has to be developed entirely and only after the design of a Petri net is finished the analysis can start. Similar remarks hold for reductions presented by other authors in [ES91, Esp94, CT90, FWL]. Authors also use reductions algorithms to decrease the size of the net. J. Esparza, etc. in [ES91, Esp94] investigated the class of free-choice Petri nets, J. Favrel in [FWL] is focused on coloured Petri nets based on the results for generalized Petri nets proposed earlier. In [CT90] the certain subclass of Petri nets, called

regular blocks, is investigated. A work on synthesis methods preserving liveness properties has also been made. In the area of free-choice Petri nets the synthesis preserving liveness properties was described along the reduction methods by J. Esparza, etc. in [ES91, Esp94]. Similar ideas not only for P/T Petri nets can be found in [BGV91, Sou91]. In the area of workflow modeling the notion of soundness which comprises liveness has turned out to be of special importance. In a row of papers, see e.g. [vdA97, vdA98, vdAtH98], v.d. Aalst has stated important results concerning composition of sound nets [vdA98].

In cited papers, the synthesis and/or reduction rules are depicted on more intuitive level. Although our approach is very close to these papers, formalization and net class make an important difference.

2 Examples for Rule-Based Refinement of Petri Nets

In order to illustrate the concepts of rule-based refinement described in Section 1.2 we will present stepwise development of two small systems in this section. The first example is a development of a token ring based on place/transition nets. The another one is a development of a train control system based on coloured Petri nets. A short version of this one can be found in [11]. Together with the development of the systems we gradually introduce system properties as liveness or safety properties. In further development steps these system properties are preserved and thus do not have to be proven again. The process of developing the system is technically based on rules and transformations in the sense of [2]. More formal introduction of used notions and morphisms follows in Section 3.

2.1 Place/Transtion Nets: Token Ring

A token ring is a classical example of a communication protocol between several computational units (CU) on a shared medium. All computational units are connected such that they form a cycle. There is a so-called token sent from one unit to another. The unit which possesses the token can decide either to pass the token through to the next unit or to keep the token and send a message. The sent message propagates through the medium from one unit to another reaching the sending unit sooner or later. Before the sending unit can pass the token to the next unit, the sent data has to be removed from the medium. When a unit receives a data, it retransmit a message to the next unit.

In order to model the token ring system with rule based refinement, the usage of place/transition nets as a basic class of Petri nets is sufficient. The necessary conditions this system has to have are:

1. There exists a unit which possesses a token.
2. The token cannot be doubled, i.e. only one unit has a token.
3. The system is live (without livelocks and deadlocks).

The first proposal of a token ring system with three computational units is given in the Figure 2. This proposal is really a very coarse model of a system.

Nevertheless, it can be refined to get a model which is more suitable. The three conditions of the system stated above can be expressed in this model as follows.

1. $\square(\mathbf{CU}_1 \vee \mathbf{CU}_2 \vee \mathbf{CU}_3)$
2. $\square((\mathbf{CU}_1 \Rightarrow \neg\mathbf{CU}_2 \wedge \neg\mathbf{CU}_3) \wedge (\mathbf{CU}_2 \Rightarrow \neg\mathbf{CU}_1 \wedge \neg\mathbf{CU}_3) \wedge (\mathbf{CU}_3 \Rightarrow \neg\mathbf{CU}_1 \wedge \neg\mathbf{CU}_2))$
3. The net is live.

The first two conditions can be shortened by using the exclusive-or operator (non-equivalence) as: $\square((\mathbf{CU}_1 \text{ xor } \mathbf{CU}_2 \text{ xor } \mathbf{CU}_3) \wedge \neg(\mathbf{CU}_1 \wedge \mathbf{CU}_2 \wedge \mathbf{CU}_3))$.

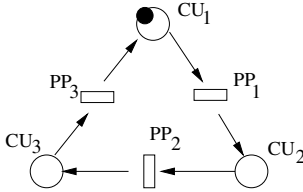


Fig. 2. Token ring - first proposal

Modeling within a rule-based framework starts with a special kind of a rule, which introduces a first proposal of the model. In our example such a rule looks like the one in the Figure 3. This rule is a so-called si- and li-rule, short for safety introducing and liveness introducing rule, respectively (for description see Section 3.1).

The rule introduces a system, which is live and has a safety- property:

$\square((\mathbf{CU}_1 \text{ xor } \mathbf{CU}_2 \text{ xor } \mathbf{CU}_3) \wedge \neg(\mathbf{CU}_1 \wedge \mathbf{CU}_2 \wedge \mathbf{CU}_3))$. The dashed arrow from the left-hand side to the right-hand side expresses the existence of a place preserving morphism between these nets (as formally introduced in Section 3.1).

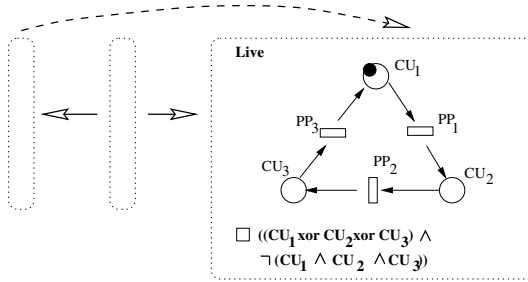


Fig. 3. Introducing the token ring

The proposal of the model does not express the functionality of the computational units. It only expresses the fact that there is a cycle with three units which are connected together. To model full functionality, the first coarse model must be decomposed. The decomposition is done via transformations based

on rules. The modeled system has a safety property and is live. We would like to preserve the properties during transformations and omit the check of these properties for final system. We employ so-called liveness preserving rules, which also translate safety properties in some sense. The special kind of transition refinement rules called lp-rules are used (see Section 3.1). We can then perform three transition refinements, one for each unit $i = 1 \dots 3$, as sketched in the Figure 4. The dashed arrow means that there exists a collapsing morphism going from the right-hand side net to the left-hand side one in the rule. Surely, in the case of the refinement of $\mathbf{PP}_1$ both places $\mathbf{CU}_1$ in the corresponding rule (for $i = 1$) are marked.

Explanation of used abbreviations:

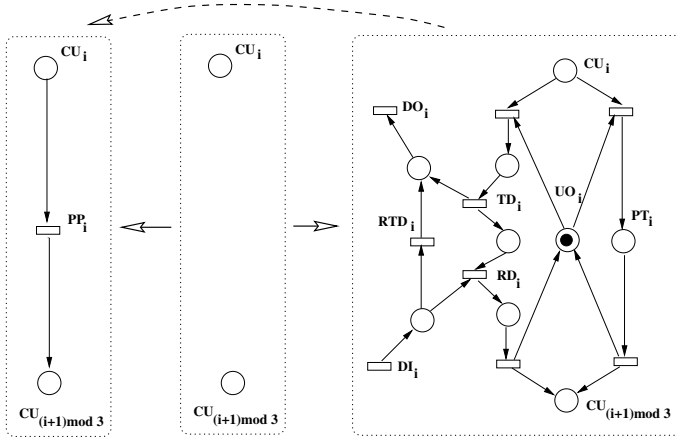


Fig. 4. Transition refinements

- PT** ... passing a token to next unit, no transmission of data
TD ... transmitting data
RD ... removing data from the medium
RTD... retransmitting data received via medium
UO ... unit not operating
DI ... receiving data
DO ... sending data

The system after refinement is depicted in the Figure 5. Performed transition refinements preserve liveness. The safety property is transformed to:

$$\square(((\mathbf{CU}_1 \vee \neg \mathbf{UO}_1) \text{ xor } (\mathbf{CU}_2 \vee \neg \mathbf{UO}_2) \text{ xor } (\mathbf{CU}_3 \vee \neg \mathbf{UO}_3)) \wedge \neg((\mathbf{CU}_1 \vee \neg \mathbf{UO}_1) \wedge (\mathbf{CU}_2 \vee \neg \mathbf{UO}_2) \wedge (\mathbf{CU}_3 \vee \neg \mathbf{UO}_3)))$$

as lp-rules preserve liveness and transform safety properties (as formally explained in the Section 3.1). The transformation of the safety property expresses the fact that when certain computational unit $\mathbf{CU}_i$ possessed the token in the simple original system then an adequate expression after refinement is that either a token is on the place $\mathbf{CU}_i$ or the corresponding added subnet (refinement of $\mathbf{PP}_i$) is running, i.e. there is no token on the place $\mathbf{UO}_i$. The last step in our example is to model a shared medium for data transmission by connecting of appropriate transitions in the Figure 5. For this reason it is necessary to glue a transition $\mathbf{DI}_i$ of one unit with a transition $\mathbf{DO}_{(i+1) \bmod 3}$ of the next unit. The gluing can be obtained using three l-transition gluing rules as sketched in the Figure 6.

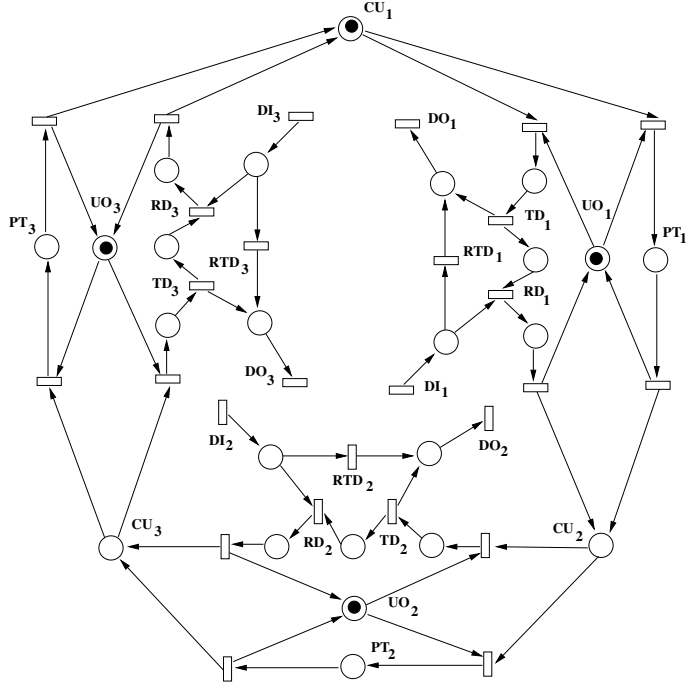


Fig. 5. Refined system

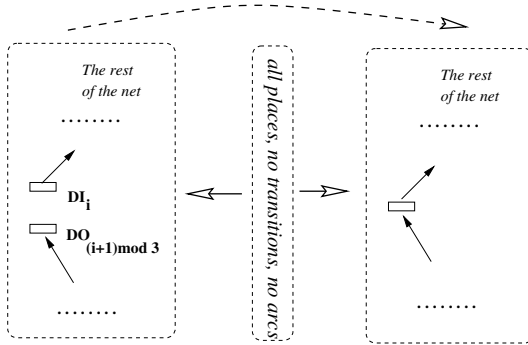


Fig. 6. Transition gluing

The dashed arrows express the fact that there exists a l-gluing morphism (and therefore also plain gluing morphism) going from the net on the left-hand side to the one on the right-hand side. In order to simplify the figure, the transitions after gluing are without labels. The final system then is shown in the Figure 7. Liveness and

safety properties are preserved via l-gluing morphisms. So, the model developer does not have to check liveness and safety-property for the final model. These properties have been preserved (transformed) during the transformation process and are therefore valid for the final net. Surely, this may save time and effort needed to check system properties, especially in the case of large and complex systems. This approach was also used for rule-based modeling of a large medical information system as presented in [4, 9]. The model developer can, of course, continue in the detailed specification of the model, if necessary.

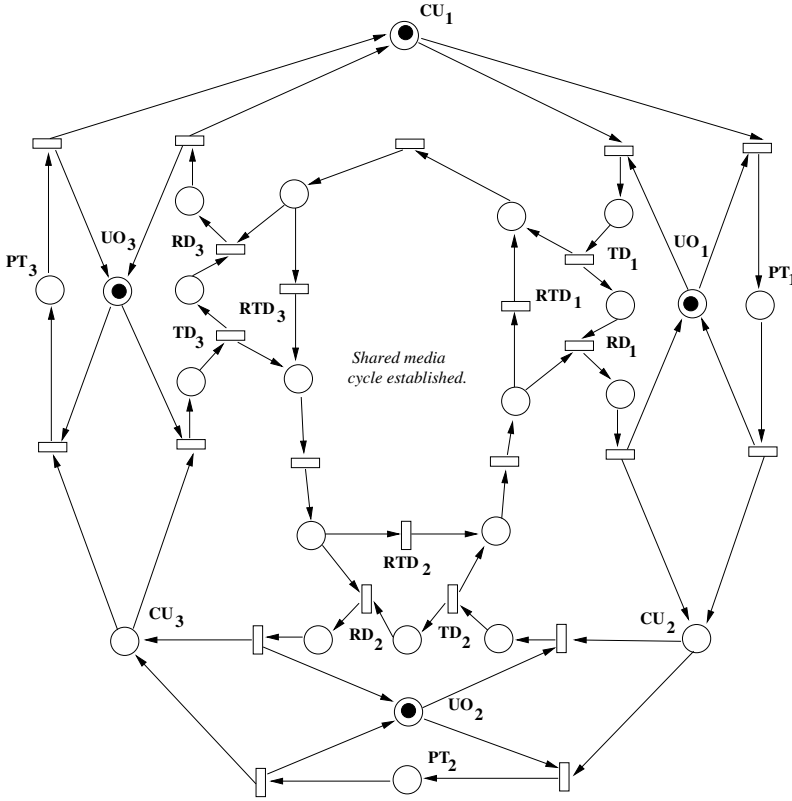


Fig. 7. Final system

2.2 Coloured Petri Nets: Stepwise Development of a Train Control System

High-level nets can be considered as the integration of process and data type description. Some of the most prominent classes are coloured Petri nets [Jen92, Jen94, Jen97], Predicate/Transition nets [GL81, Gen91] and algebraic high-level nets [Vau87, Rei91, 10]. Coloured Petri nets are the focus of this subsection, since they are widely known and constitute a very popular class of high-level Petri nets. The practical relevance of coloured Petri nets is considered to be very high, not at least due to the successful tool Design/CPN [JCHH91]. The presented example concerning train control has been inspired by a case study of a level crossing in a large research project³ [Ehr97, Jan97], but is of course incomparable with respect to complexity.

³ DFG-Schwerpunktprogramm “Integration von Techniken der Softwarespezifikation für Ingenieurwissenschaftliche Anwendungen”, see also WWW-page <http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/>

Simple Model In our example we develop a coloured Petri net concerning a train's passage of a level crossing, which is one of several situations the train has to meet while traveling to its destination. Figure 8 depicts a simple coloured Petri net, called CPN_1 , of the train's behavior in this situation. The used data enclosed in the upper right corner are

1. The train's data consisting of its identity (integer), the distance to a critical section of the track (DISTANCE), and some other data (string) like the schedule or the number of wagons which are not relevant for this example.
2. The distance data is a positive interval of integers, where the lower limit denotes the minimal distance at which the train has to start braking in order to come to a full stop at the critical section. The upper limit is considered to be infinite represented by the value w .
3. The state of the critical section, resp. level crossing, being either safe or unsafe.
4. A black token for requesting the state.

Initially, the train travels along with no knowledge of the next critical section. When it approaches a level crossing, the remaining distance is set by firing of the transition *approach to level crossing* in the Figure 8. Consequently, the train is alarmed and requests the state of the level crossing by the transition *request state of level crossing*. The level crossing state can be either safe or unsafe modeled by the arc inscription of the transition *set state*. Until the state of the level crossing is set, the train awaits the result. There are three possibilities of action: The train can still come nearer to the level crossing modeled by *coming nearer to level crossing* decreasing the distance to the level crossing; or it has already reached the point at which it must begin to brake in which case the transition *braking* is enabled. Or — provided the state of the level crossing is safe — it can proceed regularly by the transition *proceed* resulting in its passage of the level crossing.

In case of braking, it comes to a full stop before the level crossing. The train has to wait until the state of the level crossing is made safe (by the driver). Subsequently, the train may pass. Finally, it exits the level crossing returning to its initial state. Together with the net there is given a safety property, depicted at the bottom line of the Figure 8. For security reasons it should always be guaranteed that the state of the level crossing is safe when the train passes. This fact is expressed as an appropriate logic formula. Let $\alpha \in C(\text{passing level crossing}) = \text{TRAIN}$ be a variable of the colour of the place *passing level crossing*. The safety property $\Box(\alpha, \text{passing level crossing}) \implies (\text{safe}, \text{state of crossing})$ in the Figure 8 means that “In any case, if a train is passing the level crossing, the level crossing is safe”. And it is satisfied for the initial marking based on the initialization function. Intuitively, we can argue as follows: The formula $(\alpha, \text{passing level crossing}) \implies (\text{safe}, \text{state of crossing})$ is satisfied in the initial state. Moreover, the two transitions *proceed* and *making l.c. safe* are the only ones to put a token on *passing level crossing*, which makes the premise of the formula true.

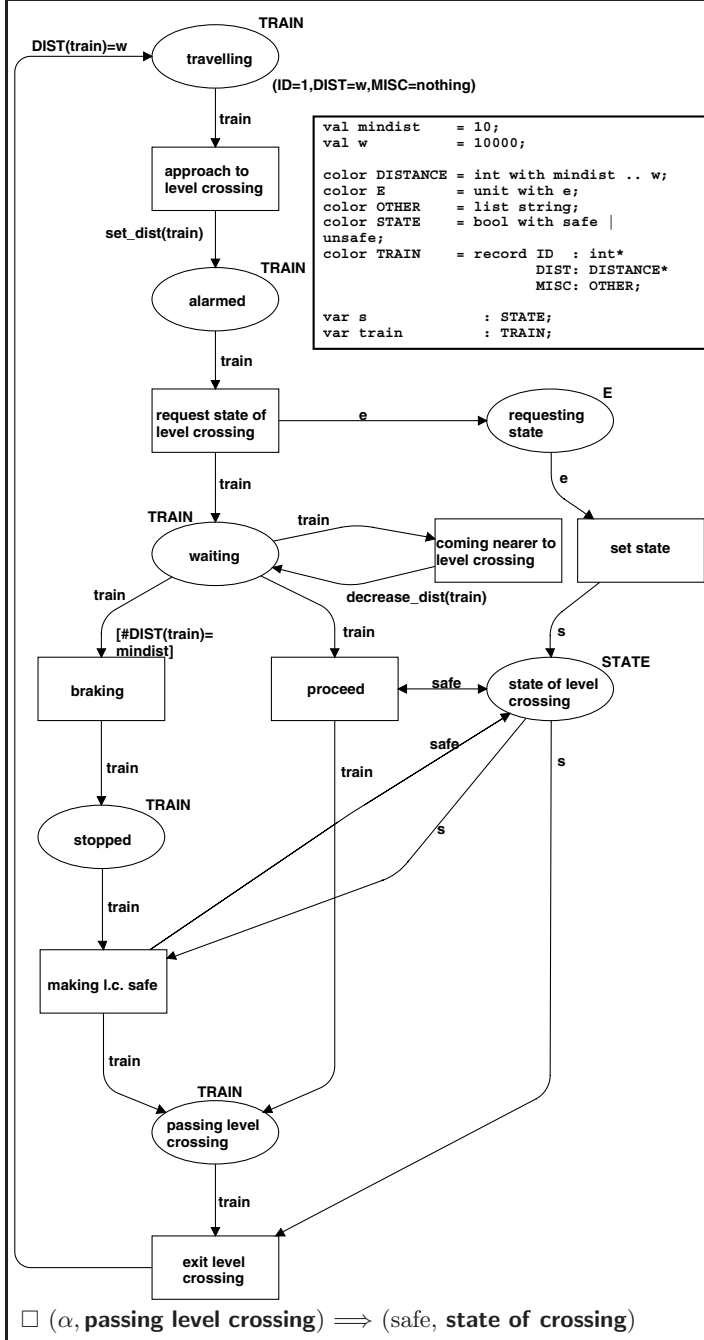


Fig. 8. CPN_1 Train passing a level crossing

Both transitions put a token coloured ‘safe’ back to the place **state of crossing**, such that also the conclusion is fulfilled. Next, the only transition (namely *exit*) withdrawing that token from **state of crossing** also removes the train from the place **passing level crossing** and leads to the initial marking. Finally, as the net is cyclic and each cycle satisfies the formula $(\alpha, \text{passing level crossing}) \implies (\text{safe}, \text{state of crossing})$, this formula is always satisfied, i. e. for each reachable marking. In the following we will modify, respectively enrich this model CPN_1 by complexity. Of course, we also want the resulting net to satisfy the important safety property. In other words, the already proven safety property should be preserved by the modification. Thus we employ safety property preserving and introducing transformations.

Developing Steps We are now going to enhance the model by gates for detaining road traffic from crossing the tracks. This is done by adding the coloured Petri net description of gates and connecting it to the first net CPN_1 . The addition of gates to the simple model is realized by application of the rule r_{gates} depicted in the Figure 9.

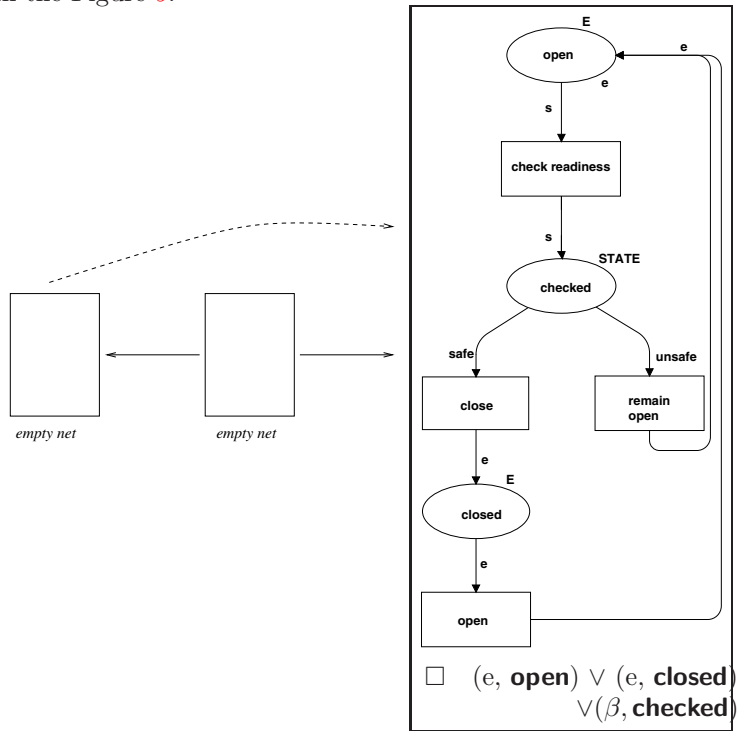


Fig. 9. Rule r_{gates} for adding gates

Initially, the gates are open. There can be a check of readiness, see transition *check readiness*, which results in the token coloured safe, if the gates are ready, or unsafe if they are not. Depending on this result, the gates may either remain

open, or they can be closed. If so, they might be opened again leading to the initial state. Again, there is a safety property given together with the net. For $\beta \in C(\mathbf{checked}) = STATE$ the safety property $\square (e, \mathbf{open}) \vee (e, \mathbf{closed}) \vee (\beta, \mathbf{checked})$ informally states, that the state of gates is either open or closed or checked. Obviously, this safety property holds in the net on the right-hand side of the Figure 9. As the rule r_{gates} not only adds a subnet but also *introduces a new safety property* we call it *si-rule*, short for safety introducing. Applying this rule, of course, we do not want to lose the safety property “In any case, if a train is passing the level crossing, the level crossing is safe”, which we already proved for the first net. Moreover, the introduced safety property should be propagated to the net resulting after applying the rule r_{gates} . The preservation of the old safety properties and the satisfaction of the newly introduced safety properties is stated in the next section. This means that the net gained by application of the rule r_{gates} to the first net satisfies the safety property introduced by the rule and also the originally stated safety property in the first net.

The application of the rule r_{gates} to the first net CPN_1 yields a disconnected coloured Petri net CPN_2 . Intuitively, we would suppose the two transitions **check readiness** and **set state** to be somehow synchronized, i.e. the state should conform to the result of the check. Therefore, we are now going to *glue these transitions* together. This is achieved by the rule r_{glue} in the Figure 10. It is compatible with the safety properties as stated in the next section meaning that in the resulting net still all safety properties hold. The resulting net CPN_3 after applying r_{glue} to the net CPN_2 .

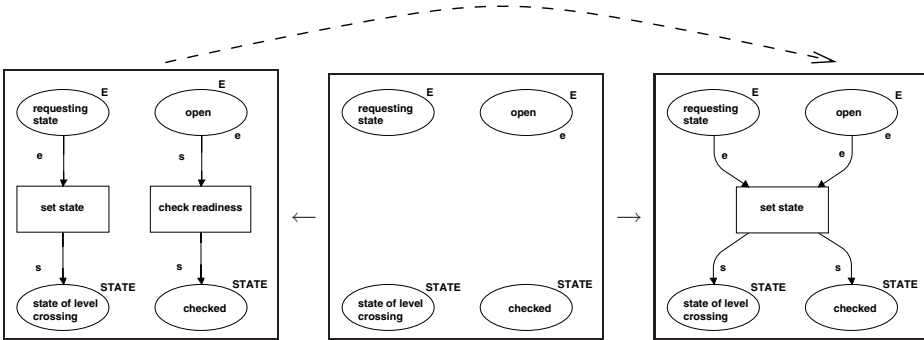
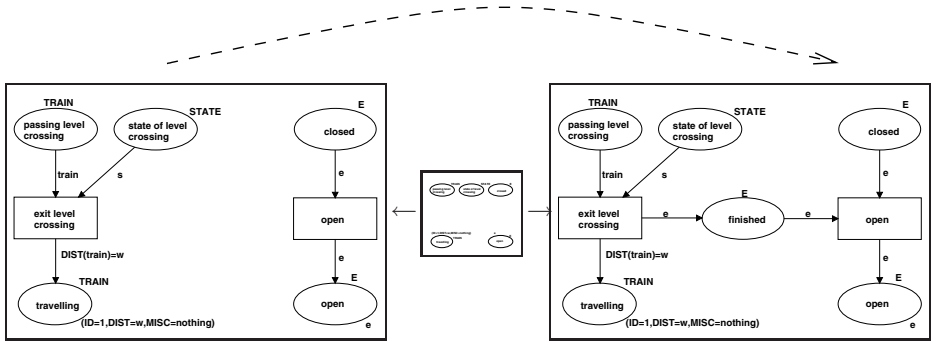
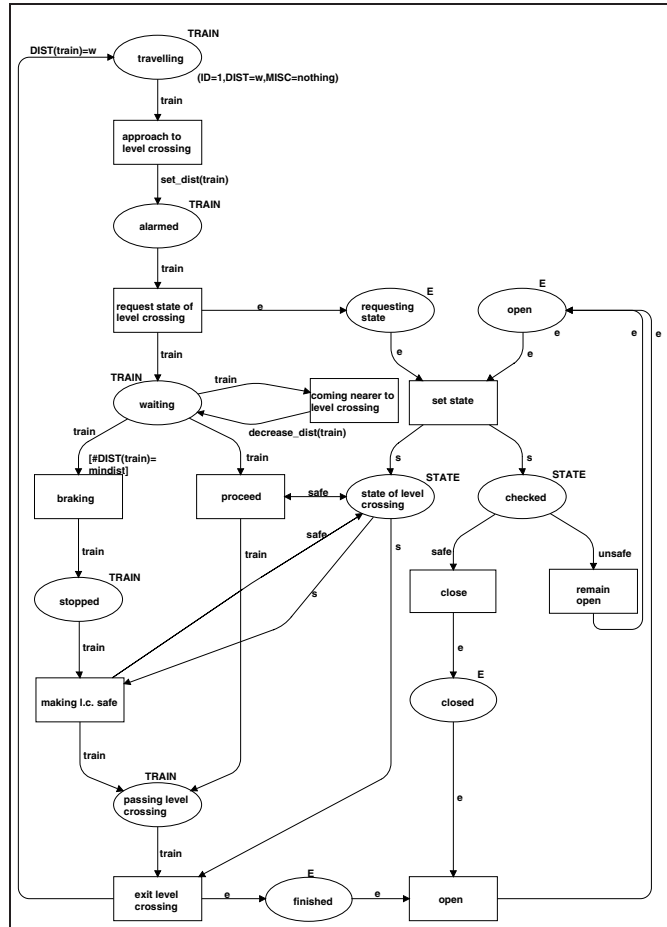


Fig. 10. Rule r_{glue} for gluing transitions

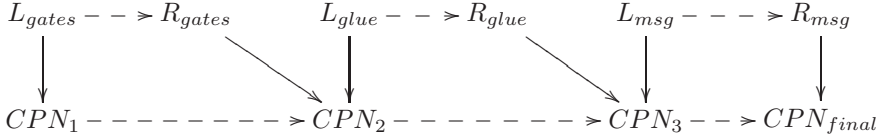
Last, we want the gates to open only after the train has left the level crossing. Therefore, we send a message to the gates indicating the exit of the train. Only then, the gates may be opened. The rule r_{msg} in the Figure 11 takes the two corresponding transitions **exit level crossing** and **open** and connects them by a place **finished**. This rule does not change the pre- and postdomains of the neighboring places and thus it is *place preserving*. By the result presented in the following section it also preserves safety properties.

Fig. 11. Rule r_{msg}

The final net CPN_{final} , which is the result of applying the rule r_{msg} to CPN_3 is presented in Figure 12. It models a train's passage of a level crossing where the opening and closing of the gates is taken into account.

Fig. 12. CPN_{final} Final net

Summarizing, we have developed the model as follows: starting with CPN_1 in Figure 8 we iteratively applied the rules r_{gates} , r_{glue} , and r_{msg} depicted in the Figures 9, 10, and 11, respectively. A graphical representation of this transformation sequence is given below, where all arrows depict (different kinds of) morphisms:



L_r and R_r denote the left-hand side of the rule r , resp. the right-hand side of r and we omitted the interface nets for brevity. More precisely, we have gradually developed a coloured Petri net modeling a train's passage of a level crossing. We proved an important safety property that "In any case, if a train is passing the level crossing, the level crossing is safe" for the first net CPN_1 . This safety property has been preserved by the subsequent refinement of the first net. Additionally, we introduced a new safety property for the gates, which then has been preserved as well. The main advantage of our approach is that we do not have to prove the safety properties in the final net but just for the first net and for the rule introducing a new safety property. By this we could add further safety properties, which were preserved by transition gluing as well as place preserving rules.

3 Morphism Classes and System Properties

In next subsections we introduce used notions on a formal basis. The achieved results concerning horizontal structuring, safety property and liveness preservation are summarized in the Section 4.

3.1 Place/Transition Nets

In this section we summarize main notions about the class of place/transition nets and its properties. The proper definitions are beyond the scope of this contribution. Presented results can be found fully elaborated in [6, 8, 17]. First we give a short intuition of the underlying basics. The precise definition can be found in [8]. Next we introduce formally different classes of net morphisms. We will also define the notions of specific systems properties such as safety properties and liveness. Finally, we list all the features which the introduced morphisms classes have. Technical details and proofs can be found in the cited papers.

Preliminaries. Here we use the algebraic notion of place/transition systems (shortly p/t systems) as introduced in [MM90]. Hence a place/transition system is given by the set of transitions and the set of places and the pre- and

post-domain function. $N = (T \xrightleftharpoons[post]{pre} P^\oplus, \widehat{m})$, where $P^\oplus$ is the free commutative monoid over P , or the set of finite multisets over P and markings are elements of $P^\oplus$, especially $\widehat{m}$ stands for the initial marking. The category **PTSys** of place/transition nets has plain morphisms, which consist of two functions $f = (f_T, f_P)$. Elements of the free commutative monoid over P are given mostly as finite linear sums, hence we use the extended operations $\oplus, \ominus, \leq, < >$ etc. Moreover we need to state how often a basic element is given within an element of the free commutative monoid. We define this for $p \in P$ and $w \in P^\oplus$, $\lambda \in \mathbb{N}$ with $w|_p = \lambda p \in P^\oplus$. This is extended to subsets $P' \subseteq P$ with $w|_{P'}$ so that there is $w = w' \oplus w''$ with $w|_{P'} = w' \in (P \setminus P')^\oplus$ and $w'' \in P'^\oplus$. The notions $\bullet e, e \bullet$ stand for sets of input and output elements of an element e (transition or place) of a net. For the firing of Petri nets we use the usual notations, that is $m[t]$ means marking m enables transition t ; $m[t]m'$ denotes firing of transition t under marking m yielding marking m' . Reachable markings from m are given by the set $[m]$. Paths of firing steps are denoted by arrows with the following special meanings $m \xrightarrow{*} m'$ arbitrary, but possibly no firing steps from m to m' . $m \xrightarrow{t_1, \dots, t_n} m'$ denotes firing steps using transitions $t_1, \dots, t_n$. A path $m \xrightarrow{\dots, t, \dots} m'$ denotes firing steps where the transition t occurs.

Sequentially Independent Transitions [17]. Given $N = (P, T, pre, post, \widehat{m})$ a place/transition system. The set of transitions $T_I = t_1, \dots, t_n \subseteq T$ is called set of sequentially independent transitions if the following holds:

For every reachable marking $m \in [\widehat{m}]$ such that some transition $t \in T_I$ is enabled, there exists a path $m \xrightarrow{t'_1, \dots, t'_n} m'$ satisfying following conditions:

1. $\sum_{i=0}^n pre(t_i) \leq m'$
2. $m' \oplus \sum_{i=0}^n (post(t_i) \ominus pre(t_i)) \geq m \ominus pre(t) \oplus post(t)$
3. $t'_1, \dots, t'_n \notin T_I$.

The Item 2 states that every transition sequence $t_1 \dots t_n$ fireable from $m^1 : m[t]m^1 \xrightarrow{t_1 \dots t_n}$ is also fireable from $m' : m' \xrightarrow{t_1 \dots t_n} m^2$.

Basic and Derived Place/Transition System-Morphisms and Categories. Given $N_i = (P_i, T_i, pre_i, post_i, \widehat{m}_i), i \in \{1, 2\}$ two place/transition systems.

A morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ with functions $f_P : P_1 \rightarrow P_2$ and $f_T : T_1 \rightarrow T_2$ is called

loose [6] if the following embedding conditions hold for all $t \in T_1$ and $p \in P_1$:

1. $f_P^\oplus(pre_1(t)) \leq pre_2(f_T(t))$ and $f_P^\oplus(post_1(t)) \leq post_2(f_T(t))$
2. $f_P^\oplus(\widehat{m}_1|_p) \leq \widehat{m}_2|_{f_P(p)}$

transition preserving [6] if it is loose and the following condition holds:

3. $f_P^\oplus \circ pre_1 = pre_2 \circ f_T$ and $f_P^\oplus \circ post_1 = post_2 \circ f_T$

- A transition preserving morphism is called **marking strict** [6] or **plain** [8] if $f_P^\oplus(\widehat{m}_1|_p) = \widehat{m}_2|_{f_P(p)}$ for all $p \in P_1$.
- It is called **strict** if it is marking strict and injective.

place preserving [6] if it is a loose morphism and the following place preserving conditions hold:

4. $\bullet(f_P(p)) = f_T^\oplus(\bullet p)$ and $(f_P(p))\bullet = f_T^\oplus(p\bullet)$ for all $p \in P_1$
 where $\bullet p = \sum_{t \in T} \text{post}(t)(p) \cdot t$ and $p\bullet = \sum_{t \in T} \text{pre}(t)(p) \cdot t$ define the pre and post sets of p .
5. f_T and f_P are injective
6. $\widehat{m}_2|_{f_P} = f_P^\oplus(\widehat{m}_1)$

transition gluing [6] if it is a loose morphism and the following holds:

7. f_P is isomorphism
8. $f_P^\oplus(\widehat{m}_1) = \widehat{m}_2$
9. f_T is surjective s.t. $\text{pre}_2(t_2) = \sum_{t_1 \in f_T^{-1}(t_2)} \text{pre}_1(t_1)$
 with $t_1 \in T_1$ and $t_2 \in T_2$. *post* analogously.

l-transition gluing [17] if it is transition gluing and the set of transitions $T_G = \{t \in T_1 \mid \exists t' \neq t : f_T(t) = f_T(t')\}$ (the set of transitions not mapped bijectively) is a set of sequentially independent transitions.

Example 1. In the Figure 3 the dashed arrow expresses a special case of a place preserving morphism. The domain of the morphism is empty, therefore all the places are preserved and new places are added.

In the Figure 6 the transition gluing morphism is used. It glues two transitions to one and keeps the rest of the net unmodified. This morphism is also l-transition gluing as it glues transitions with empty preset and postset respectively.

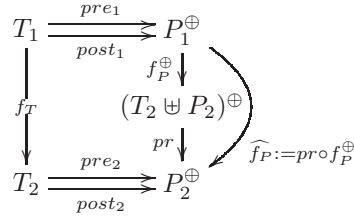
In [8] it is shown that a special type of transition refinement preserves liveness in Petri nets. The idea is based on abstracting morphisms, which are closely related to vicinity respecting morphisms (introduced in [DM90]). Abstracting morphisms allow abstracting transitions and places to a single transition. A certain subclass of abstracting morphisms, called collapsing morphisms, allows the description of transition refinement as collapsing of a special subnet (called live in-out cycle) to one transition. The formal introduction of abstracting and collapsing morphisms follows.

Abstracting Morphisms [8]. Given two place/transition systems $N_i = (T_i$

$\xrightarrow[\text{post}_i]{\text{pre}_i} P_i^\oplus, \widehat{m}_i)$ for $i = 1, 2$. An **abstracting morphism** $f : N_1 \rightarrow N_2$ is given

by $f = (f_T, f_P)$ with functions $f_T : T_1 \rightarrow T_2$ and $f_P : P_1 \rightarrow (T_2 \uplus P_2)$ such that the following conditions are satisfied:

1. for all $t \in T_1$ we have $f_P(\bullet t) = \{f_T(t)\}$
or
 $pr \circ f_P^\oplus \circ pre_1(t) = pre_2(f_T(t))$
 where $pr : (T_2 \uplus P_2)^\oplus \rightarrow P_2^\oplus$ is the corresponding projection
 analogously for the *post* function.
2. for all $t_2 \in f_T(T_1)$ we have:
 $\exists t_{in} \in T_1$ with $f_T(t_{in}) = t_2$ and $pr \circ f_P^\oplus \circ pre_1(t_{in}) = pre_2(t_2)$
 analogously for the *post* function.
3. marking strict:
 $\forall p \in P_1$ with $f_P(p) \in P_2$ we have $f_P^\oplus(\widehat{m}_1|_p) = \widehat{m}_2|_{f_P(p)}$
4. for all $p \in P_1$ with $f_P(p) \in T_2$ we have $f_T(\bullet p) = \{f_P(p)\}$
 analogously for the *post* function.



Collapsing Subnet [8]. Given an abstracting morphism $f : N_1 \rightarrow N_2$. We have for all transitions $t \in T_2$ the collapsing subnet $subst_f(t) = (\tilde{P}^{t,f}, \tilde{T}^{t,f}, \widetilde{pre}^{t,f}, \widetilde{post}^{t,f}, \widehat{m}_t) \subseteq N_1$ with

- $\tilde{P}^{t,f} = \{p_1 \in P_1 \mid f_P(p_1) = t\}$
 these are all places mapped to t
- $\tilde{T}^{t,f} = \{t_1 \in T_1 \mid f_T(t_1) = t\}$
 these are all transitions mapped to t
- $\widetilde{pre}^{t,f}(t_1) = pre_1(t_1)|_{\tilde{P}^{t,f}}$ for all $t_1 \in \tilde{T}^{t,f}$
 the pre- and post-domain restricted to collapsing places
- $\widetilde{post}^{t,f}$ is defined analogously.
- $\widehat{m}_t = \widehat{m}_1|_{\tilde{P}^{t,f}}$

Moreover, we define the following sets:

- $\mathbb{S}^f \subseteq T_2$ with $\mathbb{S}^f = \{t \mid \tilde{P}^{t,f} \neq \emptyset\}$
- $\mathbb{T}^f \subseteq T_1$ with $\mathbb{T}^f = \bigcup_{t \in \mathbb{S}^f} \tilde{T}^{t,f}$
- $\mathbb{P}^f \subseteq P_1$ with $\mathbb{P}^f = \bigcup_{t \in \mathbb{S}^f} \tilde{P}^{t,f}$

We omit the superscripts $(-)^{t,f}$ and $(-)^f$, if unambiguous.

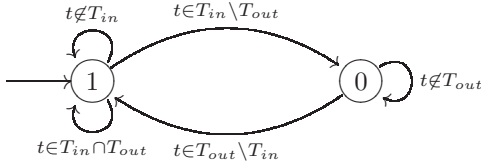
Live In-Out Cycles describe those subnets that are live, and are equipped with a guarding place. This guarding place ensures that each run within the subnet has to be completed before it may run again.

Live In-Out Cycle [8]. Given a place/transition system $N = (T \xrightarrow[\text{post}]{\text{pre}} P^\oplus, \widehat{m})$.

We call N *live in-out cycle* if the following conditions hold:

1. there are two distinguished subsets T_{in} and T_{out} of T , called set of in-transitions T_{in} and set of out-transitions T_{out} of N such that there is a place $c \in P$, called *guarding place*, which is in the predomain of all in-transitions $t_i \in T_{in}$, and in the post-domain of all out-transitions $t_o \in T_{out}$
with $pre(t)|_c = \begin{cases} 1 * c ; t \in T_{in} \\ \epsilon ; t \notin T_{in} \end{cases}$ and $post(t)|_c = \begin{cases} 1 * c ; t \in T_{out} \\ \epsilon ; t \notin T_{out} \end{cases}$
2. $\widehat{m}|_c = 1 * c$
3. N is live
4. place c is safe (1-bounded)

With respect to the number of tokens in the place c , the evolution of the live in-out cycle can be expressed in the form of a finite automaton below.



The number inside the particular state is the number of tokens which reside in the place c . Due to the definition the state 1 is the initial state. From the behavior of this automaton one can infer important property of every live in-out cycle. The firing of one of the input transitions alternates (sooner or later) with the firing of one of the output transitions.

Collapsing Morphisms [8]. A **collapsing morphism** $f : N_1 \rightarrow N_2$ is an abstracting morphism which additionally satisfies the following conditions:

1. f_T is surjective,
and f_P is quasi-surjective, i.e.
the restriction $f_P : P_1 \setminus \mathbb{P} \rightarrow P_2$ is surjective
2. f_T and f_P are quasi-injective, i.e.
For all $t, t' \in T_1 \setminus \mathbb{T}$ we have $f_T(t) = f_T(t')$ implies $t = t'$
For all $p, p' \in P_1 \setminus \mathbb{P}$ we have $f_P(p) = f_P(p')$ implies $p = p'$
3. $\forall t \in \mathbb{S}$ the following holds:
 $subst_f(t)$ is a live in-out cycle with c^t the guarding place so that
it is only connected to the rest of N_1 via the in- and out-transitions:
 - (a) for all $t_i \in T_{in} \subseteq \widetilde{T}^t$ holds $pre_2(t) = \widehat{f_P}(pre_1(t_i) \ominus c^t)$
 - (b) for all $t_o \in T_{out} \subseteq \widetilde{T}^t$ holds $post_2(t) = \widehat{f_P}(post_1(t_o) \ominus c^t)$
 - (c) for all $t_s \in \widetilde{T}^t \setminus T_{in}$ holds $\bullet t_s \subseteq \widetilde{P}^t$
 - (d) for all $t_s \in \widetilde{T}^t \setminus T_{out}$ holds $t_s \bullet \subseteq \widetilde{P}^t$

Example 2. In the Figure 4 the dashed arrow from right-hand side to the left-hand side of the depicted rule is a collapsing morphisms as it collapses a subnet to a one transition. This morphism is also abstracting.

Next, we specify the meaning of a safety property and liveness in order to be able to present our main results concerning preservation (translation) of these properties. We recall safety formulas over markings (in the sense of [MP92]), their translations via morphisms and used notion of liveness.

Safety Properties, Translations [6]. Consider a place/transition system N with a set of places P .

1. A *static formula* λp over N is given for $\lambda \in \mathbb{N}$ and $p \in P$. The set of all static formulas over P is denoted by $\mathcal{F}$; static formulas are build up using the logical operators $\neg$ and $\wedge$:

$$\begin{aligned}\varphi_1 \in \mathcal{F} &\implies \neg \varphi_1 \in \mathcal{F}, \\ \varphi_1 \in \mathcal{F}, \varphi_2 \in \mathcal{F} &\implies \varphi_1 \wedge \varphi_2 \in \mathcal{F}\end{aligned}$$

The validity of formulas is given w.r.t. the marking of a net. Let $m \in P^\oplus$ be a marking of N then:

$$\begin{aligned}m \models_N \lambda p &\text{ iff } \lambda p \leq m, \\ m \models_N \neg \varphi_1 &\text{ iff } \neg(m \models_N \varphi_1) \text{ and} \\ m \models_N \varphi_1 \wedge \varphi_2 &\text{ iff } (m \models_N \varphi_1) \wedge (m \models_N \varphi_2).\end{aligned}$$

2. Let φ be a static formula over N . Then $\Box \varphi$ is a *safety property*. The safety property $\Box \varphi$ holds in N under m iff φ holds in all states reachable from m :

$$m \models_N \Box \varphi \iff \forall m' \in [m] : m' \models_N \varphi.$$

If m is the initial marking $\hat{m}$ we also write $N \models \Box \varphi$ instead of $\hat{m} \models_N \Box \varphi$.

3. The *translation* $\mathcal{T}_f$ of formulas over N_1 along a morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ to formulas over N_2 is given for atoms by

$$\mathcal{T}_f(\lambda p) = \lambda f_P(p).$$

The translation of formulas is given recursively by

$$\begin{aligned}\mathcal{T}_f(\neg \varphi) &= \neg \mathcal{T}_f(\varphi), \\ \mathcal{T}_f(\varphi_1 \wedge \varphi_2) &= \mathcal{T}_f(\varphi_1) \wedge \mathcal{T}_f(\varphi_2) \text{ and} \\ \mathcal{T}_f(\Box \varphi) &= \Box \mathcal{T}_f(\varphi).\end{aligned}$$

4. The *l-translation* (in [17]) $\mathcal{LT}_f$ of formulas over N_2 along a collapsing morphism $f = (f_T, f_P) : N_1 \rightarrow N_2$ to formulas over N_1 is given for atoms by

$$\begin{aligned}\mathcal{LT}_f(\lambda p) &= \lambda f_P^{-1}(p) & if(p \bullet) \cap \mathbb{S}^f &= \emptyset, \\ \mathcal{LT}_f(\lambda p) &= \lambda f_P^{-1}(p) \vee \bigvee_{t_i \in (p \bullet) \cap \mathbb{S}^f} \neg(1 * c^{t_i}), & if(p \bullet) \cap \mathbb{S}^f &\neq \emptyset.\end{aligned}$$

The l-translation of formulas is given recursively by

$$\begin{aligned}\mathcal{LT}_f(\neg \varphi) &= \neg \mathcal{LT}_f(\varphi) \\ \mathcal{LT}_f(\varphi_1 \wedge \varphi_2) &= \mathcal{LT}_f(\varphi_1) \wedge \mathcal{LT}_f(\varphi_2) \quad \text{and} \\ \mathcal{LT}_f(\Box \varphi) &= \Box \mathcal{LT}_f(\varphi).\end{aligned}$$

Liveness. A place/transition system $N = (T, P, pre, post, \widehat{m})$ is called *live* if for arbitrary $m_1 \in [\widehat{m}_1]$ and arbitrary $t_1 \in T_1$ there exists some $m'_1 \in [m_1]$ such that $m'_1[t_1]$.

Theorem *Properties of Morphisms.*

1. *Place Preserving Morphisms Preserve Safety Properties [6]*

Let $f : N_1 \rightarrow N_2$ be a place preserving morphism. Let $\Box\varphi$ be a safety property. Then the following holds:

$$N_1 \models \Box\varphi \implies N_2 \models \mathcal{T}_f(\Box\varphi)$$

2. *Transition Gluing Morphisms Preserve Safety Properties [6]*

Let $f : N_1 \rightarrow N_2$ be a transition gluing morphism and let $\Box\varphi$ be a safety property then we have

$$N_1 \models \Box\varphi \implies N_2 \models \Box\varphi$$

3. *L-Transition Gluing Morphisms Preserve Liveness [17]*

Given $f : N_1 \rightarrow N_2$ a l-transition gluing morphism and let N_1 be live. Then the net N_2 is live as well.

4. *Collapsing Morphisms Respect Liveness [8]*

Given a collapsing morphism $f : N_1 \rightarrow N_2$ and let N_2 be live. Then the net N_1 is live as well.

5. *Collapsing Morphisms Translate (Backwards) Safety Properties [17]*

Let $f : N_1 \rightarrow N_2$ be a collapsing morphism. Let $\Box\varphi$ be a safety property. Then the following holds:

$$N_2 \models \Box\varphi \implies N_1 \models \mathcal{LT}_f(\Box\varphi)$$

Achieved results based on foregoing theorems are summarized in the Section 4.

3.2 Coloured Petri Nets

In this section we are going to present the results leading to preservation of safety properties in coloured Petri nets by transformations. We use coloured Petri nets [Jen92] and define different kinds of morphisms. These morphisms have different properties and are used for different purposes. We define formulas expressing safety properties and their preservation by a special kind of morphism, the safety preserving morphisms.

Coloured Petri nets [Jen92]. A **non-hierarchical coloured Petri net** is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ according to the description below:

1. Σ is a finite set of non-empty types, called **colour sets**.
2. P is a finite set of **places**.
3. T is a finite set of **transitions**.

4. A is a finite set of **arcs** s.t:
 $P \cap T = P \cap A = T \cap A = \emptyset$.
5. $N : A \rightarrow P \times T \cup T \times P$ is a **node** function.
6. $C : P \rightarrow \Sigma$ is a **colour** function.
7. $G : T \rightarrow \mathbb{EXP}_{\mathbb{B}}(\Sigma)$ is a **guard** function, where $\mathbb{EXP}_{\mathbb{B}}(\Sigma)$ denotes Boolean expressions over the colour sets in Σ .
8. $E : A \rightarrow \mathbb{EXP}(\Sigma)_{MS}$ is an **arc expression** function, where $\mathbb{EXP}(\Sigma)$ denotes expressions over the colour sets in Σ and $(\_)_{MS}$ multi-sets thereof.
9. $I : P \rightarrow \mathbb{EXP}_{\text{closed}}(\Sigma)_{MS}$ is an **initialization** function, where $\mathbb{EXP}(\Sigma)$ denotes closed expressions over the colour sets in Σ and $(\_)_{MS}$ multi-sets thereof.

A marking of a coloured Petri net is given as a multiset over the set of token elements $TE = \{(c, p) | p \in P \text{ and } c \in C(p)\}$.

Safety Property Preserving Morphisms

The morphisms we now introduce are based on mappings of the net components. That is, a CPN-morphism maps places to places, transitions to transitions, arcs to arcs, and expressions over colour sets to expressions over colour sets. Moreover, we distinguish different kinds of morphisms that have different properties.

CPN-Morphisms. A CPN-morphism $f : CPN_1 \rightarrow CPN_2$ with coloured Petri nets $CPN_i = (\Sigma_i, P_i, T_i, A_i, N_i, C_i, G_i, E_i, I_i)$ for $i = 1, 2$ is given by the mappings of the components $f = (f_{\Sigma}, f_P, f_T, f_A)$ with $f_{\Sigma} : \mathbb{EXP}(\Sigma_1) \rightarrow \mathbb{EXP}(\Sigma_2)$, $f_P : P_1 \rightarrow P_2$, $f_T : T_1 \rightarrow T_2$, and $f_A : A_1 \rightarrow A_2$.

A CPN-morphism is called:

loose if the following *compatibility conditions* hold:

1. $N_2 \circ f_A = f_{PT} \circ N_1$
 where $f_{PT} : P_1 \times T_1 \cup T_1 \times P_1 \rightarrow P_2 \times T_2 \cup T_2 \times P_2$
 with $f_{PT}(x, y) := \begin{cases} (f_P(p), f_T(t)) & (x, y) = (p, t) \\ (f_T(t), f_P(p)) & (x, y) = (t, p) \end{cases}$

That means, that the following diagram commutes:

$$\begin{array}{ccc}
 A_1 & \xrightarrow{N_1} & P_1 \times T_1 \cup T_1 \times P_1 \\
 f_A \downarrow & & \downarrow f_{PT} \\
 A_2 & \xrightarrow{N_2} & P_2 \times T_2 \cup T_2 \times P_2
 \end{array}$$

2. $C_2 \circ f_P = f_{\Sigma} \circ C_1$
3. $G_2 \circ f_T = f_{\Sigma} \circ G_1$
4. $E_2 \circ f_A = f_{\Sigma} \circ E_1$
5. $f_{\Sigma}(I_1(p)) \leq I_2(f_P(p))$

transition preserving if it is loose and the *transition preserving condition* holds:

6. no “new” arc is connected to “old” transitions, that is
for all $A \in A_2 \setminus f_A(A_1)$ we have :

$$N_2(A) \in P_2 \times T_2 \setminus f_T(T_1) \cup T_2 \setminus f_T(T_1) \times P_2$$

A special case are strict morphisms with $f_{\Sigma}(I_1(p)) = I_2(f_P(p))$

place preserving if it is loose and the *place preserving conditions* hold:

7. no “new” arc is connected to “old” places, that is
for all $A \in A_2 \setminus f_A(A_1)$ we have :

$$N_2(A) \in P_2 \setminus f_P(P_1) \times T_2 \cup T_2 \times P_2 \setminus f_P(P_1)$$

8. $\sum_{p \in P_1} f_{\Sigma}(I_1(p)) \leq \sum_{p \in P_2} I_2(p)$
9. f_P, f_T are injective
10. f_{Σ} is persistent in the sense of [EM85], meaning intuitively, that Σ_2 is
a consistent enrichment of Σ_1

transition gluing if it is loose and the *transition gluing conditions* hold:

11. f_A is bijective
12. f_T is surjective
13. f_P and f_{Σ} are identities
14. $I_1 = I_2$

Remark 1. Conditions 1 - 4 ensure the compatibility of componentwise mappings. Condition 5 states that the initialization function of the target net is place-wise greater than the source net. The difference to condition 8 concerns the identification of places by non-injective f_P . Condition 6 guarantees that the neighborhood of transitions remains unchanged, whereas condition 7 ensures this for places. Condition 11 ensures that the identification of transitions by non-injective f_T does not affect the adjacent arcs.

Example 3. In the Figures 9, 10 and 11 the morphisms denoted by $\longrightarrow$ are transition preserving, as there are no transitions in the interfaces of the rules. The morphism denoted by $--\triangleright$ is place preserving, as in the right-hand side of the corresponding rule all new arcs are connected only to new places. The morphism denoted by $\cdots\triangleright$ is a transition gluing morphism.

Safety Properties. We use a simple kind of temporal logic formulas over the marking of the places in order to express safety properties. We merely employ the always operator $\Box$ as the outermost operator. This yields safety properties in the sense of [MP92]. More precisely, we have: $\lambda(c, p)$ is a static formula for

$\lambda \in \mathbb{N}$, $p \in P$, and $c \in C(p)$. Static formulas are generated using the logical operators $\wedge$ and $\neg$. $\Box\varphi$ is a safety property, where φ is a static formula.

The validity of formulas is given w.r.t. the marking. Let M be a multi-set over the set of token elements $TE = \{(c, p) | p \in P \text{ and } c \in C(p)\}$, e.g. an arbitrary marking m of a coloured Petri net CPN then

$CPN \models_m \lambda(c, p)$ if and only if $\lambda(c, p) \leq m$.

For $CPN \models_m \neg\varphi_1$ if and only if $\neg(CPN \models_m \varphi_1)$ and $CPN \models_m (\varphi_1 \wedge \varphi_2)$ if and only if $CPN \models_m \varphi_1 \wedge CPN \models_m \varphi_2$. The safety property $\Box\varphi$ holds in CPN under m if and only if φ holds in all states reachable from m :

$CPN \models_m \Box\varphi$ if and only if $\forall m' \in [M] : CPN \models_{m'} \varphi$.

The translation of formulas $\mathcal{T}_f$ over CPN_1 to formulas over CPN_2 along a morphism $f : CPN_1 \rightarrow CPN_2$ is given for atoms by

$$\mathcal{T}_f(\lambda(c, p)) = \lambda(f_\Sigma(c), f_P(p)).$$

The translation of formulas is given recursively by $\mathcal{T}_f(\neg\varphi) = \neg\mathcal{T}_f(\varphi)$, and $\mathcal{T}_f(\varphi_1 \wedge \varphi_2) = \mathcal{T}_f(\varphi_1) \wedge \mathcal{T}_f(\varphi_2)$, and $\mathcal{T}_f(\Box\varphi) = \Box\mathcal{T}_f(\varphi)$

Theorem Properties of Morphisms.

1. *Place Preserving Morphisms Preserve Safety Properties [12].*

A place preserving morphism $f : CPN_1 \rightarrow CPN_2$ preserves safety properties, i.e. for all safety properties $\Box\varphi$ we have :

$$CPN_1 \models \Box\varphi \implies CPN_2 \models \mathcal{T}_f(\Box\varphi)$$

2. *Transition Gluing Morphisms Preserve Safety Properties [12].*

A transition gluing morphism $f : CPN_1 \rightarrow CPN_2$ preserves safety properties, i.e. for all safety properties $\Box\varphi$ we have :

$$CPN_1 \models \Box\varphi \implies CPN_2 \models \mathcal{T}_f(\Box\varphi)$$

4 Summary of Results

In this section we summarize the results we have obtained for rule-based refinement of Petri nets. Moreover, we give for these results an interpretation of its meaning. For proofs and technical details we refer to the corresponding papers.

The presented results comprise preservation of system properties via transformations by different rules and compatibility of transformations with horizontal structuring.

4.1 Preservation of Properties

Subsequently we refer to conference papers [15, 11, 16], but due to space limitations there many proofs are given in more detail in the corresponding technical reports [6, 12, 8].

Theorem [15, 11]. Given a rule $(r = (L \leftarrow K \rightarrow R), f : L \rightarrow R)$ with f being

- either a place preserving morphism
- or a transition gluing morphism

then for a transformation step $N_1 \xrightarrow{(r, f)} N_2$ holds:

$$N_1 \models \Box\varphi \implies N_2 \models \mathcal{T}_f(\Box\varphi)$$

These rules are called *sp-rules*.

Theorem [15, 11]. Given a rule (r, f) with $r = (\emptyset \leftarrow \emptyset \rightarrow R)$ so that $R \models \Box\varphi$ with a occurrence $g : R \rightarrow N_2$. Then for a transformation step $N_1 \xrightarrow{(r,f)} N_2$ holds:

$$N_2 \models \mathcal{T}_g(\Box\varphi).$$

We call these rules *si-rules*.

Theorem [16, 17]. Given a rule $(r = (L \leftarrow K \rightarrow R), f : R \rightarrow L)$ with f being a collapsing morphism. Then for a transformation step $N_1 \xrightarrow{(r,f)} N_2$ holds:

$$N_1 \models \Box\varphi \implies N_2 \models \mathcal{LT}_f(\Box\varphi)$$

$$N_1 \text{ live} \implies N_2 \text{ live}^1.$$

We call these rules *lp-rules*.

Theorem [17]. Given a rule $(r = (L \leftarrow K \rightarrow R), f : L \rightarrow R)$ with f being a l-transition gluing morphism. Then for a transformation step $N_1 \xrightarrow{(r,f)} N_2$ holds:

$$N_1 \text{ live} \implies N_2 \text{ live}.$$

These rules are also called *lp-rules*.

Theorem [17]. Given rule (r, f) with $r = (\emptyset \leftarrow \emptyset \rightarrow R)$ so that R is live. Then for a transformation step $N_1 \xrightarrow{(r,f)} N_2$ holds:

$$N_1 \text{ live} \implies N_2 \text{ live}.$$

These rules are called *li-rules*.

Interpretation. Foregoing paragraphs contain an overview of achieved results. The interpretation of these theorems can be done in a succinct way as proof-rules, which are of interest for model developers.

Consider a net transformation $N_1 \xrightarrow{(r,f)} N_2$. Then there are the following proof-rules (according to the results):

$\frac{(r, f) \text{ is a sp-rule; } N_1 \text{ satisfies } \Box\varphi}{N_2 \text{ satisfies } \Box\mathcal{T}_f(\varphi)}$
$\frac{(r, f : L \rightarrow R) \text{ is a si-rule; } g : R \rightarrow N_2; R \text{ satisfies } \Box\varphi}{N_2 \text{ satisfies } \mathcal{T}_g(\Box\varphi)}$

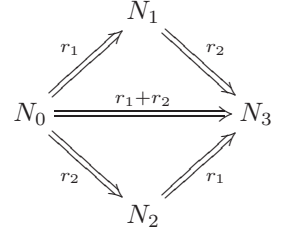
¹ under appropriate occurrence morphisms, see [8] for details

$\frac{(r, f) \text{ is a lp-rule; } N_1 \text{ is live}}{N_2 \text{ is live}^1}$
$\frac{(r, f : L \rightarrow R) \text{ is a li-rule; } R, N_1 \text{ are live}}{N_2 \text{ is live}}$
$\frac{(r, f) \text{ is a lp-rule; } N_1 \text{ satisfies } \Box\varphi}{N_2 \text{ satisfies } \Box\mathcal{LT}_f(\varphi) \text{ resp. } \Box\mathcal{T}_f(\varphi)}$

4.2 Compatibility Results

Theorem Church-Rosser and Parallelism.

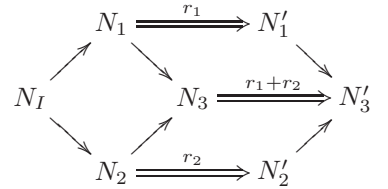
Given two rules r_1 and r_2 that satisfy certain independence conditions (see [2]) then we have: If we can apply r_1 and r_2 sequentially to N_0 and then to N_1 , then we can apply them in parallel as well as in the opposite order.



Interpretation. These Church-Rosser theorems state a local confluence in the sense of formal languages. In the context of specification techniques they are important as they state conditions for the independent development of different parts or views of the system. These independence conditions hold for refinement rules as well. The parallel independence states that the matches of both rules overlap in parts that are not deleted only. Sequential independence states that those parts created by the first transformation step are not deleted in the second.

Theorem Union and Fusion Theorems.

Union theorem [9] states that - provided that certain independence conditions hold - given two rules r_1 and r_2 and a union $N_1 \leftarrow N_I \rightarrow N_2$ of N_1 and N_2 then we obtain the same result (up to isomorphism) whether we derive first $N_1 \xRightarrow{r_1} N'_1$ and $N_2 \xRightarrow{r_2} N'_2$ and then construct the union of N'_1 and N'_2 resulting in N'_3 or whether we first construct the union N_3 of N_1 and N_2 followed by parallel transformation step $N_3 \xRightarrow{r_1+r_2} N'_3$.



Fusion theorem [9] is expressed similarly. Provided that certain independence conditions hold and given a rule r and a fusion $F \rightrightarrows N_1$ then we obtain the same result (up to isomorphism) whether we derive first $N_1 \xRightarrow{r} N'_1$ and then construct the fusion $F \rightrightarrows N'_1$ resulting in N'_2 or whether we construct the fusion $F \rightrightarrows N_1$ resulting in N_2 and then perform the transformation step $N_2 \xRightarrow{r} N'_2$.

$$\begin{array}{ccccc} F & \rightrightarrows & N_1 & \longrightarrow & N_2 \\ & & \Downarrow r & & \Downarrow r \\ & & N'_1 & \longrightarrow & N'_2 \end{array}$$

Interpretation. Both theorems state that transformation of Petri nets is compatible with the corresponding structuring technique under reasonable independence conditions. These conditions ensure that the gluing affects those subnets only that remain unchanged.

4.3 Open Problems

We have introduced several morphisms that preserve some property of the Petri net. So we obtained various morphisms classes and different categories. Based on these morphisms we can define rules and transformations that preserve stated property as well. The open question is how do the morphisms interact?

- Different morphisms can be used for the same development as exemplified in our examples in Section 2. We can use different proof-rules for the deduction of certain property although we achieve the transformations in different categories.
- Another question is whether it is possible to find adequate morphism classes that comprise the morphisms we have not yet defined and at the same time still preserve those properties. We do not think so, as morphisms would need to be so general that they will not preserve anything.
- Last but not least we think it would be promising to extend the abstract theory in order to cope with different morphism classes.

Very often are properties of communicating and reactive systems stated as temporal logic formulas. Although safety-properties are such formulas, they do not provide necessary expressive power. The open problem is how to integrate temporal formulas preservation into the rule-based refinement.

5 Conclusion

We conclude with a summary and a discussion of future work. Summarizing, we have for different net classes the following results for rule-based refinement in table 1. This table illustrates we have already developed quite a complex theory. There are various possibilities how to proceed open. Future work can comprise among others the following activities.

Table 1. Achieved results

Notion/Results	PT-nets	AHL-nets	CPNs
Rules, Transformations	✓	✓	✓
Safety property preserving transformations with			
transition-gluing morphisms	✓	✓	✓
place-preserving morphisms	✓	✓	✓
Safety property introducing transformations	✓	✓	✓
Liveness preserving transformations	✓	?	?
Liveness introducing transformations	✓	?	?
Independence Conditions	✓	✓	✓
Church Rosser I + II	✓	✓	✓
Parallelism	✓	✓	✓
Union	✓	✓	✓
Fusion	✓	✓	✓

- The extension of our approach of liveness preserving transformation to high-level Petri nets is an obvious way to continue. The basic ideas remain the same but the data type part has to be dealt with.
- Other system properties, especially liveness properties in the sense of temporal logic [MP92, Lam94] are most promising to integrate into this approach. In order to do so, morphisms have to be defined that preserve such properties. Then a specific set of conditions have to be satisfied in order to achieve similar results for rules and transformations.
- Tool support is for the practical use the main precondition. The involved morphisms are quite complex, so the user needs tool support for defining and applying rules. The tool should assist the choice as well as the execution of rules and transformations.
- The integration of the various morphisms classes as discussed in Subsection 4.3 is one of the demanding and urgent tasks.
- Collapsing morphisms are very specialized and thus quite restrictive. We think that it is possible to extend the notion of collapsing morphisms in a natural way in order to obtain more expressive notion of liveness respecting morphisms.
- Preservation of system properties under horizontal structuring is also of interest for future investigation.

References

- [BDH92] E. Best, R. Devillers, and J. Hall. The Box Calculus: a new causal algebra with multi-label communication. In *Advances in Petri Nets*, pages 21–69. Lecture Notes in Computer Science 609, 1992. [167](#)
- [BGV91] W. Brauer, R. Gold, and W. Vogler. A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets. *Advances in Petri Nets*, Lecture Notes in Computer Science 483:1–46, 1991. [167](#), [168](#)
- [BS90] J. Bradfield and C. Stirling. Verifying temporal properties of processes. In J. C. M. Baeten et al., editor, *Lecture Notes in Computer Science; CONCUR'90, Theories of Concurrency: Unification and Extension. (Conference, 1990, Amsterdam, The Netherlands)*, pages 115–125, Berlin, Germany, 1990. Springer Verlag. [167](#)
- [CT90] Y. Chen and W. T. Tsai. An algebraic approach to Petri net reduction and its application to protocol analysis. Technical report, University of Minnesota, 1990. [167](#)
- [DA92] R. David and H. Alla, editors. *Petri Nets and Grafcet*. Prentice Hall (UK), 1992. [167](#)
- [DDGJ90] W. Damm, G. Döhmen, V. Gerstner, and B. Josko. Modular verification of petri nets: The temporal logic approach. In J. W. de Bakker et al., editors, *Lecture Notes in Computer Science; Proceedings of the REX Workshop on Stepwise Refinement, 1989, Mook, The Netherlands*, pages 180–207, Berlin, Germany, 1990. Springer-Verlag. [167](#)
- [DM90] J. Desel and A. Merceron. Vicinity Respecting Net Morphisms. In *Advances in Petri Nets*, pages 165–185. Springer Verlag, 1990. Lecture Notes in Computer Science 483. [167](#), [180](#)
- [Ehr97] H. Ehrig et al. Integration von Techniken der Software Spezifikation für ingenieurwissenschaftliche Anwendungen. Antrag für ein Schwerpunktprogramm an die DFG, <http://tfs.cs.tu-berlin.de/SPP/index.html>, 1997. (akzeptiert als DFG-SPP von Januar 1998 bis Dezember 2003). [172](#)
- [EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1985. [186](#)
- [ERW02] H. Ehrig, W. Reisig, and H. Weber et al. The Petri Net Baukasten of the DFG-Forscherguppe PETRI NET TECHNOLOGY. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors. *Advances in Petri Nets: Petri Net Technologies for Modeling Communication Based Systems*. LNCS. Springer, 2002. To Appear. [162](#), [163](#)
- [ES91] J. Esparza and M. Silva. On the analysis and synthesis of free choice systems. *Lecture Notes in Computer Science; Advances in Petri Nets 1990*, 483:243–286, 1991. [167](#), [168](#)
- [Esp94] J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23:151–195, 1994. [167](#), [168](#)
- [FWL] J. Favrel, H. Wu, and K. H. Lee. Reduction method of coloured Petri nets. In *Proc. IEEE Int. Conf. on Systems, Man, and Cybernetics*. [167](#)
- [Gen91] H. J. Genrich. Predicate/Transition Nets. In *High-Level Petri Nets: Theory and Application*, pages 3–43. Springer Verlag, 1991. [172](#)
- [GG90] R. J. van Glabbeek and U. Goltz. Equivalences and Refinement. In *Semantics of Systems of Concurrent Processes*, pages 309–333. Springer Verlag, 1990. Lecture Notes in Computer Science 469. [167](#)

- [GL81] H. J. Genrich and K. Lautenbach. System Modelling with High-Level Petri Nets. *Theoretical Computer Science*, 13:109–136, 1981. 172
- [HRH91] R. R. Howell, L. E. Rosier, and Chun Yen Hsu. A taxonomy of fairness and temporal logic problems for Petri nets. *Theoretical Computer Science*, 82(2):341–372, 1991. 167
- [Jan97] L. Jansen. Referenzfallstudie Verkehrsleittechnik. <http://www.ifra.ing.tu-bs.de/m33/spezi/>, 1997. 172
- [JCHH91] K. Jensen, S. Christensen, P. Huber, and M. Holla. *Design/CPN. A Reference Manual*. Meta Software Cooperation, 125 Cambridge Park Drive, Cambridge Ma 02140, USA, 1991. 172
- [Jen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1: Basic Concepts. Springer Verlag, EATCS Monographs in Theoretical Computer Science edition, 1992. 172, 184
- [Jen94] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 2: Analysis Methods. Springer Verlag, EATCS Monographs in Theoretical Computer Science edition, 1994. 167, 172
- [Jen97] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 3: Practical Use. Springer Verlag, EATCS Monographs in Theoretical Computer Science edition, 1997. 172
- [Lam94] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems* 16, 3:872–923, 1994. 191
- [MM90] J. Meseguer and U. Montanari. Petri Nets are Monoids. *Information and Computation*, 88(2):105–155, 1990. 178
- [MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer Verlag, 1992. 167, 183, 186, 191
- [Peu01] S. Peuker. Halbbordnungs-basierte Verfeinerung zur Verifikation verteilter Algorithmen. PhD thesis, Humboldt University Berlin, 2001.
- [Peu02] S. Peuker. Concurrency based transition refinement for the verification of distributed algorithms. In H. Ehrig, W. Reisig, G. Rozenberg, and H. Weber, editors. *Advances in Petri Nets: Petri Net Technologies for Modeling Communication Based Systems*. LNCS. Springer, 2002. To Appear. 162
- [Rei91] W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80:1–34, 1991. 172
- [Sch96] K. Schmidt. *Symbolische Analysemethoden für algebraische Petri-Netze*, volume 4. Bertz Verlag, versal edition, 1996. 167
- [Sou91] Y. Soussy. Deterministic systems of sequential processes: a class of structured Petri nets. In *Applications and Theory of Petri Nets, Gjærn, Juni 1991*, pages 62–81. Springer Verlag, 1991. 168
- [Vau87] J. Vautherin. Parallel System Specification with Coloured Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets* 87, pages 293–308. Springer Verlag, 1987. Lecture Notes in Computer Science 266. 172
- [vdA97] W. M. P. van der Aalst. Verification of workflow nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 407–426. Springer-Verlag, June 1997. 168
- [vdA98] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8:21–66, 1998. 168
- [vdAtH98] W. M. P. van der Aalst and A. H. M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based approach. Forschungsberichte des AIFB 380, Universität Karlsruhe, November 1998. 168

- [WER99] H. Weber, H. Ehrig, and W. Reisig, editors. *Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems, Part II: The "Petri Net Baukasten"*. Fraunhofer Gesellschaft ISST, October 1999. 162
- [WER01] H. Weber, H. Ehrig, and W. Reisig, editors. *2nd Int. Colloquium on Petri Net Technologies for Modelling Communication Based Systems*, Berlin, Germany, Sept. 2001. Researcher Group Petri Net Technology, Fraunhofer Gesellschaft ISST. 162
- [WLB01] H. Weber, S. Lembke, and A. Borusan. Improving the Usability of Petri Nets with the Petri Net Baukasten. In H. Ehrig, G. Juhás, J. Padberg, G. Rozenberg, editors, *Advances in Petri Nets: Unifying Petri Nets*, volume 2128 of *LNCS*, pages 54–78. Springer-Verlag, 2001.

Our Work on Transformations and Rule-Based Refinement

- [1] H. Ehrig, M. Gajewsky, and F. Parisi-Presicce. *High-Level Replacement Systems with Applications to Algebraic Specifications and Petri Nets*, chapter 6, pages 341–400. Number 3: Concurrency, Parallelism, and Distribution in Handbook of Graph Grammars and Computing by Graph Transformations. World Scientific, 1999. 163
- [2] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. From graph grammars to high level replacement systems. In *4th Int. Workshop on Graph Grammars and their Application to Computer Science, LNCS 532*, pages 269–291. Springer Verlag, 1991. 168, 189
- [3] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991. 163
- [4] C. Ermel, J. Padberg, and H. Ehrig. Requirements engineering of a medical information system using rule-based refinement of petri nets. In *Proc. IDPT Conference (International Design and Process Technology)*, pages 186–193, 1996. 171
- [5] M. Gajewsky. Concepts and Requirements for Transformations within Petri Net Based Process Models. In A. Ertas, editor, *5th World Conference on Integrated Design and Process Technology, Special Session on Model Integration*, 2000. CD-ROM, 8 pages. 163
- [6] M. Gajewsky, K. Hoffmann, and J. Padberg. Place Preserving and Transition Gluing Morphisms in Rule-Based Refinement of Place/Transition Systems. Technical Report 99-14, Technical University Berlin, 1999. 178, 179, 180, 183, 184, 187
- [7] M. Gajewsky and F. Parisi-Presicce. On Compatibility of Model and Class Transformations. In M. Cerioli, and G. Reggio, editors, *15th International Workshop on Algebraic Development Techniques and General Workshop of the CoFI WG*, Lecture Notes in Computer Science 2267. Springer Verlag, 2001. 163
- [8] M. Gajewsky, J. Padberg, and M. Urbásek. Rule-Based Refinement for Place/Transition Systems: Preserving Liveness-Properties. Technical Report 2001-8, Technical University of Berlin, 2001. 167, 178, 180, 181, 182, 184, 187, 188
- [9] J. Padberg. Categorical Approach to Horizontal Structuring and Refinement of High-Level Replacement Systems. *Applied Categorical Structures*, 7(4):371–403, December 1999. 163, 165, 171, 189, 190

- [10] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic high-level net transformation systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995. 172
- [11] J. Padberg and M. Gajewsky. Rule-Based Refinement of Petri Nets For Modeling Train Control Systems. In Š. Kozák and M. Huba, editors, *Petri Nets in Design, Modelling and Simulation of Control Systems, Special Session at the IFAC Conference on Control Systems Design*, pages 299–304, 2000. 168, 187, 188
- [12] J. Padberg and M. Gajewsky. Safety Preserving Transformations of Coloured Petri Nets. Technical Report 2000-13, Technical University Berlin, 2000. 187
- [13] J. Padberg, M. Gajewsky, and C. Ermel. Rule-Based Refinement of High-Level Nets Preserving Safety Properties. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering*, pages 221–238. Springer Verlag, Lecture Notes in Computer Science 1382, 1998. 163, 167
- [14] J. Padberg, M. Gajewsky, and C. Ermel. Rule-based refinement of high-level nets preserving safety properties. *Science of Computer Programming*, 40:97–118, 2001. www.elsevier.nl/locate/scico. 167
- [15] J. Padberg, K. Hoffmann, and M. Gajewsky. Stepwise Introduction and Preservation of Safety Properties in Algebraic High-Level Net Systems. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, pages 249–265. Springer Verlag, Lecture Notes in Computer Science 1783, 2000. 167, 187, 188
- [16] M. Urbášek and J. Padberg. Preserving liveness with rule-based refinement of place/transition systems. In Society for Design and Process Science (SDPS), editors, *Proc. IDPT 2002: Sixth World Conference on Integrated Design and Process Technology*, 2002. to appear. 167, 187, 188
- [17] M. Urbášek. Another Safety Property and Liveness Preserving Morphisms of P/T Systems. Technical Report, Technical University of Berlin, 2002. 178, 179, 180, 183, 184, 188

A Notation

Basics

$\mathbb{N}$	natural numbers
$\mathcal{P}$	power set

Free commutative monoids

$P^\oplus$	free commutative monoid over the set P
$\oplus, \ominus$	operations on monoid elements
$\leq, <$	comparison predicates on monoid elements
$w _P, w _{P'}$	restrictions of monoid elements with $w _{P'} := w'$ for $w \in P^\oplus$ and $w = w' \oplus w''$ where $w' \in P'^\oplus$ and $w'' \in (P \setminus P')^\oplus$

Petri nets

$\hat{m}$	initial marking
$m \xrightarrow{*} m'$	firing path from m to m'
$m[t > m'$	firing of t from m to m'
$m[t >$	t is enabled under m
$m' \in [m >$	m' is reachable from m
$\bullet x, x \bullet$	pre- and post-set of a place or a transition x
c, c^t	guarding place

Petri net examples

$\mathbf{p}$	place $\mathbf{p}$
$\mathbf{t}$	transition $\mathbf{t}$

Morphisms

$f : N_1 \longrightarrow N_2$	transition preserving Petri net morphism (i.e. mapping transition onto transition, place onto place)
$f : N_1 \dashrightarrow N_2$	property preserving or property respecting morphism

System Properties

λp	atomic formula over a place/transition net with $\lambda \in \mathbb{N}$ and $p \in P$
$\lambda(c, p)$	atomic formula over a coloured Petri net with $\lambda \in \mathbb{N}$, $p \in P$, and $c \in C(p)$
φ	static formula generated over atomic ones using the logical operators $\wedge$ and $\neg$
$\Box \varphi$	safety formula
$N \models_{\hat{m}} \Box \varphi$	net N with the initial marking $\hat{m}$ satisfies $\Box \varphi$
$\mathcal{T}_f$	translation of formulas over a net N_1 along a morphism $f : N_1 \rightarrow N_2$
$\mathcal{LT}_f$	l-translation of formulas over a net N_2 along a collapsing morphism $f : N_1 \rightarrow N_2$

High-Level Replacement Systems

$r = (L \leftarrow K \rightarrow R)$	rule
$N_1 \xrightarrow{r} N_2$	transformation step using rule r
$N_1 \xrightarrow{(r, f)} N_2$	property preserving transformation step using rule r and property preserving morphism f
$N_1 \xRightarrow{*} N_2$	transformation sequence

Modelling a Railway Crossing with Message Sequence Charts and Petri Nets

Olaf Kluge

TU Berlin, Institut für Technische Informatik und Mikroelektronik
Sekt. EN10, Einsteinufer 17, 10587 Berlin, Germany
kluge@cs.tu-berlin.de

Abstract. Message Sequence Charts (MSC) are a popular and intuitive specification language for describing scenarios. Especially the hierarchical composition of single diagrams make it very powerful. However, the process algebra based formal semantics is complicated and renders it difficult to analyse MSC specifications. Therefore, a different approach proposes to employ Petri nets as a semantic model for MSC. By employing a transformation a Petri net can be derived from each MSC specification. Thus, known analysis and simulation techniques for Petri nets can be employed to analyse properties of the MSC specification. Throughout the paper, a railway crossing serves as a case study. It is at first described informally and subsequently specified by means of MSCs. The transformation of the MSC specification into a Petri net and the subsequent analysis of the Petri net are demonstrated.

1 Introduction

Planning and constructing technical systems is a complex task. Therefore, the development needs to be supported by appropriate techniques such as diagrammatic and formal specification techniques.

Specification techniques which employ diagrams are on the one hand usually more intuitive and easier to understand as well as to apply, but on the other hand these techniques can contain ambiguities. It is, therefore, often difficult to determine properties of the specified system.

Formal specification techniques are usually more difficult to learn and require consolidated mathematical knowledge. The advantage is, however, that these techniques provide mathematical accuracy and precision for analysis, simulation and implementation of specified systems. Thus, implementations can be verified which is possible only with formal techniques.

Scenario based specification techniques become more and more popular and allow the specification of scenarios by means of diagrams. Most prominent examples for these specification techniques are UML Sequence Diagrams (UML SD) and Message Sequence Charts (MSC) [ITU99]. Each of these diagrams describes an example scenario, i.e. an example communication sequence of objects with each other in the diagram.

MSC represents probably the most powerful of these specification formalisms. The main reason is that it allows to give an hierarchical order to the diagrams and, thereby, describe parallel, sequential and alternative scenarios. A main disadvantage of MSC,

however, is the typical shortcoming of diagrammatic techniques: The descriptions of the scenarios by means of diagrams are not definite enough in order to derive a deterministic automata model. In more recent MSC standards interfaces to programming languages and other specification techniques can be defined. Thus, the use of MSC diagrams goes beyond informal description.

To make MSC more definite formal semantics for MSC was standardised by ITU-T [ITU98], which is based on a process algebra. Applying this formal semantics to MSC a process term can be derived for each MSC and each MSC specification. Unfortunately this formal semantics is very complex and an analysis of these process terms is difficult. Furthermore, no tool support for the employed process algebra is up to now available.

Petri nets are a formal specification technique which is widely studied and offers several methods and tools for analysing, simulating and verifying specifications.

This paper demonstrates how Petri nets can be employed as a semantic model for MSC. By applying a transformation a Petri net can be derived from every MSC and MSC specification. Depending on the MSCs and the applied analysis techniques even a low-level Petri net is sufficient in order to represent the correct behaviour. However, in general high-level Petri nets are necessary to represent the correct behaviour as described in the MSC specification.

A railways crossing serves as the case study throughout the paper. In the following section this railway crossing is described informally. Subsequently, it is specified by means of MSCs. Thereby, some background information on MSC is given. The transformation of an MSC specification into a Petri net is outlined in Section 4 and applied to the previously defined MSC specification of the railway crossing. Finally, some properties of MSC specifications and how this specification can be analysed by means of the derived Petri net are discussed and demonstrated employing the railway crossing scenario.

2 Railway Crossing Example

An important issue in the European Union is to develop a common European Train Control System (ETCS) in order to provide safe, efficient and smooth train traffic across European borders. A joint project was initiated by *Deutsche Bahn AG (DB)* and *Institut für Regelungs- und Automatisierungstechnik (IFRA)* in Brunswick to study the design process of the ETCS. As a result approximately 50 scenarios have been identified and specified [ETCS97]. A scenario describes a situation, the involved objects and the actions which are performed by the objects.

In the radio-based train operation (FunkFahrBetrieb FFB) two reference case studies were identified. These are studied within the priority programme of the German Research Council concerning *Integration of Software Specification Techniques for Applications in Engineering* [Ehr97]. One of these scenarios is the railway level crossing control system (RLCCS). This scenario describes a train passing a crossing of a road and a railway track, whereby, there is no bridge over or tunnel under the track is present as illustrated in Figure 1. In order to prevent a collision, it is required that the train only enters the crossing when the crossing is safe.

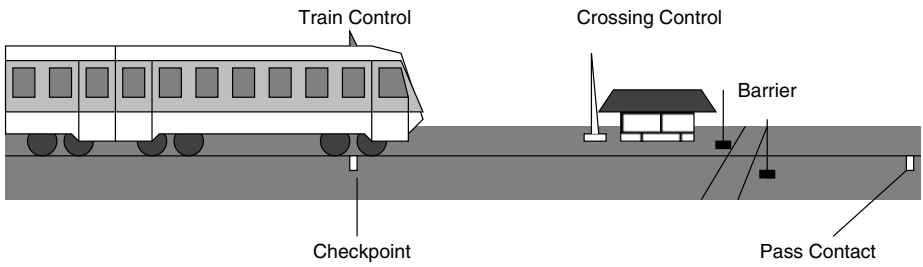


Fig. 1. Picture of an RLCCS Scenario

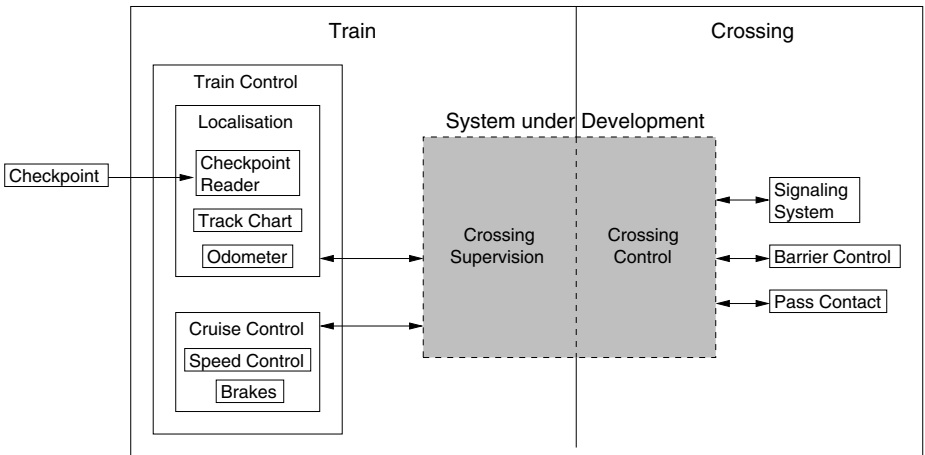


Fig. 2. Block diagram of the RLCCS Case Study

The objects involved in the scenario, their placement as well as their relation to each other is depicted in a block diagram in Figure 2. The checkpoint allows the determination of the current position and announces a potential hazard point. In this scenario the train communicates with a railway crossing. The systems specified by the scenarios are the crossing supervision of the train and the crossing control. Thereby, these two system parts receive data from sensors and control actuators.

In the following a simplified version of RLCCS [PJES99] is studied. The model contains the specifications of the correct behaviour and only one of many possible failure cases of the RLCCS scenarios.

The modelled setting is: At first a train passes a checkpoint which is a passive device on the track. A special reading device in the train reads the unique identifier (ID) of the checkpoint when the train passes by and sends this ID to the train control unit. With the ID of the checkpoint the train control calculates its position and the announced events. As in this case the announced event is a railway level crossing, the train control initiates communication with the crossing control in order to have the signalling started and the barriers lowered. Furthermore, the speed of the train is to be controlled. These actions

involve the exchange of many messages as described in greater detail in the following section.

3 Message Sequence Charts for RLCCS Scenario Specification

In this section the MSCs for the RLCCS example are introduced and explained. Thereby, the focus is on the graphical representation of MSCs.

MSC is a trace description language which was standardised and is maintained by the ITU-T as Z.120 [ITU99]. Every MSC specification has an equivalent graphical and textual representation. Especially the graphical representation and the intuitive meaning of MSCs are reasons for their growing popularity.

Each MSC describes a set of traces. It contains objects called MSC instances. Each of these instance has an own time line running from top to bottom. Along these time lines actions, so called MSC events, are arranged giving a total order to the events connected to this instance. Such events can be message send and receive events, timer and local events. A message is depicted by an arrow from the send to the receive event. The MSC message exchange is asynchronous, i.e. the send event can also be executed if no receive event is waiting for this message. The fact, that a message must be sent before it can be received imposes a total order on the send and receive event of a message and, furthermore, a partial order on all events in an MSC.

Previous works proved that MSC is a suitable formalism in order to specify such scenarios. The scenarios specified in Figure 3, Figure 4 and Figure 5 by means of MSCs were first presented in [Klo98, DDK99] in terms of LSCs. Each involved object is projected to an MSC instance. These instances as well as the exchanged messages and the order of these messages are depicted in the diagrams.

MSC conditions such as the `train_passed` box in Figure 3 are in [Klo98] and [DDK99] employed to compose MSCs. Thereby, two MSCs with a corresponding condition connected to the same set of instances may be glued together at the condition. In recent MSC standards [ITU99] this implicit composition was replaced by explicit composition (see Figure 6). Therefore, in the context of this example MSC conditions are used for documentation purposes only.

The four MSCs for the simplified version of the RLCCS scenario can be seen in Figures 3 to 6. The MSC `crossing` describes the sequence of events for a scenario in which no failure occurs. The passing of the checkpoint induces sending a message containing the checkpoint ID to the train control unit. By using this ID the train control is informed of the approaching train at the railway level crossing. Thus, the train control informs the crossing supervision process. Then the crossing supervision renders the train control to set a hazard point. If the crossing is not confirmed as being secure within a specific time frame the train must stop at this hazard point. When receiving this hazard point message the train control starts to ensure a secure passage via the crossing. A request to lower the barrier is send to the trains crossing supervision. Subsequently, the train passes it to the crossing control. The crossing control initiates the signalling as well as the lowering of the barrier and, when the crossing is secured, sends a confirmation to the crossing supervision.

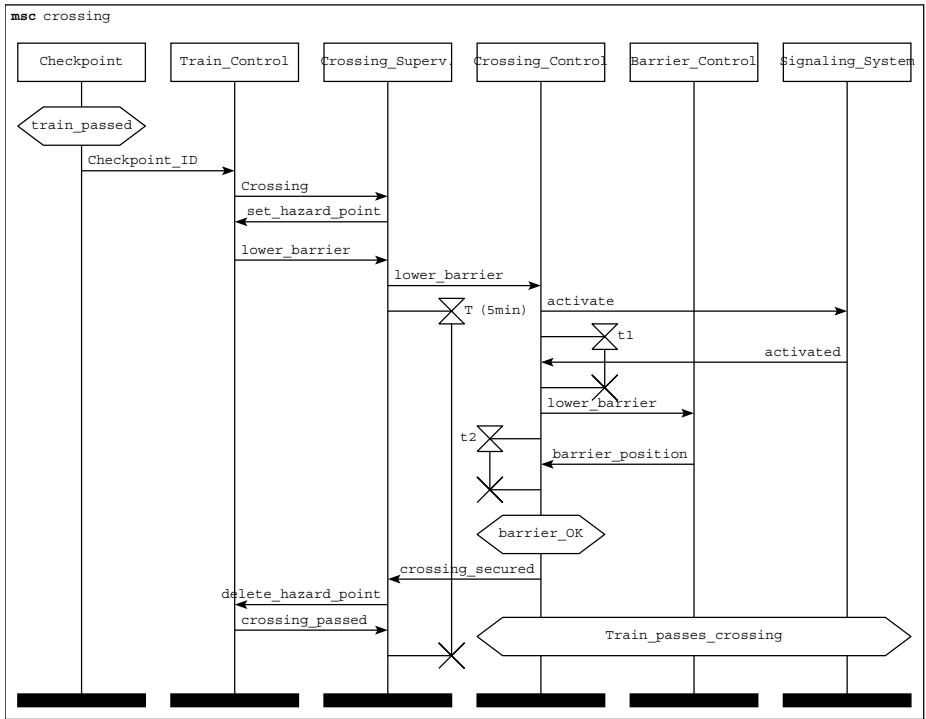


Fig. 3. Scenario - Level Crossing with Long Gates, Full System Supervision, MSC for the Regular Case

The failure scenario is specified in the MSC `crossing_error1`. The behaviour corresponds to the described case in the previous MSC up to the point where the barrier control sends the barrier position to the crossing control. In this case the position of the barrier does not fulfil the requirement of being closed. Therefore, a message is sent to an external control, which must correct the error as well as subsequently send a confirmation to the crossing control. Thus, the train can pass the now secured crossing.

Finally, the MSC `open_barrier` specifies the interaction after the train has passed the crossing. Thereby, the barriers are raised and the signalling is stopped.

In contrast to previous approaches [Klo98, DDK99], the MSC specification of the RLCCS scenario as presented here describes the composition explicitly. Figure 6 shows the High Level MSC (HMSC) for the discussed RLCCS scenario. It specifies the sequences in which the MSCs from Figures 3 to 5 can be executed. At first either MSC `crossing` or `crossing_error1` is executed. Following this alternatively sequentially the MSC `open_barrier` is executed.

The operator for the sequential composition of MSCs has a weak sequencing semantics. This means that there exists no synchronisation of the instances at their composition points. In other words, an instance might execute an event of the second MSC if the instances event order is preserved even if another instance still has to execute an event

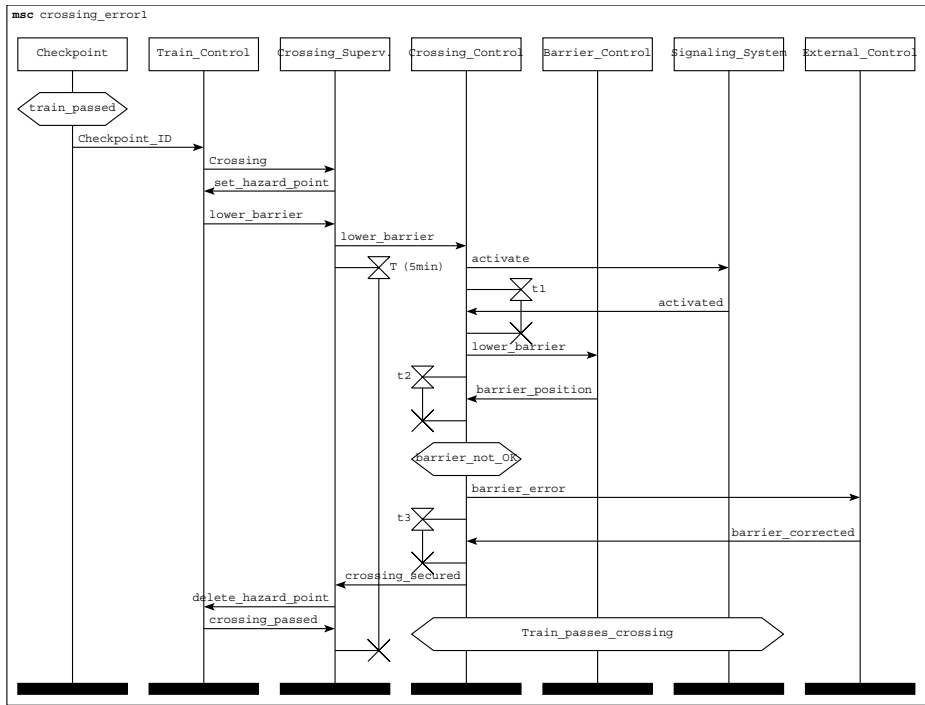


Fig. 4. Scenario - Level Crossing with Long Gates, Full System Supervision, MSC for the Failure Case

of the first MSC. This weak sequencing operator represents real systems very well as there is usually no synchronisation of autonomous system parts before the execution of a new task has started. This is a major difference to many other specification languages and has the disadvantage that it entails a much more expensive analysis compared to other formalisms with strong sequencing.

The weak sequencing semantics of the sequential MSC composition corresponds to reality. Therefore, there is no need for the MSC instance `Pass_Contact` in MSC `open_barrier` to wait for the MSCs `crossing` or `crossing_error1` to complete all events before it sends the message `train_passed`. It would make no sense if the sensor must wait for the train control system to finish its activity. Because the MSC composition points impose no synchronisation among the objects, the weak sequencing is the only sensible choice.

This presented scenario can be easily extended. In order to do so MSCs for the additional behaviour must be added. Furthermore, the description of the execution order of the MSCs in the scenario must be extended. Thereby, the MSC RLCCS needs to be changed in order to refer to additional MSCs.

A refinement of the scenario is also possible. Either an MSC of the scenario is replaced by a more detailed one or an MSC's instance is marked as decomposed and an MSC refining this instance is added to the scenario.

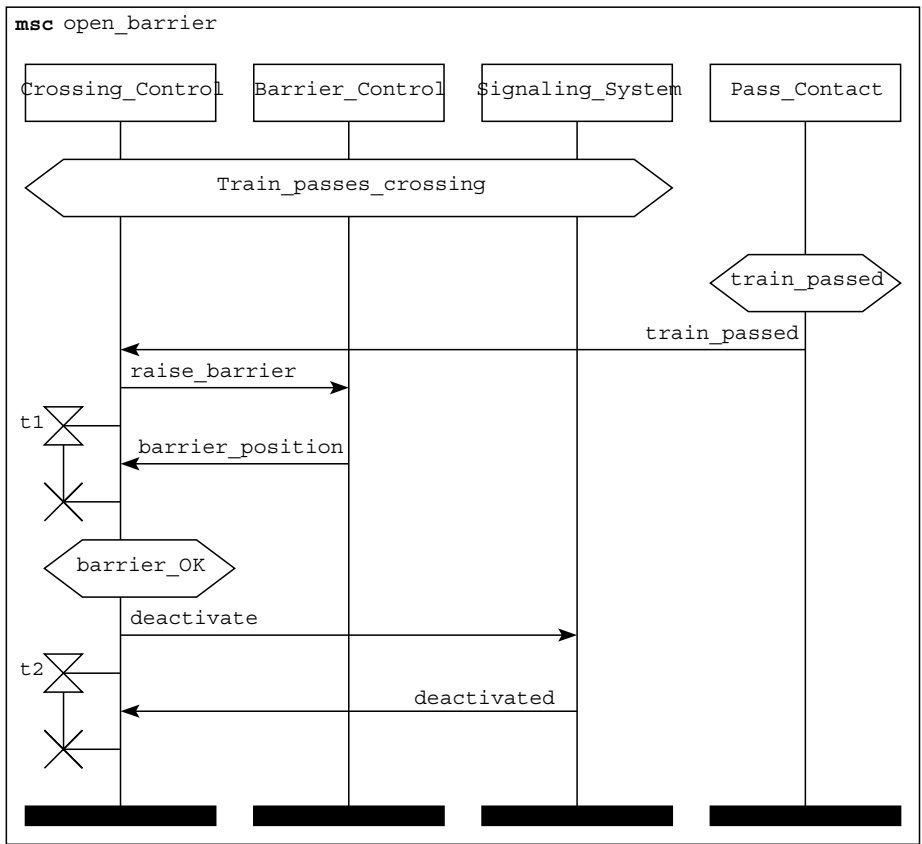


Fig. 5. Scenario: Level Crossing with Long Gates, Full System Supervision, MSC for opening the barrier after the train has passed

Another important difference to other formalisms is the alternative composition of MSCs. It has a delayed choice semantics, which means that as long as the alternative branches do not differ, both alternatives are taken into account, i.e. followed. Furthermore, it describes a global branching choice, i.e. all instances follow the same alternatives. More details are given in [ITU98]. Equivalence of MSCs or parts thereof is in general undecidable. Therefore, this paper does not regard a delayed global choice but an instant global choice.

4 Petri Nets for Message Sequence Charts

The formal MSC semantics as standardised by the ITU-T [ITU98] defines a process algebra semantics. An MSC is decomposed into its single events according to the MSC grammar. These events are then mapped to process algebra actions and composed to process terms employing process algebra composition operators. This semantics has

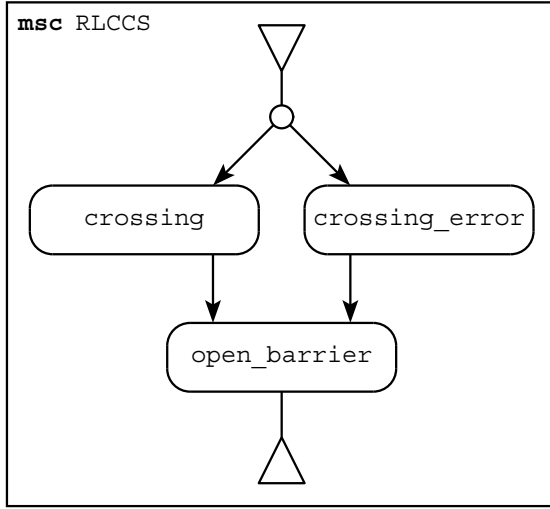


Fig. 6. High Level MSC giving the sequence for the scenarios in Figures 3 to 5

turned out to be not very practical, since the process algebra and its process terms are quite hard to grasp. Especially the use of the semantic model for further specification purposes suffers from the lack of analysis and simulation methods and according tool support.

This paper proposes to employ Petri nets as a formal model for MSC. Petri nets have the advantages of being formally precise as well as intuitively understandable. Furthermore, Petri nets are well established and provide analysing and simulation algorithms and tools. MSC specifications with loops and alternatives require high-level Petri nets to represent the specified scenarios adequately and correctly. However, due to space restrictions loops and alternatives are omitted in this paper. Therefore, low-level Petri nets are sufficient and are employed in this paper to achieve a better understanding of the transformation. In general the presented approach employs AHL-nets [PER94] which combine Petri nets with algebraic specifications.

The idea for a Petri net representing an MSC instance is as follows: The top and the bottom symbol in the Petri net are represented by a start and an end place. The transitions represent the MSC events. A control flow, i.e. an execution of the instance, is represented by a token moving through the net. Beginning at the start place the token moves along the transitions and places to the end place. The passing of a transition represents the execution of an MSC event. The Petri nets for the instances are composed according to the MSC specification. This idea is analogous to the approach presented in [GGR93]. However, MSC has changed substantially and the approach presented here is more formal.

To associate a Petri net representation $\mathcal{N}[[M]]$ with each MSC specification M the following steps are performed. First, the MSC specification is decomposed into single events. Then all events are mapped to Petri net fragments. Consecutively, those

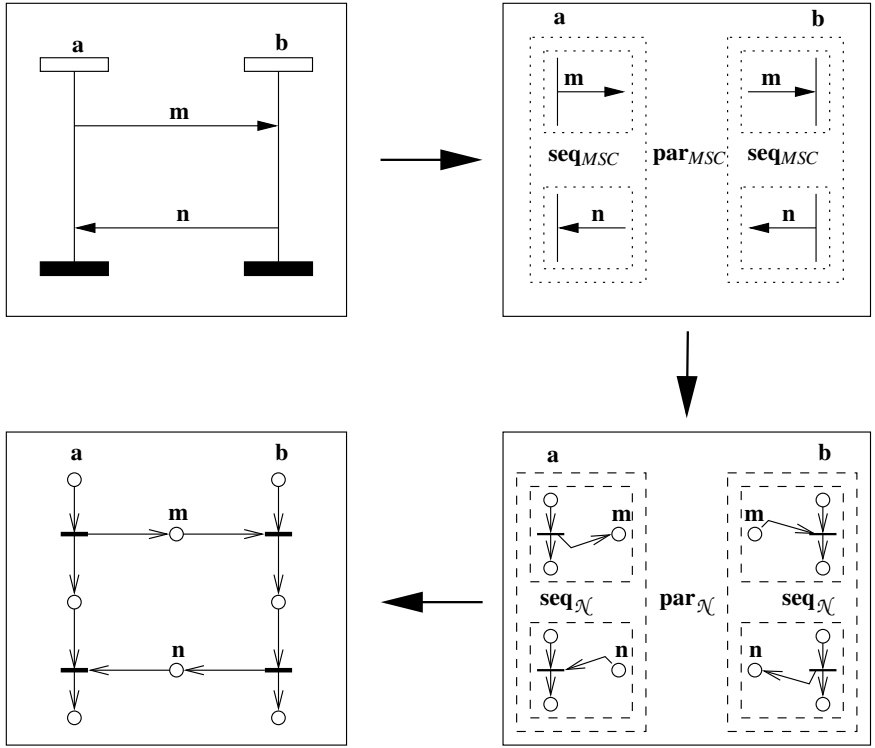


Fig. 7. Transformation of an MSC into a Petri net by decomposing the MSC into single MSC events, mapping these events to Petri net fragments and composing these fragments in correspondence to the decomposition

fragments are composed again (Figure 7). Thereby, the MSC operators op_{MSC} are represented by operators on Petri nets $op_{\mathcal{N}}$ such that

$$\mathcal{N}[[op_{MSC}(M_1, M_2, \dots, M_n)]] = op_{\mathcal{N}}(\mathcal{N}[[M_1]], \mathcal{N}[[M_2]], \dots, \mathcal{N}[[M_n]])$$

We consider labelled Petri Nets defined as (P, T, F, i, l, e) where P is the set of places, T is the set of transitions and F is the flow relation $F \subseteq (P \times T) \cup (T \times P)$. The meaning of the labelling functions i , l and e is given in the following paragraphs. An initial marking is not given. A token represents the point of control. Thus, a process starts if a token is in the begin place.

An MSC event is represented by a Petri net transition. To be able to compose Petri net fragments, the mapping i from places and transitions to the set of all MSC instance names I and the mapping l from places to controlling labels and the set of all MSC message identifiers C are introduced. The mapping e from transitions to MSC events E is necessary in order to match a Petri net transitions with the corresponding MSC event. Thereby, the mapping $i : P \cup T \rightarrow I$ assigns the MSC instance name to places and

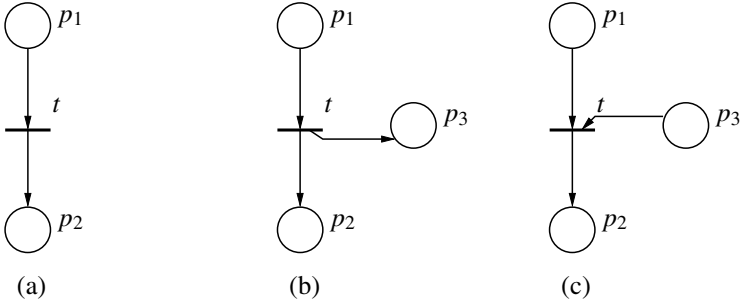


Fig. 8. Structure of Petri net fragments for MSC events: (a) local action, (b) send event and instance creation respectively and (c) receive event

transitions. The labelling $e : T \rightarrow E$ connects the event names to the transitions and $l : P \rightarrow \{\text{begin}, \text{end}, \text{init}\} \cup C$ connects controlling labels or MSC message identifiers to several places.

Three different types of events are distinguished in the Petri net representation (see Figure 8): local events, events with an outward effect and events reacting to an outward effect.

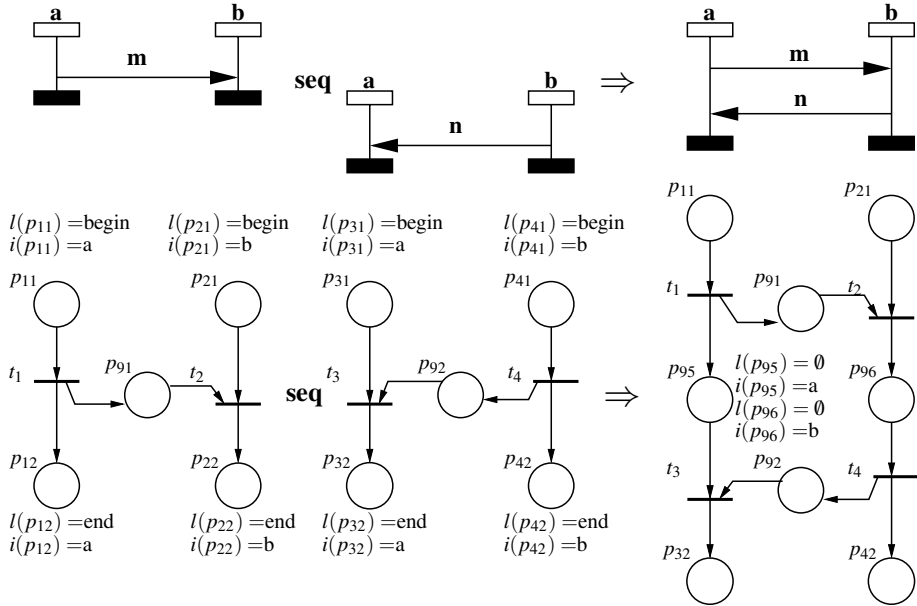


Fig. 9. Sequential composition of MSC and of Petri net fragments with common instances, p_{12} and p_{31} are agglutinated as p_{95} , p_{22} and p_{41} are agglutinated as p_{96}

Basically three different MSC composition operators exist: the sequential, the parallel and the alternative composition operator. The operators for constructing loops and describing optional and exceptional behaviour can be constructed from these three operators.

These composition operators assemble the Petri net fragments in order to obtain the resulting Petri net for the complete MSC specification. The events along an instance axis are composed sequentially. The resulting Petri nets for the MSC instances are composed in parallel. Furthermore, the subnets are composed according to the MSC composition operators as specified in the MSC specification.

Open communication places must be checked for each composition. If a place exists for a send message in one Petri net and a place for a received message in the other Petri net and the message name, sender, and receiver are identical, then these two places are replaced by a new place with the same labels. Then this place is linked to both Petri net fragments accordingly.

Sequential composition with the weak sequencing semantics is the simplest operation. For every instance which occurs in both subnets the end place of this instance in the first net and the start place of this instance in the second net are agglutinated. Figure 9 shows an example for an informal representation of the sequential composition of MSCs and according Petri net fragments.

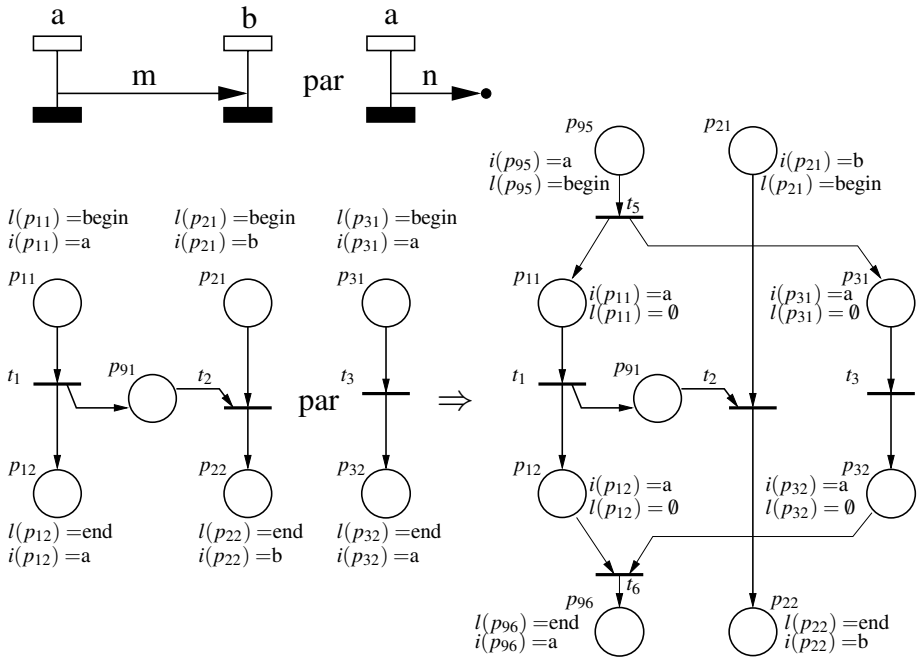


Fig. 10. Parallel composition of Petri net fragments with common instances which correspond to the MSC fragments depicted above

The parallel composition operator doubles the control flow for all instances which occur in both subnets and joins the control flows at end as demonstrated in the example which is depicted in Figure 10.

More details on the composition operators can be found in [Klu00].

Figure 11 shows the low-level Petri net which is derived from the application of the Petri net semantics to the MSC `crossing`.

Figures 12 and 13 demonstrate the corresponding results for the MSCs `crossing_error1` and `open_barrier`. These Petri nets are composed alternatively and sequentially according to the HMSC RLCCS (Figure 6). The result is given in Figure 14.

Note that in the alternative the control flow branches for every instance. A subnet is employed as global control structure to ensure that all instances choose the same alternative. Since only one run of the scenario is considered, one token on a place called “init” is sufficient. Depending on the global branching the token passes one of the two transitions following the “init” place.

If this global structure is omitted, then the control flows of communicating instance can run into different alternatives leading to deadlocks, so called race conditions.

The instance `External_Control` is also present in the left alternative although it does not come into action. However, it is necessary in order to represent the alternative of not acting or of correcting the barrier position.

The point at which it is decided which alternative is to be followed is a further problem of this system model. The decision is made long before the barrier position is known. In actual fact the barrier position toggles in between the regular and the failure behaviour. Unfortunately, this delayed choice cannot be generally resolved due to the undecidable equivalence of MSCs or parts thereof [MPS98, Mus99].

So far the presented low-level Petri net can only execute one instantiation of the RLCCS scenario since the subnet for the global branching control cannot distinguish tokens belonging to different instantiations. Therefore, instances of all instantiations follow the same alternative as the first one. Hence, a high-level Petri net must be employed if the Petri net for the RLCCS scenario is instantiated more than once.

5 Analysis of the Resulting Petri Net

The composition of MSCs by means of MSC inline and reference expressions and High-level MSCs renders MSC much more powerful than other similar specification languages. It also allows the specification of infinite event sequences. Unfortunately, it is the composition that sometimes leads to MSC specifications containing unintended deadlocks or process divergence. Furthermore, the weak sequential composition together with the non-deterministic branching choice can lead to syntactically correct MSC specifications that either cannot be realised or specifications with unintended behaviour.

The Petri net resulting from the transformation of an MSC specification can be employed as a semantic model to detect these properties.

Deadlocks in Petri nets are usually detected by exploring the reachability graph, i.e. the possible markings of the Petri net. Unfortunately, this method works almost only for

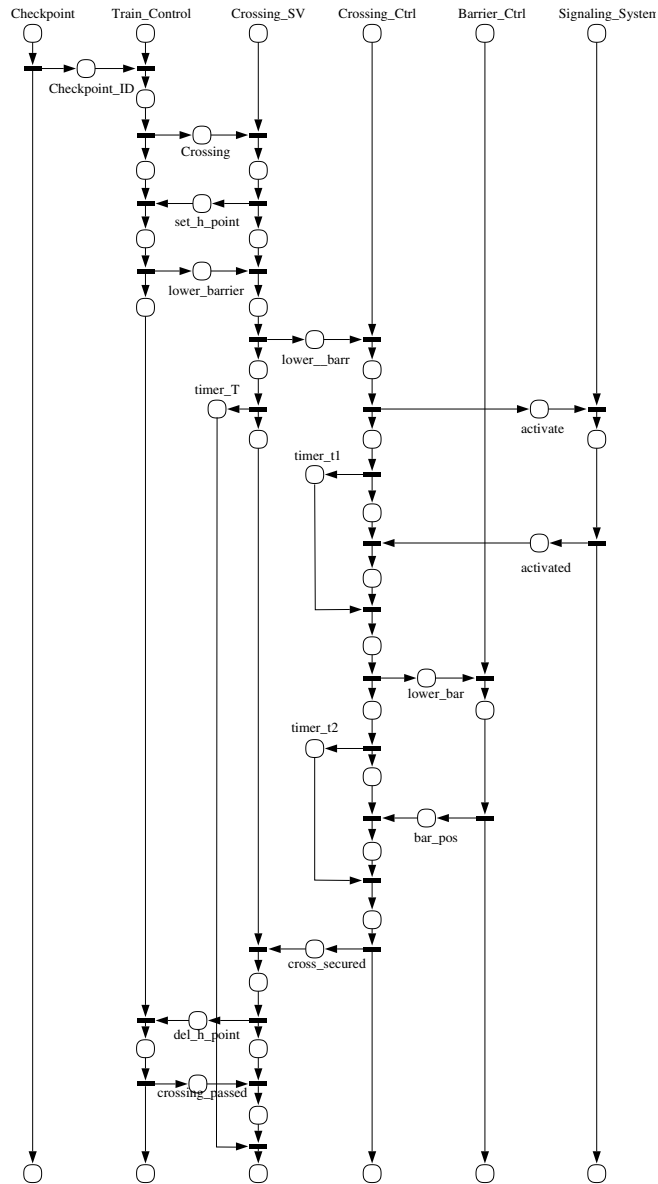


Fig. 11. Petri Net for the Regular RLCCS Case

finite Petri nets. Only a high-level Petri net represents the correct behaviour of an MSC specification if the MSCs contain branchings and loops or if multiple net instantiations are regarded. As these high-level Petri nets for MSC usually have infinite reachabil-

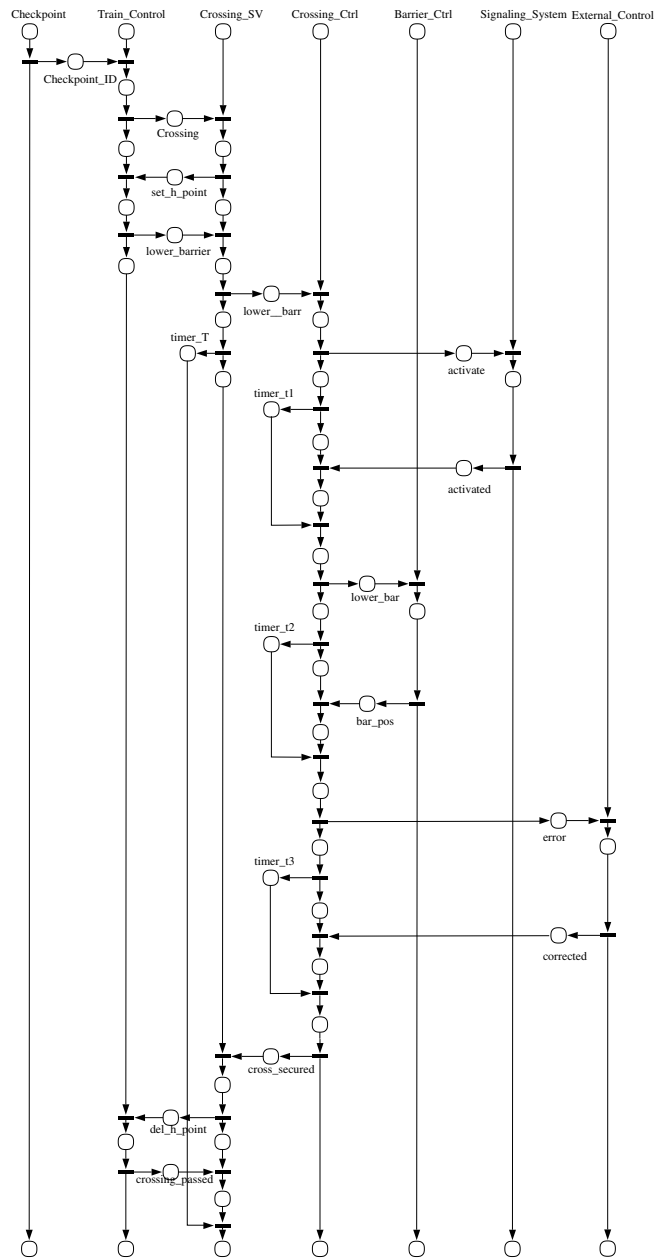


Fig. 12. Petri Net for the Failure RLCCS Case

ity graphs the exploration of the nets complete reachability graphs is impossible and, therefore, the application of this method for detecting deadlocks cannot be employed.

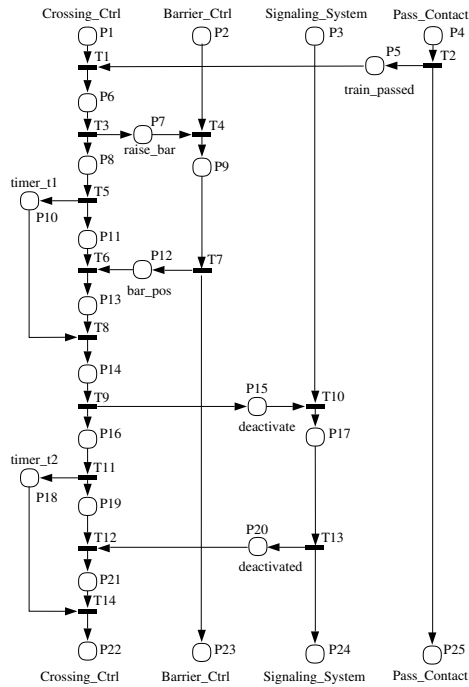


Fig. 13. Petri Net for Opening the Barrier

In order to be able to detect deadlocks in an MSC specification, this paper suggests the characterisation of these situations in MSC specifications which lead to deadlocks. The only source for a deadlock is an instance, which tries to receive a message that is never send. The MSC standard restricts the use of send and receive events. Therefore, only one possibility for a deadlock for a syntactically correct MSC specification remains. This is the case when a message exchange via an MSC gate takes place where more receive events are executed than messages are send. Figure 15 shows an example MSC which embodies such a situation. In order to detect these situations the boundaries of the surrounding loops and the optional executions of the send and of the receive events are detected and, subsequently, the number of minimal repetitions of the send and the number of maximal repetitions of the according receive events are calculated. The advantage of employing the constructed Petri net instead of the MSC specification for this analysis method is, that the syntactic matching of send with receive events and of MSC references with MSCs in loops stretching across several MSCs is not necessary for each analysed pair of send and receive events.

In the MSC context a global branching choice is made at each alternative branching. However, usually no global branching control exist in a real system. Therefore, it is important to study if a global branching can be replaced by local branchings.

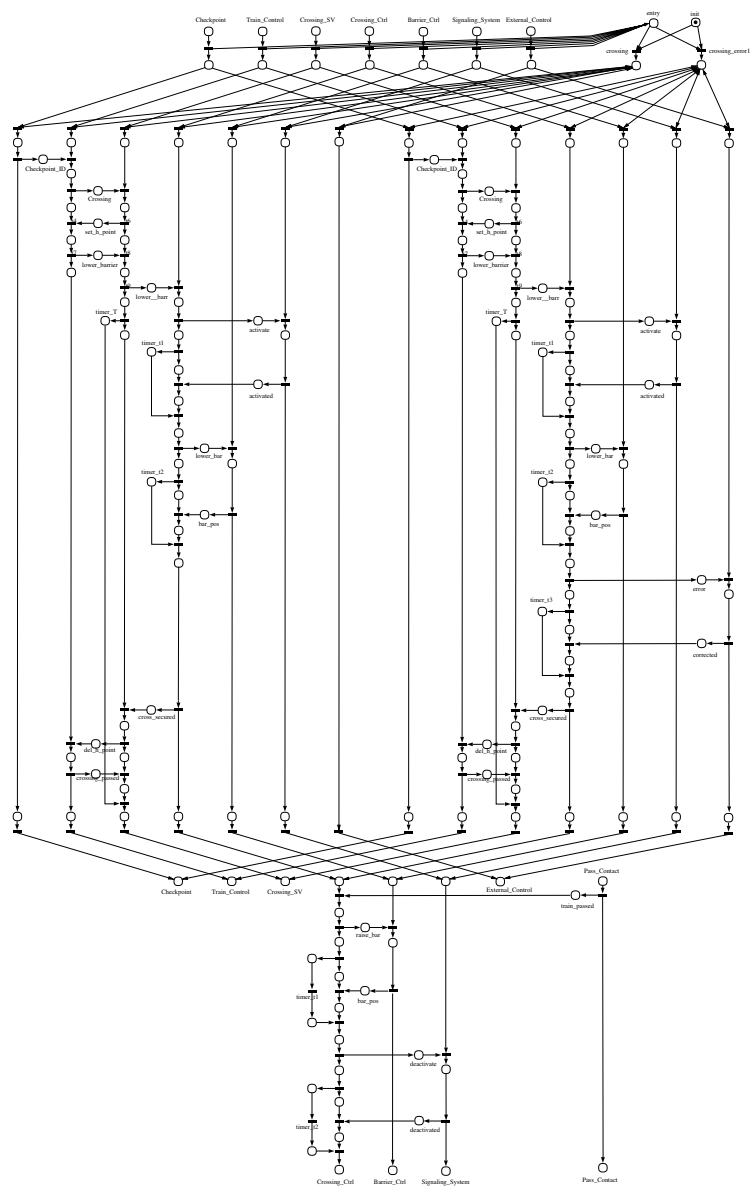


Fig. 14. Low-level Petri net for the complete RLCCS scenario as specified in Figures 3 to 6

A global branching can only be replaced by a local alternative if only one out of all connected instances actively decides which alternative is to be chosen. In the following this instance is called the active instance of an alternative. All other instances must run

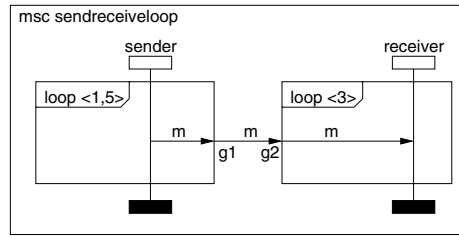


Fig. 15. MSC with two MSC loop inline expressions exchanging messages via MSC gates containing a deadlock situation

into this alternative chosen by the active instance. These are, therefore, called passive instances of the alternative.

The information, which alternative branch is chosen by the active instance, must be propagated to all passive instances. This can only be specified by sending specific MSC messages from the active to all passive MSC instances. In this case the assumption is that a receive event triggers the alternative of the passive instances. Therefore, different messages must be send for the different alternatives.

In this section the Petri net for the RLCCS scenario is analysed. Analysing the low-level Petri net for the scenario is sufficient, because the RLCCS model does not contain any loops. However, only the AHL-net gives the correct behaviour if more than one system run at a time is regarded. If not stated otherwise, the following facts apply to the low-level and the AHL-net.

Both, the low-level and the AHL-net, are deadlock free due to the sufficient reason that no MSC gates are employed for the message exchange and, furthermore, that the MSC semantics does not allow race-conditions, i.e. instances running into different alternatives. The standardised formal MSC semantics as well as the Petri net semantics are constructed as such that race-conditions are not allowed and that the Petri net models do not contain deadlocks resulting from race-conditions.

As mentioned in the previous paragraph MSC gates are not employed for the message exchange. Therefore, a one-to-one mapping of send and receive events is always possible and the Petri net model cannot contain open send or receive places.

Process divergence describes the situation in which two concurrent processes run in an infinite loop and one process does not need to wait for the other one but it can iterate faster then the other. If the faster process sends a message which is received by the slower one, this can lead to flooding the receiver with messages. In connection with MSC process divergence was first described in [BL96a]. It may occur in connection with loops or cycles in MSC specifications. The MSC specification studied in this chapter does not contain cycles or loops and, therefore, process divergence cannot occur.

The approach presented in this publication assumes for every alternative to represent a global, i.e. non-local, branching choice. Further on such a global branching choice op-

erator can be examined whether it can be replaced by a local branching choice without changing the semantics of the specification.

The approach discussed above differs from other approaches as these assume an MSC branching choice to be purely local and test if this leads to race-conditions, i.e. is not be chosen locally.

Comparing the left and right branches of the alternative in the studied RLCCS-example shows that the behaviour is identical apart from the correction of the barrier position in the right scenario. This implies, that only the instances directly controlling and correcting the barrier position, i.e. `Crossing_Control` and `External_Control` are involved in alternative behaviour. Furthermore, their behaviour differs in just one specific section. Before and after this section they behave identically.

However, according to the MSC specification a global choice between the MSCs `crossing` and `crossing_error` must be made. In the Petri net a subnet controls the global branching choice.

The presented approach assumes that receive events are never executed on their own, but their execution is triggered by a sent message waiting to be received. A global branching choice can be replaced by local branching choices if these conditions are fulfilled by the subnets corresponding to parts of the MSC instances, which are involved in this global alternative.

Performing these tests on the Petri net from Figure 14 yields that the global, non-local choice cannot be replaced by a local choice. Although just one instance sends a message as a first event in the alternative and all other instances receive a message and the instance sending the message is the same in both alternatives, the condition, that different messages are sent in different alternatives does not hold. Therefore, an instance receiving a message cannot decide locally which of the alternatives with an equal prefix must be chosen.

Still, the Petri net can be manually changed so that the branching only covers the two involved instances. The branching point for both instances can also be moved to the point at which the alternatives differ. The control flows can be joined when the behaviour for both alternatives is again the same. This simplified Petri net for our RLCCS scenario is given in Figure 16.

In this simplified Petri net the instance `Crossing_Control` decides which branch of the alternative is to be chosen. Thus the global control, which in this net is still present, could be moved to this instance. Although it is still necessary for the external control to know which alternative it has to choose, a Petri net results which can be implemented by distributed objects.

Unfortunately, the simplification of the branchings cannot be automated due to the undecidable MSC equivalence [Mus99]. In general, therefore, it cannot be decided which is the equal pre- or postfix of alternatives.

The results of the presented analysis can be employed to revise the MSC specification of Figures 3 to 6. In order to do so, the common prefix of the alternative MSCs `crossing` and `crossing_error` is moved out of the alternative. Figure 17 demonstrates one of numerous possibilities to do this. Thereby, the alternative is moved into an alternative inline-expression. The depicted MSC `crossing_new` replaces the alternative in between the two MSCs mentioned above and is followed sequentially by the

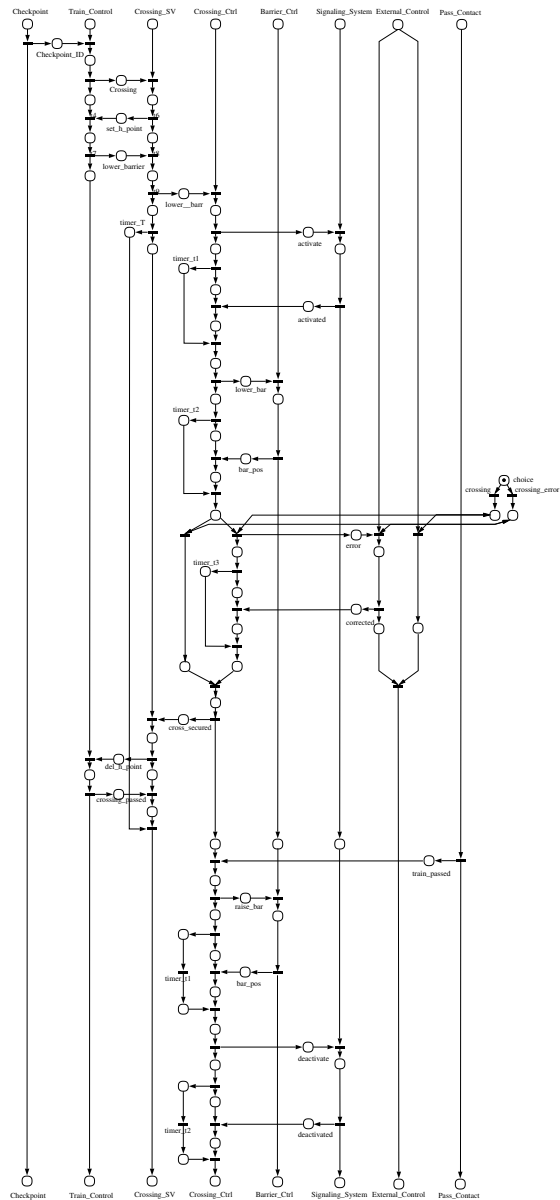


Fig. 16. Simplified Petri net for the complete RLCCS scenario

MSC open_barrier. One of the alternatives in the MSC inline-expression contains only an MSC condition box. Therefore, this alternative could be omitted and an op-

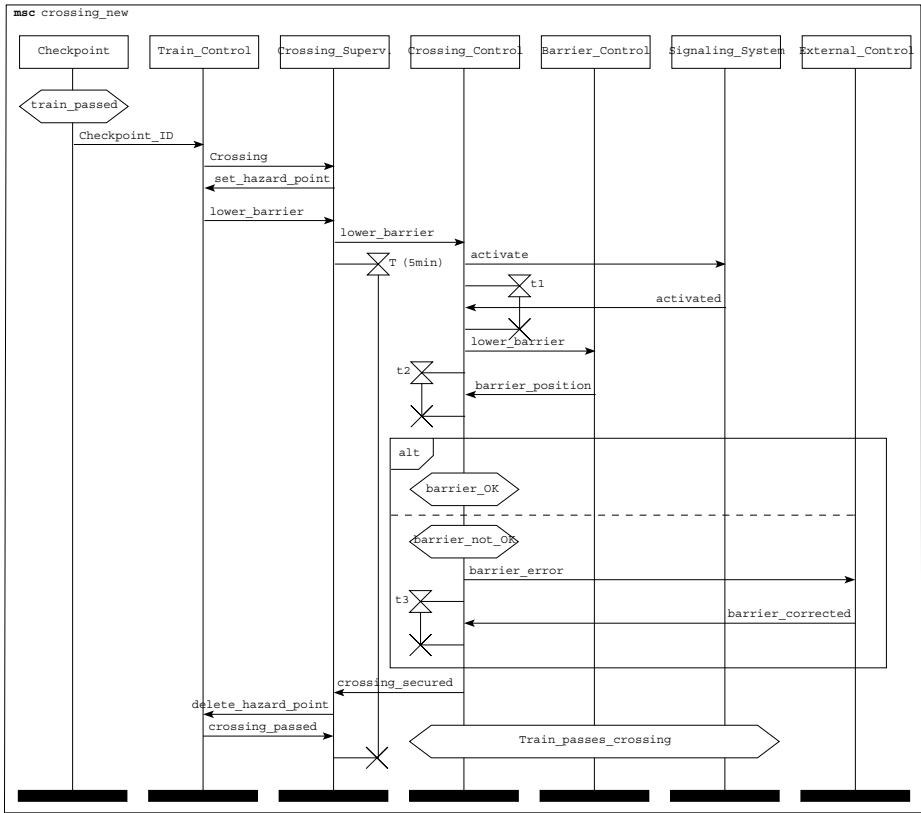


Fig. 17. Scenario - Level Crossing with Long Gates, Full System Supervision, revised Regular and Failure Case

tional inline-expression could be used instead. The alternative is used in order to increase readability.

Nevertheless, the non-local branching choice covering two MSC instances cannot be replaced by a local branching choice. This can only be done if also in the regular case the MSC instance `Crossing_Control` sends a message to `External_Control`, e.g. in case the lowering of the barrier succeeded a message `barrier_ok` send (Figure 18). Only then the `External_Control` can decide locally for the correct alternative when it received the message containing the information which alternative was chosen locally by the `Crossing_Control`.

6 Conclusion

Employing an example scenario this paper demonstrated that Petri nets are a suitable semantic model for MSCs. The railway crossing serving as the example throughout the

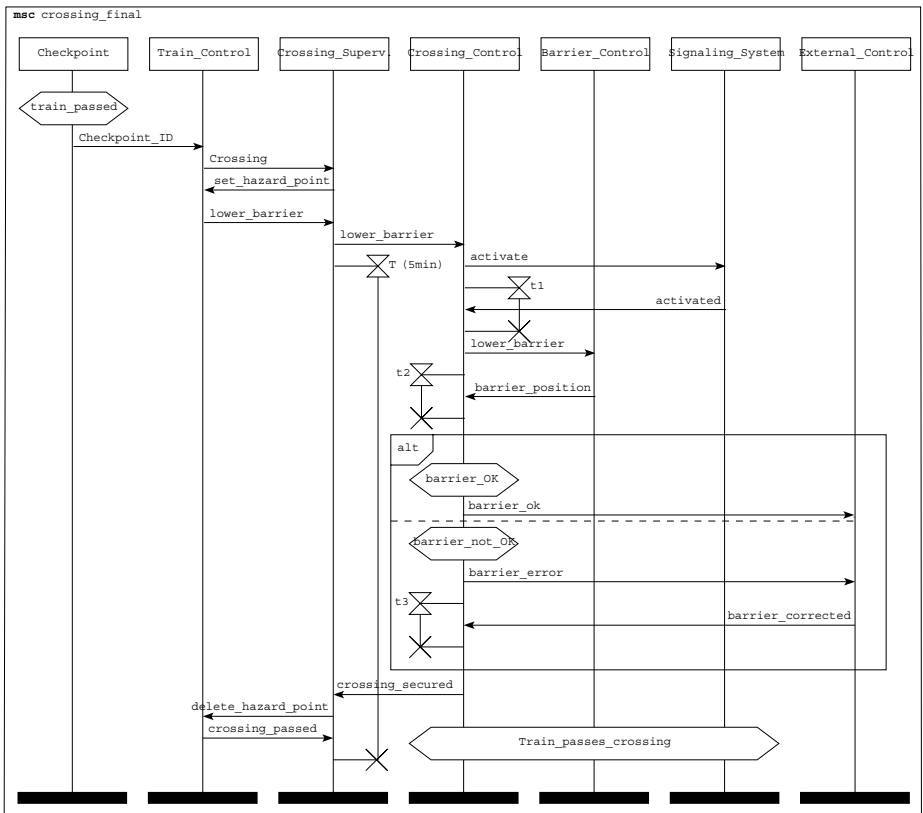


Fig. 18. Scenario - Level Crossing with Long Gates, Full System Supervision, revised Regular and Failure Case with local-branching choice

paper was introduced informally and by means of an MSC specification. The transformation of this MSC specification into a Petri net was outlined and the resulting Petri net was presented. This resulting Petri net is more concrete and intuitive than the process term resulting from applying the official process algebra semantics for MSC. Furthermore, methods and tools for the analysis and simulation of Petri nets exist. Therefore, the Petri net can be easily employed to analyse properties of the specified system.

Additionally, different approaches for analysing MSC specifications in order to derive properties of the specified system are presented, as available methods and tools for analysing MSCs are not yet sufficient to analyse the Petri nets for MSC specifications. Employing the derived Petri net for analysing the MSC specification has the advantage that only one model is analysed, instead of several MSCs with alternatives possibly stretching across several MSCs. The Petri net for the example was employed to demonstrate how properties of an MSC specification can be tested. However, advanced analysis and simulation tools and methods for the employed Petri nets could simplify the analysis of MSC specifications even more.

References

- [BL96a] H. Ben-Abdallah and S. Leue. *Syntactic Detection of Process Divergence and non-Local Choice in Message Sequence Charts*. University of Waterloo, Canada, 1996. 213
- [DDK99] W. Damm, G. Dhmen, J. Klose. *Secure decentralized control of railway crossings*. In: *Proceedings of the Fourth International ERCIM Workshop on Formal Methods in Industrial Critical Systems*, 1999. 200, 201
- [Ehr97] H. Ehrig et. al., *Integration von Techniken der Software Spezifikation für ingenieurwissenschaftliche Anwendungen*, <http://tfs.cs.tu-berlin.de/SPP/index.html>, 1997. 198
- [ETCS97] *ETCS Phase I - Modellbildung, Simulation und Schnittstellenbetrachtung*. Technische Universität Braunschweig, Institut für Regelungs- und Automatisierungstechnik. Projektdokumentation, 1997. 198
- [GGR93] J. Grabowski, P. Graubmann, and E. Rudolph, *Towards a Petri net based semantics definition for Message Sequence Charts*, In O. Frgemand and A. Sarma, editors, *SDL'93: Using Objects*, *Proceedings of the Sixth SDL Forum*, pages 179-190, Darmstadt, 1993, Amsterdam, North-Holland. 204
- [ITU98] ITU, *Annex B to Recommendation Z.120: Message Sequence Charts (MSC)*, ITU Genral Secreteriat, 1998. 198, 203
- [ITU99] ITU, *MSC2000, Recommendation Z.120: Message Sequence Charts (MSC)*, ITU Genral Secreteriat, 1999. 197, 200
- [Klo98] J. Klose, *Sichere dezentrale Steuerung von Bahnübergängen*, Diplomarbeit, Universität Oldenburg, 1998. 200, 201
- [Klu00] O. Kluge. *Time in Message Sequence Chart Specifications and How to Derive Stochastic Petri Nets*. In: *Proceedings of the Third International Workshop on Communication Based Systems (CBS3)*, Berlin, March 2000. 208
- [KPE00] O. Kluge, J. Padberg, H. Ehrig. *Modeling Train Control Systems: From Message Sequence Charts to Petri Nets*. In: *FORMS 2000 - Formale Techniken für die Eisenbahnsicherung*, Fortsch.-Ber. VDI Reihe 12 Nr. 441. pages 25-42. Düsseldorf: VDI Verlag, 2000.
- [MPS98] A. Muscholl, D. Peled and Z. Su. *Deciding Properties for Message Sequence Charts*. *Proceedings of FoSSaCS'98*, 1998. 208
- [Mus99] A. Muscholl. *Matching Specifications for Message Sequence Charts*. *Proceedings of FoSSaCS'99*, 1999. 208, 214
- [PER94] J. Padberg, H. Ehrig and L. Ribeiro. *Algebraic High-level net transformation systems*. In: *Mathematical Structures in Computer Science*, vol.5, pp. 217-256, 1995. 204
- [PJES99] J. Padberg, L. Jansen, H. Ehrig and E. Schnieder. *Cooperability in Train Control Systems: Specification of Scenarios using Open Nets*, Special Issue, Society for Design and Process Science (SDPS) '99, H. Ehrig, M. Goedicke, J. Padberg (eds.) 199
- [SSP-BU97] L. Jansen. *Referenzfallstudie Verkehrsleittechnik*, <http://www.ifra.ing.tu-bs.de/~m33/spezi>, 1997.

Quantitative Engineering of Business Processes with *VIPbusiness*

Jörg Desel¹ and Thomas Erwin^{1,2}

¹Katholische Universität Eichstätt-Ingolstadt
Lehrstuhl für Angewandte Informatik
85071 Eichstätt, Germany
joerg.desel@ku-eichstaett.de

²KPMG Deutsche Treuhand-Gesellschaft
Information Risk Management
Marie-Curie-Straße 30
60439 Frankfurt am Main, Germany
terwin@kpmg.com

Abstract. The *VIP*¹ approach [8, 7] aims at modeling and validating process specifications with High-level Petri nets in different application domains. It is based on simulative creation of partially ordered runs and employs advantages of these semantics with respect to efficiency and expressive power compared to sequential simulation. This contribution presents a new component to the *VIP* architecture, namely *VIPbusiness*. This component allows to apply the *VIP* approach and the corresponding *VIPtool* for the creation, validation and quantitative analysis of business process specifications. The approach aims at supporting early phases of projects where business processes and information systems have to be integrated.

1 Introduction

In recent years we have seen a rising need for the integration of information systems and business processes. For information systems there has been a paradigm shift from supporting only isolated activities towards supporting a company's business processes as a whole. However, corresponding projects such as the deployment of *enterprise resource planning (ERP)* or *workflow management (WFM)* systems have shown that the integration of information systems and business processes can be very complicated tasks to accomplish in a successful way. First problems already occur in very early stages of such projects. The main objective of these stages is to come up with a design of the business processes, e.g., a *descriptive model*, that can serve as a basis for the further steps towards the actual implementation and integration of the information systems. In order

¹ Verification of Information systems by evaluation of Partially ordered runs, the corresponding research project was funded by DFG and conducted at the Universities of Karlsruhe and Frankfurt, Germany.

to avoid the manifestation of bad business process designs by using them as an implementation basis, the (re-)design of the affected processes in the context of a *Business Process (Re-)Engineering (BPR)* project becomes a crucial task to be completed before system-oriented phases such as implementation can begin. The quality of the business process designs yielded in early project phases is one of the key success factors for the integration effort as a whole. It mainly concerns the following three aspects:

Correctness w.r.t. the Application. This means that the process specification should be a correct image of the specified process. And it also means that this process really does what it is supposed to do. Correctness w.r.t. the application can be checked by means of *validation* of the specification, i.e. the specification itself or runs or properties of the specification are visualized and discussed with the experts of the application domain.

Consistency w.r.t. Logical Specifications. Consistency concerns the logical structure of a process specification. The process should not run into a deadlock, it should finally reach an end, and some more application dependent requirements have to be met. Consistency can be checked by means of *verification* given that the specification or its behavior is not too complex.

Efficiency w.r.t. Performance Parameters. This property mainly concerns the time, e.g., throughput time, and the cost necessary. Efficiency can be checked by means of (possibly iterative) application of *quantitative analysis methods*.

So far work in the context of the *VIP* project has mainly dealt with issues concerning *correctness* of specifications, although some concepts have been developed to ensure *consistency* of specifications as well [8, 11]. The contribution of the present paper is to motivate the *VIPbusiness* approach, an extension of previous *VIP* concepts with a strong focus on *efficiency* of specifications, especially specifications of business processes.

The approach is based on High-level Petri nets [26, 17]. There are several reasons for using Petri nets for specification of business processes: formal semantics, graphical nature, expressiveness, vendor independence and the availability of many analysis techniques and tools (cf. [1, 9]) Hence Petri nets have emerged as a premier choice for specification when it comes to finding a compromise between user-friendly features such as a graphical modeling formalism and formal semantics needed for simulation and analysis of rather technical aspects. What makes our approach different from similar approaches based on Petri nets (cf., e.g., [24, 13]) or closely related formalisms such as *event-driven process chains (EPCs)* [27] is its combination of partial-order based simulation and a state-based timing concept. This combination allows for an efficient way of performing quantitative analysis for conceptual specifications in early phases of integration projects.

Before we start with presenting the approach we first settle some basic terms and give a brief overview and assessment of related concepts.

The remainder of this contribution is organized as follows. In Section 2 we introduce our 3-step approach for business process specification and analysis. We

give an overview of the framework and then briefly introduce all three steps in more detail with a focus on the concepts for quantitative analysis. In Section 3 we introduce the *VIPtool*, a software prototype we have built in order to validate our concepts. Using a small example from handling claims at an insurance company we demonstrate how our approach can be used to gain useful insight into the design of a business process specification.

1.1 Terms

There are different types of business processes which can be classified according to the dimensions *structuredness* and *frequency*. In this paper we focus on *structured* business processes (referred to as *workflow processes* in [2]).

When dealing with the specification of business processes, usually first the specification (or model) is created, often using Petri nets or related formalisms such as EPCs, and then runs are defined as a derived concept. This way of specifying runs follows an algorithmic paradigm; a business process definition can be executed and each execution generates a run. In this paper we stick to this algorithmic paradigm but like to mention that business process definitions can also be constructed from a set of specified runs (cf. [6]), allowing for more flexibility in the specification formalism, for the modeling of *non-structured* parts of business processes as well as for the modeling of *known exceptions*.

We view business processes as being *case-based*, i.e., every piece of work is executed for a specific *case*. Examples of cases are a mortgage loan, an insurance claim, a tax declaration, an online-order or a request for information. A business process is designed to handle similar cases by executing *tasks*. The execution of a task is referred to as an *activity*. A *business process specification* describes which tasks need to be executed to get the work for cases done as well as the causal relation that exists between these tasks.

1.2 Related Approaches

Methods to support the (re-)engineering of business process specification in the context of information systems should fulfill a number of requirements:

- graphical specification/modeling,
- formal semantics for the execution/simulation of the model,
- support for systematic simulation and analysis in early design stages,
- specification of non-structured process parts and known exceptions,
- resulting in a descriptive model of the business process that can more or less directly be used as a starting point for implementation activities.

Since the original works of Hammer and Champy in [14] there have been numerous publications on the subject of business processes and their modeling/specification. However, most approaches from the area of *Business Process Engineering* (e.g., [5, 18, 22, 23]) focus on *qualitative* aspects (i.e., organizational

aspects, media gaps etc.) as opposed to *quantitative* (*performance*-related) aspects such as throughput time, costs, resource utilization and the like. Although the corresponding specification formalisms tend to be quite simple and are hence suitable for the conceptual work that is to be done in early project phases, they hardly contain any kind of formal semantics that could be the basis for quantitative analysis involving simulation. In a lot of approaches only a set of rules for (re-)design of business processes is provided (e.g., [14, 5, 25]) without any suggestion on how to create specifications at all. The paper [4] is one of the few papers targeting at quantitative redesign rules for business processes. It uses heuristics based on results from queueing theory.

Most approaches from the area of *Workflow Management* (e.g., [15, 19–21]) concentrate on one of the numerous considered technical issues (e.g., aspects of distributed systems and data integrity). A common task is to create specifications that can directly be used as input for control units of corresponding systems. However, the rather detailed level of the resulting specifications (as well as their sometimes non-graphical nature) tends to make them less applicable in *early* stages of corresponding projects where conceptual issues are more important.

2 The *VIPbusiness* Approach

2.1 A 3-step Approach

In this section we briefly present the basic ideas of our framework for the (re-)engineering of business processes. The approach has been discussed in more detail in [7]. It consists of three steps that are performed iteratively until the model is valid and satisfies desired properties (cf. Figure 1).

In *Step 1* a specification of a business process is created using High-level Petri nets. At this stage values for time and costs do not necessarily have to be considered. The modeling should rather concentrate on building an appropriate *logical structure*, i.e., identifying necessary tasks and putting them in a meaningful order.

The behavior of a Petri net can be described using *sequential* or *causal semantics*. We will use causal semantics and causal nets for the representation of the behavior of the specified business processes. We will refer to an execution of a business process specification as a *concurrent run*. A concurrent run of a work process results from the execution of *tasks* for a given case, i.e., it is made up by the corresponding set of activities and the order in which these activities are carried out, starting from an initial configuration (e.g., the arrival of a customer request) and ending with some defined final state (e.g., the completion of the request). For the representation of runs we use a class of (low-level) Petri nets, namely *causal nets*. Causal nets are characterized by the following properties:

- Each transition has at least one ingoing and at least one outgoing arc,
- each place has at most one ingoing and at most one outgoing arc,
- there is a distinguished place c_i which has no ingoing arc,
- there is a distinguished place c_f which has no outgoing arc,

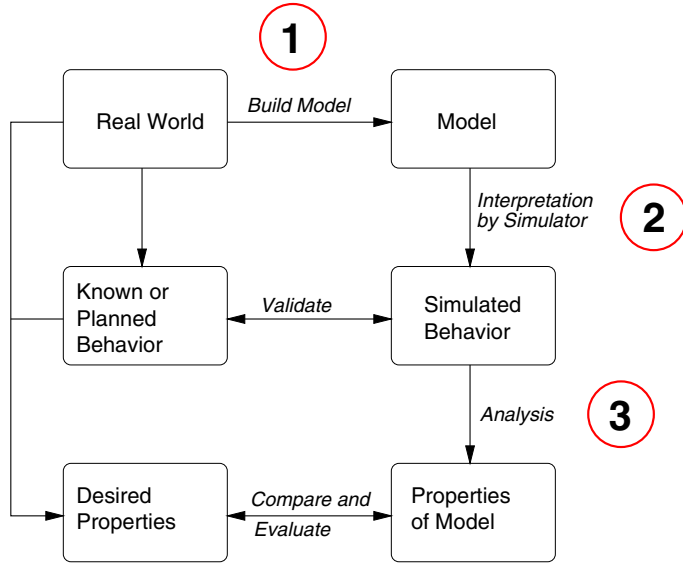


Fig. 1. *VIPbusiness* : a 3-step approach for business process engineering.

- the transitive closure of the flow relation is irreflexive and thus a partial order on the set of net elements, i.e., there are no directed cycles.

The firing of a transition, i.e., the execution of the activity modeled by the transition, is displayed by an *event*. An event is a transition in the causal net representing the corresponding concurrent run. Likewise, *conditions*, i.e., places in the causal net, model the local states that hold between the execution of activities and constitute their causal relations.

In order to get an idea of the possible behavior of the specified business process, in *Step 2* of our approach a relevant set of concurrent runs of the High-level Petri net model is generated and stored for later analysis. This (in most cases) time-consuming procedure can be performed independently from any values chosen for time and costs. The generation of runs can be done automatically or interactively. Note that this step only has to be repeated if the structure of the business process is changed. As long as this is not the case, all further analysis steps can be performed for the set of runs generated once.

Step 3 of our approach starts with transferring time and cost values from the business process specification (which by now should have been included into the High-level Petri net model) to the set of runs generated in Step 2. Now these generated runs can be used for performance analysis of the business process.

Since the same set of runs can be used for different sets of values for time and costs, the results for performance measures are based on the same causal structures and therefore are comparable. This is a key difference to approaches where

the timing concept affects the behavior and hence the set of runs is different for different time values.

As the analysis techniques use runs that have already been generated, the time consumed by the analysis itself is significantly reduced. The effects of values chosen for time and costs can be examined immediately. Since rather little time is needed for repeating this step for all kinds of time and cost values (compared to generating a new set of runs each time) this step can be easily be performed numerous times in order to experiment with different value estimates for time and costs.

In the following we will have a closer look at each of the three steps.

2.2 Modeling

The starting point for the building blocks we use for creating our business process specifications are *VIPmodel* nets, a High-level Petri net class introduced in [11]. This net class aims at combining the compact notation of classic Predicate/Transition nets [12] with the implementation focus of Colored Petri nets [17]. Formally, VIP model nets employ data types for places, (tuples of) variables and operator symbols for arc inscriptions as well as corresponding syntactical means for transition guards. Their semantics are given in the usual way: A transition can occur for an assignment of the variables of surrounding arcs and its guard if the necessary tokens are available in the input places and the guard evaluates to true. Its occurrence removes respective tokens from the input places and adds respective tokens to the output places, according to the evaluation of the arc inscriptions, respectively.

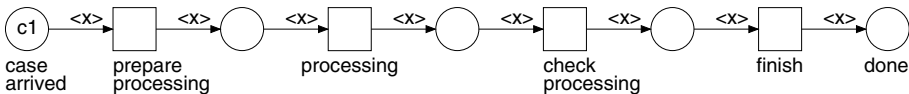


Fig. 2. Handling insurance claims: a coarse specification.

Example. Figure 2 shows a very coarse specification of a business process for handling insurance claims, given in terms of a *VIPmodel* net.

We will now introduce some enhancements, namely *activity transitions*, *routing primitives* and a *timing concept*.

Activity Transitions. Although they are important for the actual execution of a business process, technical (or organizational) details usually do not matter for the analysis of the structure of a business process. In most cases, by modeling too many details the expressiveness of the model with respect to the underlying structure of the business process is spoiled. Therefore it is of great value for the modeler to have a choice between several levels of abstraction. This enables him

to emphasize different views of the model which might be on a more *detailed* or on a more *conceptual* level. In our approach we introduce *activity transitions* for modeling sequences of *(part-)activities* that have to be performed for the execution of an activity as a whole. Activity transitions allow for the representation of activities in both, a detailed and an abstracting view. Places that are within the detailed view of an activity transition are called *internal* places, all other places are referred to as *external*. The execution of the (part-)activities is modeled by internal places. A marked internal place p_{int} can be interpreted as a (local) state where the system is in the state of carrying out the (part-)activity that is modeled by p_{int} . Note that as a result the execution of a task does not correspond to the firing of a transition but to marked internal places. External places model the states between the execution of activities, i.e., waiting states. *Example.* In Figure 3 the task *processing* has been modeled using an activity transition. Note the distinction between *internal* and *external* places with respect to the activity transition.

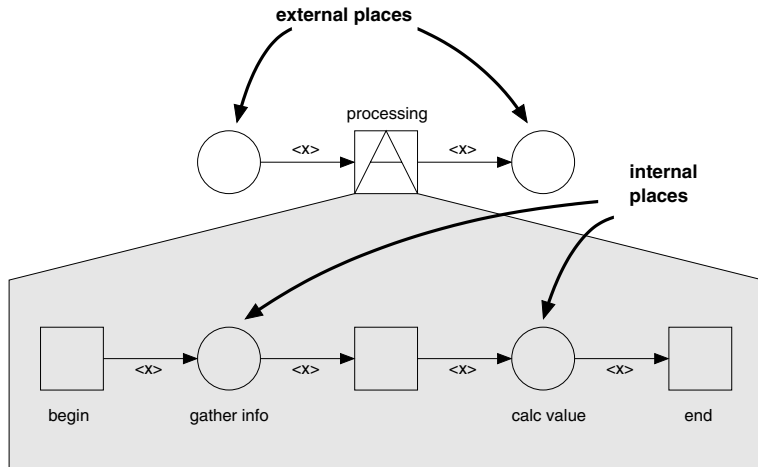


Fig. 3. Detailed view of an activity transition.

Routing Primitives. The main objective of a business process specification is to describe the (partial) causal order in which tasks have to be carried out to handle cases successfully and to define which tasks have to be executed at all, once, repeatedly etc. Hence for our formalism we need to introduce means for specifying the *routing* of cases. To this purpose we adopt a set of *routing primitives* which we use to model the most important routing concepts from [2]: *AND-Splits*, *OR-Splits* as well as their respective *Joins*. In addition to this we introduce *iteration* as a concept derived from OR-Splits. Note that we only use *explicit* OR-Splits where the choice between alternatives depends on the values of variable or on the user. Although we could do without special modeling

primitives, experience shows that it is rather hard for inexperienced users to model the corresponding concepts directly with traditional Petri net building blocks. However, the semantics of our routing primitives can be given using the High-level Petri net formalisms (e.g., Figure 4 shows the semantics of an *OR-Split*). Hence these routing primitives can be viewed as encapsulations of the underlying Petri net concepts.

Example. In Figure 5 the task *prepare processing* has been split up in two concurrent tasks using an AND-Split. Figure 6 shows how an OR-Split can be used to model routing decisions. In Figure 7 an OR-Split is used to model iteration.

Resources. The execution of tasks requires *resources*. A resource can either be a machine (e.g., a printer or a fax) or a person (e.g., a worker, an employee). A *resource class* is a group of resources with similar characteristics. There may be many resources in the same class and a resource may be a member of multiple resource classes. However, in this contribution we assume that each resource belongs to a single resource class. Resource classes are modeled by *resource places* which are additional places in our model. Additional arcs specify which resources are needed by the tasks. Single resources, i.e., *instantiations* of a resource class, correspond to the marking of a resource place. Note that by simply omitting the resource-related places and arcs we automatically get "normal" *VIPmodel* nets as defined above. As a result, the simulation concepts of *VIPsim* can be applied in a straightforward way. Furthermore, when compared to the runs generated without consideration of resources, the effects of limited and/or shared resources on the possible behavior of the business process specification become quite visible (as we will see in more detail in Section 3).

Example. In Figure 8 a resource representing an *employee* has been added to the tasks *create file* and *acknowledge receipt*. We will examine the effects of shared resources later on.

Time. A notion of *time* is a mandatory prerequisite for answering any questions related to the performance of real systems. The integration of time into Petri nets has been discussed extensively (cf. [2, 3, 16, 28]). The basic idea of our timing concept is to link time to the places of our nets. Time is not consumed by the firing of transitions (as in most other approaches) but *between* firings of transitions. The firing itself does not consume any time. This timing concept corresponds to the idea of modeling (part-)activities by internal places of activity transitions (see above). That way our approach is completely *state-based* [1], i.e., the state of the specified business processes can be completely determined by the marking of places. This allows to distinguish between the enabling and the actual execution of (part-)activities which is a mandatory prerequisite for the modeling of waiting time. Time values can be fixed or variable. For analysis purposes with respect to the dynamic behavior of the model, time values will be transferred to runs and then be interpreted there.

There is an explicit distinction between time values for internal and external places. Internal places are inscribed with a fixed (or stochastic) *activity time* value which corresponds to the target value for the time needed to execute the (part-)activity that is modeled by the place. Waiting time is linked to external

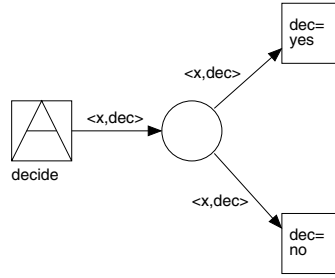


Fig. 4. Internal view revealing the semantics of our *OR-Splits*.

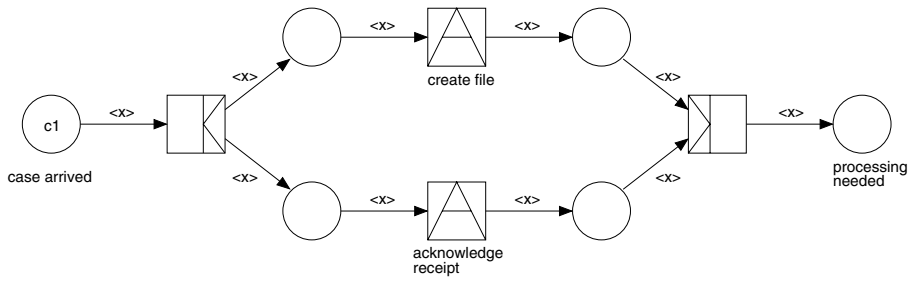


Fig. 5. AND-Split for preparing the processing.

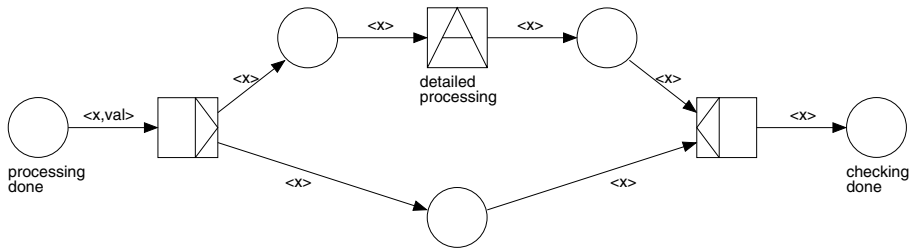


Fig. 6. OR-Split for deciding on detailed processing.

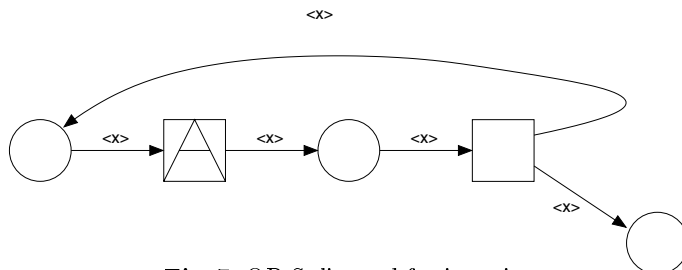


Fig. 7. OR-Split used for iteration.

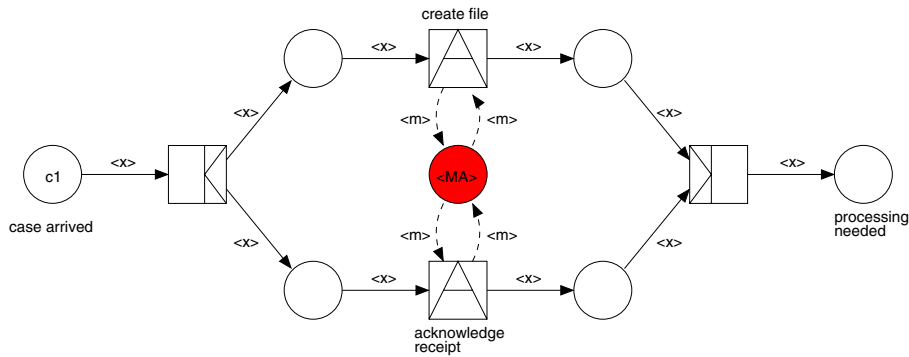


Fig. 8. Adding resources.

places. Waiting time results from the synchronization of (sequences of) concurrent activities that differ with respect to the time needed for execution. If there are alternative activities with different respective activity times, the values for waiting time can be different from run to run. Hence, when building a model of a business process we use variables for waiting time. Waiting time can only be determined for a single run.

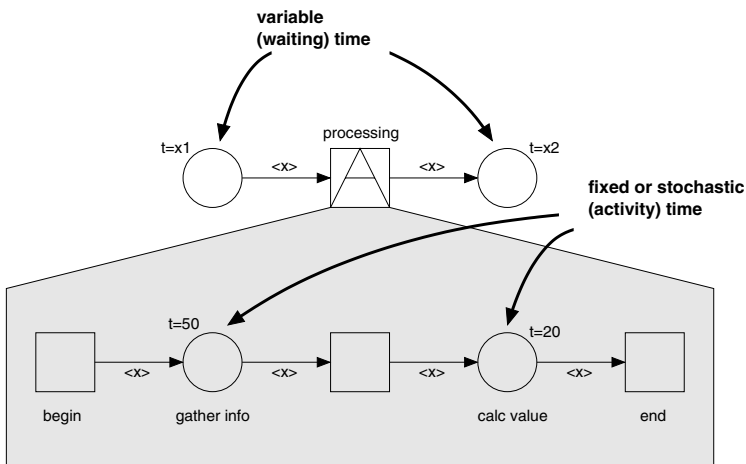


Fig. 9. Adding time.

Example. In Figure 9 time inscriptions have been added to internal and external places of the activity transition *processing*.

The possible behavior of a business process specification can now be described by a set of *time-inscribed runs*. Each time-inscribed run is a run of the business process model where conditions have a natural number as an inscription. This number reflects the life-span of a condition, i.e., the amount of time the condition holds (which corresponds to the amount of time a token remains at the corresponding place). For internal places this value corresponds to the time inscription (the activity time) of this place. Conditions that have no successor and therefore "hold forever" have no associated number as they represent final states of the business process execution. In addition we require that the sum of life-spans of conditions for each pair of paths that lead to an event be equal. Furthermore the sum of life-spans of conditions on maximal paths has to be equal as well. This sum is the throughput time of the time-inscribed run. This characteristic is exploited by our methods for performance analysis.

The integration of *costs* into our models is accomplished using an approach similar to the one used for time. As we assume costs to be a function of time we use fixed values for *activity costs* and variable values for *waiting costs*, which are assigned to internal and external places, respectively.

2.3 Simulation

In our terminology, simulation means *generation of runs*. This is precisely what the *VIPtool* does in an efficient way. For details on the *VIPsim* simulation concepts refer to [8, 11]. Note that during the generation of runs time and cost values as well as resource limitations are completely ignored. The yielded set of runs is exclusively based on the causal relations of the business process specification.

2.4 Analysis

Qualitative Analysis. Qualitative analysis of runs concerns properties of runs such as given by invariants, termination in a given finite state etc. These properties should be fulfilled by all runs in order to make the specification consistent with given requirements. If possible, this consistency is verified by analysis of the specification. However, since this often is a quite complicated and complex task, it makes sense to check the consistency at least for the subset of simulated runs.

The *VIPtool* contains features to verify *invariants*, *goals* (i.e., specific liveness properties demanding that a defined state is finally reached) and *causal chains* (i.e., specific patterns of causal relationships between events and conditions occurring in a run [8]).

Performance Analysis. In this contribution we concentrate on *time-related* performance measures with *throughput time* being the most distinct one. In most cases the analysis of time-related performance measures is the basis for any subsequent analysis related to other important fields such as *costs* or *resources*. The starting point for our time-related analysis are *time configurations*. A time configuration is a set of time values that represent *activity time* for the tasks

within the business process specification. In accordance with our timing concept only internal places get values for activity time (see above). Hence a configuration contains either a fixed value or a stochastic function for each internal place within the Petri net model of a business process specification. No time values are assigned to external places as waiting time values result from activity time and the causal relations within a run. If a stochastic function has been chosen as a time assignment for at least one of the internal places, different sets of random values can be generated and stored for each configuration. We will refer to configurations containing stochastic functions as *stochastic configurations*. Each business process specification can have multiple configurations. For each run the throughput time can easily be determined for each configuration by using *critical path* techniques [7, 10].

Since we allow the use of multiple configurations and (in case of stochastic configurations) arbitrary sets of random numbers during time-related analysis we can reduce the risk of severe design mistakes caused by faulty assumptions for time and cost values. As we are able to test all kinds of values for time and costs rather quickly, we help to avoid situations where decisions on a business process design have to be made based on few (possibly faulty) values for time and costs.

In addition to this, dependencies between the measured quality of the business process specification and estimated or *target* values (often referred to as *service levels*) used for time and costs can be examined. When introducing time and cost values to the specification model, the values used will either have to be based on assumptions or experiences from business practice. Since these values might significantly differ from values that occur once the business process specification is put to practice, it is certainly a good idea to examine the relation between chosen values and analysis results. This is especially important in case decisions between design alternatives have to be made. Depending on the specific requirements of the company either the alternative with the best average results for all tested time and cost values or the alternative with the best value for one time and cost combination can be chosen.

In the following we introduce an approach for analysis of a business process specification with respect to throughput time. As a prerequisite we assume that a business process specification has been modeled and a set of runs has been generated. The basic idea is to subsequently examine the runs of the business process specification using a set of four different scenarios:

- 1 run / 1 configuration (*Scenario 1*)
- 1 run / all configurations (*Scenario 2*)
- all runs / 1 configuration (*Scenario 3*)
- all runs / all configurations (*Scenario 4*)

Usually examining all runs using one configuration (*Scenario 3*) with fixed time values is a good way to start. The result is a first impression of the performance of the business process specification for all different kinds of cases and the corresponding routing decisions. Significant differences between the values

yielded for throughput time can be the basis for some first analysis with respect to mandatory or possible design changes. In case the throughput time for a run is well below average for all runs, the run might either be a very efficient way of handling cases (a "role" run) or might contain flaws that allow the skipping of important tasks. If flaws are the reason for short throughput time the business process specification has to be changed in a way as to prohibit the behavior that is represented by the run, i.e., by modifying the routing decisions. If the run appears to be valid it might be a good idea to understand the reasons for its efficiency by doing some in-depth analysis using *Scenario 1* and *Scenario 2*. Testing this run with single configurations allows for analyzing very specific scenarios, e.g. worst-case or best-case assumptions for the activity times. Testing multiple configurations (especially with stochastic ones) gives an idea of the robustness of the run in terms of performance changes resulting from changes of testing values. Depending on the situation either a design that shows the best average results for all values or a design that produces best results for some values (and possibly worse for others) will be preferred. If the run appears to be an efficient way of handling cases for all sorts of testing values it can be used as a starting point for re-design cycles by exploring the reasons for its efficiency, e.g., a high degree of parallelism or the skipping of unnecessary tasks. An important step in redesign will then be to examine whether it is possible to change the business process specification in a way that more cases are handled using this particular run.

If all runs fail to provide performance that conforms to a given or planned service level (e.g., a *maximal* acceptable throughput time), the design of the business process has to be revoked as it is obviously not any good.

Once the throughput time of a run has been determined it is possible to derive a number of additional performance measures. One of the most distinct ones is *waiting time*. Waiting time is caused by synchronisation of parallel branches within a run. As already explained, waiting time is assigned to conditions in the causal net representation of a run that correspond to external places in the business process specification. The waiting time of a run can easily be calculated while the critical path for the determination of throughput time is examined [7, 10]. Waiting time is one of the prime indicators for potential improvements of throughput time. In addition to this it can be very interesting to examine total waiting time values for specific conditions since they indicate the exact places where waiting time occurs and where it might hence be avoided. Even if the value for total waiting time has been optimized, there is still room for improvement: if cost functions are assigned to the external places it is possible to distribute waiting time in a cost-optimal way [7, 10].

The *total service time* for a run is the sum of activity times for all conditions of the causal net representing the run. The ratio between total service time and throughput time is also a very important performance measurement. It combines the effects of parallel processing of a case and waiting time. Runs where the value for service time significantly exceeds the value for throughput time are marked by a high degree of parallelism and might serve as a "role" run for the handling

of cases. If the value for throughput time is higher than the value for service time it might be a good idea to examine the waiting time values and their occurrence, especially if the ratio between the total waiting time and throughput time of a run is rather large.

So far we have not yet considered *resources*. As the supply of resources can be limited, we might observe that tasks which could be executed concurrently have to be performed in a sequential order due to the sharing of resources. Hence the sharing of resources causes linearizations of partially ordered runs (cf. Figure 13 in Section 3). These linearizations are of particular interest for runs that deal with the handling of multiple cases. If resources are not considered, all cases can be handled concurrently as we assume that there is no interaction between similar cases based on causal relations or information exchange. However, as the same resources have to perform the work the cases are no longer independent from each other. As a result, for each single run without resources we get a set of runs with resources (corresponding to different ways that resources are allocated to cases), often with significantly increased values for throughput time.

If a task requires a resource of a given resource class this is modeled by adding the place representing the resource class to the preset of the activity transition representing the task (see the above subsection on modeling). Hence the use of resources technically resembles a synchronisation. As a result, for the calculation of resource-related performance measures we can apply the same algorithm as for the calculation of throughput time for runs without resources. By distinguishing the conditions in the preset of synchronisations via their mappings on places (as we have split up the set of places when introducing the set of resource places) we can distinguish between waiting time caused by the causal relation of the business process specification and waiting time that actually resembles a resource not being used. Given this information, it is straightforward to determine the resource utilization for each resource within each resource class by calculating the ratio between the sum of waiting times for the corresponding conditions and the throughput time of the run.

3 Working with *VIPtool*

In order to validate our approach with some (real-life) examples we have integrated the *VIPbusiness* methods into an already existing software for partial-order-based simulation, the *VIPtool*. In this section we will very briefly introduce some of the basic ideas behind this prototype. We then utilize a small case study from the area of insurance claim handling to demonstrate how the *VIPtool* can be used to perform the analysis steps introduced in the previous section.

3.1 The Tool

The *VIPtool* originated from a software prototype which was used for validating the simulation concepts of *VIP*. In the meantime the original prototype has been improved and developed quite extensively. Due to its modular design it has been

possible to exchange and add modules such as the business features for time and cost analysis very easily. Based on the object-oriented, interpreted scripting language *PYTHON*, the *VIPTool* is available as an (immediately) executable binary for Linux and Windows NT/95/98/2000.

Figure 10 gives an overview of the *VIPTool* architecture and its components. A graphical *editor* (Figure 11) provides all functionality that is needed for creating and editing the business process specification. Runs are generated by the simulation component *Vipsim*. In order to make the behavior of the modeled business process specification more transparent, all generated runs can be viewed using the *browser* component (Figure 12). Via export/import-interfaces the *VIPTool* has been integrated with a number of other tools, e.g., *Woflan*, a tool for the verification of *workflow nets* [2] developed at Eindhoven University of Technology, The Netherlands.

3.2 Case Study

In this section we use a small example to illustrate the use of our concepts for business process engineering. The case study is derived from the examples in the previous sections, it is a (quite coarse) model of a process for the handling of insurance claims. After introducing the example in some detail and creating a specification we will then show some examples on how our simulation and analysis approach can be used for gaining useful insight in our specification design.

The Example. We are looking at the department of an insurance company that processes claims stemming from liability insurances. The creation of a new case is triggered by the arrival of a customer's claim at the insurance company. The reception of the claim is acknowledged, in parallel a file for storing all related information is created. After these provisions the actual processing of the claim can begin. Three tasks have to be completed: the damaged third party has to be notified that the claim is being dealt with, the customer is contacted in order to receive and more detailed information and some data has to be looked up (e.g., whether the insurance policy of the customer is still valid or whether the customer has already filed other claims with the company). These three tasks can also be carried out independently of each other (if we do not consider shared resources). Based on the value of the claim further checking may be required (further checking is required if the claim value exceeds 10000). After the processing is done it is decided whether the insurance company pays for the claim or not. For our example we only consider the processing part and omit the payment decision.

Modeling. Following our 3-step approach (cf. Figure 1) we start with modeling the business process specification. To this purpose we use the *editor* component of *VIPTool* that provides all means we need to create business process specifications as described in the previous sections. Figure 11 shows a screenshot of the editor component containing a specification of our business process.

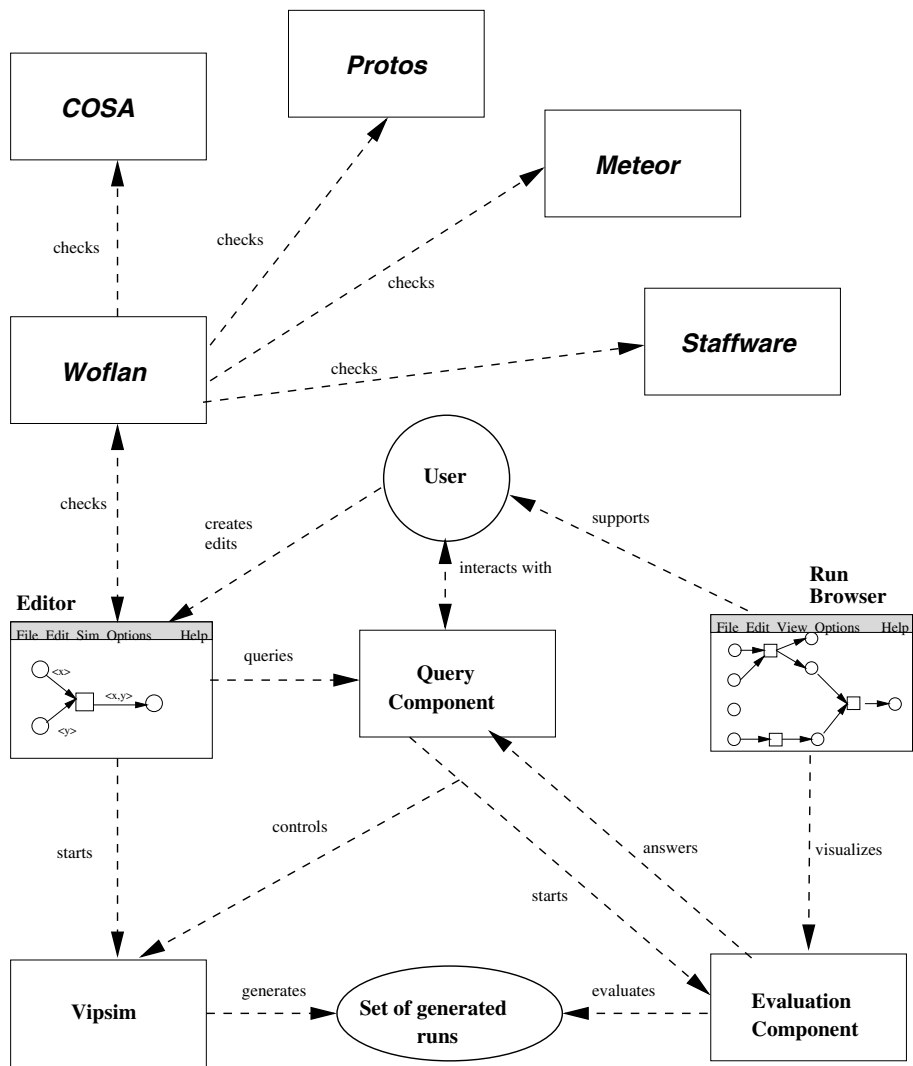


Fig. 10. *VIPtool* and friends.

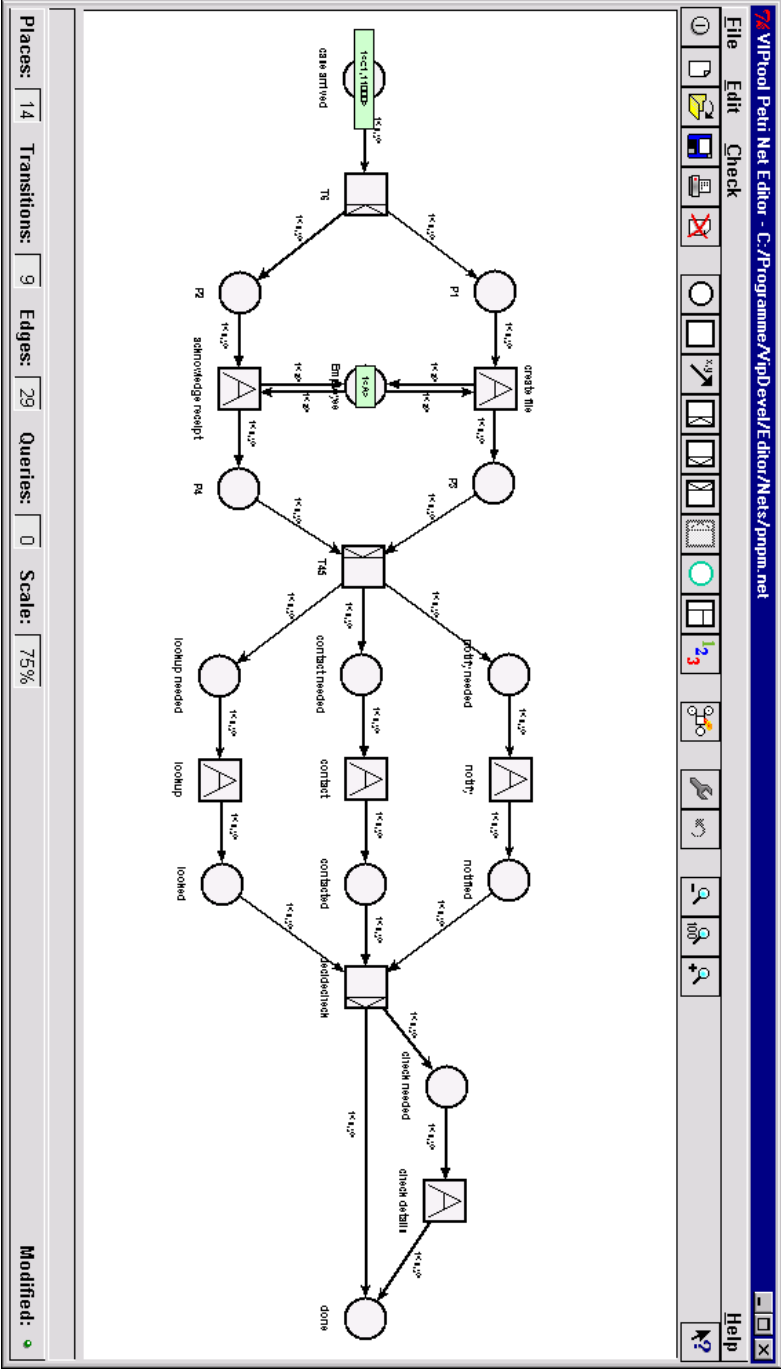


Fig. 11. The *editor* component of *VIPtool*.

At this stage we could export the model to *Woflan* in order to check some structural properties such as *soundness* or *well-handledness*. This way we can eliminate a number of potential design errors before actually having to perform any simulation. Since our example is rather small, ensuring some basic correctness is quite straight-forward, so we proceed with some simulation-based analysis.

Basic Testing. The first thing we do is to use a single case as an initial marking in our Petri net model. In Figure 11 we have inserted $\langle c1; 11000 \rangle$ in place *case arrived*, *c1* being the case identifier and 11000 being the value of the claim. The value of the claim is the information needed for deciding whether further checking of the claim is needed. Of course, this information is not known from the beginning but is one of the outcomes of the processing of the case. However, as we do neither incorporate external data sources nor (possibly random) generators for information values in our models we have to provide all information needed for the routing of cases in advance. We compensate this drawback by systematically creating initial markings in a way that all possible combinations of routing decisions can occur. That way we ensure that all possible paths through the business process specification can actually be taken, i.e., all possible runs for the handling of single cases may be generated. In our example we only have one routing decision: if the claim value exceeds 10000, further checking is required. So by introducing another marking $\langle c2; 5000 \rangle$ we get both possible outcomes of routing decisions. Note that for this small example it is rather straightforward to anticipate the outcome of the generation of runs. However, as specifications grow larger the effects of different routing decisions might be surprising for some runs.

Using the *VIPsim* component of *VIPtool* we generate the corresponding runs for each case. Figure 12 shows a screenshot of the *Browser* component containing the causal net representation of a run for the processing of $\langle c1; 11000 \rangle$. At this point we do not consider resources, so for our example we completely ignore the resource *employee* that is needed in order to perform the tasks *create file* and *acknowledge receipt*. As the sharing of resources is the only way in which similar cases interact with each other we could of course have used $\langle c1; 11000 \rangle + \langle c2; 5000 \rangle$ as initial marking. As a result, a run showing the (concurrent) handling of both cases would have been yielded. We will use this initial marking later on as we check the processing of several cases with resources taken into consideration. As for now, we settle for validating that all kinds of relevant cases can be dealt with by the business process specification, i.e., for each case we get a causal net representation of a run that starts with a condition corresponding to the input place of the specification (in our example: *case arrived*) and ends with a condition corresponding to the output place (*done*).

Introducing Time. Having checked that all cases can be dealt with in a sensible way we now take a look at throughput time as one of the most important performance measures. Since our example is rather simple and contains no choices that are not pre-determined by the values chosen for the initial markings, as long as we omit resources there exists only one run (since all cases can be dealt

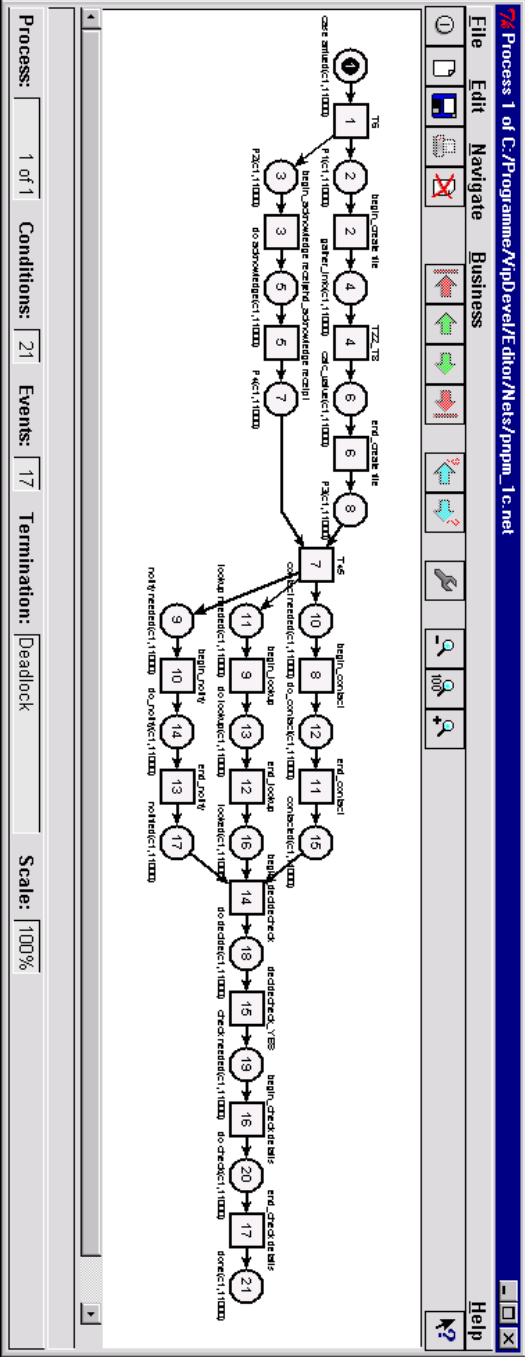


Fig. 12. The browser component of VIPtool.

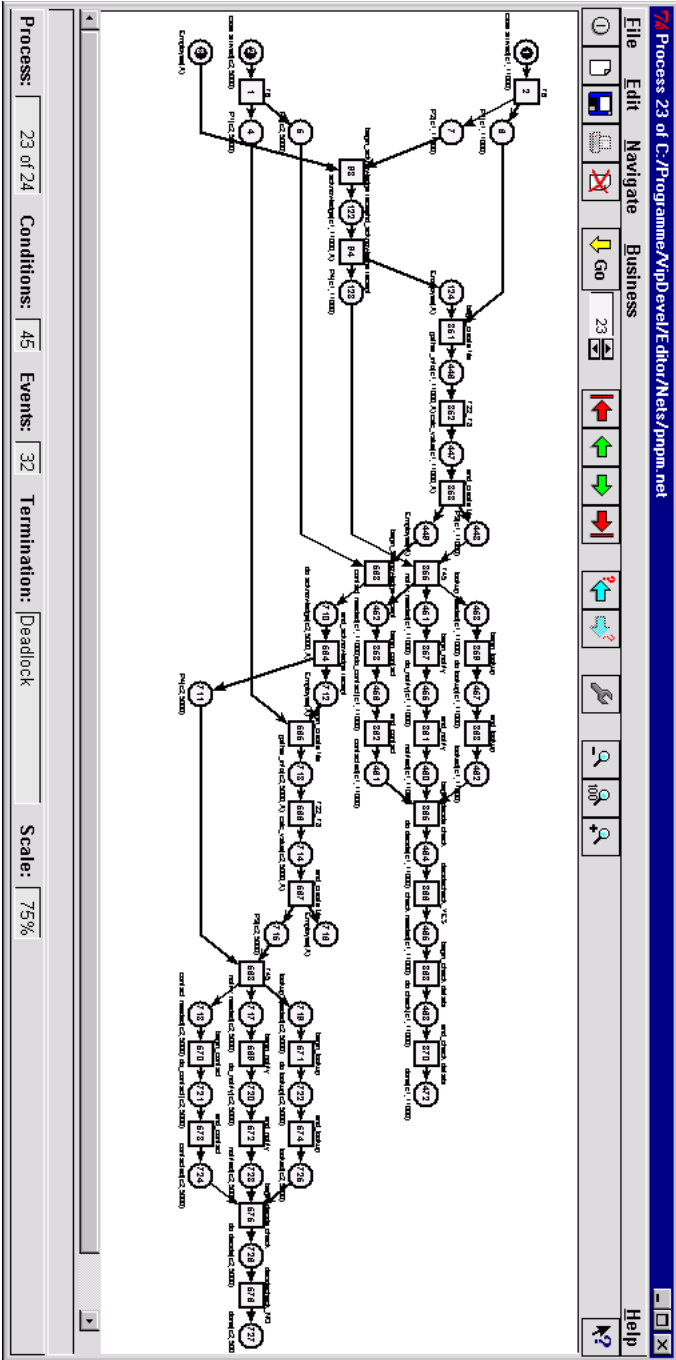


Fig. 13. Linearizations due to sharing of resources.

with independently). So we first introduce a resource to our model in order to increase the number of runs due to linearization effects.

Introducing Resources. As explained in the previous sections the sharing of resources causes linearizations of partially ordered runs. In our example, we take the resource class *employee* into consideration. In order to yield the maximum number of linearizations we add only one resource to *employee*.

Figure 13 shows the effects the sharing of resources has on the runs. Besides the fact that the number of runs has increased from 1 to 24 (due to the possible variations for allocating resources to cases), the length of the runs has also increased significantly. This simple example illustrates how our approach can be used to develop a very clear understanding of what effects different resource settings have on the handling of cases. This helps in identifying or testing worst case scenarios.

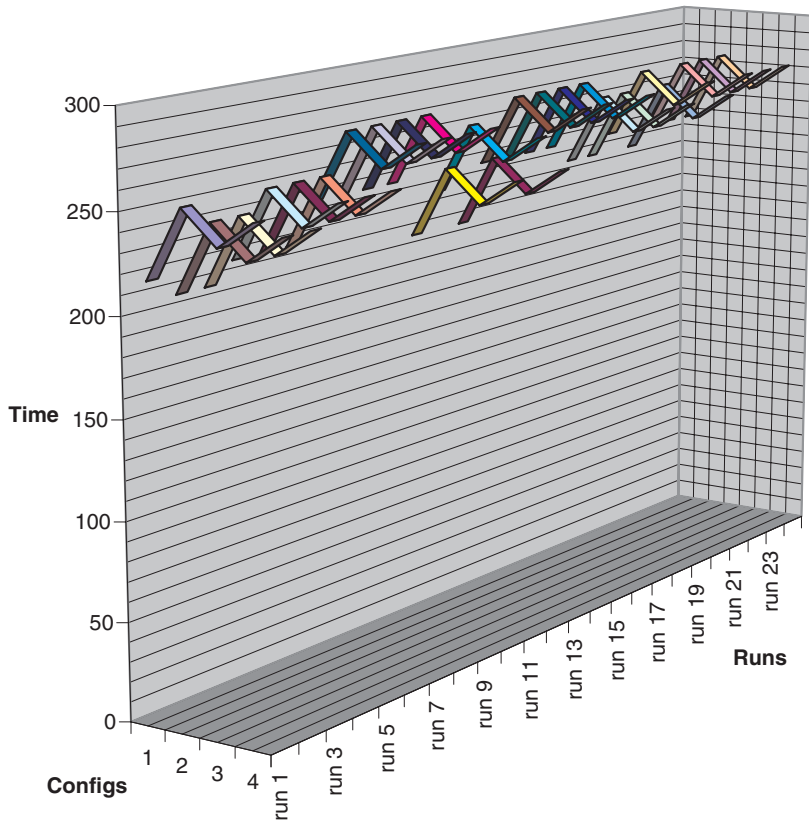


Fig. 14. Throughput times for all runs and a stochastic configuration.

In combination with our concept of multiple configurations for activity times it thus becomes possible to determine all kinds of performance measures as described in Section 2.4. Figure 14 shows the throughput times for all runs and four sets of random numbers generated for a configuration with stochastic activity times. All 24 runs show similar performance characteristics with respect to throughput time, so there is no "best" run that might serve as a "role" run for handling cases. As the room for variety in our example is rather limited (due to its size), this result is not too spectacular. Note, however, that even for this little example there are some points that might be worth investigating. As configuration 1 consistently yields the best (and configuration 2 the worst) results for throughput times it might be important to take a closer look at the actual values within these configuration in order to decide which might be more realistic. If, for example, the company wishes to guarantee a throughput time below 250 as service level, configuration 2 should better not be the more probable scenario. It might also be useful to understand why throughput times for some runs are a little lower (e.g., runs 1, 11 and 13). Also, the numbers could now be used to fix some service level, e.g., a time-span in which claims are being dealt with, that could be guaranteed to customers.

4 Conclusion

In this contribution we have introduced *VIPbusiness*, a new component in the *VIP* framework, that supports the reengineering of business process with a focus on quantitative aspects. To this purpose, we have introduced a class of High-level Petri nets, *VIPmodel* nets, which we have enhanced with important features that facilitate the creation of business process specifications, namely *activity transitions*, *routing primitives*, *resource places* and a state-based *timing concept*. Starting from a set of runs generated using the simulation techniques developed in the course of the *VIP* project we were able to calculate a number of important performance measures. By repeatedly applying our 3-step framework the design of the business process can thus be improved in an iterative manner.

Our approach is meant to be applied in early phases of projects where business processes and information systems have to be integrated. Since these projects usually contain an *implementation* phase, specifications of the business process at hand become a sooner or later mandatory prerequisite. By providing a specification formalism that permits specification on a conceptual level while at the same time allowing for simulation and quantitative analysis we hope to improve the overall quality of the business process specification that can then be used as a solid starting point for implementation efforts.

References

1. van der Aalst W.M.P.: Chapter 10: Three Good Reasons for Using a Petri-Net-Based Workflow Management System. In: Wakayama T. et al. (eds.): Information

- and Process Integration in Enterprises: Rethinking Documents. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, Norwell (1998) 161–182
2. van der Aalst W.M.P.: The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers* **8(1)** (1998) 21–66
 3. Ajmone Marsan M., Bobbio A., Donatelli S.: Petri Nets in Performance Analysis: An Introduction. In: Reisig W., Rozenberg G. (eds.): *Lectures on Petri Nets I: Basic Models*. Lecture Notes in Computer Science, Vol. 1491. Springer-Verlag, Berlin Heidelberg New York (1998) 211–256
 4. Buzacott J.A.: Commonalities in Reengineered Business Processes: Models and Issues. *Management Science* **42(5)** (1996) 768–782
 5. Davenport T.H.: *Process Innovation: Reengineering Work Through Information Technology*. Harvard Business School Press, Boston (1993)
 6. Desel J., Erwin T.: Hybrid Specifications - Looking at Workflows from a Run-Time Perspective. *International Journal of Computer Systems Science & Engineering* **15(5)** (2000) 291–302
 7. Desel J., Erwin T.: Modeling, Simulation and Analysis of Business Processes. In: van der Aalst W., Desel J., Oberweis A. (eds.): *Business Process Management: Models, Techniques, and Empirical Studies*. Lecture Notes in Computer Science, Vol. 1806. Springer-Verlag, Berlin Heidelberg New York (2000) 129–141
 8. Desel J.: Validation of Process Models by Construction of Process Nets. In: van der Aalst W., Desel J., Oberweis A. (eds.): *Business Process Management: Models, Techniques, and Empirical Studies*. Lecture Notes in Computer Science, Vol. 1806. Springer-Verlag, Berlin Heidelberg New York (2000) 110–128
 9. Desel J., Juhas G.: What is a Petri Net? In: Ehrig H., Juhas G., Padberg J., Rozenberg G. (eds.): *Unifying Petri Nets*. Lecture Notes in Computer Science, Vol. 2128. Springer-Verlag, Berlin Heidelberg New York (2002) 1–25
 10. Erwin T.: *Leistungsbewertung von Geschäftsprozessen durch Auswertung halbgeordneter Petrinetz-Abläufe*. Diplomarbeit, Universität Karlsruhe (1998)
 11. Freytag T.: *Softwarevalidierung durch Auswertung von Petrinetz-Abläufen*. Dissertation, Universität Karlsruhe (2001)
 12. Genrich H.J.: Predicate/Transition-Nets. In: Brauer W., Reisig W., Rozenberg G. (eds.): *Advances in Petri Nets 1986, Part I: Petri Nets, Central Models and Their Properties*. Lecture Notes in Computer Science, Vol. 254. Springer-Verlag, Berlin Heidelberg New York (1987) 207–247
 13. Gruhn V., Kampmann M.: Modellierung unternehmensübergreifender Geschäftsprozesse mit FUNSOFT-Netzen. *Wirtschaftsinformatik* **38** (1996) 369–381
 14. Hammer M., Champy J.: *Reengineering the Corporation*. Nicolas Brealey Publishing, London (1993)
 15. Jablonski S., Bussler C.: *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, Bonn Albany (1996)
 16. Jensen K.: *Coloured Petri Nets, Vol. 2: Analysis Methods*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg New York (1995)
 17. Jensen K.: *Coloured Petri Nets. Vol. 1: Basic Concepts*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg New York (1996)
 18. Johansson H.J., McHugh P., Pendlebury A.J., Wheeler W.A.: *Business Process Reengineering: Breakpoint Strategies for Market Dominance*. John Wiley and Sons, New York (1993)

19. Khoshanfian an S., Buckiewicz M.: Introduction to Groupware, Workflow and Workgroup Computing. John Wiley and Sons, New York (1995)
20. Koulopoulos T.M.: The Workflow Imperative. Van Nostrand Reinhold, New York (1995)
21. Lawrence P. (ed.): Workflow Handbook 1997, Workflow Management Coalition. John Wiley and Sons, New York (1997)
22. Manganelli R.L., Klein M.K.: The Reengineering Handbook: A Step-By-Step Guide to Business Transformation. Amacom, New York (1996)
23. Morris D., Brandon J.: Reengineering Your Business. McGraw-Hill, New York (1993)
24. Oberweis A.: Modellierung und Ausführung von Workflows mit Petri-Netzen. Reihe Wirtschaftsinformatik. Teubner, Stuttgart Leipzig (1996)
25. Poyssick G., Hannaford S.: Workflow Reenigneering. Adobe Press, Mountain View, CA (1996)
26. Reisig W.: Petri Nets: An Introduction, Monographs in Theoretical Computer Science: an EATCS Series, Vol. 4. Springer-Verlag, Berlin Heidelberg New York (1985)
27. Scheer A.W.: Business Process Engineering, Reference Models for Industrial Enterprises. Springer-Verlag, Berlin Heidelberg New York (1994)
28. Starke P.: Analyse von Petri-Netz-Modellen. Leitfäden und Monographien der Informatik. Teubner, Stuttgart Leipzig (1990)

Inter-organizational Business Process Management with XML Nets

Kirsten Lenz and Andreas Oberweis

Institute of Information Systems, Goethe-University
D-60054 Frankfurt/Main, Germany
{lenz, oberweis}@wiwi.uni-frankfurt.de

Abstract. Due to the fast growth of internet based electronic business activities, languages for modeling as well as methods for analyzing and executing distributed business processes are becoming more and more important. Efficient inter-organizational business processes in the field of e-commerce require the integration of electronic document interchange and inter-organizational process management. We propose to support inter-organizational business processes by so-called **XML nets**, a new variant of high-level Petri nets. **XML nets** are a formal, graphical modeling language that allows to model both the flow of XML documents and the control flow of the underlying business process. **XML nets** are based on **GXSL**, a novel graphical XML schema definition language, and its related graphical XML document manipulation language **XManiLa**. They can be directly executed by a respective workflow engine. Advantage of the formal foundation of **XML nets** can be taken, e.g., when analyzing a (global) inter-organizational workflow model. **XML nets** support the identification of relevant process fragments and their allocation to the appropriate organizational units. Thus **XML nets** help to derive an improved, process-oriented organizational structure.

1 Introduction

Due to the fast growth of internet based electronic business activities, languages for modeling as well as methods for analyzing and executing distributed business processes are becoming more and more important. Specific requirements concerning the expressibility of process modeling languages arise from application fields such as e-commerce, virtual organizations, supply chain management, tele-cooperation, mobile computing, and application service providing.

Efficient inter-organizational business processes require the integration of electronic document interchange and inter-organizational process management: From the document point of view, there is a need for electronically interchanging structured documents and data between the involved organizations. From a process-oriented perspective, modeling, analysis, and automated execution of distributed workflows

gain more and more importance. Moreover, the advantages of workflow driven electronic document interchange are relevant for the intra-organizational process management, especially with the emerging integration of mass data in traditional databases and document management.

This paper proposes *XML nets*, a new variant of high-level Petri nets, to support inter-organizational business processes. *XML nets* are a formal, graphical modeling language that allows to model both the flow of XML documents and the control flow of the underlying business process. *XML nets* are based on *GXSL*, a graphical XML schema definition language, and the corresponding XML document manipulation language *XManiLa*.

XML nets can be directly executed by a respective workflow engine and they allow for analysis and simulation of distributed business processes. Additionally, *XML nets* support the identification of relevant process fragments and their allocation to the appropriate organizational units. Thus *XML nets* help to derive an improved, process-oriented organizational structure.

Related concepts for graphical XML data modeling languages based on UML and the entity-relationship model can be found for example in [4, 9, 18], those for XML query languages [3, 7, 11, 23] focus rather on the information retrieval than on the manipulation of existing documents. Many of the *XML net* concepts proposed in this paper rely on so-called SGML nets [26], a variant of Petri nets for the modeling of workflows based on SGML documents.

This paper is structured as follows: First, we give a brief introduction to XML, Petri nets, and workflow management. In the following section, we present a graphical XML schema definition language and propose a graphical XML document manipulation language. Finally, we introduce *XML nets* for the inter-organizational business process management.

1.1 XML – Extensible Markup Language

The Extensible Markup Language XML [12] is a document description standard proposed by the World Wide Web Consortium (W3C, <http://www.w3.org/>) as a simplification of the Standard Generalized Markup Language (SGML, ISO 8879).

```
<!-- DTD for employee documents-->
<!ELEMENT employee (empname, work_in_proj*, skill*)>
<!ATTLIST employee empid ID #REQUIRED sex (male|female) "female">
<!ELEMENT empname (title?,(firstname|initial)+, lastname)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT initial (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT work_in_proj (#PCDATA)>
<!ATTLIST work_in_proj projid IDREF #IMPLIED>
<!ELEMENT skill (#PCDATA)>
```

Fig. 1. Document Type Definition for Employee Documents

An XML document type definition (DTD) allows to specify a document type with application specific markups as a class of XML documents conforming to that specific DTD, i.e. a set of documents that follow the respective document design rules. Figure 1 shows a simple DTD for an employee document with personnel

information, a list of projects the employee works in and a list of the employee's skills.

A corresponding valid example document for the employee Mary L. Miller is shown in Figure 2.

```
<?xml version="1.0"?>
<!DOCTYPE employee SYSTEM "http://...empdoc.dtd">
<employee empid="0815" sex="female">
  <empname>
    <firstname>Mary</firstname>
    <initial>L.</initial>
    <lastname>Miller</lastname>
  </empname>
  <work_in_proj projid="328">20 </work_in_proj>
  <work_in_proj projid="85">10 </work_in_proj>
  <skill>Java</skill>
  <skill>C++</skill>
  <skill>Oracle 8</skill>
</employee>
```

Fig. 2. Valid Employee Document

XML DTDs may be complemented by XML schemas [25, 28], a new standard promoted by W3C which aims at overcoming the limits of a DTD, e.g. concerning the data-oriented role of XML in e-business applications. XML schemas, a functional superset of DTDs, are written themselves in XML and can therefore be parsed and validated like XML documents. Moreover, they provide concepts to specify data types (like in traditional databases) and thereby ensure data integrity for valid XML documents.

1.2 Petri Nets

For the modeling of business processes several more or less formal description languages have been proposed. In contrast to other semi-formal languages, Petri nets [6, 22] combine the advantages of a graphical representation of business processes with a formal semantics of the modeled behaviour. Beyond visualization of processes Petri nets allow for the analysis and validation of business processes and workflows [2, 10]. Notations such as event-driven process chains (EPC) [24], which are widely used in practical applications for business process modeling, might be regarded as a semi-formal variant of Petri nets with a specific graphical interface.

A simple example of a Petri net is shown in Figure 3:

A planned project is initialized. Afterwards, employees can be assigned to the project. At the beginning of a project the project name is assigned to the responsible department. After finishing the project, the employee information must be updated: new skills learned in the project have to be added, whereas the project has to be removed from the list of projects the employee works in.

High-level Petri nets such as predicate/transition nets (Pr/T-nets) [13] or nested relation/transition nets (NR/T-nets) [20, 21] integrate behavior- and object-related aspects of business processes. Hence high-level Petri nets might be interpreted as dynamic extensions to static data models to represent processes on the respective business objects. In Pr/T-nets or NR/T-nets the places are interpreted as relation

schemas according to which the marking of the net assigns a relation to each place. A transition represents a class of operations on the relations in the adjacent places.

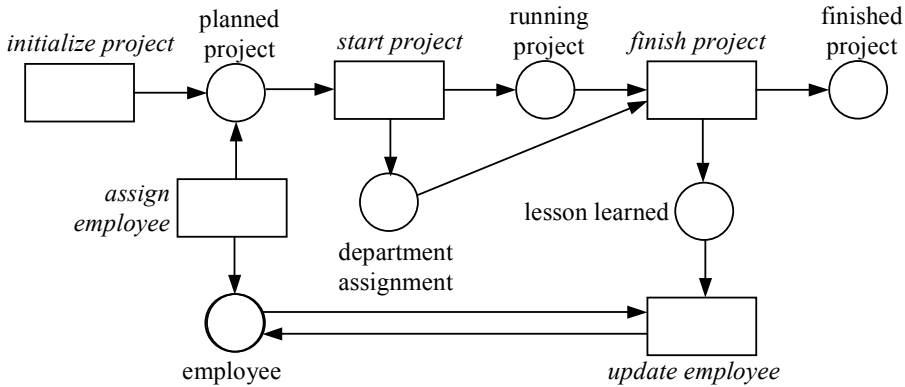


Fig. 3. Petri Net Example

When a transition occurs, the respective tuples are removed from its input places and inserted into its output places as specified by the edge inscriptions. Hence from a database point of view a transition occurrence might be regarded as a combination of insert and delete operations on the data in the transition's output and input places. Arc and transition inscriptions might serve as additional filters to select tuples which are to be inserted and deleted (see Figure 4).

1.3 Workflow Management

According to the Workflow Management Coalition, a workflow is defined as "the automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules" [27]. Workflow management includes the specification of the workflow model, its analysis and verification, the execution of the workflow by a suitable workflow architecture, and the development of workflow applications [16]. A workflow management system supports these tasks and controls the automated aspects of business processes [27].

Petri nets and related languages are well suited and frequently proposed in the literature for workflow modeling [1]. They provide a set of graphical modeling concepts for the basic workflow behavior patterns such as sequence, iteration, concurrency and choice. The formal foundation of Petri net based workflow models enables the application of analysis techniques (e.g. to check whether a certain system state can be reached or not) and the model-driven execution of the workflow. Moreover, they integrate data and behavior aspects and therefore meet the need to model for example inter-organizational workflows based on electronic document interchange. The use of Petri nets at different formalization levels allows for a stepwise refinement of a workflow model.

A survey of different approaches for Petri net based business process and workflow management can be found for example in [1].

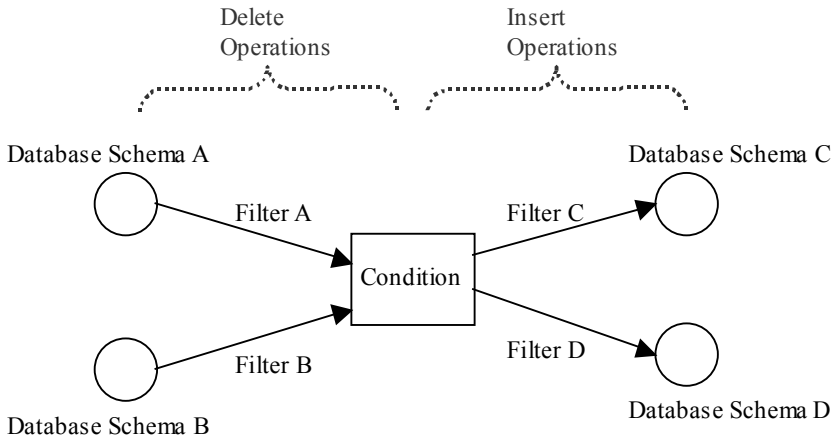


Fig. 4. Database Interpretation of a High-Level Petri Net

2 Graphical Language for XML Document Management

In this section, we introduce the graphical XML schema definition language **GXSL** for graphically specifying XML schemas that represent a document type definition DTD by XML schema diagrams (XSDs). *XManiLa*, an extension to the graphical XML schema definition language, is proposed for the retrieval and manipulation, especially insertion and deletion, of XML documents.

2.1 XML Schema Definition with GXSL

The XML document type definition represents a grammar for the declaration of documents. It consists of a set of markup declarations of different types: element, attribute list, entity or notation declaration. However, this grammar is only textual so that the DTDs for objects with complex structures often may become quite unreadable. The advantages of a graphical schema definition language (like the entity-relationship model [8] for the traditional database design) are missing.

In the following, we propose a graphical schema definition language for the design of XML document types, that we call *graphical XML schema definition language* (**GXSL**). The presented version of **GXSL** is DTD-based, i.e. a DTD can be unambiguously derived from the *XML schema diagram* XSD. This version is a subset of the formally founded **GXSL** for XML Schema. The full **GXSL** provides concepts for graphically modeling datatype-based XML schemas with enhanced features like an additional kind of element aggregation, element substitution, element and association references, and finally the concept of keys and foreign keys.

Instead of creating a completely new graphical modeling language for XML document types, we rely on well known data modeling concepts (of the entity-relationship model and other semantic data models), which all had their impact on the static modeling concepts of the Unified Modeling Language (UML) [5]. The main

advantage of UML is that it is in comparison to other language proposals a broadly accepted integration of well-known modeling concepts and guidelines and that it comprises a generic notation for the derivation of new graphical modeling languages for specific purposes. A proposal similar to the DTD-based part of **GXSL** can be found in [9]. Due to space limitations, we focus on the logical structure of XML document types here and omit XML entity declarations, internal subsets and INCLUDE or IGNORE markups. We also propose a simplification of **GXSL** conforming to UML which has been adapted to the quick and rather intuitive modeling of XML document types.

The main principles of XML DTDs are classification of documents with comparable structure and aggregation of document components. We present XML documents and their elements by UML classes, hierarchically (i.e. non-recursively) structured by aggregation. In the following, the UML concepts for **GXSL** are explained in detail.

Simple element types: In general, element type declarations of the XML DTD are represented by classes in **GXSL**. By simple element types we understand element types without children (i.e. they are not composed of other element types), namely the EMPTY-element type, the ANY-element type and the predefined #PCDATA. When the element type declaration contains the keyword ANY, both other tags and general characters are allowed within the element tags. Empty elements have no content at all. Figure 5 shows the **GXSL** representation of simple element type declarations in an XSD by stereotyped «any»- and «empty»-classes.

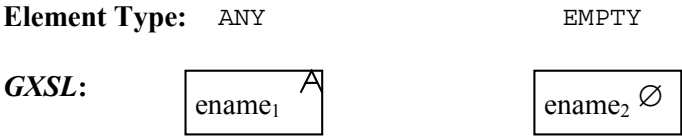


Fig. 5. **GXSL** Class Representation of the ANY- and EMPTY-Element Type Declaration

#PCDATA stands for *parsed character data* and symbolizes any sequence of characters that does not contain any tag. The **GXSL** class «pcdata» has no name and provides its own icon as shown in Figure 6.

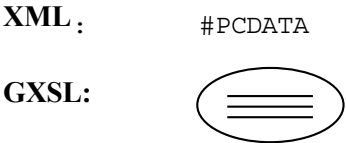


Fig. 6. Icon for the «pcdata»-Class

In addition to element type declarations, a DTD may include attribute list declarations in order to specify element types. The attribute list declaration for an element type ename has the following pattern:

```
<!ATTLIST ename aname1 atype1 defaultdecl1
...
anamen atypen defaultdecln>.
```

aname is the attribute name, atype the attribute type and defaultdecl an optional default declaration. In an XSD, the attribute list is added to the element class (see Figure 7).

ename
aname ₁ : atype ₁
aname ₂ : atype ₂ = default ₂
...
aname _n : atype _n

Fig. 7. Element Class with Attribute List

The default declaration consists of a default value and/or a default modifier. In Figure 7, the second attribute of the element type has a default value. Table 1 shows the translation of XML default modifiers into **GXSL**.

In a survey diagram it is recommended to restrict the XSD class representation to the element name with or even without the list of the attribute names.

Table 1. GXSL Class Attribute for XML Attribute Declaration with Default Modifier

XML	GXSL	Explanation
#IMPLIED	aname: atype	Attribute value may remain unspecified
#REQUIRED	aname [1]: atype	Mandatory attribute value, specified by the occurrence indicator
#FIXED "dvalue"	aname: atype = dvalue {frozen}	Attribute value must not be changed; default value required

Nested Element Types: Element types in XML can be nested in order to describe hierarchically structured documents, i.e. the element type declaration refers to other element type declarations of the DTD:

```
<!ELEMENT ename (...)>.
```

UML offers a special kind of association to adequately model the hierarchical relationship between different element types: *aggregation*. An aggregation is represented by an open diamond in the XSD (see Figure 8). Additionally, a name and a name direction can be assigned to each association.

XML occurrence indicators as appendix of an element name or a group of elements can be translated into a cardinality value that can be assigned to the associated subclass(es) as shown in Table 2.

The cardinality corresponding to the parent classes is always 1..* for the aggregation. For example, the element type declaration `<!ELEMENT qualification (skill*)>` for the qualification of an employee as a set of skills has the XSD in Figure 8.

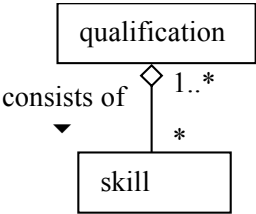


Fig. 8. XSD for a Nested Element Type

Table 2. GXSL Cardinalities for XML Occurrence Indicators

GXSL cardinality	Element occurrence is ...
1 (default value)	... mandatory
0..1	... optional
*	... zero or more times
1..*	... one or more times

Choice: The XML element type can be defined as a choice between two or more element types. For example, the DTD contains the element type declaration `<!ELEMENT ename1 (ename2|ename3)>` and respective element type declarations for `ename2` and `ename3`. Within a document which is valid for this DTD there has to appear either an `ename2` element or an `ename3` element inside the `ename1` element (and must not appear both). With GXSL a choice can be expressed by a constraint with `{or}`-condition between the respective alternative subclasses (see Figure 9).

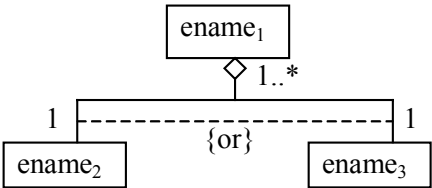


Fig. 9. XSD for Alternative Element Types

Sequence: A sequence of elements that have to appear in a predefined order within another element can be declared as a list of element names, separated by commas:

`<!ELEMENT ename (ename1, ename2, ..., enamen)>.`

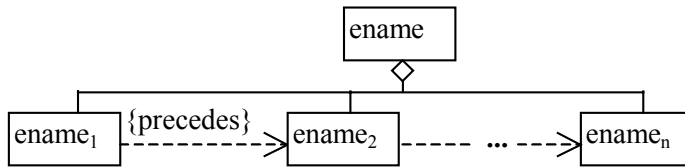


Fig. 10. XSD for a Sequence of Element Types

We use the dependency modeling concept of UML to model a sequence with **GXSL**. Dependency is a (possibly directed) association and indicates a sort of a using relationship between elements. In **GXSL** we introduce a directed dependency including the label **{precedes}** to order the subclasses of a superclass (see Figure 10) with the following intuitive semantics: The document elements must follow the same order as their element types (respective their cardinality).

Cascading Nesting: XML allows for element type declarations with cascading nesting, i.e. element types are not declared explicitly but are implicitly given by an expression enclosed in parenthesis. For example, we can declare by $(\text{ename}_1, (\text{ename}_2 | \text{ename}_3)^+)$ that an element of type ename_1 must be followed by an iteration of either an ename_2 element or an ename_3 element. In this case, we have to introduce an abstract class to model the hierarchy of the element type (see Figure 11). This means splitting the expression into $(\text{ename}_1, \text{abstract}^+)$ and $< !\text{ELEMENT abstract } (\text{ename}_2 | \text{ename}_3)^+ >$.

The abstract class is a stereotyped class and has its own icon. It is not possible to assign attributes to it, the class name is irrelevant and thus can be omitted. However, the abstract class may have a cardinality.

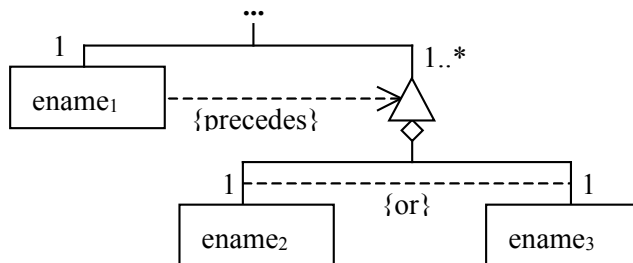


Fig. 11. XSD for Nested Element Types with an Abstract Class

Referential Integrity: Attributes may be of type **IDREF** or **IDREFS**. Their values must be included in (or be a subset of respectively) the set of **ID** attribute values of the referenced element type. The attribute types **IDREF** and **IDREFS** therefore implicitly define referential integrity constraints between XML elements of a DTD. **GXSL** also allows for explicit referential integrity constraints by **{ref}**-dependencies (see Figure 12). **{ref}**-dependencies are labeled by the referencing and the referenced attribute and the *****-cardinality for attributes of type **IDREFS**.

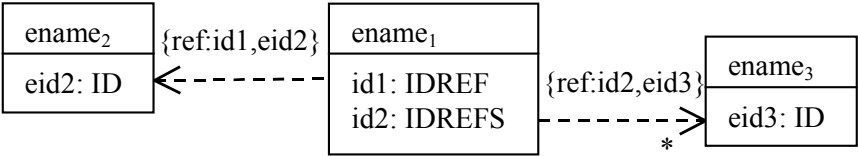


Fig. 12. *GXSL* Dependencies for Referential Integrity Constraints

Simplifications of *GXSL*: In this section, we propose some simplifications for the *GXSL* concepts described above. With these simplifications, *GXSL* does not completely coincide with UML anymore. But a simplified XSD still provides all necessary information without notational overload and the XML schema design becomes more intuitive for modelers who are familiar with XML.

- The association name and direction can be omitted.
- Cardinality of the superclass of composition or aggregation is fixed and thus can be omitted.
- For *GXSL*, we declare the default cardinality to be '1' (instead of '*' for UML), which is identical to the default occurrence indicator of XML.
- We propose a stereotyped class and a special icon for the element type declaration `<!ELEMENT ename (#PCDATA) >` (Figure 13b) instead of the XSD conforming to UML (Figure 13a). Note that the `#PCDATA`-element class is not identical to the `#PCDATA`-class introduced before.

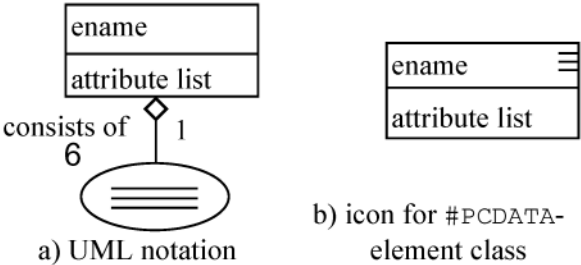


Fig. 13. Simplification for `#PCDATA`-Element Types

- The constraint for a choice is simply expressed by a dashed line. The `{or}`-condition can be omitted.
- The XSD for a sequence is simplified by omitting the `{precedes}`-dependencies. Nevertheless, elements of a valid document must appear in the same order as the respective classes in the XSD.
- The label of a `{ref}`-dependency may be omitted if either class attributes have not been specified in the XSD or both referencing and referenced attributes can be uniquely identified without any label.

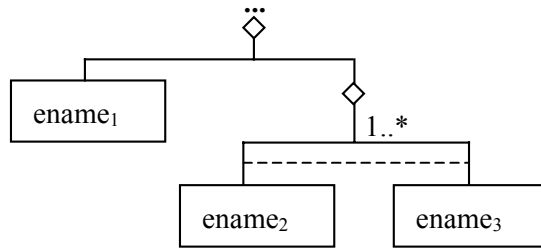


Fig. 14. Simplification for cascading nesting of element types referring to Figure 11

- Cascading nesting of elements is represented without any abstract class. We allow for a tree structure of aggregation (see Figure 14). The cardinality of the abstract class is written above the respective branch.

Example: We are now able to construct an XSD (shown in Figure 15) which is equivalent to the employee DTD of Figure 1.

Compared to the textual representation in Figure 1, the graphical representation gives a better overview of the defined elements and the logical structure of a valid employee document.

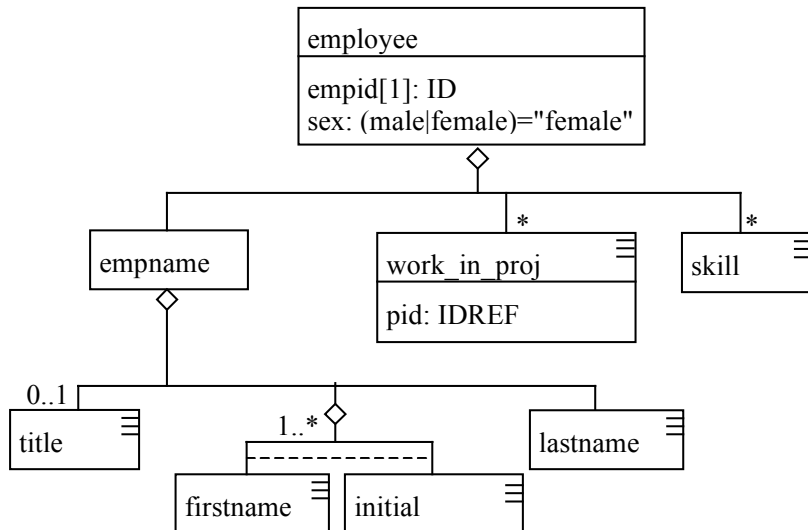


Fig. 15. XSD for the Employee DTD of Figure 1

2.2 Manipulation of XML Documents With *XManiLa*

We can use *GXSL* with some extensions also for querying and manipulating XML documents. The *GXSL*-based document manipulation language is called *XManiLa*. An XSD can be interpreted as a template for a set of XML documents that specifies the structure of matching documents. In order to enable also a content based query,

we allow for assigning constants or variables to an element or an attribute. The idea for this type of query formulation is taken from Query-By-Example, a graphical query language for relational databases [29].

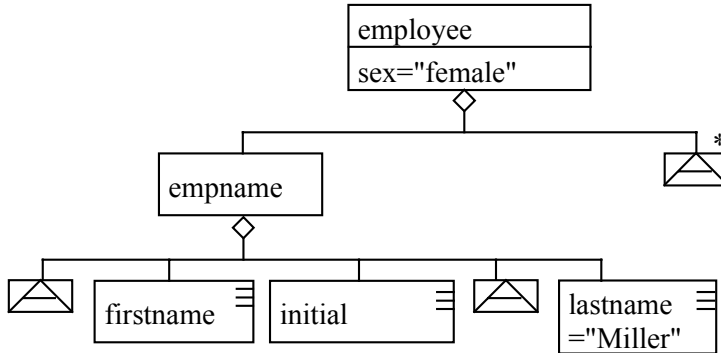


Fig. 16. Female Employees Named Miller with a Firstname Followed by an Initial

For example, we may want to search for all female employees that have a first name followed by an abbreviation and the lastname "Miller". The XSD shown in Figure 16 represents this query. In order to express that matching documents may contain within the `empname`-element any element preceding the `firstname`-element, we use the element wildcard that is graphically represented by a rectangle inscribed with an "A".

XManiLa is not only suited for document retrieval, but also for insertion and deletion (and thereby also for updating). These operations either concern a whole document or elements on lower hierarchy levels. For that purpose, the original XSD is extended by depicting elements to be inserted or deleted by a solid line on the left side of the element's rectangle. The insert- and delete-qualification is inherited by all subclasses. An extended XSD as edge label of a Petri net describes precisely the insert or delete operation of a document or document elements. Examples of insert and delete operations on the employee documents are given in Figure 17.

3 Inter-organizational Workflows

The interchange of XML documents between organizations can be implemented in different ways:

- The XML document can be sent to another organization without any DTD. From the recipient's point of view the document only fulfills the weaker requirements of being well-formed (i.e. it has a correct syntax), supplementary information is missing.

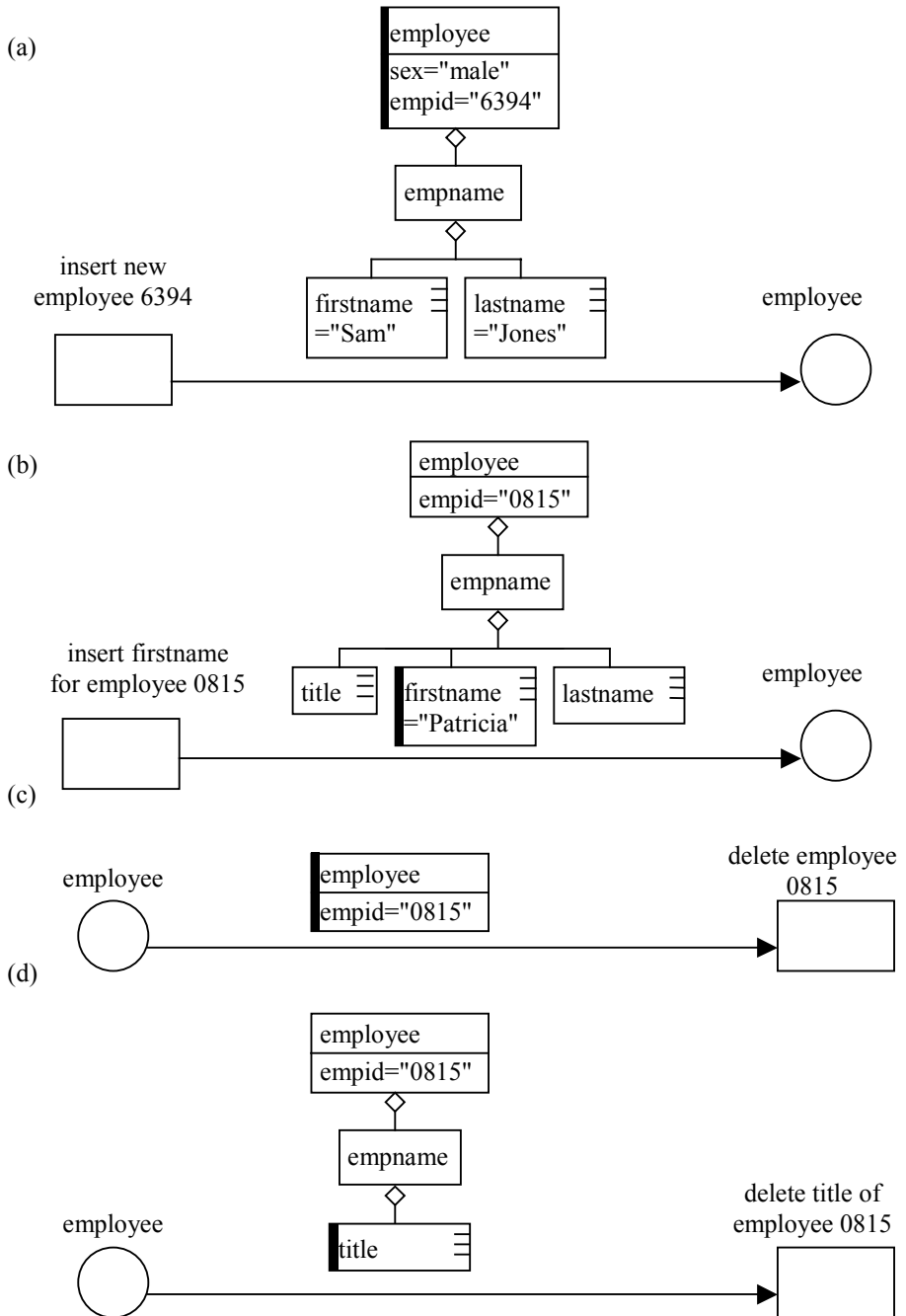


Fig. 17. Insert and Delete Operations for the Employee Documents

- The DTD can be stored internally within the document type declaration of the document and therefore be sent to the recipient together with the document itself. In this case the XML document can be processed anywhere without any supplementary information. However, reuse of the DTD is not supported.
- External DTDs (either public or private) are stored as independent files. They can be identified via uniform resource identifiers (URIs) which must be contained in the document type declaration of the valid XML document. External DTDs can be distinguished through so called namespaces [19]. They are reusable and thus support more efficient document management.

The extent of publishing organizational DTDs depends on many aspects, for example on technological aspects as well as on the chosen organizational strategy. DTDs can be completely hidden to other process participants, exchanged between two partner organizations or published world wide. For an introduction into XML and namespaces see for example [15]. In the following, we suppose that the set of process relevant DTDs is known to all participants of the business process.

3.1 XML Nets

In the sections above, we have described how to graphically model XML DTDs with *GXSL* and how to specify document retrieval and document manipulation with *XManiLa*. We can now combine both techniques for the definition of *XML nets*.

XML nets, a variant of high-level Petri nets, are a formal, graphical modeling language that allows to model both the flow of XML documents and the underlying business process. The static components of *XML nets* (i.e. the places of the Petri net) are typed by XML schema diagrams, each of them representing a DTD. Places can be interpreted as containers for XML documents which are valid for the corresponding DTD. The flow of XML documents is defined by the occurrences of transitions which thereby manipulate (create, change or delete) documents of their adjacent places. The enabledness of a transition, which is prerequisite for the transition's occurrence, depends on the labels of the adjacent edges constructed with *XManiLa* and on an optional transition inscription. *XML nets* have the following characteristics:

- Each place is typed over an XSD which identifies the type of documents contained in the place.
- An edge between a place and a transition is labeled with an extended XSD that does not conflict with the schema of the adjacent place. For each instantiation of the variables of the edge label it is possible to decide whether a document of the adjacent place matches the extended XSD or not.
- A transition may be inscribed by a logical expression over all or some of the variables that appear in the labels of the adjacent edges. The expression evaluates either to `true` or `false` for an instantiation of the variables.
- The initial marking assigns to each place of the *XML net* a (possibly empty) set of valid XML documents.

The behavior of *XML nets* is defined by transition occurrences. A transition is enabled for a given marking, i.e. it may occur, if the following three conditions hold

for an instantiation of the variables in the transition's environment (note, that if a variable appears more than once in the vicinity of a transition it must be instantiated with the same value for the same transition occurrence):

- All places in the preset of the transition, i.e. with an outgoing edge to the transition, contain at least one document that matches with the schema of the adjacent edge under the given instantiation of the variables,
- All places in the postset of the transition, i.e. with an edge from transition to place, contain (at least) one document that matches with the label of the adjacent edge under the given instantiation of the variables in case of document manipulation or do not contain a matching document in case of document creation,
- The transition inscription validates to `true` for these documents and the chosen instantiation of the variables.

If a transition occurs, it removes the matching documents or parts of the documents as specified by the edge label from the places in the preset of the transition and inserts new documents into the places in the postset of the transition or new elements into the matching documents.

To continue the example of Figure 3, we can now assign to each place an XSD and to each edge an extended XSD as label. Figure 18 shows the schemas for `planned`, `running` and `finished` projects, the `department`, and the `lessons learned`. The schema for `employee` is shown in Figure 15.

Besides the definition of all process relevant document type definitions, the behavior of the *XML net* has to be specified in detail for all transitions.

- *initialize project*: A new document of type `planned project` is created.
- *assign employee*: The assignment consists of inserting the employee ID as team member for the project and the time required for the task execution together with the project ID for the employee. In contrast to the previous transition, the number of documents remains the same: the respective documents are only partially changed.
- *start project*: When a project is started, a responsible department is chosen for the project. This is only possible for departments with an identifier value that is higher than 100. The document of type `planned project` is deleted and a new document of type `running project`, including the department ID, is created. The project ID is added to the department's list of running projects.
- *finish project*: Delete the respective document from the list of running projects and create a new document with the information about the finished project. Moreover, delete the project ID from the department document. Create a new document that contains all information about the skill that an employee has learned from the project.
- *update employee*: Finally, the information about project participation is deleted in the employee document. At the same time, the new skill is appended to the employee's list of skills.

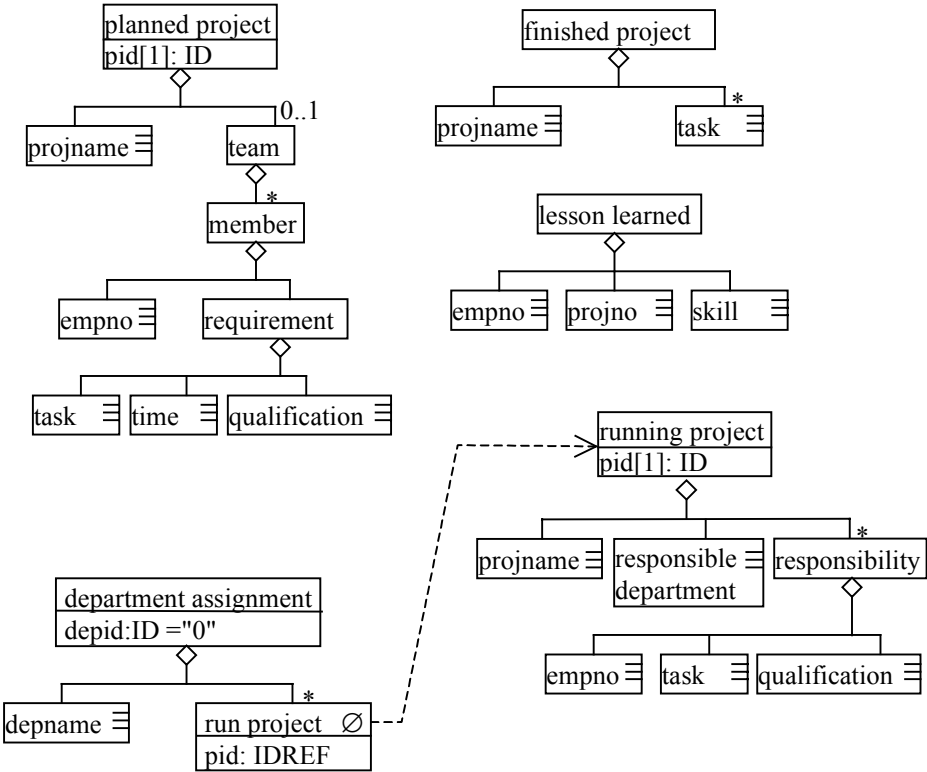


Fig. 18. XML Schema Diagrams for the Example

The *XML net* refining the Petri net of Figure 3 is shown in Figure 19. For the sake of readability, we omitted the XSD place labels. Furthermore, the initial marking is not shown in the graphical representation. It could for example consist of a set of employee documents and a set of department assignments.

An inter-organizational workflow management system should be able to deal with both simply structured data ("flat tuples") and documents. Although places of predicate/transition nets are interpreted as containers for relations, i.e. structured data of relational databases, predicate/transition nets can also be modeled as *XML nets*: the XSD corresponding to a relation schema or an edge inscription is a diagram with a one level hierarchy of all attributes. This allows to combine both concepts, data based workflow modeling with predicate transition nets and document based workflow modeling with *XML nets*.

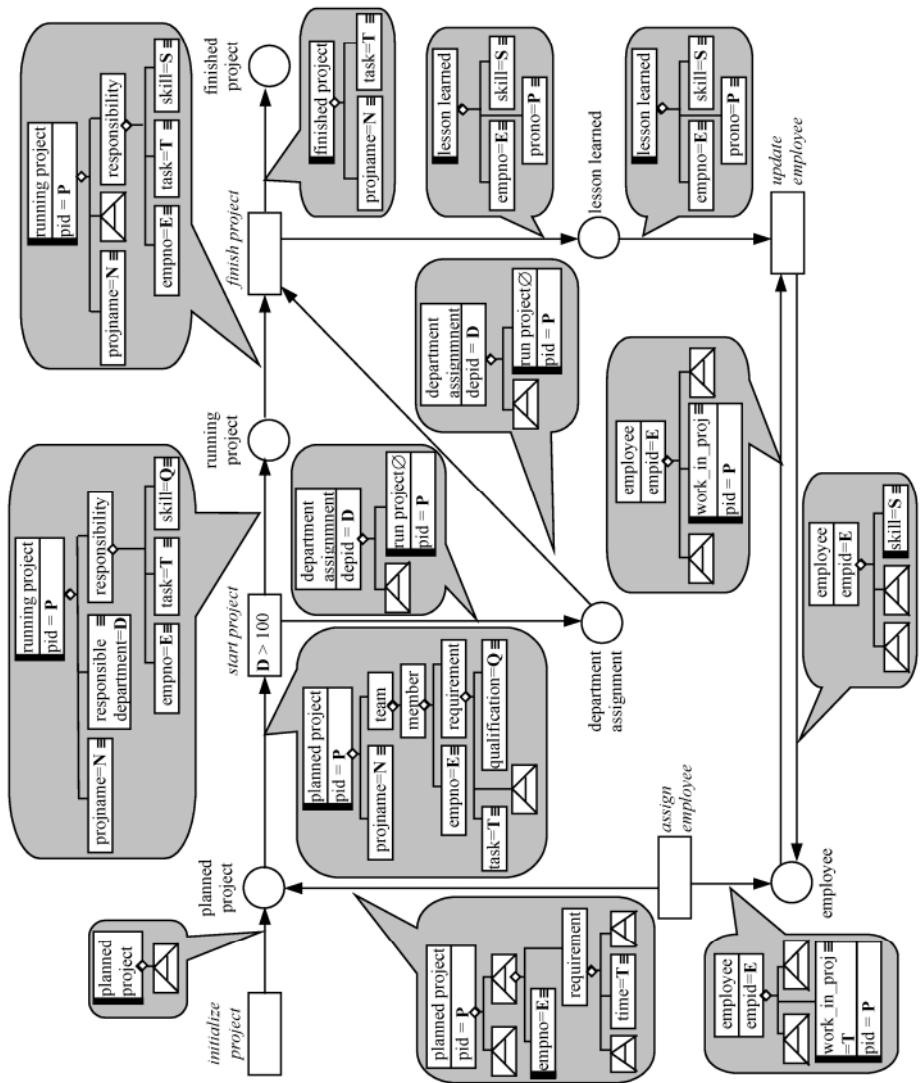


Fig. 19. XML Net for the Process of Project Execution

3.2 Fragmentation of XML Nets

The globalization of projects and companies, flexibilization of organizational structures, cooperation between companies and business-to-business commerce, mobile computing, and the development towards distributed business processes require the application of geographically distributed information system technologies. The distribution of a global, inter-organizational workflow implies a distribution of the workflow management system on the execution level and on the design level methods for the fragmentation of centralized workflows.

The fragmentation of predicate/transition nets as a kind of Petri nets is described in [14] and can easily be transferred to *XML nets*: A global *XML net* can be decomposed into several local net fragments that can be allocated to different execution sites. In general, we decompose Petri nets by splitting the places, i.e. by duplicating them and assigning the copies to the respective fragments. The initial marking of an interface place in the global net (for *XML nets* a set of valid documents) can be assigned to the corresponding places of the fragments in two ways: by replication of the documents or by allocation to one of the fragments.

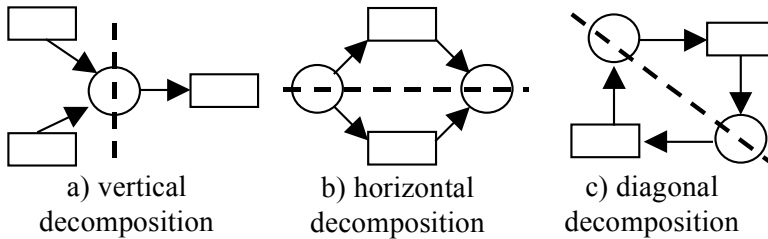


Fig. 20. Vertical, Horizontal, and Diagonal Decomposition of Petri Nets

In [14] three types of decomposition are described based on the dependencies between the fragments (i.e. the data flow in the global net): vertical, e.g. for sequential processes, horizontal as for alternative processes, and diagonal for decomposition into processes with mutual dependencies (see Figure 20 for examples).

Referring again to the workflow example for the project execution, we can identify three different organizational units involved in the execution of the workflow: the management, the department that is responsible for the execution of the specific project and the human resources department (see Figure 21). The big arcs describe the document flow and thus the dependencies between the workflow fragments. For example, the project department needs documents from the management for the execution of their part of the workflow (the running project document).

For the execution of a distributed workflow based on a Petri net several architectures are conceivable, with the degree of distribution ranging from centralized to completely distributed. Organizational structures, the distribution of workflow relevant data and documents, as well as technical restrictions are crucial criteria for the allocation of the workflow fragments. Whether a workflow engine runs in a central place, is installed at all process participants' sites, or is sent to the site on demand (together with the workflow fragment) depends for example on the size of the workflow engine in relation to the size of the workflow fragments, the average number of workflow fragments to be executed at the site, the maintenance and update cost of the workflow engine, and the flexibility of the workflow fragment allocation. For example, distribution of the workflow engine is a suitable architecture for the workflow support of business-to-business applications because of the small number of participating organizations in contrast to workflows e.g. for business-to-consumer applications like internet shopping malls.

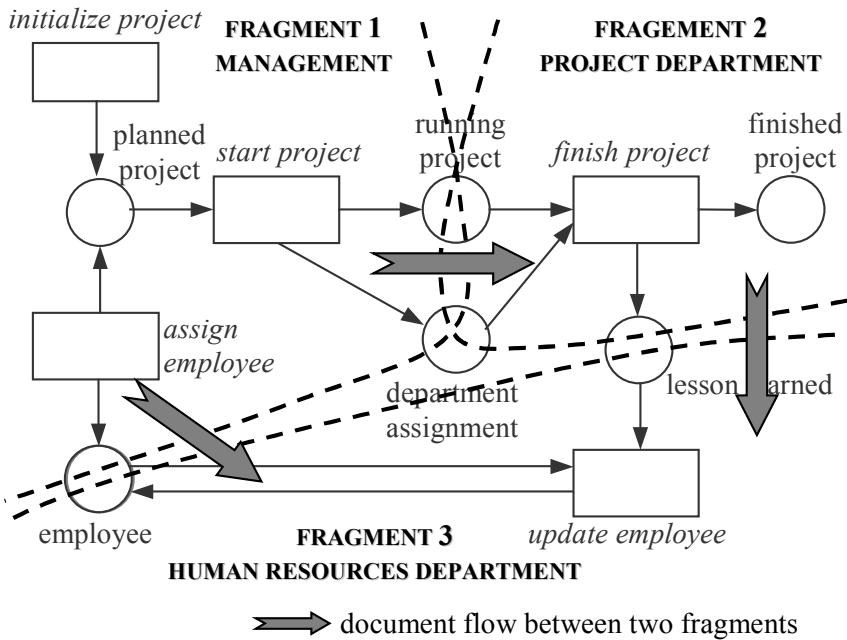


Fig. 21. Workflow Fragments for the Distributed Project Execution

4 Outlook

In this paper, we have introduced *XML nets*, which integrate behavior- and document-related aspects of distributed workflows. Recently, the formal definition of *GXSL* has been extended to cope with the concepts of XML Schema (<http://www.w3.org/XML/Schema>). In the future, *XManiLa* will be extended to querying the sequential order of elements of the same element class. Moreover, analysis methods for Petri net types will be transferred to XML nets.

For the geographical distribution of XML net-based workflows, principles of distributed database management systems may be transferred to the management of distributed XML documents, focused for example on the management of document replication. Finally, criteria for the allocation of workflow fragments depending on the allocation of documents, mass data and other relevant resources have to be found. To support the application of *XML nets* in business process management, we are currently developing a prototype tool environment which imports functionalities of existing systems:

- for the graphical representation of XML schemas, XML Spy (<http://www.xmlspy.com>) is used.
- for the graphical editing of *XML nets*, the INCOME Suite (<http://www.get-income.com>) is extended.

- a simulator for *XML nets* is implemented based on the native XML database management system TAMINO (<http://www.softwareag.com/tamino/>). The marking of a given XML net is stored as a set of XML documents in the database, each transition occurrence is implemented as a database operation on the respective XML documents.

Current work also includes developing concepts for contract management based on *XML nets* which supplements process management in the area of business to business e-commerce [17].

References

- [1] van der Aalst, W.M.P.; Desel, J.; Oberweis, A.: Business Process Management – Models, Techniques and Empirical Studies, Lecture Notes in Computer Science, Vol. 1806, Springer-Verlag, 2000.
- [2] van der Aalst, W.M.P.: Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques, in: [1], pp. 160-182.
- [3] Bonifati, A.; Ceri, S.: Comparative Analysis of Five XML Query Languages, in: SIGMOD Record 29(1), March 2000, pp. 68-79.
- [4] Booch, G.; Christerson, M.; Fuchs, M.; Koistinen, J.: UML for XML Schema Mapping Specification, 1999.
(http://www.rational.com/media/uml/resources/media/uml_xmlschema33.pdf)
- [5] Booch, G.; Rumbaugh, J.; Jacobson, I.: The Unified Modeling Language User Guide, Addison Wesley, 1999.
- [6] Brauer, W., Reisig, W., Rozenberg, G. (eds.): Advances in Petri Nets, Part I, Lecture Notes in Computer Science, Vol. 254, Springer-Verlag, 1986.
- [7] Ceri, S.; Comai, S.; Damiani, E.; Fraternali, P.; Paraboschi, S.; Tanca, L.: XML-GL: a Graphical Language for Querying and Restructuring XML Documents, in: Proc. 8th Int. World Wide Web Conference, Toronto, Canada, 1999.
(<http://www8.org/w8-papers/1c-xml/xml-gl/xml-gl.html>)
- [8] Chen, P.P.: The entity-relationship model: Toward a unified view of data, in: ACM Transaction on Database Systems 1 (1976) 1, 166 - 192.
- [9] Conrad, R.; Scheffner, D.; Freytag, Ch.: XML Conceptual Modeling Using UML, in: Laender, A.H.F.; Liddle, S.W.; Storey, V.C. (eds.): ER2000 Conference, Lecture Notes in Computer Science, Vol. 1920, Springer-Verlag, 2000, pp. 558-571.
- [10] Desel, J.; Erwin, Th.: Modeling, Simulation and Analysis of Business Processes, in [1], pp. 129-141.
- [11] Deutsch, A.; Fernandez, M.; Florescu, D.; Levy, A.; Suciu, D.: A Query Language for XML, in: Proc. 8th Int. World Wide Web Conference, Toronto, Canada, 1999.
(<http://www8.org/w8-papers/1c-xml/query/query.html>)
- [12] Extensible Markup Language (XML) 1.0 (Second Edition), World Wide Web Consortium Recommendation, Technical Report, 6-October-2000.
(<http://www.w3.org/TR/REC-xml>)

- [13] Genrich, H.J.: Predicate/Transition Nets, in [6], pp. 207-247.
- [14] Guth, V.; Lenz, K.; Oberweis, A.: Distributed Workflow Execution Based on Fragmentation of Petri Nets, in: Traummüller, R.; Csuháj-Varjú, E. (Eds.): Proc. 15th IFIP World Computer Congress 'Telecooperation – The Global Office, Teleworking and Communication Tools', Vienna, Budapest, September 1998, pp. 114-125.
- [15] Harold, E.R.; Means, W.S.: XML in a Nutshell, O'Reilly, 2001.
- [16] Jablonski, S.: Workflow Management Between Formal Theory and Pragmatic Approaches, in: [1], pp. 345-358.
- [17] Lenz, K.; Oberweis, A.; Schneider, S.: Trust Based Contracting in Virtual Organizations: A Concept Based on Contract Workflow Management Systems, in: Schmid, B.; Stanoevska-Slabeva, K.; Tschammer, V. (Eds.): Towards the E-Society: E-Commerce, E-Business, and E-Government; Proc. First IFIP Conference on E-Commerce (I3E 2001), Zurich/Switzerland, Oct. 2001, pp. 3-16.
- [18] Mani, M.; Lee, D.; Muntz, R.R.: Semantic Data Modeling Using XML Schemas, in: Kunii, H.S.; Jajodia, S.; Sølvyberg, A. (Eds.): ER2001 Conference, Lecture Notes in Computer Science Vol. 2224, 2001, pp. 149-163.
- [19] Namespaces in XML, World Wide Web Consortium Recommendation, Technical Report, 14-January-1999.
(<http://www.w3.org/TR/REC-xml-names/>)
- [20] Oberweis, A.: An Integrated Approach for the Specification of Processes and Related Complex Structured Objects in Business Applications, in: Decision Support Systems, 17, 1996, pp. 31-53.
- [21] Oberweis, A.; Sander, P.: Information System Behavior Specification by High-Level Petri Nets, in: ACM Transactions on Information Systems, 14(4), 1996, pp. 380-420.
- [22] Reisig, W.: Petri Nets: an Introduction, EATCS monographs 4, Springer-Verlag, 1985.
- [23] Robie, R.; Lapp, J.; Schach, D.: XML Query Language (XQL), in: Proc. of the Query Language Workshop, Cambridge, Massachusetts, 1998.
(<http://www.w3.org/TandS/QL/QL98/pp/xql.html>)
- [24] Scheer, A.-W.; Nüttgens, M.: ARIS Architecture and Reference Models for Business Process Management, in [1], pp. 376-389.
- [25] Walmsley, P.: Definitive XML Schema, Prentice Hall, 2002.
- [26] Weitz, W.: Combining Structured Documents with High-level Petri-Nets for Workflow Modeling in Internet-based Commerce, Int. Journal of Cooperative Information Systems (IJCIS), 7(4), December 1998, pp. 275-296.
- [27] Workflow Management Coalition: Terminology and Glossary, Brussels 1996.
(<http://www.wfmc.org>)
- [28] XML Schema Part 1: Structures, Part 2: Datatypes, World Wide Web Consortium Recommendation, Technical Report, 2-May-2001.
(<http://www.w3.org/TR/xmlschema-1/> and <http://www.w3.org/TR/xmlschema-2/>)
- [29] Zloof, M.M.: Query-By-Example: The invocation and definition of tables and terms, in: Kerr, D.S. (Ed.): Proc. Int. Conference on Very Large Data Bases, Framingham/Massachusetts, 1975, pp. 1-24.

Three-Level Process Specification for Dynamic Service Outsourcing: From Petri Nets to ebXML and WFPDL

Paul Grefen and Samuil Angelov

Computer Science Department, University of Twente
P.O. Box 217, 7500 AE Enschede, Netherlands
www.cs.utwente.nl/~{grefen,sangelov}
{grefen,sangelov}@cs.utwente.nl

Abstract. Service outsourcing is the business paradigm, in which an organization has part of its business process performed by a service provider. In dynamic markets, service providers are selected on the fly during process enactment. The cooperation between the parties is specified in a dynamically made electronic contract. This contract includes a process specification that is tailored towards service matchmaking and cross-organizational process enactment and hence has to conform to market and specification standards. Process enactment, however, relies on intra-organizational process specifications that have to comply with the infrastructure available in an organization. In this paper, we present a three-level process specification framework for dynamic contract-based service outsourcing. This framework relates the two process specification levels through a third, conceptual level. Petri nets are used for conceptual process specification, whereas the external and internal levels are based on the ebXML standard respectively workflow management technology. We show how the framework can be placed in the context of infrastructures for cross-organizational process support.

1 Introduction

Nowadays, many organizations focus on specific parts of their business process and rely on partners in a market to perform the additional parts of the process required to reach their business goals. A common business paradigm is that of service outsourcing, in which an organization focuses on its core business process (the central part of its primary process) and has secondary process parts enacted on its behalf by service provider organizations. In this paradigm, the outsourcing organization is referred to as service consumer. The details of service outsourcing are specified in a contract between both parties.

Service outsourcing can be found in many market segments. An example is an insurance company that focuses on the core insurance process and outsources secondary activities like customer call handling, and insurance claim loss assessment. Other ex-

amples can be found in the logistics domain, where companies outsource their logistic subprocesses to specialist organizations in this field. We will elaborate an example from this domain in the sequel of this paper.

The combination of service consumer and service provider can be seen as a virtual enterprise that presents itself to a third party (for example a customer) as a single entity. Traditionally, these virtual enterprises have a more or less stable character over time. In modern dynamic e-commerce markets, however, players in a market and general market conditions change that fast, that a more dynamic approach is required to service outsourcing to create or retain a competitive position. This means that in service outsourcing, service consumers dynamically determine which service providers to use in the enactment of their business processes.

Dynamic service outsourcing implies a number of steps: services have to be identified and defined, compatible business partners have to be found in an efficient way, contracts have to be made dynamically, process enactment infrastructures have to be set up and coupled, and the process has to be actually enacted. Requirements to speed and costs dictate that all of this takes place electronically, preferably in a fully automated fashion.

To enable this, clear specifications are necessary that describe business processes in the context of the above-mentioned steps in service outsourcing. Given the fact that these steps are quite diverse in nature, a single process specification will not be adequate for all these activities. Hence, we propose a multi-level process specification approach to be used in dynamic service outsourcing. In this approach, each level has its own specification characteristics and related specification techniques. The level of process specification aimed at collaboration has to be embedded in a framework for electronic contracts to support the proper specification of the business relationship between partners in an outsourcing relationship. As such, the electronic contracts are the specification of the dynamic virtual enterprise, in which the process specification provides the operational interaction specification.

The multi-level framework we propose is inspired by the ANSI-SPARC model [18] that is well known in the database community for describing three levels of data management. The general approach to service outsourcing used in this paper is inspired by the approach taken in the CrossFlow project [12, 14].

1.1 Structure of this Paper

In Section 2, we first present our approach in terms of a three-level process model, explaining what the global purpose of each of the levels is. In Sections 3 to 5, we discuss each of the three levels and the mappings between them in more detail. At the central level, Petri Nets are used as specification technique. At the two other levels, we use UML activity diagrams in the context of ebXML respectively a workflow process definition language of a commercial workflow management system. The levels are illustrated by means of a running example from the telecom and logistics area. We place the process model in the context of process support infrastructures in Section 6. We end the paper with conclusions. Main purpose of this paper is to paint the overall picture of our approach to multi-level process support for service outsourcing and show the position of various process specification techniques in this approach.

2 A Three-Level Process Model

In this section, we introduce our three-level process specification framework for dynamic service outsourcing. We first explain the need for a separate conceptual process specification level. Then, we present the three levels of the framework and their relationships.

2.1 The Need for a Conceptual Specification Level

In service outsourcing, we require process specifications that specify all necessary details of process and data structures with respect to these services and their enactment. These specifications exist on two levels from an operational point of view.

Firstly, we require process specifications that can be shared between multiple organizations (service consumer and service providers). These specifications should allow for service brokering between partners when setting up a virtual enterprise on the one hand, and service enactment in a running virtual enterprise on the other hand. Brokering is used to relate service consumers and providers, either through simple matching of specification characteristics or through more advanced matchmaking. In service enactment, services are actually executed in an inter-organizational setting. Interoperability is the main characteristic of this specification level. The nature of this level is often imposed by market standards. If an organization operates in multiple markets, it may be confronted with multiple standards.

Secondly, we require process specifications at the implementation level of the process enactment systems employed by each of the organizations cooperating in a virtual enterprise. Executability is the main characteristic of this specification level. The nature of this level is often imposed by current process and data management systems – be it of an up-to-date or legacy nature. Again, an organization having a heterogeneous infrastructure may be confronted with multiple specification types here.

None of the above two levels is focused on conceptual process specification, however, that is independent from both practical collaboration and implementation aspects. A conceptual specification is required, though, for three main reasons:

1. A conceptual specification is necessary to guard the quality of processes through proper design, analysis, and validation of process structures. Certainly, in the field of e-business where automated processes are the backbone of an organization, quality of these processes is of the highest importance.
2. A conceptual specification is used as a 'semantic bridge' between the standards used at the interoperability level on the one hand and the implementation level on the other hand. This bridge provides a separation of concerns with respect to specific details of semantics at the two levels.
3. A conceptual specification is also required as a unifying level to bridge multiple specification types at both interoperability and implementation levels. As such, it is a basis for portability of process specifications across multiple markets with diverse standards on the one hand and multiple process and data management systems with diverse technologies on the other hand.

2.2 The Three-Level Framework

For the reasons discussed above, we introduce a third, conceptual process specification level between the two levels discussed above. Consequently, we arrive at the following three-level process specification model:

External Level: The external level is geared towards communicating a process specification between different organizations. It can be considered a projection of the conceptual level, where projection uses hiding and translation operations.

Conceptual Level: The conceptual level is the centerpiece of process specification. It is independent from external use and internal implementation. It is used for conceptual reasoning about the process, e.g. for design and analysis purposes. The conceptual level is a combination of abstraction and aggregation of the internal level.

Internal Level: The internal level is geared towards enactment of processes in the context of a specific organization, e.g., by means of workflow management systems. The internal level is a mapping of the conceptual level, where mapping is a combination of translation (specialization for a specific platform) and refinement.

The relation between the three process levels in service outsourcing is illustrated in Figure 1. In this figure, we see that two organizations can each have local conceptual and internal process specifications, but that they share a common external specification. This common specification can be based on a market standard, can be exported by one organization and imported by the other, or can be constructed in a process of negotiation between the two organizations.

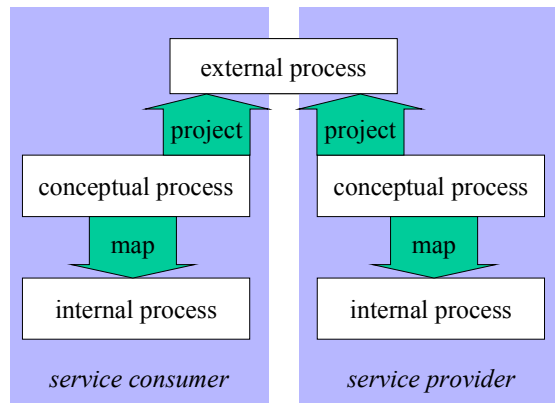


Fig. 1. Three-level framework

Note that the three above levels coincide more or less with the well-known ANSI-SPARC three-level model in the database community [18, 8]. The ANSI-SPARC model distinguishes between external, conceptual and internal levels for data management. Aims of the conceptual and internal level are similar to that in our approach – except of course for the data versus process management perspective. Main difference is the use of the external level. In the ANSI-SPARC model, the external level contains multiple views on a database that cater for different user groups in an organi-

zation. In our model, the external level contains multiple 'views' on a process to cater for different organizations in an electronic market.

In the sections below, we discuss the levels of the framework in more detail. We start with the conceptual level, as this level is the pivotal point in our framework. Next, we move 'up' to the external level, paying attention to the nature of this level and the projection from the conceptual level. After that, we move our attention 'down' to the internal level and to its mapping from the conceptual level.

3 The Conceptual Level

The conceptual process specification level is the center point in our framework for cross-organizational process support. It is used for the design, analysis and verification of cross-organizational processes. In this section, we first discuss the nature of this process specification level. We illustrate this level with an example process specified in Petri Nets. Next, we pay attention to design and analysis of conceptual process specifications.

3.1 The Nature of the Conceptual Process Specification Level

The conceptual process exhibited by an organization to the outside world is usually less detailed than the implementation of the same process actually enacted by the organization. Consequently, several levels of detail exist in a conceptual process specification. These levels of detail correspond to process aggregation levels. The mapping between these aggregation levels is dealt with by a process refinement hierarchy.

As the conceptual specification level is the design and analysis level of our process specification framework, we place the process refinement levels that have a conceptual meaning within this level. Refinement that is related to mapping to process enactment infrastructures is dealt with in the mapping between conceptual and internal process specification – see Section 5.2. This ensures a separation of concerns between the 'intention' of a process and its implementation on a specific platform.

To properly support design and analysis tasks at the conceptual level, an adequate process specification technique has to be selected. As we deal with discrete business processes, we require a discrete process modeling technique. As unambiguous reasoning about process specifications should be possible at this level, we require a specification technique with a formal background. Clearly, the chosen technique should be mappable to both the external and the internal process specification levels.

Variants on Petri Nets are widely accepted in the business process reengineering and workflow management domains (see for example [1, 2]), both from a research and from an industrial perspective. Their token-based semantics are well suitable to model properties of business processes. Also, many tools exist for design and analysis of Petri Nets. Therefore, we use Petri Nets as the specification technique for the conceptual specification level in this paper. We illustrate this below by means of an example process.

3.2 An Example Conceptual Process Specification

Our example is based on a real-world logistics scenario in the telecom industry developed in the CrossFlow project [6, 12]. In this scenario, a telecom operator sells mobile phones (GSMs) and network subscriptions for these phones. The subprocess of delivering the phones to clients is dynamically outsourced to a logistics company. The example process at the telecom operator is specified as a Petri Net at the conceptual level in Figure 2.

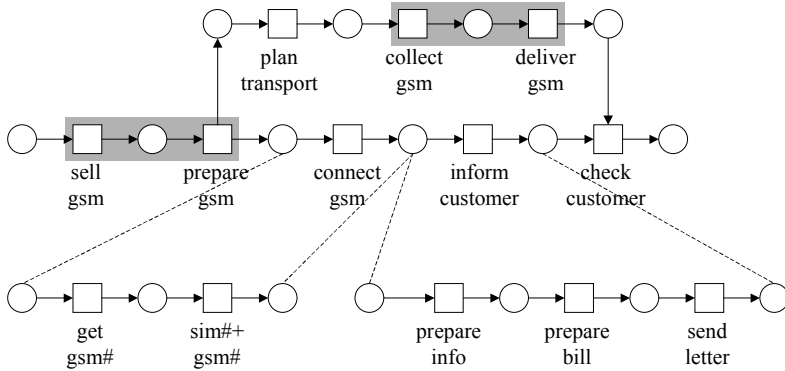


Fig. 2. Example conceptual process, levels 1 (top) and 2 (bottom)

In the upper part of the figure, we see the level 1 conceptual process model. The telecom operator sells a GSM, using telephone or web communication with its client. It then prepares the GSM for transport. After that, two parallel process branches are executed. In-house, the GSM is connected to the network and the customer is informed by mail about the connection, the personal identification code and the subscription. The process of physically delivering the GSM to the customer is outsourced to a logistics operator, who performs the delivery in three sequential steps¹. After completion of the two parallel branches, the telecom operator checks up on the customer by phone.

Apart from the control flow of the process, certain execution characteristics are required, e.g. with respect to transactional properties or timing of processes. Two examples are drawn as gray boxes in Figure 2 (following the idea of spheres from [5]). The box around 'sell gsm' and 'prepare gsm' indicates a transactional property: these two activities should be executed atomically. The box around 'collect gsm' and 'deliver gsm' indicates a timing property: these two activities should be completed within a certain timeframe (i.e., after the phone has been picked up at the telecom company, it should be delivered at the client in say three days). Note that in practice, more execution characteristics will be specified.

In the lower part of Figure 2, we see (part of) the refinement of the level 1 into the level 2 process model. Activity 'connect gsm' has been refined into two activities for retrieving a GSM phone number respectively coupling this in the administration to the

¹ Note that we have used a simple example outsourced process for reasons of brevity and clarity. The outsourced process in [6] contains more than 10 steps and alternative process flows.

SIM number of the phone. Activity ‘inform customer’ has been refined into three activities for preparing customer information, preparing a bill and sending a letter to the customer including information and bill.

3.3 Designing, Analyzing and Verifying Conceptual Process Specifications

Design of processes takes place at the conceptual level, preferably in a top-down fashion. In a service outsourcing scenario, this means that at the highest aggregation level of the conceptual level, the conceptual interaction between service consumer and service provider are determined.

At this level, the service consumer specifies its conceptual cross-organizational business process in conceptual terms. This process contains activities that are enacted in-house, which are refined in the conceptual level if necessary. It also contains activities to outsource, which are not further refined, but enacted by a service provider.

The service provider also specifies the service it offers in conceptual terms at this level. The service specification may be extended with additional activities that are internal to the service provider, for example administrative activities. All activities can be refined within the conceptual process specification level.

Refinement of processes at the conceptual level should stop at the level where independence from execution infrastructures cannot be guaranteed. This means, for example, that process properties of specific workflow management systems should not be reflected at the conceptual level.

Summarizing the above, there is a highest process aggregation level at the conceptual specification level – determined by collaboration characteristics – and a lowest aggregation level – determined by platform independence. We call these levels the conceptual upper and lower bound. This is illustrated in Figure 3, where we see three conceptual refinement levels (C1 to C3) between external (E) and internal (I1 and I2) levels.

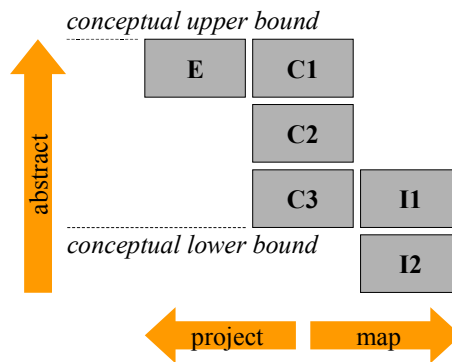


Fig. 3. Conceptual upper and lower bound

The nature of the conceptual process specification allows for analysis and verification of processes. Analysis may, for example, focus on transactional characteristics of business processes [5]. Transactional characteristics determine for example which parts of a process must be executed in an atomic fashion and under which conditions

access to shared data is possible. Analysis may also have a more quantitative character, for example by using simulations of process runs. Verification techniques may be used to prove certain characteristics of processes, e.g., reachability or termination characteristics. Analysis and verification lead to the assessment of quality characteristics of services that may be required as attributes at the external process specification level.

4 The External Level

Interoperability of processes is the main focus of the external level. Below, we discuss the nature of external process specifications and their mapping from the conceptual level. We illustrate this level with an example. Next, we discuss the role of external process specifications in electronic contracts. We end this section with the brief discussion of an example contract model that includes explicit process specification.

4.1 The Nature of External Process Specifications

Process specifications at the external level aim at defining interoperation (or collaboration) of two organizations. More concrete, these specifications are used for two purposes:

Service Matchmaking: The external process specification is used in matchmaking between service consumers and service providers, i.e., in setting up a virtual enterprise. The external process required by a consumer should match with an external process offered by a provider in order to link their processes.

Cross-Organizational Process Control: The external process specification is used for controlling the cross-organizational service enactment, i.e., it is the basis for cross-organizational process monitoring and control.

Translation of the conceptual process specification to the external level is necessary to conform to market standards – this to allow interoperability between partners. Where the conceptual process specification uses a formal process model – as explained above – the external model is based on a process specification standard that is common within a certain market. In this context, we find XML-based standards like ebXML [7] or WSDL [22]. In this paper, we assume ebXML as the basis for the external level. Consequently, we use UML activity diagrams as process specification technique (see also Section 6.2) – we illustrate this in the sequel. The semantics of activity diagrams in workflow specification is discussed in [9].

A process specification at the external level can be a high-level abstraction of a complex process that on the conceptual level is considerably refined into separate subprocesses. Still the process specifications at the external level should not be too general as they may turn the process of service outsourcing in a black box process, thereby not allowing the fine-grained cooperation that is required in dynamic virtual enterprises. Clearly, this is closely related to the discussion of the conceptual upper bound in the previous section.

The external level includes service execution characteristics that define how the service is executed. These characteristics can be of different natures. An important class of characteristics is the specification of the transactional behavior of service execution, determining for example how failures and concurrent access to shared resources are handled. A second class is the specification of monitoring and control points in the service process and available control primitives – this to specify how a service consumer can observe and influence the enactment of an outsourced service. Quality of service attributes, e.g. execution times or success probability, form another important class.

4.2 Mapping the Conceptual Level to the External Level

The external specification is a projection of the conceptual process specification, where projection is a combination of hiding and translating.

Parts of the conceptual process can be hidden in the external process because they are not relevant for a cooperation partner. A service consumer might hide everything of its conceptual process but the subprocess to be outsourced. A service provider will usually show most of its conceptual process, but might hide some administrative tasks at the end of a service specification. Process steps may also be renamed from conceptual to external level, as internal names may not coincide with standard names in a market.

Translation is necessary to transform primitives from the conceptual process specification technique into the process specification standard used at the external level. Also, execution quality characteristics determined at the conceptual level have to be translated into service attributes at the external level.

4.3 Example External Process Specification

The top part of Figure 4 shows the external process based on the conceptual process discussed before. As indicated before, we base the external level specification on ebXML. Consequently, we use UML activity diagrams as the graphical process specification technique (see also Section 6.2).

We show how the top-level conceptual process is projected onto the external level. Only the activities to be outsourced are represented at the external level in the external representation language. We see that steps in the process are renamed. Where the conceptual specification uses telecom-specific names, the external specification uses names generally used in the logistics industry: 'collect gsm' is for example renamed into 'collect parcel'.

The mapping between Petri Net and activity diagram is straightforward in this simple case: each transition in the Petri Net is mapped onto an activity in the activity diagram. In more complex cases, the mapping may be less trivial. In [10], a comparison is made between the two techniques, elaborating the differences between them in the context of workflow management.

Enactment characteristics specified in the conceptual process specification are mapped to separate enactment clauses at the external level (shown in the upper right corner in the figure). Both process specification and enactment clauses must be integrated into one transferable process specification.

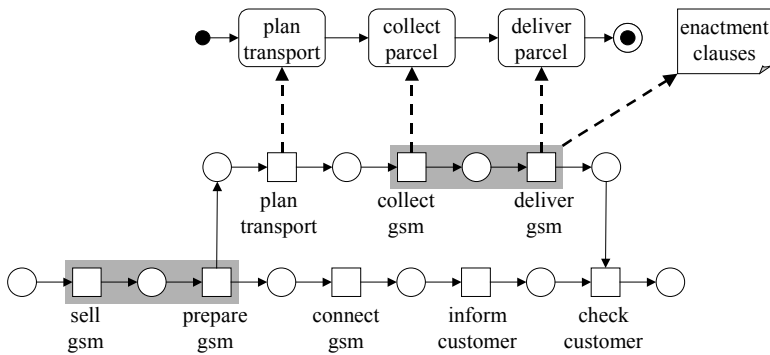


Fig. 4. Example external process (top), projected from conceptual (bottom)

4.4 External Process Specifications in Electronic Contracts

Electronic contracts form the basis for the automated formation of dynamic virtual enterprises. One of the most important aspects of this formation is the alignment of the business processes of the cooperating organizations. For this reason, the external process specification is a key element in electronic contracts in the domain of service outsourcing [3, 15].

On the external level, there are two types of process specifications that are relevant to electronic contracts: establishing contracts between service consumer and provider and enacting contracts based on established contracts. In this paper, we focus on the specification of processes related to contract enactment, i.e., processes for enacting outsourced services. For reasons of completeness, however, we briefly describe both types of processes below.

The first type of process specification describes the establishment of an electronic contract, i.e., the process of negotiation and exchanging preliminary data between two organizations. This process is specified in an electronic offer and subsequently – if necessary – in an electronic contract. The reason to include this process specification in a contract is for example to achieve contract reusability. The established contract can be stored and subsequently, when new business relations between the parties appear, the preliminary process can be repeated in an automated manner and will lead to a new instance of the contract. This kind of process specification can also be reflected at the conceptual and internal specification levels.

The second type of process specification concerns a specification of the outsourced service. This specifies the process execution chronology and components, and supports the cross-organizational process monitoring and control. Specification of these processes at the right level of granularity at the external level is of high importance for the proper execution of the services accompanying contract enactment, e.g., contract monitoring, contract enforcement, etc. The granularity is determined in the design process at the conceptual level, as described in Section 3. The design determines which process specifics can remain hidden from the contractual relation between a service consumer and a provider. These observations for the process specifications embedded in the electronic contract are an underpinning for the need of a conceptual level that will play a mediating role and will allow independence between the internal and external levels.

4.5 Example Contract Model

In Figure 5, the top level of the CrossFlow contract model [15] is shown as an example contract structure. This model consists of five main entity clusters: a concept space that defines concepts relevant in the contract, a workflow schema that describes a service process, enactment clauses that specify how to execute the workflow schema, usage clauses that specify how to establish a contract, and a natural language description of the contract for human interpretation. The concept space is essential for specifying the context of the service outsourcing, e.g., to identify the parties involved in service outsourcing. The various clusters are interlinked to describe relations between them.

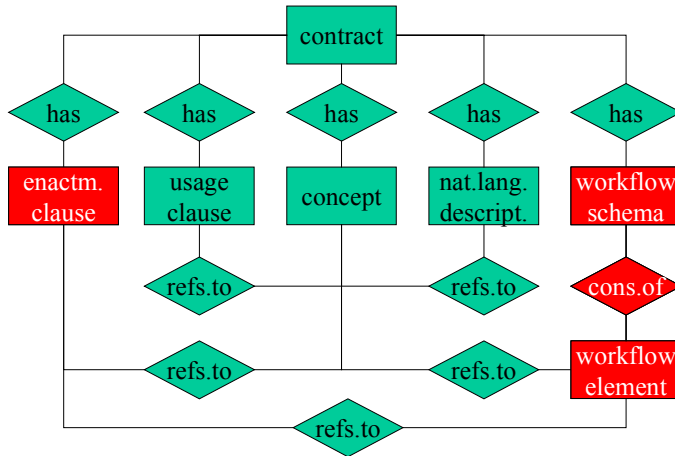


Fig. 5. CrossFlow contract model

The workflow schema and enactment clauses coincide with the external process specification as proposed in this paper. The workflow schema describes the service process in an abstract process specification language, consisting of activities and transitions. This abstract language is easily mappable to UML activity diagrams as used in this paper. Enactment clauses come in different ‘flavors’ to describe specific classes of execution characteristics, for example transactional behavior and quality of service parameters.

5 The Internal Level

The internal process specification is used to have local parts of cross-organizational processes enacted by process support systems. Below, we first describe the nature of internal process specifications. Next, we discuss the mapping of specifications from the conceptual level to the internal level. Finally, we show an example internal process specification based on our running example.

5.1 The Nature of Internal Process Specifications

Internal process specifications are used to describe enactment processes by organizations participating in cross-organizational processes.

Workflow management systems are a general infrastructure for the automated support of business processes enactment. Often, these systems are separate entities in an information system infrastructure; sometimes they are embedded in other systems, like ERP systems. Hence, the internal process specification level can be based on a workflow process definition language (WFPDL). WFPDLs are discrete in nature and often use Petri Net based process models, so they match the requirements as discussed at the conceptual level. They usually contain primitives allowing easy specification of various process constructs, for example various forms of process splits and joins. We show an example WFPDL in the sequel of this section.

The internal process specification contains a mapping of the activities in the process to the enactment resources available in the organization. These resources include the employees in the organization and the back-end information systems employed in the business processes. WFPDLs contain primitives that cater for this specification.

In practice, a WFPDL is usually WFMS-specific: each WFMS has its own WFPDL. Most of the languages, however, conform to standard concepts [20] defined by the Workflow Management Coalition [21].

5.2 Mapping the Conceptual Level to the Internal Level

Abstract process models defined at the conceptual level have to be mapped to workflow models in a WFPDL. This requires a syntactical translation, but also the addition of resource specifications as discussed above. Given the global similarity of the natures of process primitives in conceptual specification language and WFPDL, process translation is in most practical cases not problematic. In specific cases, however, the precise semantics of constructs at both levels have to be analyzed to obtain a correct translation. Clearly, quality characteristics identified at the conceptual level have to be taken into account in the translation – such that guarantees specified in contracts at the external level are in fact delivered at the internal level.

In some cases, the process aggregation level available at the internal level is lower than that of the lowest refinement level of the conceptual level. The conceptual level may, for example, contain multi-user tasks, whereas the workflow system employed at the internal level only allows single-user tasks. In this case, an additional refinement step has to take place to go from conceptual to executable internal level (as illustrated in Figure 3 where level I1 is refined into I2). Note that refinement only takes place here if technology limitations dictate this. Refinement from a functional point of view takes place at the conceptual level. We further illustrate this below.

5.3 Example Internal Process Specification

In Figure 6, we show level 1 of the internal process specification in the graphical version of the WFPDL of the FORO workflow management system [11]. The graphical representations of most other WFPDLs are more or less comparable – the reason for

choosing the FORO notation is that it makes some details explicit (as explained below).

We see that the conceptual process specification can be mapped almost directly onto the internal specification. Start nodes, end nodes, splits and joins in the process are, however, modeled explicitly in the FORO FWPD. Also, the difference between atomic tasks and subprocesses is explicitly modeled (note that tasks ‘connect gsm’ and ‘inform customer’ are represented by a different symbol than the other tasks; this symbol indicates subprocesses that are exploded at a lower level). For reasons of brevity, we do not illustrate how specifications of resource mappings (roles in the process) are added to the internal process specification.

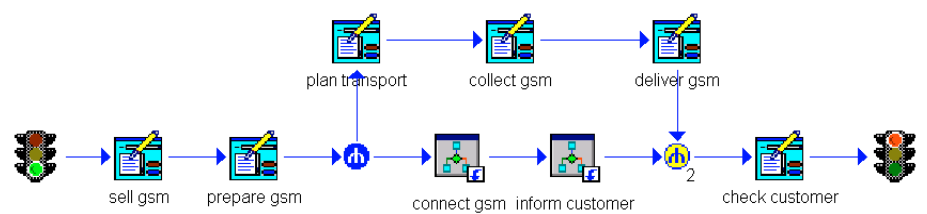


Fig. 6. Example internal process in FORO WFPDL (level 1)

Note that the above example mapping from conceptual to internal specification is rather straightforward. Depending on the functionality of the process enactment platform, the mapping may be more complex. If the process enactment platform would not support nested processes (subprocesses), for example, the nested structure of the conceptual specification as shown in Figure 2 would need to be ‘flattened’ into a single level. If the process enactment platform would not allow access to multiple back-end systems in one task, the task ‘sim#+gsm#’ at the second conceptual level would have to be split into two tasks ‘get sim#’ and ‘enter sim#’. The resulting process specification at the internal level is shown in Figure 7. Note that this level is a refinement of levels 1 and 2 of the conceptual level. This refinement is made at the internal level, as it is driven by platform specifics, not by conceptual process functionality. As such, the distinction between refinement at conceptual and internal process levels caters for both a separation of concerns and portability of conceptual process specifications across implementation platforms.

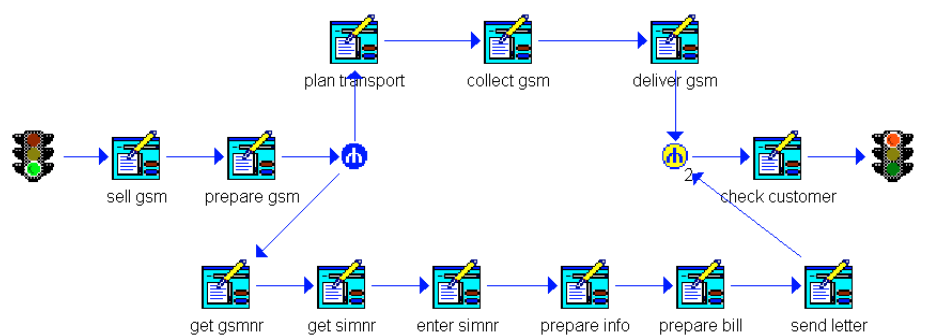


Fig. 7. Example internal process specification for limited platform

6 Process Support Aspects

In this section, we focus on more practical aspects of the proposed three-level modeling approach by placing it in the context of two example process support environments – note that there are more relevant environments. To take the research point of view, we first place the approach in the context of the CrossFlow architecture. To take the industrial standardization point of view, we place the approach in the context of the ebXML architecture. As our approach leads to the existence of multiple interrelated process specifications per process, storage of process specifications is an important practical aspect. Therefore, we end this section by relating storage aspects to our framework.

6.1 CrossFlow Architecture

In the CrossFlow project, concepts and technology for workflow support in dynamic virtual enterprises have been developed [12, 23]. In the context of this project, the formation of virtual enterprises is based on dynamic service outsourcing, as advocated in this paper as well.

Service offerings and service requests are specified in electronic contract templates [15], which are matched by a service matchmaker. An established electronic contract is the basis for the dynamic generation of a service enactment infrastructure [13], based on workflow management technology.

In the CrossFlow approach, external process specification is part of contract specification. For this purpose, the contract model contains a process specification submodel, based on the standard of the Workflow Management Coalition. The process specification submodel is augmented with an extensible set of enactment clauses, which specify service enactment characteristics, as discussed in Section 4.1. Apart from this, the contract model includes usage clauses, describing aspects of establishing contracts – this is a first step towards specification of establishment processes as discussed in Section 4.4. The contract model is reflected in a dedicated, XML-based contract specification language [15].

The internal process specification in CrossFlow is formed by the workflow specification language of the workflow management system, in the case of the CrossFlow prototype IBM MQSeries Workflow [16].

In the CrossFlow approach, no separate conceptual level for process specification is used. An internal enactment specification is used for a direct mapping of external onto internal level [13]. This means that conceptual process design and analysis are not integrated into the CrossFlow approach. This can be illustrated by the support for advanced cross-organizational transaction management in CrossFlow [19]. This functionality is supported both at the external and internal levels, but without conceptual analysis facilities integrated in the model. In complex situations, however, this analysis is desirable to determine the precise semantic effects of transaction management.

6.2 ebXML Architecture

ebXML is an initiative governed by UN/CEFACT and OASIS, joined by major companies and standardizing consortiums, that aims to provide a framework for establishment of business relations and subsequent execution of business transactions [7].

ebXML provides a process modeling technique based on the UN/CEFACT Modeling Methodology (UMM) that utilizes UML. The modeling technique employs UML activity diagrams and sequence diagrams in both a graphical and a textual, XML-based format. In Section 4.3, we have shown an example activity diagram.

The process modeling and specification stage in the ebXML framework relates to the conceptual level of our three-level process model. However, ebXML does not differentiate process specifications at the conceptual level from process specification at the external level. As these two levels are mixed up and there is no explicit attention for refinement of processes in the approach, detailed mappings to internal level process specifications may be hard to obtain.

As a result of process modeling in the ebXML approach, Collaboration Protocol Profiles (CPPs) are created at the service consumer and service provider sides that reflect the process specification at the external level of both parties. The CPP includes specifications of the supported processes, transaction chronology descriptions, etc. and is the contract offer specification described in Section 4.3. On the base of the CPPs of both parties, a Collaboration Protocol Agreement (CPA) is created. The CPA is the electronic contract that specifies the agreed processes that will be performed, their chronology and possible interactions between parties.

The existence of ebXML as a global standard specification allows achieving the interoperability required on the external level. ebXML does not cover in its specification process specifications at the internal level. Other standardizing efforts exist with comparable goals as ebXML, for example WSDL [22] and BizTalk [4].

6.3 Process Specification Storage

The proposed approach to service outsourcing leads to complex process specifications at three levels, with possibly one-to-many mappings between the levels. Consequently, an infrastructure for specification storage and management is needed, paying due attention to consistency requirements between specifications and version management of specifications. Storage of all specifications in central process specification database is an approach to match these requirements. This database will have a semi-structured data format and might be supported by a database management system with an XML front end.

As discussed in this paper, specifications at the external level are usually stated in a language based on XML, e.g., ebXML. Mapping to XML infrastructures is thus trivial at this level. At the conceptual level, this requires a mapping of the used specification technique, for example Petri Nets, into XML. At a number of places, XML representations of Petri nets are currently being developed. An example is the Petri Net Markup Language [17]. At the internal level, an XML representation of a WFPDL is required. The application of XML for workflow management currently finds some application in Wf-XML standard of the Workflow Management Coalition [21], although this language does not cover a complete WFPDL. Commercial workflow

management systems, e.g. IBM's MQSeries Workflow [16], are being developed towards XML support.

7 Conclusions

The use of contract-based dynamic service outsourcing opens ways to efficient handling of fine-grained cross-organizational processes. Efficient means that contractual outsourcing can take place in a fast and cheap way – quite different from the traditional paper-based situation. This paradigm allows the creation and dismantling of short-term virtual enterprises, thereby supporting highly dynamic cooperation in fast-changing markets. This can even enable completely new business models in situations where formal service outsourcing is required but traditional ways to do so are too costly or slow.

The proposed three-level approach to business process specification provides a clear separation of concerns in business process support, thereby increasing quality, flexibility and reusability of process specifications in cross-organizational settings. Having a conceptual process specification as a center point and external and internal specifications as derivatives, provides a separation of concern in process design on the one hand and portability over both market standards and implementation platforms on the other hand. The separation of concerns is becoming increasingly important, as the complexity of automated cross-organizational processes grows through the advent of e-business and e-commerce. Portability in both dimensions is required to deal with fast changes in this domain, both with respect to business and technical aspects.

We have shown how Petri Nets can be used as an adequate specification technique for the conceptual process specification level. In this paper, we have illustrated the external and internal specification levels by means of UML activity diagrams in the context of ebXML respectively a specific workflow process definition language.

A spectrum of work is being performed at the University of Twente that is related to the approach presented in this paper. Support for e-contract handling is being researched, using the three-level framework for contract specification (both data and process aspects). Research into flexible architectures for cross-organizational process support is being performed, both on the conceptual and technical architecture levels. Models and architecture for transactional cross-organizational workflow processes are a further topic of research.

Acknowledgements

All former CrossFlow project members are acknowledged for their role in the CrossFlow project, on which part of the ideas presented in this paper are based. Juliane Dehnert is thanked for her feedback on the draft version of this paper.

References

- [1] W. van der Aalst; *The Application of Petri Nets to Workflow Management*; Journal of Circuits, Systems and Computers, Vol. 8, No. 1, 1998; pp. 21-66.
- [2] N. Adam, V. Atluri, W. Huang; *Modeling and Analysis of Workflows using Petri Nets*; Journal of Intelligent Information Systems, Vol. 10, No. 2, 1998; pp. 131-158.
- [3] S. Angelov, P. Grefen; *B2B eContract Handling - A Survey of Projects, Papers and Standards*; CTIT Technical Report 01-21; University of Twente, 2001.
- [4] *BizTalk Framework*; <http://www.biztalk.org>.
- [5] W. Derks, J. Dehnert, P. Grefen, W. Jonker; *Customized Atomicity Specification for Transactional Workflows*; Procs. 3rd Int. Symp. on Cooperative Database Systems for Advanced Applications; Beijing, China, 2001; pp. 155-164.
- [6] M. Duitshof; *Logistics Prototype Deployment Report*; CrossFlow Project Deliverable D13; KPN Research, The Netherlands, 2000 (available via <http://www.crossflow.org>).
- [7] ebXML Technical Architecture Specification v1.0.4; <http://www.ebxml.org>.
- [8] R. Elmasri, S.B. Navathe; *Fundamentals of Database Systems*; Benjamin/Cummings, 1994.
- [9] R. Eshuis, R. Wieringa; *A Formal Semantics for UML Activity Diagrams - Formalising Workflow Models*; CTIT Technical Report 01-04; University of Twente, 2001.
- [10] R. Eshuis, R. Wieringa; *A Comparison of Petri Net and Activity Diagram Variants*; Procs. 2nd Int. Coll. on Petri Net Technologies for Modelling Communication Based Systems; Berlin, 2001; pp. 93-104.
- [11] *FORO Workflow Management System*; <http://dis.sema.es/projects/FORO/foro.html>; Sema Group, Spain.
- [12] P. Grefen, K. Aberer, Y. Hoffner, H. Ludwig; *CrossFlow: Cross-Organizational Workflow Management in Dynamic Virtual Enterprises*; Int. Journ. of Computer Systems Science & Engineering, Vol. 15, No. 5, 2000; pp. 277-290.
- [13] Y. Hoffner, H. Ludwig, C. Gülcü, P. Grefen; *Architecture for Cross-Organisational Business Processes*; Procs. 2nd Int. Worksh. on Advanced Issues of E-Commerce and Web-Based Information Systems; Milpitas, CA, USA, 2000; pp. 2-11.
- [14] Y. Hoffner, S. Field, P. Grefen, H. Ludwig; *Contract Driven Creation and Operation of Virtual Enterprises*; Computer Networks, Vol. 37, No. 2; Elsevier, 2001; pp. 111-136.
- [15] M. Koetsier, P. Grefen, J. Vonk; *Contracts for Cross-Organizational Workflow Management*; Procs. 1st Int. Conf. on Electronic Commerce and Web Technologies; London, UK, 2000; pp. 110-121.
- [16] *IBM MQSeries Workflow*; <http://www-4.ibm.com/software/ts/mqseries/workflow/>; IBM.
- [17] *Petri Net Markup Language*; <http://www.informatik.hu-berlin.de/top/pnml/>.
- [18] D. Tsichritzis, A. Klug; *The ANSI/X3/SPARC DBMS Framework*; AFIPS Press, 1978.
- [19] J. Vonk, W. Derks, P. Grefen, M. Koetsier; *Cross-Organizational Transaction Support for Virtual Enterprises*; Procs. 5th Int. Conf. on Cooperative Information Systems; Eilat, Israel, 2000; pp. 323-334.
- [20] Workflow Management Coalition; *Workflow Standard Interface 1: Process Definition Interchange Process Model*; Doc. Nr. WfMC TC-1016-P; 1998.
- [21] Workflow Management Coalition; <http://www.wfmc.org>.
- [22] W3C; *Web Services Description Language (WSDL) 1.1*; <http://www.w3.org/TR/wsdl>.
- [23] *CrossFlow Project*; <http://www.crossflow.org>; CrossFlow Consortium.

Open Petri Nets as Semantic Model for Workflow Integration^{*}

Reiko Heckel

University of Paderborn, Dept. of Mathematics and Computer Science
D-33095 Paderborn, Germany
reiko@upb.de

Abstract. The integration of workflows of different enterprises requires to resolve conceptual inconsistencies which are best dealt with using a high-level modelling language. Languages like the UML provide dedicated notations for workflow modelling (e.g., by means of activity diagrams), but due to the lack of formal semantics, only little support for detecting and resolving inconsistencies can be provided.

We propose open Petri nets as a suitable semantic model for workflows spanning different enterprises, and discuss the use of this model to formalise a notion of consistent evolution.

1 Introduction

One of the major challenges for software engineering today is the integration of computerised business processes across organisational boundaries over the world wide web. Besides the integration of computational platforms and data formats, often conflicting conceptual models and workflows have to be reconciled and related. Thereby, technical, semantic, and administrative inconsistencies have to be resolved. This cannot be done by computer scientists alone but requires negotiation between domain experts, users, and managers of the different organisations.

For this purpose, a language is needed which is *abstract*, so that it does not force to deal with low-level technical detail, *intuitive*, so that it is understandable to non-computer scientists, and *precise* in order to allow to detect inconsistencies between the views of different enterprises and to support the implementation of the integration.

Mainstream software processes, like the Unified Process [13], propose UML class diagrams for modelling the integration of conceptual data models and activity diagrams for specifying workflows [18]. Then, these models are used as requirements specification to drive the subsequent development and integration of applications through analysis, design, and implementation activities. The alternative approach of workflow management consists in the use of the workflow specification as an executable model interpreted by a workflow management system.

Both interpretations of workflow models, as requirements specifications or executable models, benefit from a formal interpretation of the semantics of models, e.g.,

^{*} Research partially supported by the ESPRIT Working Group APPLIGRAPH (Applications of Graph Transformation).

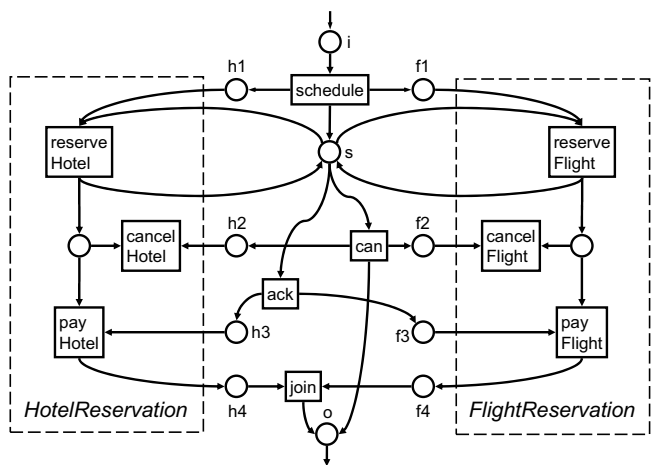


Fig. 1. Workflows of hotel and flight reservation coordinated through travel agency

for analysis purposes. Rather than using “ad-hoc mathematics”, such a formalisation should be based on an established language which provides a semantic framework, e.g., for comparison of different approaches.

For the workflow domain, Petri nets have been shown to be able to capture many of the typical patterns of behaviour and interaction [2, 3, 5]. Processes of Petri nets [12] provide a formal semantics which is particularly well-suited for the workflow domain, for two reasons.

- 1. It is *truly concurrent*. That means, it distinguishes between potentially simultaneous and interleaved execution of activities—a fact which is of high relevance for workflow models as witnessed, for example, by two different workflow patterns for *interleaving* and *concurrency* in [5].
- 2. It is *visual*. That means, the history of a case (the execution of a workflow) can be shown in a similar notation that is also used for the specification. (Observe that concurrent executions (see above) are not faithfully represented by a linear simulation of the model, which is only able to show one particular trace at a time.)

As an example for a Petri net modelling a workflow consider Fig. 1. It shows the planning of a trip by a travel agency. This includes the reservation of a flight and the booking of a room, two workflows which are performed by external organisations like an airline and a hotel. The travel agency is responsible for coordinating these workflows on behalf of the traveller.

To support the integration of workflows across organisational boundaries, it is important that each organisation has a local view of the joint workflow (for integration with other local activities) and that a global view exists which contains all activities relevant for the coordinator. Moreover, the local and global views should be compatible, that is, the behaviour of each local view should reflect the possible influence of the behaviour

of the global view and, vice versa, the global behaviour should be obtained by a suitable combination of local behaviours. Such a *compositional semantics* shall allow the modeller to understand, execute, and change the local workflow of one enterprise without complete knowledge of the internals of the workflows of other enterprises [4, 1, 6].

Unfortunately, classical Petri net processes are not compositional in this sense, and existing compositional semantics for Petri nets do not consider processes, but other mathematical models like partial orders or traces [9, 17, 14, 15, 7]. Therefore, in [8] open Petri nets have been proposed as a compositional approach to Petri nets equipped with a semantics based on deterministic processes satisfying the above requirements.

In this paper, we will give a light-weight introduction to the theory of open nets based on familiar set-theoretic notions (rather than categorical constructions). Below, in Section 2, we demonstrate the modelling of workflows across organisational boundaries with open nets and discuss their semantic interpretation in terms of deterministic open processes. Then, in Section 3, we discuss the problem of the decentralised evolution of workflow models and draw connections with the theory of graph transformation as one possibility to specify such evolutions in a rule-based way. Finally, we conclude with a discussion of related and future work.

2 A Compositional Approach to Petri Nets

An *open net* is an ordinary P/T Petri net with a distinguished set of places which represent the interface of the net towards the environment. As a consequence of the (hidden, implicit) interaction between the net and the environment, some tokens can freely appear and disappear at the open places. To distinguish these cases, an open place can be either an *input* or an *output* place (or both), where the environment can put or remove tokens, respectively.

2.1 Open Petri Nets

We assume the standard definition [20] of a place-transition Petri net N defined as a bipartite multigraph over disjoint sets of places S and transitions T . By $\sigma, \tau : T \rightarrow S^\oplus$ we denote the functions assigning to each transition its pre- and post-multiset, where $X^\oplus$ is the set of finite multisets over a set X . The union of multisets A and B is denoted by $A \oplus B$. Thus, a net is at the same time a hypergraph with places as vertices and transitions as hyperedges [16]. Sometimes we will find it convenient to use this hypergraph presentation. The powerset over X shall be denoted by 2^X .

Given a place $s \in S$, the pre- and post-set of s are defined by $\bullet s = \{t \in T \mid s \in t^\bullet\}$ and $s^\bullet = \{t \in T \mid s \in t^\bullet\}$. A net N_0 is a *subnet* of N_1 , denoted by $N_0 \hookrightarrow N_1$ if $T_0 \subseteq T_1, S_0 \subseteq S_1$ such that source and target functions $\sigma_0 = \sigma_1|_{T_0}, \tau_0 = \tau_1|_{T_0}$ are defined by restriction of the corresponding functions of N_1 to the smaller set of transitions. That means, the connections of a given transition in N_0 are exactly the same as in N_1 .

Definition 1 (open net). An open net is a pair $Z = (N_Z, O_Z)$, where

- $N_Z = (S_Z, T_Z, \sigma_Z, \tau_Z)$ is an ordinary P/T Petri net and
- $O_Z = (O_Z^+, O_Z^-) \in 2^{S_Z} \times 2^{S_Z}$ are the input and output open places of the net.

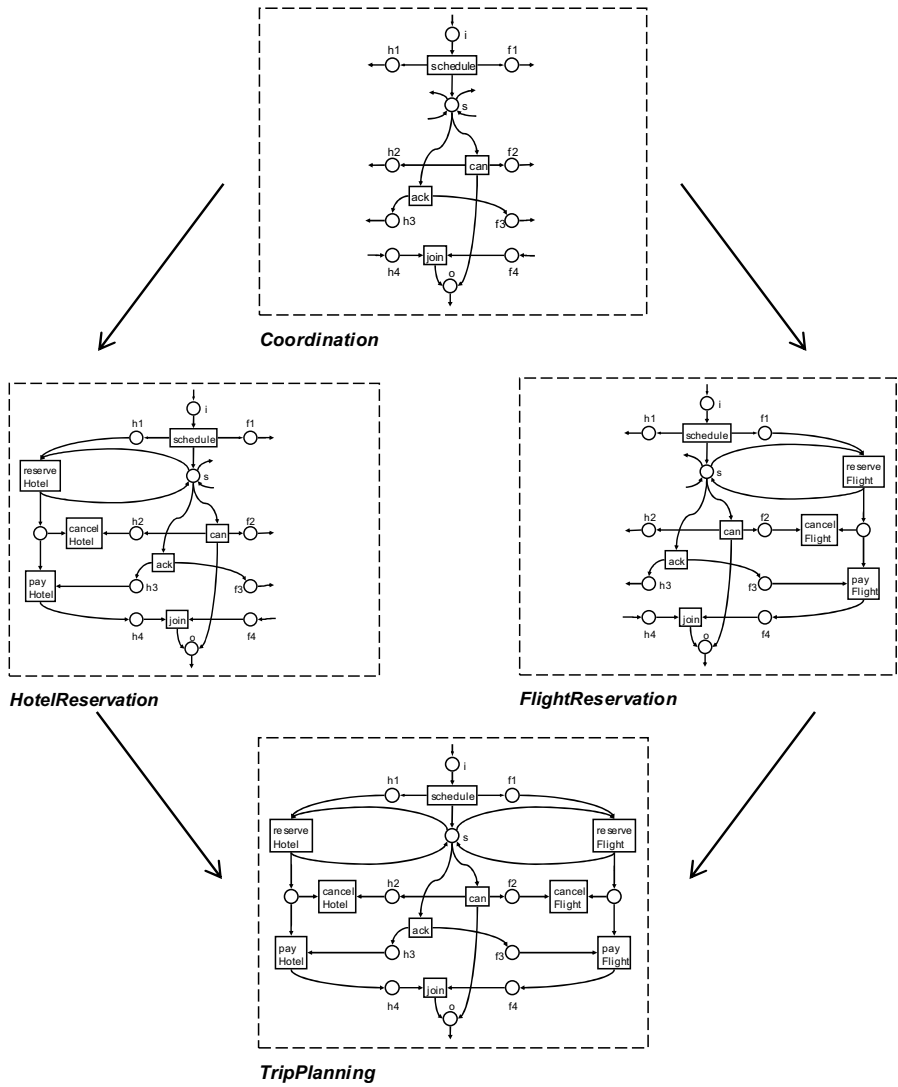


Fig. 2. Open nets and their composition

Observe that the sets O_Z^+ and O_Z^- are not necessarily disjoint, hence a place can be both an input and an output open place at the same time.

The open nets for hotel and flight reservation workflows of Fig. 1, including the coordinating workflow of the travel agency, are shown in the middle of Fig. 2. Ingoing and outgoing arcs without source or target designate the input and output places, re-

spectively. The travel agency workflow in isolation is represented by the open net in the top of the figure.

The notion of enabledness for a transition (or multiset of transitions) of an open net is the usual one, but, besides the changes produced by the firing of transitions, we represent interactions with the environment by spontaneous actions producing/consuming tokens in input/output places of the net. The actions of the environment which produce and consume tokens in an open place s are denoted by $+_s$ and $-_s$, respectively. Thus, if Z is an open net, a (sequential) *firing step* can be either

- (i) the firing of a transition, i.e., $m \oplus \bullet t \rightarrow m \oplus t \bullet$, with $m \in S_Z^\oplus$, $t \in T_Z$;
- (ii) the creation of a token by the environment, i.e., $m [+_s] m \oplus s$, with $s \in O_Z^+$, $m \in S_Z^\oplus$;
- (iii) the deletion of a token by the environment, i.e., $m \oplus s [-_s] m$, with $m \in S_Z^\oplus$, $s \in O_Z^-$.

Embeddings of open nets are based on the notion of subnet. They formalise the idea of an “insertion” of a net into a large context where part of the unknown environment gets more specified. To capture this idea, we have to know for an inclusion of Petri nets $N_0 \hookrightarrow N_1$ those places where the net N_1 interferes with the context provided by N_0 . Let $\text{in}(N_0 \hookrightarrow N_1) \subseteq S_0$ be the set of all places s_0 of N_0 such that there exists a transition $t_1 \in T_1 \setminus T_0$ with $s_0 \in \tau(t_1)$. These are the places where context transitions may drop tokens, i.e., potential input places. Dually, $\text{out}(N_1 \hookrightarrow N_2)$ is the set of all places s_0 of N_0 such that there exists $t_1 \in T_1 \setminus T_0$ with $s_0 \in \sigma(t_1)$. These are potential output places where context transitions may remove tokens.

Definition 2 (open net embedding). An open net embedding $Z_1 \hookrightarrow Z_2$ is a Petri net inclusion on the underlying nets $N_0 \hookrightarrow N_1$ such that

- (i) $O_1^+ \cup \text{in}(N_0 \hookrightarrow N_1) \subseteq O_0^+$ and
- (ii) $O_1^- \cup \text{out}(N_0 \hookrightarrow N_1) \subseteq O_0^-$.

Conditions (i) and (ii) require that open places are reflected and hence that places which are “internal” in Z_0 cannot be promoted to open places in Z_1 . Furthermore, the context in which Z_0 is inserted can interact with Z_0 only through open places.

As an example of open net embedding, consider the embedding of net *Coordination* into net *FlightReservation* of Fig. 2. Observe that the constraints characterising open nets embeddings have an intuitive graphical interpretation:

- The connections of transitions to their pre-set and post-set have to be preserved. New connections cannot be added;
- In the larger net, a new arc may be attached to a place only if the corresponding place of the subnet has a dangling arc in the same direction. Dangling arcs may be removed, but cannot be added in the larger net. For instance, without the outgoing dangling arc from place $h1$ in net *Coordination* (i.e., if place $h1$ were not output open) the embedding into *HotelReservation* in Fig. 2 would have not been a legal open net embedding.

Open net embeddings form a special case of open net morphisms [8]. In particular, they are closed under composition, and they induce a projection of the behaviour of the larger net onto the smaller one. This fact is made precise in the next subsection in terms of deterministic processes of open nets.

2.2 Deterministic Processes of Open Nets

A deterministic process represents a unique concurrent computation of a Petri net [12]. In analogy to ordinary deterministic processes, a deterministic process of an open net Z is given by a Petri net K , called occurrence net and satisfying suitable acyclicity and conflict freeness requirements, together with a mapping $\pi : K \rightarrow Z$ to the original net Z . The idea is that each place of the occurrence net K represents a token occurrence, and each transition of K represents a firing of a transition in Z . Therefore, a place may be in the post-set of at most one transition, i.e., the firing who generated the corresponding token. This is called *absence of backward conflicts*. Dually, *absence of forward conflicts* means that each place is in the pre-set of at most one transition. Otherwise this place would represent a branching point which would violate the idea of a deterministic computation. Formally, an open occurrence net is an open net K such that N_K is a deterministic occurrence net [12] which satisfies the following additional conditions for open places.

The open places in K are intended to represent tokens which are produced/consumed by the environment in the considered computation. Consequently, every input open place is required to have an empty pre-set, i.e., to be minimal with respect to the causal order. In fact, an input open place in the post-set of some transition would correspond to a kind of generalised backward conflict: a token on this place could be generated in two different ways and this would prevent us to interpret the place as a token occurrence.

Analogously, each output open places must be maximal with respect to the causal order, i.e., an output open place cannot be in the pre-set of any transition. In fact, an output open place s which is in the pre-set of a transition t represents a token occurrence which can be consumed either by the environment or by transition t .

The class of all deterministic processes over an open net Z is denoted by $\mathbf{Proc}(Z)$. In order to simplify the notation, in the following we will sometimes identify a process $\langle K, \pi \rangle$ with the mapping π .

Processes for the open nets in Fig. 2 are shown in Fig. 3. The mappings back to the original nets in Fig. 2 are implicitly represented by the labeling of places and transitions of the occurrence nets. Observe that the requirement that each input place is minimal and each output place is maximal w.r.t. to the causal order of the process has a natural graphical interpretation: the absence of backward and forward conflicts extends to dangling arcs, i.e., in total, each place may have at most one ingoing and one outgoing arc.

Let $Z_0 \hookrightarrow Z_1$ be an open net embedding. As mentioned before each process of Z_1 can be projected to Z_0 by considering only that part of the computation which is visible in the smaller net. This *projection* π_1 to Z_0 yields a process $\pi_1|_{Z_0} = \pi_0 : K_0 \rightarrow Z_0$ whose open places are determined by taking the smallest sets of open places which makes the inclusion $K_0 \hookrightarrow K_1$ an open net embedding, namely

- $O_{K_0}^+ = O_{K_1}^+ \cup \text{in}(K_0 \hookrightarrow K_1)$
- $O_{K_0}^- = O_{K_1}^- \cup \text{out}(K_0 \hookrightarrow K_1)$

The embedding of Coordination into HotelReservation in Fig. 2 induces a projection of open net processes. For instance, the left part of Fig. 3 shows a process of HotelReservation. Its projection to the net Coordination is shown in the left and upper part of the same figure. Notice how transition `cancelHotel`, which consumes a token from place `h2`,

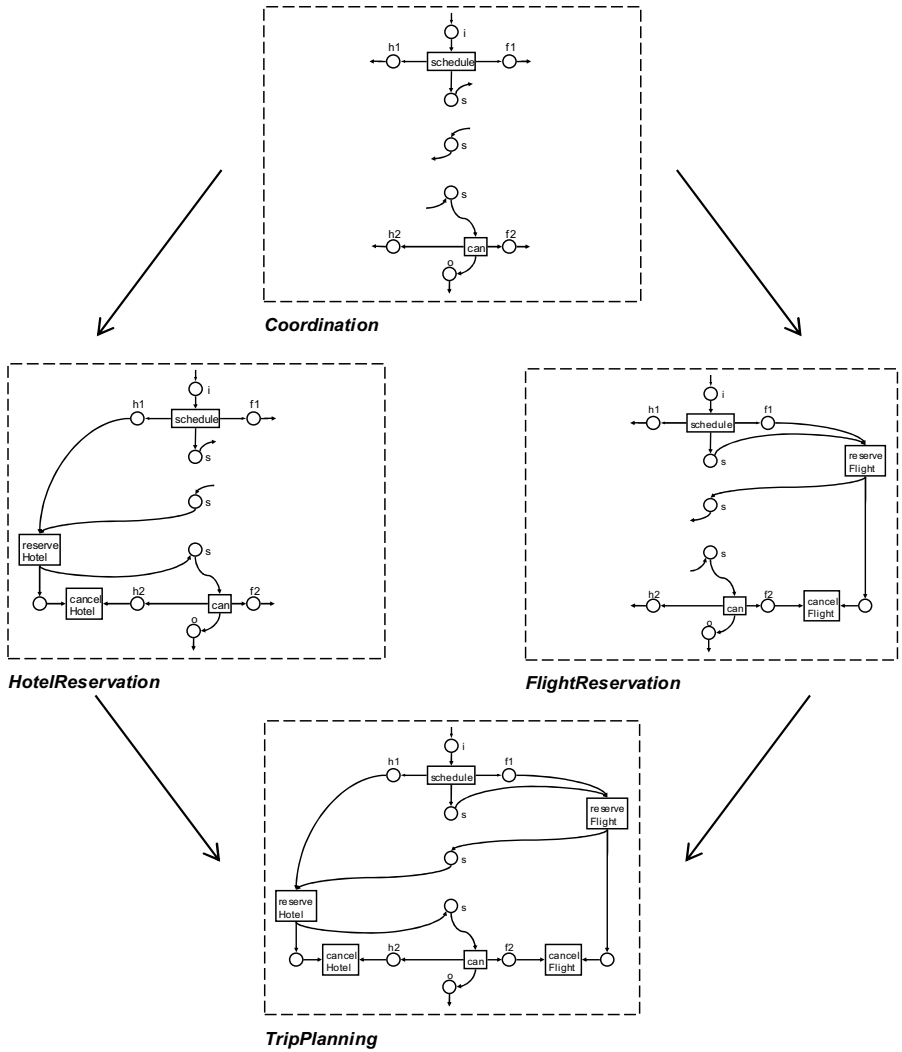


Fig. 3. Open deterministic processes and their amalgamation

is replaced in the projection by a dangling output arc: an internal action in the larger net becomes an interaction with the environment in the smaller one.

2.3 Composing Open Nets

In this section we introduce a basic mechanism for composing open nets. It is based on the gluing of two nets sharing a common subnet that represents the intended overlap.

Consider, for instance, the open nets for the local workflows *HotelReservation* and *FlightReservation* in the middle of Fig. 2. The two nets share the subnet *Coordination* depicted in the top of the same figure, which represents the “glue” between the two components. The net *TripPlanning* resulting from the composition of *HotelReservation* and *FlightReservation* over the shared subnet *Coordination* is shown in the bottom part of Fig. 2. This composition is only defined if the embeddings of the components into the resulting net satisfy the constraints of open net embedding. For example, if we remove the outgoing dangling arc of the place *h1* in the net *FlightReservation*, the embedding of *Coordination* into *Airline* would still represent a legal open net embedding. However, in this case the embedding of *FlightReservation* into *TripPlanning* would become illegal because of the new arc from *h1* to *reserveHotel* (cf. condition (i) of Definition 2).

Next we formalize the conditions under which the construction is defined and the way the open places for the composed net are determined.

Definition 3 (composable nets). *Two open nets Z_1 and Z_2 are called composable if their intersection $Z_0 = Z_1 \cap Z_2$ is defined, such that $Z_1 \hookleftarrow Z_0 \hookrightarrow Z_2$ are open net embeddings and*

1. $\text{in}(Z_0 \hookrightarrow Z_1) \subseteq O_{Z_2}^+$ and $\text{out}(Z_0 \hookrightarrow Z_1) \subseteq O_{Z_2}^-$;
2. $\text{in}(Z_0 \hookrightarrow Z_2) \subseteq O_{Z_1}^+$ and $\text{out}(Z_0 \hookrightarrow Z_2) \subseteq O_{Z_1}^-$.

Conditions 1 and 2 mean that places which are used as interfaces by Z_1 , namely the places $\text{in}(Z_0 \hookrightarrow Z_1)$ and $\text{out}(Z_0 \hookrightarrow Z_1)$, are input and output open places in Z_2 , and symmetrically. If these condition are satisfied the composition of Z_1 and Z_2 is defined as follows.

[composition of open nets] Let Z_1, Z_2 be composable open nets and $Z_1 \hookleftarrow Z_0 \hookrightarrow Z_2$ be the corresponding embeddings. Then, the composition of Z_1 and Z_2 , denoted by $Z_3 = Z_1 \oplus Z_2$, is computed by forming the union of the underlying nets of Z_1 and Z_2 , and then taking as open places, for $x \in \{+, -\}$,

$$O_{Z_3}^x = \{s_3 \in S_3 \mid s_3 \in S_1 \Rightarrow s_3 \in O_{Z_1}^x \wedge s_3 \in S_2 \Rightarrow s_3 \in O_{Z_2}^x\}.$$

Thus, the open places of Z_3 are determined in the most permissible way that is consistent with the constraints imposed by Z_1 and Z_2 (i.e., such that the conditions for open net embeddings are satisfied).

In [8] this construction is formalised as a pushout of open net morphisms. Such axiomatic characterisation of the idea of a union allows to apply this concept to more sophisticated structures (like nets) while abstracting from “implementation issues” like name handling. In fact, category-theoretic constructions, like the pushout, are only defined up to isomorphism (consistent renaming) of structures. In our presentation, which is set-theoretic in flavour, we implicitly assume renaming of places and transitions whenever appropriate, relying on [8] for the consistency of this assumption.

2.4 Amalgamating Deterministic Processes

For a composition of nets like in Fig. 2, we would like to establish a relationship between the behaviours of the involved nets: The processes of the composed net should

be constructed by merging matching pairs of local processes. Given two deterministic processes π_1 of Z_1 and π_2 of Z_2 which *agree* on Z_0 in the sense that they could be seen as two local views of the same global computation, we construct a deterministic process π_3 of Z_3 by *amalgamating* π_1 and π_2 . Vice versa, each deterministic process π_3 of Z_3 shall be projected to deterministic processes π_1 and π_2 of Z_1 and Z_2 , respectively, which can be amalgamated to produce π_3 again.

Since processes are essentially open nets (subject to some additional constraints and equipped with a mapping to another net), the amalgamation of processes is based on the composition of these occurrence nets. [8] identifies a condition which ensures that the composition of occurrence nets is defined and produces a net in the same class: Two open nets K_1, K_2 are *consistent* if they are composable in the sense of Def. 3 such that the composed net $K_1 \oplus K_2$ is an open deterministic occurrence net (that is free of conflicts and cycles).

Given a composed net $Z_3 = Z_1 \oplus Z_2$ of open nets Z_1 and Z_2 over Z_0 , two deterministic processes $\pi_1 : K_1 \rightarrow Z_1$ and $\pi_2 : K_2 \rightarrow Z_2$ *agree* on Z_0 if $\pi_1|_{Z_0} = \pi_2|_{Z_0} = \langle K_0, \pi_0 \rangle$, that is, their projections to Z_0 coincide, and the corresponding occurrence nets K_1 and K_2 are consistent.

In this case, the two processes can be *amalgamated* to a process $\pi_3 = \pi_1 \oplus \pi_2 : K_3 \rightarrow Z_3$ given by the composed occurrence net $K_3 = K_1 \oplus K_2$ together with the mapping $\pi_3 = \pi_1 \cup \pi_2$ obtained as the union of the given process mappings. For processes P_1 over Z_1 and P_2 over Z_2 , we denote by $P_1 \oplus P_2$ the set $\{\pi_1 \oplus \pi_2 \mid \pi_i \in P_i \text{ and } \pi_1, \pi_2 \text{ agree on } Z_0\}$.

In [8] it is shown that each deterministic process π_3 of the composed net Z_3 can be decomposed into processes π_1 and π_2 over Z_1 and Z_2 , respectively, such that π_1 and π_2 agree on Z_0 . Moreover, amalgamation and decomposition of processes are inverse to each other, up to isomorphism of processes. That means, the set of processes of Z_3 can be computed from the processes of Z_1 and Z_2 .

Theorem 4 (amalgamation). *Let Z_1 and Z_2 be composable open nets with intersection Z_0 . Then, up to isomorphism of processes, $\mathbf{Proc}(Z_1 \oplus Z_2) = \mathbf{Proc}(Z_1) \oplus \mathbf{Proc}(Z_2)$.*

This construction, called the amalgamation theorem in [8] is exemplified in Fig. 3. Two processes for the component nets HotelReservation and FlightReservation which agree on the shared subnet Coordination can be amalgamated to produce a process for the composed net TripPlanning. Vice versa, each process of the net TripPlanning can be reconstructed as amalgamation of compatible processes of the component nets.

3 Evolution of Workflows Based on Open Nets

One fundamental problem of workflow modelling and management is the handling of evolution of workflows [4]. In particular, if the evolution is not centralised, as it is typical with workflows where more than one organisation is involved, we cannot assume that a single authority exists which could ensure consistency after a change. Following [1], in this section we approach this problem by requiring local evolution steps to be performed in such a way that the coordinating workflow does not notice a change in behaviour. This idea is formalised by means of a notion of equivalence of open nets over a common view.

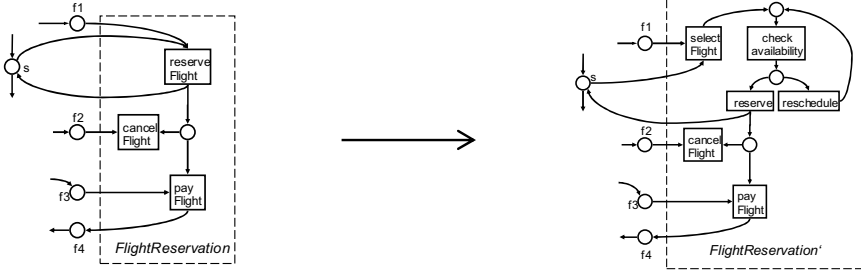


Fig. 4. Replacing a local workflow by an equivalent one

3.1 Simulation and Equivalence of Open Nets

A workflow can be replaced or implemented by another one if, in the given context, their behaviours cannot be distinguished from each other. Modelling workflows in context with open nets, this idea can be formalised as a notion of semantic equivalence of two nets Z_1 and Z_2 from the point of view of a shared subnet Z_0 representing the common context.

Definition 5 (simulation and equivalence). Assume open nets Z_1 and Z_2 , and let $Z_0 = Z_1 \cap Z_2$. Then, we say that Z_1 is simulated by Z_2 (over Z_0) if, for each process π_1 of Z_1 with projection $\pi_1|_{Z_0} = \pi'_1$, there exists a process π_2 of Z_2 with projection $\pi_2|_{Z_0} = \pi'_2$ such that p'_1 and p'_2 are isomorphic. The two nets are called equivalent (over Z_0) if Z_1 is simulated by Z_2 and vice versa.

Fig. 4 provides an example where the net *FlightReservation* which represents an abstract view of the workflow, is replaced by a slightly more detailed version which allows iterations of flight selection and trip rescheduling, if no suitable flight is available at the time scheduled. Both nets are equivalent over the shared context, a fragment of the Coordination net.

3.2 Consistent Evolution

The notion of equivalence of nets allows to replace component nets with equivalent ones in a given context. Using our composition operator, this replacement shall now be lifted to the global workflow. This requires that, after the evolution, the new local components are still composable.

Consider once again our trip planning example. The refinement of the *FlightReservation* net can be embedded into the global net *TripPlanning* leading to the net *TripPlanning'* in Fig. 5. Composability of the new component net *FlightReservation'* with the other local component is ensured because

$$\text{in}(\text{Coordination} \hookrightarrow \text{FlightReservation}') \subseteq \text{in}(\text{Coordination} \hookrightarrow \text{FlightReservation})$$

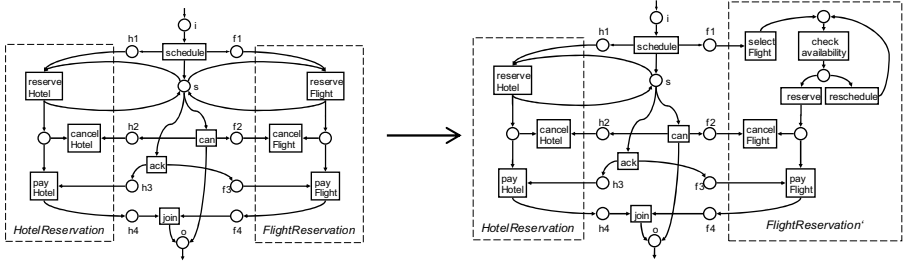


Fig. 5. Evolution of global workflow

and analogously for $\text{out}(\text{Coordination} \hookrightarrow \text{FlightReservation}')$. That means, the new component does not require any “access rights” beyond those already granted to the old one.

As a result, evolution steps can be performed locally as long as the local workflows before and after the evolution are equivalent from the point of view of the context they are embedded in and the required input and output places are not extended. The interaction of the new components among each other (and with old components) follows the same protocol like before the evolution.

3.3 Towards Rule-Based Evolution

The replacement of the open net *FlightReservation* by *FlightReservation'* within the Trip-Planning workflow in Fig. 5 could be seen as the application of a rewrite rule like in Fig. 4. Rules for transforming Petri nets have been considered, for example, in [19] in the framework of high-level replacement systems [10], a categorical generalisation of the double-pushout (DPO) approach to graph rewriting which includes, beside other structures, hypergraphs.

A transformation rule in the DPO approach (cf. Fig. 6 on the left) is a triple of graphs $L \hookrightarrow K \hookrightarrow R$ where L represents a pattern for the structure to be rewritten, R is the replacement for L , and $L = L \cap R$ specifies the intended intersection. A transformation consists of three steps, the matching of the left-hand side L with the structure G to be transformed, the removal of the image of $L \setminus R$ in G and the gluing of the resulting structure D with R over the shared structure K .

Think of these structures as open nets and of the mappings involved as open net embeddings (suitably generalised to allow renaming of places and transitions). Then, the gluing of D and R over K corresponds to a composition of open nets, while the removal of $L \setminus R$ represents a decomposition of G into L and D , so that $L \oplus_K D = G$.

This idea of *deletion as inverse of gluing* is classical for the DPO approach to graph transformation [11]. It requires a so-called gluing condition which ensures that the given structure G can indeed be decomposed into L and D such that their gluing over K leads again to G . In the case of open nets (which are essentially hypergraphs) this would mean that a place (vertex) may only be removed if it is not connected to a transition (hyperedge), or if all connected transitions are removed as well. Moreover, on the right-hand side, we have to make sure that the composition of D and R over K exists, that

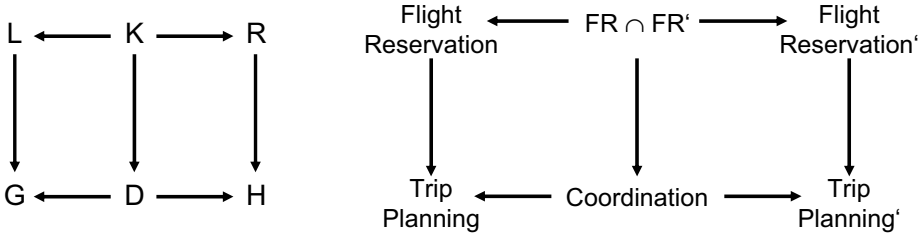


Fig. 6. DPO Transformation of open nets

is, D and R should be composable in the sense of Def. 3. This can be done by requiring, like in the previous subsection, that the required input and output places of R are subsets of those of L , so that no extra permissions are required.

In this way, the extension of the local evolution from `FlightReservation` by `FlightReservation'` to the global `TripPlanning` workflow can be seen as a DPO transformation. Now, the most interesting question is the equivalence of the given and derived nets assuming that the left- and right-hand sides of the rule are equivalent. More formally, given a transformation like in Fig. 6, we would like that, if L and R are equivalent over K , G and H are equivalent over D . Under which side conditions this is a case is a question of future work.

4 Conclusion

In this paper, we have presented an approach to the modelling of workflows across organisational boundaries based on open Petri nets. Some of the notions, in particular in the previous section, are motivated by work on workflow integration based on inheritance of workflow specifications [4, 1]. The approach uses branching bisimilarity of workflow nets to formalise the equivalence of two nets from the abstract point of view of a coordinator. It also defines transformation rules for constructing equivalent nets representing subclasses.

Branching bisimilarity provides an established model of concurrency including a variety of theoretic concepts and results, e.g., for compositionality and analysis. It may be used to compare and relate a Petri net-based workflow language with other formal methods, like process algebras.

However, in contrast with deterministic processes, branching bisimilarity does not provide a truly-concurrent semantics. That means, it is not able to distinguish between the interleaved execution of `reserveHotel` and `reserveFlight` on the one hand and the concurrent execution of `payHotel` and `payFlight` on the other hand. As stated in the introduction, we believe that the visual semantics provided by processes provides means for the visualisation of cases which is also accessible to non-experts. Finally, we prefer the visual specification of evolution rules, following concepts of graph transformation, over the set-based, textual description in [4].

References

- [1] W. M. P. van der Aalst and K. Anyanwu. Inheritance of interorganizational workflows to enable business-to-business E-commerce. In *Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC'99)*, pages 141–157. Nashville, Tennessee, October 1999. 283, 289, 292
- [2] W. M. P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998. 282
- [3] W. M. P. van der Aalst. Interorganizational workflows: An approach based on message sequence charts and Petri nets. *Systems Analysis - Modelling - Simulation*, 34(3):335–367, 1999. 282
- [4] W. M. P. van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Computing Science Reports 99/06*, Eindhoven University of Technology, 1999. 283, 289, 292
- [5] W. M. P. van der Aalst, A. P. Barros, A. H. M. ter Hofstede, and B. Kiepuszewski. Advanced workflow patterns. In *Conference on Cooperative Information Systems*, pages 18–29, 2000. 282
- [6] W. M. P. van der Aalst and M. Weske. The P2P approach to interorganizational workflows. In K. R. Dittrich, A. Geppert, and M. C. Norrie, editors, *Proc. 13th Intl. Conference on Advanced Information Systems Engineering (CAiSE 2001)*, Interlaken, Switzerland, June 2001, volume 2068 of *LNCS*, pages 140–156. Springer-Verlag, 2001. 283
- [7] T. Basten. *In terms of nets*. PhD thesis, Eindhoven University of Technology, The Netherlands, 2000. 283
- [8] P. Baldan, A. Corradini, H. Ehrig, and R. Heckel. Compositional modeling of reactive systems using open nets. In *Proc. CONCUR 2001, Aarhus, Denmark*, volume 2154 of *LNCS*, pages 502–518. Springer-Verlag, August 2001. Full version available at <ftp://ftp.di.unipi.it/pub/techreports/TR-01-21.ps.z>. 283, 285, 288, 289
- [9] E. Best, R. Devillers, and J. Hall. The Box Calculus: a new causal algebra with multi-label communication. In *Advances in Petri Nets*, pages 21–69. LNCS, 1992. 609. 283
- [10] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991. 291
- [11] H. Ehrig, M. Pfender, and H. J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973. 291
- [12] U. Golz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57:125–147, 1983. 282, 286
- [13] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999. 281
- [14] E. Kindler. A compositional partial order semantics for Petri net components. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets*, volume 1248 of *LNCS*, pages 235–252. Springer-Verlag, 1997. 283
- [15] E. Kindler, A. Martens, and W. Reisig. Inter-operability of workflow applications: Local criteria for global soundness. In W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *LNCS*, pages 235–253. Springer-Verlag, 2000. 283
- [16] J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, 1990. 283
- [17] M. Nielsen, L. Priese, and V. Sassone. Characterizing Behavioural Congruences for Petri Nets. In *Proc. CONCUR'95, LNCS 962*, pages 175–189. Springer-Verlag, 1995. 283

- [18] Object Management Group. UML specification version 1.4, 2001.
<http://www.celigent.com/omg/umlrtf/>. 281
- [19] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic high-level net transformation systems. *MSCS*, 2:217–256, 1995. 291
- [20] W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1985. 283

Team Automata for CSCW

– A Survey –

Jetty Kleijn

LIACS, Leiden University, P.O.Box 9512
NL-2300 RA Leiden, The Netherlands
`kleijn@liacs.nl`

Abstract. Team automata have been proposed as a formal framework for modelling both the conceptual and the architectural level of groupware systems. They are defined in terms of component automata (labelled transition systems) together with an interconnection mechanism which is based on shared actions (synchronizations). Components can be combined in different fashions depending on which actions are to be shared, and when. This set-up makes it possible to distinguish between different types of synchronizations and allows to represent hierarchical systems. A survey is presented including a brief comparison with some related models.

1 Introduction

Computer Supported Cooperative Work (CSCW for short) is concerned with understanding how people work together, and ways in which technology can assist. By the nature of the field this technology mostly consists of multi-user software, so-called groupware. CSCW is a rapidly developing field which benefits from the evolution and formulation of basic ideas and intuitions. Our understanding of such conceptual ideas is enhanced when they are accompanied by formal versions which require an exact formulation of basic features. Such formalizations make analysis possible and properties can be rigourously proved. With the increase of the complexity of groupware systems, abstractions tend to be especially useful. Thus, CSCW has a need for developing a precise and consistent terminology. Moreover, at the architectural level, CSCW needs a rigorous framework to describe, compare and contrast groupware systems.

Team automata have been introduced in [11] explicitly for the specification and analysis of CSCW phenomena and groupware systems. The model is inspired by and forms an extension of Input/Output automata (see, e.g., [19]). The set-up of the team automata model makes it possible to clarify and capture precisely notions related to coordination and collaboration in distributed systems. Team automata consist of an abstract specification of the components of a system and different interconnection mechanisms can be described based upon the concept of “shared action”. Components can be combined in a loose or more tight fashion depending on which actions are to be shared, and when. Such aggregates

of components can then in turn be used as components in a higher-level team. Team automata thus fit nicely with the needs and the philosophy of groupware systems. Moreover, thanks to the formal automata theoretic set-up, results and methodologies from automata theory are available for team automata. While inappropriate for capturing aspects of group activity such as social aspects and informal unstructured activity, the model has proved useful in various CSCW modelling areas. A spectrum from hardware components to protocols for interacting groups of people can be modelled by team automata.

This paper is a tutorial overview of research conducted in the last 5 years concerning a theory of team automata and their applications. In particular, it covers some of the main ideas of [5], further investigated in [4] and employed in [7], where access control mechanisms are considered in the context of the team automata model. The overview presented is a curtailed version of [6] and because of the restrictions on its size, the presentation is rather sketchy. The best reference for a detailed elaboration of the material presented here and for more results concerning team automata will be the forthcoming Ph.D. thesis [4] of Maurice ter Beek. The structure of the current paper is as follows. First we introduce team automata and discuss their possibilities to define hierarchies by iteration. Next we identify different types of synchronization and consider how to (uniquely) construct team automata satisfying certain synchronization constraints. Team automata are compared with some related models, in particular Input/Output automata, Vector Controlled Concurrent Systems, and labelled Petri nets. The conclusion of the paper reflects on the use of team automata in the design of CSCW systems. The paper as a whole explains rather than proves. Some basic definitions are provided in detail to avoid ambiguities. A few small examples are given intended to illuminate some of the technical details.

1.1 Some Notation

The following notations are used frequently. Let $\mathcal{I} \subseteq \mathbb{N}$ be a set of indices given by $\mathcal{I} = \{i_1, i_2, \dots\}$ with $i_j < i_l$ if $1 \leq j < l$. For sets V_i , $i \in \mathcal{I}$, we denote by $\prod_{i \in \mathcal{I}} V_i$ the Cartesian product $\{(v_{i_1}, v_{i_2}, \dots) \mid v_{i_j} \in V_{i_j}, \text{ for all } j \geq 1\}$. If $v_i \in V_i$, for all $i \in \mathcal{I}$, then $\prod_{i \in \mathcal{I}} v_i$ denotes the element $(v_{i_1}, v_{i_2}, \dots)$ of $\prod_{i \in \mathcal{I}} V_i$. If $\mathcal{I} = \emptyset$, then $\prod_{i \in \mathcal{I}} V_i = \emptyset$. In addition to the prefix notation $\prod_{i \in \mathcal{I}} V_i$ or $\prod_{i \in \mathcal{I}} v_i$ for a Cartesian product, we also use the infix notation $V_{i_1} \times V_{i_2} \times \dots$ or $v_{i_1} \times v_{i_2} \times \dots$, respectively. Let $A \subseteq \prod_{i \in \mathcal{I}} V_i$. Then, for $j \in \mathcal{I}$, $\text{proj}_j : A \rightarrow V_j$ is defined by $\text{proj}_j(a_{i_1}, a_{i_2}, \dots) = a_j$. For $J \subseteq \mathcal{I}$, $\text{proj}_J : A \rightarrow \prod_{i \in J} V_i$ is defined by $\text{proj}_J(a) = \prod_{i \in J} \text{proj}_i(a)$. We use $\text{proj}_i^{[2]}$ and $\text{proj}_J^{[2]}$ as shorthand notations for double projections, thus $\text{proj}_i^{[2]}(a, b) = (\text{proj}_i(a), \text{proj}_i(b))$ and $\text{proj}_J^{[2]}(a, b) = (\text{proj}_J(a), \text{proj}_J(b))$.

2 Team Automata

A team automaton consists of component automata, combined in a coordinated way such that they can perform shared actions. Within a team, component

automata can simultaneously participate in an action (i.e. synchronize on this action) or remain idle. The choice for a specific interconnection strategy (which components synchronize on what actions, and when) is based on what one wants to model, and this possibility to choose gives the team automata framework a high level of flexibility.

The basic concept underlying both team and component automata is a labelled transition system, which captures the idea of a system with states (configurations, possibly an infinite number of them), together with actions the executions of which lead to (non-deterministic) state changes as described by the labelled transitions in its transition relation. These transition systems come equipped with a set of initial states from which they can start their executions.

Definition 1. An initialized labelled transition system (or *ilts*) is a construct $\mathcal{A} = (Q, \Sigma, \delta, I)$, where Q is a set of states, possibly infinite, Σ an alphabet of actions, such that $Q \cap \Sigma = \emptyset$, $\delta \subseteq Q \times \Sigma \times Q$ a set of (labelled) transitions, and $I \subseteq Q$ a set of initial states.

A computation of $\mathcal{A}$ is a (finite) sequence $q_0 a_1 q_1 a_2 \cdots a_n q_n$ where $q_0 \in I$, $n \geq 0$, and $(q_i, a_{i+1}, q_{i+1}) \in \delta$, for all $0 \leq i < n$. $\square$

For an *ilts* $\mathcal{A} = (Q, \Sigma, \delta, I)$ and an action $a \in \Sigma$, we define $\delta_a = \{(q, q') \mid (q, a, q') \in \delta\}$ to be the set of *a-transitions* of $\mathcal{A}$; an *a-transition* $(q, q') \in \delta_a$ is called a *loop* (on a); and a is said to be *enabled* in state $q \in Q$ if $(q, q') \in \delta_a$ for some $q' \in Q$.

A component automaton is an initialized labelled transition system with a classification of its actions. The actions are divided into two main categories, one of which is subdivided into two more categories. A distinction is made between *internal* actions which have strictly local visibility and can thus not be used for communication with other components, and *external* actions which are observable by other components. The external actions can be used for communication between components and are divided into *input* actions and *output* actions. As formulated in [11]: "input actions are not under the local system's control and are caused by another non-local component, the output actions are under the system's control and are externally observable by other components, and internal actions are under the local system's control but are not externally observable". Thus when specifying a subsystem as a component automaton, a design issue that has to be considered is the role of the actions within that component in relation to the other components within the system as a whole (the team).

Definition 2. A component automaton is a construct

$$\mathcal{C} = (Q, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, I)$$

such that $(Q, \Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}, \delta, I)$ is an *ilts*, and Σ_{inp} , Σ_{out} and Σ_{int} are mutually disjoint alphabets called the input, output and internal alphabet of $\mathcal{C}$, respectively. $\square$

The computations of a component automaton are the computations of its underlying ilts. Given a computation one may choose to focus on certain details while filtering away others. In this way, records are made of computations and a certain notion of behaviour can be chosen. A standard behavioural notion obtained in this way is the *language* of the system, which is derived from the computations by deleting the states while preserving the occurrences of the actions. Given the different roles of actions one may also opt, e.g., to define “external” or “internal” behaviour. In this exposition we will mainly refer to computations and only occasionally say something about behaviours.

A team automaton consists of component automata which work together by means of synchronizations. Let us fix $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$, a collection of component automata $\mathcal{C}_i = (Q_i, (\Sigma_{i,inp}, \Sigma_{i,out}, \Sigma_{i,int}), \delta_i, I_i)$. For each $i \in \mathcal{I}$, we write Σ_i to denote the set of actions $\Sigma_{i,inp} \cup \Sigma_{i,out} \cup \Sigma_{i,int}$ of $\mathcal{C}_i$. Here $\mathcal{I}$ is a (possibly infinite) set of positive integers used to index the component automata involved. In order to compose a team automaton from $\mathcal{S}$, we require first of all that the internal actions of these components are private, i.e. uniquely associated to one component automaton. This is formally expressed by stating that, for all $i \in \mathcal{I}$, $\Sigma_{i,int} \cap \bigcup_{j \in \mathcal{I} \setminus \{i\}} \Sigma_j = \emptyset$. Thus no internal action of any component automaton from $\mathcal{S}$ may appear as an action in any of the other component automata in $\mathcal{S}$. If this is the case, then $\mathcal{S}$ is called a *composable system*. Note that every subset of a composable system is again a composable system. So, let us assume for the sequel that $\mathcal{S}$ is a composable system.

The state space of any team automaton $\mathcal{T}$ formed from $\mathcal{S}$ is the product $\prod_{i \in \mathcal{I}} Q_i$ of the state spaces of the component automata of $\mathcal{S}$, with the product $\prod_{i \in \mathcal{I}} I_i$ of their initial states forming the set of initial states of $\mathcal{T}$. A transition in the transition relation of $\mathcal{T}$ may be *any* combination of component transitions with the same label. When a team transition occurs, the state of the team automaton is changed according to the local state changes. It is not required that all components which have a given action in common participate in every team transition with that label nor that a team automaton has all possible combinations in its transition relation.

Definition 3. Let $a \in \bigcup_{i \in \mathcal{I}} \Sigma_i$. The complete transition space of a in $\mathcal{S}$ is denoted by $\Delta_a(\mathcal{S})$ and is defined as

$$\begin{aligned} \Delta_a(\mathcal{S}) = \{ & (q, q') \in \prod_{i \in \mathcal{I}} Q_i \times \prod_{i \in \mathcal{I}} Q_i \mid \\ & \exists j \in \mathcal{I} : \text{proj}_j^{[2]}(q, q') \in \delta_{j,a} \text{ and} \\ & \forall i \in \mathcal{I} : [\text{proj}_i^{[2]}(q, q') \in \delta_{i,a}] \text{ or } [\text{proj}_i(q) = \text{proj}_i(q')]\} . \end{aligned}$$

□

Thus $\Delta_a(\mathcal{S})$ consists of all possible combinations of a -transitions from components of $\mathcal{S}$, with all non-participating components remaining idle. It is an explicit requirement that at least one component is active, i.e. contributes an a -transition. The transitions in $\Delta_a(\mathcal{S})$ are referred to as *synchronizations* (on a).

$\Delta_a(\mathcal{S})$ is called the *complete* transition space of $\mathcal{S}$ because whenever a team automaton $\mathcal{T}$ is constructed from $\mathcal{S}$, then for each action a , a specific subset of $\Delta_a(\mathcal{S})$ has to be chosen. Consequently, the transition relation of a team automaton is defined by allowing certain synchronizations, while excluding others. For an internal action however, it is stipulated that each component retains all its possibilities to execute that action. Note that since $\mathcal{S}$ is a composable system, synchronizations on internal actions never involve more than one component. The alphabets of actions of any team automaton formed from $\mathcal{S}$ are uniquely determined by the alphabets of actions of the components of $\mathcal{S}$. The internal actions of the components will be the internal actions of the team automaton. Each action which is output for one or more of the component automata is an output action of the team automaton. Hence an action that is an output action of one component and also an input action of another component, is considered an output action of the team. The input actions of the component automata that do not occur at all as an output action of any of the component automata, are the input actions of the team. The reason for this construction of alphabets is again based on the intuitive idea of [11] that when relating an input action a of a component automaton to an output action a of another component, the input may be thought of as being caused by the output. On the other hand, output actions remain observable as output to other automata.

Definition 4. A team automaton over $\mathcal{S}$ is a construct

$$\mathcal{T} = (\prod_{i \in \mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i \in \mathcal{I}} I_i)$$

such that $\Sigma_{int} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,int}$, $\Sigma_{out} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$, $\Sigma_{inp} = (\bigcup_{i \in \mathcal{I}} \Sigma_{i,inp}) \setminus \Sigma_{out}$, $\delta \subseteq \prod_{i \in \mathcal{I}} Q_i \times \Sigma \times \prod_{i \in \mathcal{I}} Q_i$ where $\Sigma = \Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}$ and $\delta_a \subseteq \Delta_a(\mathcal{S})$ for all $a \in \Sigma$, and moreover $\delta_a = \Delta_a(\mathcal{S})$ if $a \in \Sigma_{int}$. $\square$

The definition of the alphabets of actions of a team automaton over $\mathcal{S}$ together with the composability of $\mathcal{S}$ guarantee consistency in the sense that in a team automaton every action appears exclusively as an input, output or internal action. As a consequence, every team automaton is again a component automaton, which in its turn could be used as a component in a higher-level team, an issue to which we return later. Since team automata are component automata, their behaviours can be described using computations.

Summarizing, we note that all team automata over a given composable system thus have the same set of states, the same alphabet of actions — including the distribution over input, output and internal actions — and the same set of initial states. They only differ by the choice of the transition relation, which is based on but not fixed by the transition relations of the component automata. Due to the freedom of choosing a δ_a for each external action a , a composable system does not uniquely define a single team automaton. Instead, a flexible framework is provided within which one can construct a variety of team automata over the composable system.

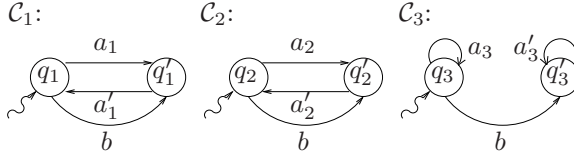


Fig. 1. Three component automata

When designing a system as a team automaton, one chooses a specific transition relation with a specific protocol in mind. Synchronizations between component automata may be excluded even if the action could occur at the current local states. This allows one, e.g., to forbid the execution of an action at certain combinations of local states. Also, predetermined synchronization constraints for the actions can be described, which lead to uniquely defined transition relations. An example is the rule to include, for all actions a , all and only those a -transitions in which all component automata participate that have a as one of their actions. This leaves no choice for the transition relation, and thus leads to a unique team automaton. Constructing the transition relation according to this particular strategy is very natural and often presupposed implicitly in the literature (see, e.g., [19], [13] and [8]). Note that the freedom of the team automata model to choose transition relations offers the flexibility to distinguish even the smallest nuances in the meaning of one's model. Leaving the set of transitions of a team automaton as a modelling choice is perhaps the most important feature of team automata.

Example 1. Let us now consider a very simple example, originally from [11]. Consider three component automata $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$. Each $\mathcal{C}_i$ has two states q_i, q'_i of which only q_i is an initial state (as indicated by a wavy arc in Figure 1). The actions a_i and a'_i are the internal actions of $\mathcal{C}_i$ and all a_i and a'_i are distinct symbols and different from b . Hence $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ is a composable system.

Action b is a common observable action of the three component automata. It is an output action of $\mathcal{C}_1$ and $\mathcal{C}_2$ and an input action of $\mathcal{C}_3$. The transition relations of $\mathcal{C}_1$, $\mathcal{C}_2$, and $\mathcal{C}_3$ are as depicted in Figure 1.

Each team automaton over $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ has output alphabet $\Sigma_{out} = \{b\}$, input alphabet $\Sigma_{inp} = \emptyset$, and internal alphabet $\Sigma_{int} = \{a_1, a'_1, a_2, a'_2, a_3, a'_3\}$ and is of the form $(\prod_{1 \leq i \leq 3} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \{(q_1, q_2, q_3)\})$ with only δ as a variable parameter. In fact, $\delta_a = \Delta_a(\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\})$, for each $a \in \Sigma_{int}$ and there is only a choice of which synchronizations on b to include in δ .

We consider some protocols which each define a team automaton over $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$.

First, all and only those synchronizations on b are allowed that involve all components. In this case only one b -transition is possible and we have $\delta_b = \{((q_1, q_2, q_3), (q'_1, q'_2, q'_3))\}$. From the structure of the component automaton $\mathcal{C}_3$ it follows immediately that in every computation of the resulting team automaton $\mathcal{T}_1$, action b is executed at most once.

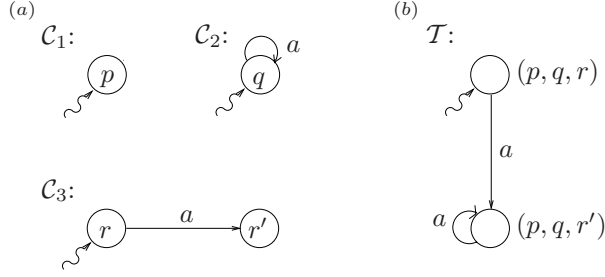


Fig. 2. Three component automata and a team automaton

Another possibility is the scenario that the automata which have b as output action always synchronize on b while b as a component's input action cannot be executed unless it is caused by an output action b of another component. This specification is ambiguous and hence has to be refined:

- if C_3 has to execute b whenever the other components do, we are back in the first situation and T_1 is the automaton being defined;
- if C_3 has to execute b whenever the other components do, provided it is ready to do so (enabled), then $\delta_b = \{((q_1, q_2, q_3), (q'_1, q'_2, q'_3)), ((q_1, q_2, q'_3), (q'_1, q'_2, q'_3))\}$, and the resulting team automaton T_2 has computations with any number of synchronizations on b , but C_3 is involved only in the first one;
- if C_3 has an option to execute b or not whenever the other components do, then $\delta_b = \{((q_1, q_2, q_3), (q'_1, q'_2, q'_3)), ((q_1, q_2, q'_3), (q'_1, q'_2, q'_3)), ((q_1, q_2, q_3), (q'_1, q'_2, q_3)), \}$, and in this case team automaton T_3 has computations with any number of synchronizations on b , with C_3 involved in at most one of them.

There are more strategies to define a team over $\{C_1, C_2, C_3\}$, e.g., the requirement that C_1 and C_2 should always synchronize on b could be dropped. $\square$

Once a team automaton has been defined, information on loops of its components may be lost. That is to say, given a synchronization on a of the team automaton one cannot in general distinguish whether or not a component automaton with a loop on a in its current local state takes part in this synchronization or remains idle.

Example 2. Consider the three component automata C_1 , C_2 and C_3 , as depicted in Figure 2(a). C_1 and C_2 each have only one state, p and q , respectively, which are their initial states. C_3 has two states, r and r' , of which r is its initial state. Both C_2 and C_3 have $\{a\}$ as their output alphabet, while all other alphabets of the three component automata are empty. Hence $\{C_1, C_2, C_3\}$ is a composable system.

Let T be the team automaton over $\{C_1, C_2, C_3\}$ with transition relation δ as depicted in Figure 2(b).

Consider the execution of the loop on a in T . Clearly C_1 does not participate as it cannot execute a at all. Also C_3 does not participate since a is not enabled

in r' (there is no a -transition leaving r'). However, since in every transition of a team automaton at least one component is required to participate, it must thus be the case that $\mathcal{C}_2$ does participate by executing its loop on a .

Secondly, consider the a -transition from (p, q, r) to (p, q, r') in $\mathcal{T}$. Clearly $\mathcal{C}_1$ is not involved. On the other hand, $\mathcal{C}_3$ is responsible for the local state change from r to r' and thus participates by executing a . But for $\mathcal{C}_2$ it is not clear whether it participates by executing its loop on a or remains idle during this execution of a by the team. $\square$

Even though information on the actual execution of loops by the components is lacking in the definition of a team automaton, in order to relate the computations of a team to those taking place in the components one can simply apply projections. This resolves the problem of loops by implicitly assuming that the presence of a component's loop in a transition of a team implies execution of that loop. This may be considered as a “maximal” interpretation of the components' participation.

Definition 5. Let $\mathcal{T} = (\prod_{i \in \mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i \in \mathcal{I}} I_i)$ be a team automaton over $\mathcal{S}$ and let $j \in \mathcal{I}$. The projection on $\mathcal{C}_j$ of a computation w of $\mathcal{T}$ is denoted by $\pi_{\mathcal{C}_j}(w)$ and is defined recursively as follows.

- (1) If $w = q$, for some $q \in \prod_{i \in \mathcal{I}} I_i$, then $\pi_{\mathcal{C}_j}(w) = \text{proj}_j(q)$.
- (2) If $w = w'qaq'$, for some computation $w'q$ of $\mathcal{T}$, $q, q' \in \prod_{i \in \mathcal{I}} Q_i$ and $(q, a, q') \in \delta$, then $\pi_{\mathcal{C}_j}(w) = \pi_{\mathcal{C}_j}(w'q)$ if $\text{proj}_j^{[2]}(q, q') \notin \delta_{j,a}$ and $\pi_{\mathcal{C}_j}(w) = \pi_{\mathcal{C}_j}(w'q)a\text{proj}_j(q')$ if $\text{proj}_j^{[2]}(q, q') \in \delta_{j,a}$. $\square$

Since computations of team automata are sequences of synchronizations, a projection on the j -th component yields computations of that j -th component. Hence team computations are composites of the components' computations. However, due to the fact that the transition relation of a team automaton is only required to be a subset of the complete transition space, not every computation of a component of a team will be represented in a computation of that team.

3 Subteams and Iteration

In this and the next section $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ is as before a fixed but arbitrary, composable system and we let $\mathcal{T} = (\prod_{i \in \mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i \in \mathcal{I}} I_i)$ be a team automaton over $\mathcal{S}$.

By focussing on a subset of the components in $\mathcal{S}$, a subteam within $\mathcal{T}$ can be distinguished. Its transitions are restrictions of the transitions of $\mathcal{T}$ to the components in the subteam. Its actions are of course the actions of the components involved. To allow for the use of the subteam as an independent team over a subset of $\mathcal{S}$, its actions are classified without the context provided by $\mathcal{T}$. Hence, whether an action is input, output or internal for the subteam only depends on its roles in the components forming the subteam rather than on how it is classified in $\mathcal{T}$. This means in particular that an action which is an output action of

$\mathcal{T}$ is an input action for the subteam whenever this action is an input action of at least one of the components of the subteam and no component automata are considered which have this action as an output action.

Definition 6. Let $J \subseteq \mathcal{I}$. The subteam of $\mathcal{T}$ determined by J is denoted by $SUB_J(\mathcal{T})$ and is defined as

$$SUB_J(\mathcal{T}) = (\prod_{j \in J} Q_j, (\Sigma_{J,inp}, \Sigma_{J,out}, \Sigma_{J,int}), \delta_J, \prod_{j \in J} I_j)$$

where $\Sigma_{J,int} = \bigcup_{j \in J} \Sigma_{j,int}$, $\Sigma_{J,out} = \bigcup_{j \in J} \Sigma_{j,out}$, $\Sigma_{J,inp} = (\bigcup_{j \in J} \Sigma_{j,inp}) \setminus \Sigma_{J,out}$, and for all $a \in \Sigma_J = \bigcup_{j \in J} \Sigma_j$, $(\delta_J)_a = proj_J^{[2]}(\delta_a) \cap \Delta_a(\{\mathcal{C}_j \mid j \in J\})$. $\square$

The transition relation of a subteam of $\mathcal{T}$ determined by some $J \subseteq \mathcal{I}$, is obtained by restricting the transition relation of $\mathcal{T}$ to synchronizations between the component automata in $\{\mathcal{C}_j \mid j \in J\}$. Hence in each transition of the subteam at least one of the component automata is actively involved. This is formalized by the intersection of $proj_J^{[2]}(\delta_a)$ with $\Delta_a(\{\mathcal{C}_j \mid j \in J\})$, for each action a , because in each transition in this complete transition space at least one component from $\{\mathcal{C}_j \mid j \in \mathcal{J}\}$ is active.

Since $\{\mathcal{C}_j \mid j \in J\}$ is a composable system, it is clear that the subteam $SUB_J(\mathcal{T})$ of $\mathcal{T}$ is again a team automaton (over $\{\mathcal{C}_j \mid j \in J\}$). Subteams can thus be used as components and, as we will see, they can be used to iteratively define the team automaton from which they are derived. Clearly, the subteam $SUB_{\mathcal{I}}(\mathcal{T})$ of $\mathcal{T}$ determined by $\mathcal{I}$, i.e. by all components, is the team itself. However, a subteam determined by a single component $j \in \mathcal{I}$ in general differs from $\mathcal{C}_j$, even if the difference between Q_j and $\prod_{j \in \{j\}} Q_j$ is ignored. This is due to the possibility that within $\mathcal{T}$ not all transitions from the complete transition spaces $\Delta_a(\mathcal{S})$ are used and hence some a -transitions from $\Delta_a(\{\mathcal{C}_j\})$ may be missing. Thus if $\mathcal{I} \neq \{j\}$, then for each action $a \in \Sigma_{\{j\}} = \Sigma_j$ we only have $proj_J^{[2]}((\delta_{\{j\}})_a) \subseteq \delta_{j,a}$ and not necessarily an equality.

It is straightforward to extend the definition of the projection of computations of $\mathcal{T}$ on a component $\mathcal{C}_j$ to projection on a subteam $SUB_{\mathcal{J}}(\mathcal{T})$ (it suffices to replace $proj_j$ by $proj_J$). The projection on $SUB_{\mathcal{J}}(\mathcal{T})$ of a team computation yields computations of $SUB_{\mathcal{J}}(\mathcal{T})$.

Until now we directly defined team automata from the component automata in $\mathcal{S}$, but other routes are also feasible. In particular, one might (iteratively) form teams from (disjoint) subsets of $\mathcal{S}$ and then use these as components for a higher-level team, until after a finite number of such iterations all components from $\mathcal{S}$ have been used.

Example 3. Let $\mathcal{S} = \{\mathcal{C}_i \mid 1 \leq i \leq 7\}$ be a composable system consisting of seven component automata and let $\mathcal{T}_{1-7} = (\prod_{1 \leq i \leq 7} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{1 \leq i \leq 7} I_i)$ be a team automaton over $\mathcal{S}$. The structure of $\mathcal{T}_{1-7}$ relative to $\mathcal{S}$ is depicted in the tree of Figure 3(a).

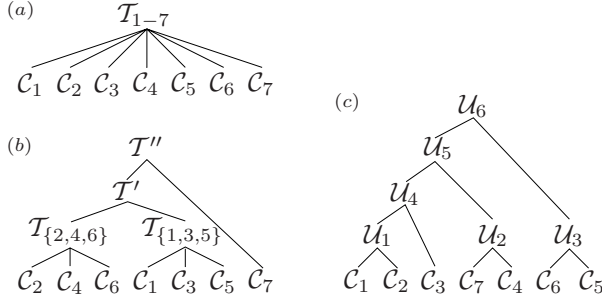


Fig. 3. Team automata constructed from $\{C_1, C_2, \dots, C_7\}$

Since every subset of a composable system is again a composable system, one could form a team automaton $\mathcal{T}_{\{2,4,6\}}$ over $\{C_2, C_4, C_6\}$ and a team automaton $\mathcal{T}_{\{1,3,5\}}$ over $\{C_1, C_3, C_5\}$. Then a new team can be formed from $\mathcal{T}_{\{2,4,6\}}$ and $\mathcal{T}_{\{1,3,5\}}$ since every team automaton is a component automaton and, moreover, $\mathcal{C}'_1 = \mathcal{T}_{\{2,4,6\}}$ and $\mathcal{C}'_2 = \mathcal{T}_{\{1,3,5\}}$ together form a composable system $\mathcal{S}' = \{\mathcal{C}'_1, \mathcal{C}'_2\}$. That $\mathcal{S}'$ is a composable system is a consequence of the composability of $\{C_i \mid 1 \leq i \leq 7\}$ and the definition of team automata by which the internal actions of a team are exactly the internal actions of its components. Hence, the internal actions of $\mathcal{T}_{\{2,4,6\}}$ do not occur as actions in $\mathcal{T}_{\{1,3,5\}}$ and vice versa. In general, we have that teams over disjoint subsets of components from a composable system form again a composable system.

Now assume that a team automaton $\mathcal{T}'$ over $\mathcal{S}'$ has been constructed. Then, with the same reasoning as above, $\mathcal{T}'$ and C_7 together form a composable system $\mathcal{S}'' = \{\mathcal{C}''_1, \mathcal{C}''_2\}$, with $\mathcal{C}''_1 = \mathcal{T}'$ and $\mathcal{C}''_2 = C_7$ and we can define a team automaton $\mathcal{T}''$ over $\mathcal{S}''$. The structure of such $\mathcal{T}''$ relative to $\mathcal{S}$, is depicted in the tree of Figure 3(b). In Figure 3(c), the tree for yet another route for constructing a team automaton $\mathcal{U}_6$, starting from the components in $\mathcal{S}$, is depicted.

Let $\mathcal{T}'' = (P'', (\Sigma''_{inp}, \Sigma''_{out}, \Sigma''_{int}), \delta'', I'')$ be a team automaton over $\mathcal{S}''$, for some $\delta'' \subseteq P'' \times \Sigma'' \times P''$ where $\Sigma'' = \Sigma''_{inp} \cup \Sigma''_{out} \cup \Sigma''_{int}$. Then by the definition of team automata we immediately have $\Sigma''_{int} = ((\bigcup_{i \in \{2,4,6\}} \Sigma_{i,int}) \cup (\bigcup_{i \in \{1,3,5\}} \Sigma_{i,int})) \cup \Sigma_{7,int} = \bigcup_{1 \leq i \leq 7} \Sigma_{i,int}$. Likewise, $\Sigma''_{out} = \bigcup_{1 \leq i \leq 7} \Sigma_{i,out}$ and $\Sigma''_{inp} = (\bigcup_{1 \leq i \leq 7} \Sigma_{i,inp}) \setminus \bigcup_{1 \leq i \leq 7} \Sigma_{i,out}$. Thus $\mathcal{T}''$ has the same input, output and internal actions as any team formed directly over $\mathcal{S}$. Its set P'' of states, however, differs from the set of states of a team over $\mathcal{S}$ by its nested structure and its ordering. By definition, $P'' = ((\prod_{i \in \{2,4,6\}} Q_i) \times (\prod_{i \in \{1,3,5\}} Q_i)) \times Q_7 = ((Q_2 \times Q_4 \times Q_6) \times (Q_1 \times Q_3 \times Q_5)) \times Q_7$. Similarly, $I'' = ((I_2 \times I_4 \times I_6) \times (I_1 \times I_3 \times I_5)) \times I_7$.

Also the team automaton $\mathcal{U}_6$ has the same input, output and internal actions as any team formed directly over $\mathcal{S}$. The set of states of $\mathcal{U}_6$ is $((Q_1 \times Q_2) \times Q_3) \times (Q_7 \times Q_4) \times (Q_6 \times Q_5)$. $\square$

As demonstrated in the above example, given the composable system $\mathcal{S}$, one may form teams over disjoint subsets of components from $\mathcal{S}$. These teams together with the component automata not involved in any of these teams are again a composable system, which can then be used as the basis for the formation of still higher-level teams. This leads to the following definition of an iterated team automaton.

Definition 7. *A team automaton is an iterated team automaton over $\mathcal{S}$ if it is either a team automaton over $\mathcal{S}$ or a team automaton over $\{\mathcal{T}_j \mid j \in \mathcal{J}\}$, where each $\mathcal{T}_j$ is an iterated team automaton over $\{\mathcal{C}_i \mid i \in \mathcal{I}_j\}$, for some $\mathcal{I}_j \subseteq \mathcal{I}$, and $\{\mathcal{I}_j \mid j \in \mathcal{J}\}$ forms a partition of $\mathcal{I}$. $\square$*

The notion of an iterated team automaton is a generalization of the notion of a team automaton: every team over a given composable system $\mathcal{S}$ is also an iterated team over $\mathcal{S}$. Conversely, a team formed iteratively over a composable system has, when compared with the teams formed directly over $\mathcal{S}$, the same alphabet of actions — including the distribution over input, output and internal actions — and “essentially” the same state space, transition space and set of initial states. The only difference lies in the ordering and grouping of states coming from different components. In [5] this difference is formally described in terms of functions to unpack and reorder Cartesian products. Here we simply assume some reorder mechanism by which the state space of any iterated automaton over $\mathcal{S}$ is identified with $\prod_{i \in \mathcal{I}} Q_i$. This reordering of the state space of an iterated automaton naturally implies a corresponding modification of its transition space which translates the transitions of the iterated team into transitions of a team formed directly over $\mathcal{S}$. Thus every iterated team over $\mathcal{S}$ can be interpreted as an ordinary team automaton over $\mathcal{S}$. Conversely, a team automaton over $\mathcal{S}$, can be constructed as an iterated team automaton (with still essentially the same transitions) according to any prescribed iteration, provided that each intermediate iterated team over a subsystem $\{\mathcal{C}_i \mid i \in \mathcal{J}\}$ of $\mathcal{S}$ has at least the transitions (after reordering) of the subteam determined by $\mathcal{J}$.

So we may conclude that team automata are naturally suited to describe hierarchical systems. Both subteams and iterated teams can be treated as team automata including the considerations concerning their computations and behaviour and it suffices to consider only the relationship between subteams and team automata.

4 Synchronizations

This section discusses various — standard — ways of synchronizing transitions within team automata. On the one hand we consider modes of synchronization which describe how actions synchronize in a given team automaton. On the other hand we consider how to (uniquely) define a team automaton (thus its transition relation) given these modes of synchronizations for its actions.

We distinguish three natural globally defined modes of synchronization for the actions of a team automaton: there may be actions that are always executed

by a single component and actions on which all components that have this action have to synchronize. In the latter case the requirement of having to participate may be weakened by requiring participation only if the components are ready (in the right state) to execute that action.

Let $\mathcal{T}$ be the team automaton over $\mathcal{S}$ as specified before and let a be an action of $\mathcal{T}$. Then

a is *free* in $\mathcal{T}$ if no a -transition of $\mathcal{T}$ is brought about by a simultaneous execution of a by two or more components. Thus, whenever a is executed by $\mathcal{T}$ only one component is active in this execution.

a is *action-indispensable* (ai for short) in $\mathcal{T}$ if all components which have a as one of their actions are involved in every execution of a by $\mathcal{T}$. This means that $\mathcal{T}$ cannot perform an a if one of the components to which a belongs is not ready for it (a is not enabled in that component at the current local state).

a is *state-indispensable* (si for short) in $\mathcal{T}$ if all executions of a by $\mathcal{T}$ involve all components in which a is currently enabled. Thus blocking may be avoided since $\mathcal{T}$ does not have to “wait” with the execution of a until all components of which a is an action are ready for it.

In Example 1, b is ai and si but not free in $\mathcal{T}_1$, it is si, not ai and not free in $\mathcal{T}_2$, and it is neither ai nor si nor free in $\mathcal{T}_3$.

As noted before, information on the actual execution of loops by components is missing in the transition relation of a team automaton. Therefore, in the above classification of actions the presence of loops on a in components is treated as if a is actually executed, which is in accordance with the maximal interpretation of the components’ involvements adopted before.

In Example 2 we thus have that action a is not free and not ai, but it is si.

Since an internal action of $\mathcal{T}$ is an action of only one of the component automata, it is immediate that every internal action is free, ai and si in $\mathcal{T}$. Furthermore, every action that is ai in $\mathcal{T}$ also satisfies the weaker requirement of being si. So ai implies si, and this is in fact the only dependency among the properties free, ai and si.

The property of an action being free, ai or si in $\mathcal{T}$ is inherited by all subteams of $\mathcal{T}$ to which this action belongs. Thus if $J \subseteq \mathcal{I}$ and $a \in \Sigma_J = \bigcup_{j \in J} \Sigma_j$, then a is free (ai or si, respectively) in $SUB_J(\mathcal{T})$ whenever a is free (ai or si, respectively) in $\mathcal{T}$. The converse is not true in general. Consider, e.g., the team automaton $\mathcal{T}_3$ defined in Example 1. The action b is neither free, nor ai nor si in $\mathcal{T}_3$, although it is free, ai and si in the subteam determined by $\{3\}$, which is a copy of $\mathcal{C}_3$. In fact, an action that is not free (ai or si) in a team automaton, can be made free (ai or si, respectively) in a subteam by dropping those components that caused it not to be free (ai or si, respectively) in $\mathcal{T}$. Observe that in every subteam determined by a single component, every action is free, ai and si. The properties of being free, ai or si are carried over from a subteam to the team as a whole if all components that the action in question belongs to, are included in the subteam.

In a team automaton where every action is ai, every transition involves all components that have the action to be executed in their alphabet. This implies that any behaviour of such automata, when projected on a component or

subteam, yields the corresponding behaviour of that component or subteam. Thus, for such automata, we can extend our earlier observation on computations to behaviours: from a team behaviour a component (subteam) behaviour can immediately be extracted by deleting all actions that do not belong to that component (or subteam)! This is not possible when an action may be executed without involving all components to which it belongs.

Until now we have discussed synchronizations while ignoring whether the action was input, output or internal in certain components. Next we turn to the different roles an action may have in different components. Since internal actions belong to only one component distinguishing between their roles in different components is indeed not very relevant. External actions, however, may be input to some components, and output to other components, and thus we consider only external actions for the rest of this section.

We denote the set of external actions of (the team automata over) $\mathcal{S}$ as Σ_{ext} . Thus $\Sigma_{ext} = \bigcup_{i \in \mathcal{I}} (\Sigma_i \setminus \Sigma_{i,int})$. First we separate their output role and their input role. Since no external action of any team automaton over $\mathcal{S}$ will ever be both an input and an output action for one and the same component, we can define disjoint input and output domains for each external action. For $a \in \Sigma_{ext}$ we have $\{j \in \mathcal{I} \mid a \in \Sigma_{j,inp}\}$ as the *input domain* of a in $\mathcal{S}$, and $\{j \in \mathcal{I} \mid a \in \Sigma_{j,out}\}$ as the *output domain* of a in $\mathcal{S}$. These two domains each determine a subteam of $\mathcal{T}$, to which we refer, respectively, as the *input subteam* of a in $\mathcal{T}$ and the *output subteam* of a in $\mathcal{T}$. Note that for every external action, at least one of its input domain and its output domain is not empty. In case the input (or output) domain of an external action is empty, then its input (output) subteam is the trivial automaton $(\emptyset, (\emptyset, \emptyset, \emptyset), \emptyset, \emptyset)$.

Having determined for each external action a its input and its output subteam, we can now identify certain modes of synchronization relating to a in its role as input or output action. First we look *within* these subteams in which by definition a has only one role and all components are peers, in the sense that they are on an equal footing with respect to a . We say that an input (output) action a is input (output) peer-to-peer, if every execution of a involving components of that subteam requires the participation of all. This obligation to participate can be explained in a strong and in a weak sense. Strong simply means that no synchronizations on a can take place unless all components in the input (output) domain of a take part. Weak means that synchronizations on a involve all of the components in the input (output) domain of a which are ready to execute a (in a state in which a can be executed). Thus the notion of strong requires that a is ai in its input (output) subteam, while the notion of weak requires that a is si in its input (output) subteam. Since ai implies si, it is immediately clear that every action which is strong input (output) peer-to-peer in a team automaton, is also weak input (output) peer-to-peer in that team automaton. This however does not hold the other way around. For the output case this follows from Example 2 since a is weak output peer-to-peer but not strong output peer-to-peer in $\mathcal{T}$.

Next we define synchronizations *between* the input and output subteams of the external action a . Here the idea is that input actions (“slaves”) are driven

by output actions (“masters”). This means that if a is an output action, then its input counterpart can never take place without being triggered (the slave never proceeds on its own). Consequently, the input subteam of an output action a cannot execute a unless a is also executed as an output action (by its output subteam). It is however possible that a is executed as an output action without its simultaneous execution as an input action. We say that a is *master-slave* if it is an output action and its output subteam participates in every a -transition of $\mathcal{T}$. In addition one could require that a in its role of input action *has to* synchronize with a as an output action (the slave has to follow the master). Since the obligation of the slave to follow the master may again be formulated in two different ways, we obtain notions of strong and weak master-slave actions. When guided by the ai principle, we get a strong notion of master-slave synchronization, while the si principle leads to a weak notion of master-slave synchronization. Formally, we have that a is *strong master-slave* if it is master-slave and its input subteam moreover participates in every a -transition of $\mathcal{T}$. For a to be *weak master-slave*, we require that it is master-slave and that its input subteam moreover participates in every a -transition of $\mathcal{T}$ whenever it can. Thus if an action is strong master-slave in a team automaton, then it is also weak master-slave.

Since the definition of a being master-slave in $\mathcal{T}$ guarantees that the output subteam of a is actively involved in every a -transition of $\mathcal{T}$, it follows immediately from the definition of subteams that the a -transitions of the output subteam of a are precisely the projections of the a -transitions of $\mathcal{T}$ on the output domain of a . Similarly, in case a is strong master-slave we have in addition that the a -transitions of the input subteam of a are precisely the projections of the a -transitions of $\mathcal{T}$ on the input domain of a . In case a is weak master-slave, there may be a -transitions in $\mathcal{T}$ in which the input subteam, even when it is not trivial, is not actively involved. In those cases, a is executed as an output action by $\mathcal{T}$ without simultaneous execution of a as an input action.

In these master-slave definitions, input subteams and output subteams are treated as given entities (black boxes). Clearly, one can combine the master-slave synchronizations with additional requirements on the synchronizations taking place within the subteams. Thus one may prescribe a master-slave synchronization on an action a which is in addition input peer-to-peer. Then *all* components with a as an input have to follow the output.

A synchronization mode describes how components synchronize on an action as a property that is satisfied by the combination of transitions chosen for that action from its complete transition space. However, in general there may be different choices which all satisfy a given property.

In Example 1, the action b is si in $\mathcal{T}_1$ as well as in $\mathcal{T}_2$ although these two automata do not have the same b transitions. Also in the extreme case that no b -transitions at all are chosen, b is si in the resulting team automaton.

Clearly, for applications of the team automata model in the field of groupware systems, it is undesirable if different team automata can be constructed according to the specification of the required mode of synchronizations for its

actions. The natural and obvious way to get around this problem is to apply a maximality principle and to include everything that is not forbidden, i.e. is in accordance with the synchronization constraints. This is the intuitive approach of [11] which generalizes the classical approach to define synchronized systems (see, e.g., [10], [13], [19] and [21]) from ai to other modes of synchronization and is often implicitly assumed. When a team automaton is to be constructed according to a specification of synchronization conditions for its actions, the strategy is to include as many transitions as possible without violating the specification. For this approach to work as desired, the result should be unique, i.e. there should exist a largest set of transitions satisfying the given constraints. We now check whether this maximality principle can be applied for the synchronization modes just discussed.

Let $a \in \Sigma_{ext}$ be an external action of $\mathcal{S}$. Note that since the transition relation of an internal action is by definition fixed to be the complete transition space $\Delta_a(\mathcal{S})$ of that action in $\mathcal{S}$, there is no need to discuss internal actions. Assume that we want to construct, using the maximality principle, a team automaton over $\mathcal{S}$ with its set of a -transitions $\delta_a \subseteq \Delta_a(\mathcal{S})$ satisfying a given constraint.

If no constraints are imposed on the synchronizations on a , all a -transitions are allowed since nothing is required, and thus no transition is forbidden. In this case we set $\delta_a = \Delta_a(\mathcal{S})$ and the maximality principle yields a unique set of a -transitions for the team automaton.

If a should be free, ai, or si, then *all and only* those a -transitions are included in δ_a that respect the specified property of a . It can easily be shown that in each case δ_a is the largest relation in $\Delta_a(\mathcal{S})$ in which a is free, ai or si, respectively.

If a is to be peer-to-peer, we have to distinguish between its input and output role in $\mathcal{S}$ and between strong (ai) and weak (si) synchronizations. For each of these four possibilities *all and only* those transitions from $\Delta_a(\mathcal{S})$ are included in δ_a in which all component automata given by the input or output domain, respectively, are forced (in the weak or in the strong sense) to participate. Thus if the input (output) domain of a is not empty and a should be strong input (output) peer-to-peer, then δ_a consists of all a -transitions from $\Delta_a(\mathcal{S})$ in which all components from the input (output) domain of a are active. If the input (output) domain of a is not empty and a should be weak input (output) peer-to-peer, then δ_a consists of all a -transitions from $\Delta_a(\mathcal{S})$ in which all components from the input (output) domain of a in which a is currently enabled are active. In all four cases, δ_a will be the largest set of a -transitions satisfying the specified constraint.

Finally, we turn to master-slave synchronizations. Recall that a is master-slave in the team automaton if it is an output action and its output subteam participates in every synchronization on a . Thus if a is to be master-slave, then we include *all and only* those a -transitions in δ_a in which a is executed by at least one of the components in its output domain. For a to be strong master-slave there is the additional requirement that a should also be executed by its input subteam in the team automaton. In this case we include *all and only* those a -transitions in δ_a in which a appears at least once in its output role and,

moreover, if the input domain of a is not empty, then also at least one component from the input domain of a is involved. In both cases δ_a will be the largest set of a -transitions satisfying the specified constraint.

If a is to be weak master-slave, it should be master-slave and moreover its input subteam should participate in the a -transitions of the team automaton whenever it can. Following the established pattern we would include *all and only* those a -transitions in δ_a in which a appears at least once in its output role and, moreover, if a component from the input domain of a is ready to perform a , then also at least one component from the input domain of a is involved. However, it is not necessarily the case that the resulting δ_a will be the largest set of a -transitions by which a is weak master-slave. This is due to the fact that a can be weak master-slave in a team automaton which has a -transitions in which the input subteam of a does not participate although a is currently enabled in a component of this subteam. Since a subteam only contains a selection of all possible a -transitions, it may happen that a is enabled in a component of the input subteam, but not in the subteam. There is no way out and in fact the maximality principle cannot work. One has to refer to *components from the input domain of a* because referring to an *input subteam* is only possible if a team automaton including the transition relation has been defined already.

Hence, for each of the various modes of synchronization considered here except for weak master-slave synchronization the maximality principle can be applied to define a unique set of transitions.

5 Related Models

Team automata are related to many familiar models for distributed, concurrent systems. In this section we demonstrate how one may view them as a model somewhere inbetween Input/Output automata and a Petri net model developed within a theory of systems with vector synchronizations.

5.1 I/O Automata

Input/Output automata (or I/O automata) were introduced in [25] (see also [18] and [19]) for modelling distributed discrete event systems consisting of components that operate concurrently. Since then they have been used extensively as a formal model for the verification of distributed algorithms (see, e.g., [17] and [22]). Originally, I/O automata are defined in terms of labelled transition systems together with an associated equivalence relation over the set of actions used to define so-called fair computations. In [25] I/O automata without such equivalence relations are called safe I/O automata and in [12] they are referred to as unfair. Here we are not concerned with fairness and we only consider safe or unfair I/O automata, to which we will simply refer as I/O automata. The model of I/O automata has a single notion of automaton composition which, as already noted in [25], is rather restrictive and may hinder a realistic modelling of certain

types of interactions. This is the main motivation given in [11] for introducing team automata for groupware systems as a generalization of I/O automata.

An I/O automaton is an ilts together with a classification of its actions as input, output or internal. Input and output actions form the interface between the automaton and its environment, including other I/O automata. Within a composition, automata which share an action a have to perform a simultaneously (synchronize on a). The intention is that simultaneous execution models a communication from the automata of which a is an output action to the automata of which a is an input action. In fact, the execution of an input action is thought of as the notification of the arrival of output from another automaton. With these considerations in mind, I/O automata are formally defined as component automata, but with the additional condition that they should be *input-enabled*. This means that, whatever the current state of the automaton, it is always capable of receiving any of its potential inputs. Thus, in every state of the automaton, every input action of that automaton is enabled.

Given a collection $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ of I/O automata, a new I/O automaton can be constructed provided $\mathcal{S}$ satisfies two conditions. These conditions only relate to the role of the actions and it is irrelevant whether or not the $\mathcal{C}_i$ are input-enabled. We can thus assume that $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ is as before a collection of component automata. The first condition is that $\mathcal{S}$ should be composable. Hence, as for the definition of a team automaton, it is required that the internal actions of any of the component automata belong uniquely to that component. Secondly, there is the idea that two components cannot be expected to synchronize on an output action. Rather than complicating the notion of composition itself, this is prohibited by the requirement that the output actions of the automata in $\mathcal{S}$ should be disjoint. This means that every external action can be output in at most one of the component I/O automata. Formally, for all $i \in \mathcal{I}$, $\Sigma_{i,out} \cap \bigcup_{j \in \mathcal{I} \setminus \{i\}} \Sigma_{j,out} = \emptyset$. If $\mathcal{S}$ satisfies both conditions, then we call it a *compatible system*. Note that every subset of a compatible system is again a compatible system.

Finally, the composition of I/O automata into a new automaton is defined according to the intuitive explanation above that automata which share an action have to synchronize on a . In terms of our framework this means that a team automaton is constructed, in which every action is ai. Moreover (although this is only implicit in the explanation) all synchronizations which do not violate this condition have to be included (maximality). Hence the constructed team automaton is unique. We adopt the terminology that an action a of a team automaton is *maximal ai* in that team, if it is ai and the set of a -transitions is maximal.

Definition 8. Let $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ be a compatible system of I/O automata. Then the team I/O automaton over $\mathcal{S}$ is the team automaton $\mathcal{T}$ over $\mathcal{S}$ such that all actions are maximal ai in $\mathcal{T}$. $\square$

Since this composition of a team automaton preserves input-enabledness, it follows that every team I/O automaton is again an I/O automaton and hence

can be used to iteratively define higher-level team I/O automata. Together with our earlier observation that subsets of a compatible system are compatible systems, this implies that the team I/O automaton over a compatible system $\mathcal{S}$ of I/O automata can be constructed by iteration. Any iterated team I/O automaton corresponds to the team I/O automaton over $\mathcal{S}$ after reordering its state space. Conversely, if $\mathcal{T}$ is the team I/O automaton over a compatible system $\mathcal{S} = \{\mathcal{C}_i \mid i \in \mathcal{I}\}$ of I/O automata, then every subteam of $\mathcal{T}$ determined by some $J \subseteq \mathcal{I}$, is the team I/O automaton over $\{\mathcal{C}_j \mid j \in J\}$. This follows from our earlier remark that the property of an action being ai in a team automaton is inherited by all of its subteams. (That maximality is also inherited is immediate.) Since all actions of a team I/O automaton are ai it follows that every output action is strong master-slave. This provides a formal description of the idea that output is always received by those component automata that have its input counterpart as an action. Since I/O automata are input-enabled, the output automaton does not even have to wait until the input automata are ready for the communication. It is however worthwhile to notice that it may be the case that an external action appears only as an input action in the system $\mathcal{S}$. Then it is again an input action of the team I/O automaton over $\mathcal{S}$ and can be used as such in a higher-level team. Note that since all input (output) actions are ai in an I/O team automaton, they are also strong input (output) peer-to-peer.

The I/O automaton model fits seamlessly in the team automata model and results and notions from team automata become available for I/O automata. In particular, team automata provide a framework in which the underlying concepts of I/O automata can be given a broader perspective and compared with other ideas. For instance, the possibility to define the language of a team I/O automaton directly (without actually considering the team) from the languages of its components is an important property in the theory of I/O automata. This property is already implied by the maximal ai construction for general team automata. Also the idea of subteams and iterative construction only marginally investigated for I/O automata, are now immediately available from the team automata framework. Team automata however allow more types of synchronizations, which is convenient when formally designing a system. As remarked in [25], for some designs it may be a disadvantage that the composition of I/O automata implies that output actions can always be traced back to a unique sender.

5.2 Vector Control and Petri Nets

Team automata are compositions of component automata which work together through synchronizations on certain actions. These synchronizations are labelled transitions which describe state changes caused by global actions of the team. As a consequence, the operational semantics in terms of computations of team automata is of a sequential nature and does not reflect the fact that they are distributed systems. By switching from global actions to actions with local information on the participation of the components it is however possible to make the potential concurrency within a team visible. This subsection shows how vector

actions can be employed to this aim and then relates team automata to Petri nets by fitting them in a theory of vector controlled concurrent systems.

For the sequel we let $\mathcal{T} = (\prod_{i \in \mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta, \prod_{i \in \mathcal{I}} I_i)$ be a given team automaton over $\mathcal{S}$ and we use Σ to denote the union $\Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}$ of the input, output and internal alphabet of $\mathcal{T}$.

By the definition of team automata each transition of $\mathcal{T}$ is of the form (q, a, q') with $a \in \Sigma$ and $q, q' \in \prod_{i \in \mathcal{I}} Q_i$. We now switch from transitions (q, a, q') to *vector transitions* (q, α, q') , where α is an element of $\prod_{i \in \mathcal{I}} (\{a\} \cup \{\lambda\})$, i.e. a vector with for each component a corresponding entry which is either a or λ . If an entry of α is a , then this indicates that the corresponding component takes part in the synchronization on a , while if it is λ , then that component is not involved. This switch is feasible since for each transition (q, a, q') the global state change from q to q' , caused by the occurrence of this transition, is described in terms of changes in the local states of the components involved. If $proj_j(q) \neq proj_j(q')$, then $proj_j^{[2]}(q, q') \in \delta_{j,a}$ and the j -th component is involved. In that case we set $proj_j(\alpha) = a$. If $proj_j(q) = proj_j(q')$ and $proj_j^{[2]}(q, q') \notin \delta_{j,a}$, then the j -th component is not involved and we set $proj_j(\alpha) = \lambda$. There is however — again — the problem of loops. If $proj_j(q) = proj_j(q')$ and $proj_j^{[2]}(q, q') \in \delta_{j,a}$, then it is unclear whether or not the j -th component is involved. Following the maximal interpretation we assume it is and set $proj_j(\alpha) = a$. This procedure transforms $\mathcal{T}$ into a *vector team automaton* over $\mathcal{S}$ which has vector transitions rather than “flat” transitions.

On the other hand, one may also directly define a vector team automaton over $\mathcal{S}$ by translating the required synchronizations straight away into vector transitions. In that case, for each action a , one chooses vector transitions from the *complete vector transition space* $\Delta_a^v(\mathcal{S})$ of a in $\mathcal{S}$ which describes all possible vector transitions for a .

Definition 9. Let $a \in \bigcup_{i \in \mathcal{I}} \Sigma_i$. The complete vector transition space of a in $\mathcal{S}$ is denoted by $\Delta_a^v(\mathcal{S})$ and is defined as

$$\begin{aligned} \Delta_a^v(\mathcal{S}) = \{ & (q, \alpha, q') \mid (q, q') \in \Delta_a(\mathcal{S}), \alpha \in \prod_{i \in \mathcal{I}} (\{a\} \cup \{\lambda\}) \text{ and} \\ & \forall i \in \mathcal{I} : [\text{if } proj_i(\alpha) = a, \text{ then } (proj_i(q), proj_i(q')) \in \delta_{i,a}] \text{ and} \\ & [\text{if } proj_i(\alpha) = \lambda, \text{ then } proj_i(q) = proj_i(q')] \} . \end{aligned}$$

□

If $(q, \alpha, q') \in \Delta_a^v(\mathcal{S})$, then α is called a *vector representation* of a in $\mathcal{S}$ or a *vector action* of $\mathcal{S}$. Observe that due to the composability of $\mathcal{S}$ every internal action has only one vector representative and this representative has exactly one entry which is not λ . Furthermore, all vector representatives of external actions have at least one entry which is not λ .

A vector team automaton over $\mathcal{S}$ is now defined exactly as an ordinary team automaton over $\mathcal{S}$, except that its transition relation consists of vector transitions.

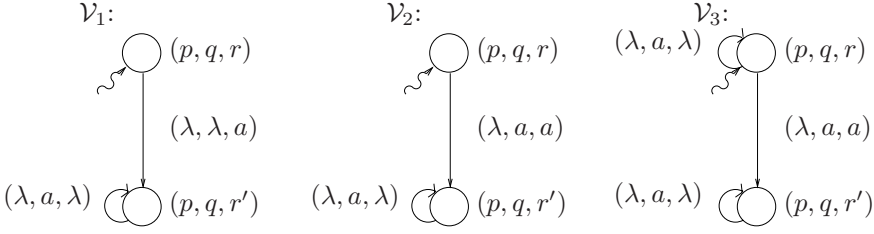


Fig. 4. Three vector team automata

Definition 10. A vector team automaton over $\mathcal{S}$ is a construct

$$\mathcal{V} = (\prod_{i \in \mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta^v, \prod_{i \in \mathcal{I}} I_i)$$

such that $\Sigma_{int} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,int}$, $\Sigma_{out} = \bigcup_{i \in \mathcal{I}} \Sigma_{i,out}$, $\Sigma_{inp} = (\bigcup_{i \in \mathcal{I}} \Sigma_{i,inp}) \setminus \Sigma_{out}$, $\delta^v \subseteq \prod_{i \in \mathcal{I}} Q_i \times [\bigcup_{a \in \Sigma} \prod_{i \in \mathcal{I}} (\{a\} \cup \{\lambda\})] \times \prod_{i \in \mathcal{I}} Q_i$ where $\Sigma = \Sigma_{inp} \cup \Sigma_{out} \cup \Sigma_{int}$ and $\delta^v \cap \prod_{i \in \mathcal{I}} Q_i \times [\prod_{i \in \mathcal{I}} (\{a\} \cup \{\lambda\})] \times \prod_{i \in \mathcal{I}} Q_i \subseteq \Delta_a^v(\mathcal{S})$ for all $a \in \Sigma$, and moreover $\Delta_a^v(\mathcal{S}) \subseteq \delta^v$ if $a \in \Sigma_{int}$. $\square$

Example 4. In Figure 4 three vector team automata over the composable system $\{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$ of Example 2 are given.

Note that in these vector team automata it is clear which components participate in each of their vector transitions. This contrasts with the team automaton $\mathcal{T}$ of Example 2. $\square$

By replacing each transition (q, α, q') of a vector team automaton $\mathcal{V}$ by the flat transition (q, a, q') if α is a vector representative of the action a , one obtains a *flattened version* of the vector team automaton. This is a normal team automaton which models essentially the same synchronizations. However, information on the role of loops is lost. In fact, each vector team automaton has a unique flattened version, whereas there may be many vector team automata that have the same flattened version. In this sense, vector team automata have more expressive power than “ordinary” team automata. As an example, note that both vector team automata of Example 4 have the team automaton $\mathcal{T}$ of Example 2 as their flattened version.

Vector team automata are an example of a model of distributed systems consisting of sequential components, the cooperation of which is controlled by synchronization vectors. In such systems one deals with independently operating processes that from time to time synchronize their actions with others. Vectors of actions describe which processes are involved in such a cooperation. No other actions are allowed in the system than those described by the vectors. Thus also individual actions of the processes appear as vectors with a single non- λ entry. In [15] and [16] a framework is proposed for the study of vector controlled systems. The approach there is based on the synchronization as modelled in the

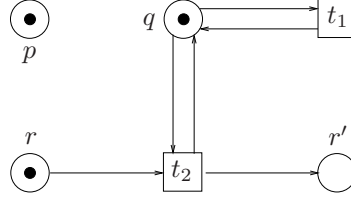


Fig. 5. A Petri net

vector firing sequence semantics of path expressions and COSY (see, e.g., [13]) and is related to the work of Arnold and Nivat (see, e.g., [1] and [2]) and the coordination of cooperating automata by synchronization on multisets in [3]. The model of Vector Controlled Concurrent Systems (VCCS for short) allows to specify, in addition to the component processes and the synchronization vectors, also a control mechanism to determine when synchronizations are to be used. This makes it possible to fit team automata and vector team automata, with their state dependent synchronizations into the VCCS framework.

As an example, we describe a particular method of vector synchronization based on Petri nets which under certain circumstances is applicable to team automata. (We assume the reader to be familiar with basic notions of Petri nets, see, e.g., [23] and [24].)

Let $\mathcal{V} = (\prod_{i \in \mathcal{I}} Q_i, (\Sigma_{inp}, \Sigma_{out}, \Sigma_{int}), \delta^v, \prod_{i \in \mathcal{I}} I_i)$ be a vector team automaton over $\mathcal{S}$. Then $\mathcal{V}$ has an immediate interpretation as a Petri net $\mathcal{N}_{\mathcal{V}}$ of a special form. This net has a place for each state of each of the component automata and, for each vector transition of the team, a Petri net transition (which we will call an event from now on in order to avoid confusion). Let (q, α, q') be a vector transition of $\mathcal{V}$ and let t be its associated event. For all $i \in \mathcal{I}$, whenever $proj_i(\alpha)$ is not λ we have $proj_i(q)$ as an input place of t and $proj_i(q')$ as an output place of t . Thus only the components actively involved in the vector transition (q, α, q') participate in the event t and have their local state changed by t according to (q, α, q') . The event t is labelled by α . The initial markings of $\mathcal{N}_{\mathcal{V}}$ are defined by the initial states of $\mathcal{V}$. An initial marking corresponding to $q \in \prod_{i \in \mathcal{I}} I_i$ assigns one token to $proj_i(q)$ for each $i \in \mathcal{I}$. This defines the structure and the labelling of $\mathcal{N}_{\mathcal{V}}$. Its dynamics is defined by the ordinary firing rule for Petri nets.

Example 5. In Figure 5 the Petri net $\mathcal{N}_{\mathcal{V}_2}$ obtained by applying the above construction to the vector team automaton $\mathcal{V}_2$ of Example 4 is given.

Event t_1 is associated to $((p, q, r'), (\lambda, a, \lambda), (p, q, r'))$ and is labelled by (λ, a, λ) , whereas event t_2 is associated to $((p, q, r), (\lambda, a, a), (p, q, r'))$ and labelled by (λ, a, a) . For $\mathcal{V}_3$, the construction associates to $((p, q, r), (\lambda, a, \lambda), (p, q, r))$ and $((p, q, r'), (\lambda, a, \lambda), (p, q, r'))$ the same event. Consequently, $\mathcal{N}_{\mathcal{V}_3}$ is identical to $\mathcal{N}_{\mathcal{V}_2}$. $\square$

The Petri net $\mathcal{N}_V$ is a composition of state machines (sequential Petri net versions of transition systems). The synchronizations between the state machines and the labelling of the events are mutually consistent. This makes $\mathcal{N}_V$ an *Individual Token Net Controller* (ITNC for short) as defined in [15] and further investigated in [14]. ITNCs are control mechanisms for vector synchronization based on state machine decomposable nets. They are designed in such a way that they can record the progress and control the synchronizations of sequential processes. The vector labels in an ITNC are the synchronization vectors and they do not have to be uniform in the sense that all non- λ entries are instances of the same action. Thus ITNCs allow more types of synchronization than team automata. However, ITNCs are not concerned with the distinction of actions into input, output and internal.

ITNCs (see [14]) allow a concurrent operational semantics according to the intuition that synchronizations that involve disjoint sets of components are independent and can be executed concurrently. This can be formalized also in terms of an independence relation (over transitions or over vector actions) similar to the independence relation used for Mazurkiewicz traces (see, e.g., [20] and [9]). In fact, as is argued in [14], (finite) ITNCs can be viewed as asynchronous automata ([26]), which are the distributed labelled transition systems that define recognizable trace languages. In a vector team automaton however whether or not a synchronization may take place also depends in general on the local states of components not actively involved in the synchronization. The construction given above to transform a (vector) team automaton into an ITNC does not take this phenomenon into account as we have seen in Example 5. There we obtained for $\mathcal{V}_2$ an ITNC in which the event t_1 can be executed independently of the current state (r or r') of the third component. Consequently, this construction should be applied directly only to team automata with synchronization constraints by which disjoint synchronizations are independent. To faithfully incorporate (global) state dependency when transforming an arbitrary (vector) team automaton into a Petri net, a construction similar to the one described in [14] may turn out to be useful.

6 Conclusion

Within the field of CSCW one deals with systems intended to support groups of people working together in collaborative projects. Such systems are often distributed and conceived as consisting of agents cooperating in a coordinated way, which leads to complex interactive behaviours. Consequently, coordination policies and their effect on behaviour are key issues for CSCW. There is a need for models which help to clarify basic notions and to develop new notions of collaboration. Team automata provide a formal, yet flexible framework for the description and analysis of protocols and groupware systems. Modelling a system as a team automaton in the early phases of design forces one to consider the intended communications and synchronizations in detail, which leads to a better understanding of the functionality of the system and to explicit and unambiguous

ous design choices. This forms the basis of further design and implementation, especially since the team automata framework allows a modular (iterative) construction and can be used also at the architectural level of system specification. At the same time the mathematically rigorous definitions provide the possibility of formal analysis tools for proving crucial design properties, without first having to implement the design.

Team automata model the logical architecture of a design. They abstract from concrete data, configurations and actions, and they describe the system solely in terms of a state-action diagram (transition system), the role of actions (input, output or internal) and synchronizations. To model a system as a team automaton, first the components have to be identified. Each of them should be given a description in the form of a labelled transition system, an easy to understand well-known model. Based on the idea of shared action, these components can be connected in order to work together. Within each component, a distinction has to be made between internal actions (not available for synchronization with other components) and external actions (which can be used to synchronize components and are subject to synchronization restrictions). Next, for each external action separately, a decision is made as to how the components should synchronize on this action. Assigning different roles to an external action makes it possible to describe different types of synchronization, such as, e.g., communications in which an action has both an input and an output role. If the action is supposed to be a “passive” action, which may not be under the local control of the component, then it can be designated as an input action of that component. Otherwise it is an output action. If such distinction between the roles of an external action is not necessary, then the choice is arbitrary. A natural option would be to make it an output action in all components in which it occurs. Once the synchronization constraints for each external action have been determined, one constructs a team automaton satisfying all constraints (perhaps applying the maximality principle).

Similarly, the architecture of the system can be described as a team automaton with team automata as building blocks (components). System properties can then be considered both at the team level and at the level of subteams. Thus, the team automata framework supports the design by making explicit the role of actions and the choice of transitions governing the coordination of the component automata. The crucial feature is the freedom of choice for the synchronizations collected in the transition relation of a team automaton.

In this paper we have presented a survey based on our theoretical work. In [7] however an example is presented of how team automata can be used to model multi-layered access control requirements for collaborative environments. Also distributed access control, where the supervisory work of granting and revoking access rights is administered by multiple agents, is modelled by team automata. In [11] a shared application example is discussed and an architecture for the GROVE document editor is presented using the design possibilities provided by team automata.

Team automata can be classified on basis of the properties of their transition relation or by imposing conditions on the construction of the transition relation. One way of viewing the team automaton model is as having a two-way mechanism to model a spectrum of group interactions. On the one hand there are master-slave synchronizations, in which output as a master may force the concurrent execution of a corresponding input action. They can be used to model asynchronous cooperation, as in workflow systems, to enact certain modules. On the other hand there are peer-to-peer synchronizations, in which all participants are considered equal. They model the group collaboration aspect that frequently occurs in synchronous groupware. Thus, exact descriptions can be given of certain groupware notions which may otherwise have an ambiguous interpretation. For example the descriptions of cooperation and collaboration of team automata as proposed in [11] can be given formal, mathematically precise definitions using the concepts of master-slave and peer-to-peer synchronizations. In fact, more and finer distinctions can be made on basis of the two different formalizations (ai or si) of the obligation for components to participate in the execution of a certain action. Combinations of cooperation and collaboration, called hybrids in [11], are also easy to define.

Our comparison with I/O automata shows that team automata are more flexible. They impose less requirements on the role of the actions and have the option to choose a transition relation. Hence it is possible, e.g., for components to be unavailable for communications and to model output peer-to-peer synchronizations. Of course one has to pay for this. To describe the behaviour of a team automaton in terms of its component behaviours is much more difficult when one cannot assume that the composition is based on maximal sets of ai synchronizations, as is the case for I/O automata. This is a topic currently under investigation for [4].

Team automata are distributed systems consisting of labelled transition systems which work together through synchronizations. Since the concept is so natural, there are many variants of these models (see, e.g., [26], [2], [21]) including state machine decomposable nets like ITNCs, each emphasizing a special aspect of sequential systems working together. Team automata contribute to this spectrum as argued above their flexibility and potential to capture and give a semantics to a broad range of synchronization protocols which may be hierarchical, globally defined, role dependent, state dependent or individually defined.

Acknowledgement

The author is very grateful to the other members of the team automata team, Maurice ter Beek, Clarence Ellis, and Grzegorz Rozenberg for the fruitful cooperation that has led to the notions and results discussed in this paper. Special thanks go to Maurice also for his help in compiling and editing its first (colloquium) version [6].

References

- [1] A. Arnold, Synchronized Behaviours of Processes and Rational Relations. *Acta Informatica* 17 (1982), 21 – 29. 315
- [2] A. Arnold, *Finite Transition Systems*. Prentice Hall, 1994. 315, 318
- [3] E. Badouel, Ph. Darondeau, D. Quichaud, and A. Tokmakoff, Modelling Dynamic Agent Systems with Cooperating Automata. Publication Interne 1253, Institut de Recherche en Informatique et Systèmes Aléatoires, Rennes, 1999. 315
- [4] M. H. ter Beek, *Team Automata*. Ph.D. thesis, Leiden University, in preparation. 296, 318
- [5] M. H. ter Beek, C. A. Ellis, J. Kleijn, and G. Rozenberg, Synchronizations in Team Automata for Groupware Systems. To appear in *Computer Supported Cooperative Work*. 296, 305
- [6] M. H. ter Beek, C. A. Ellis, J. Kleijn, and G. Rozenberg, Team Automata for CSCW. In *Proceedings of the 2nd International Colloquium on Petri Net Technologies for Modelling Communication Based Systems, Berlin, Germany* (H. Weber, H. Ehrig, and W. Reisig, eds), 2001, 1 – 20. 296, 318
- [7] M. H. ter Beek, C. A. Ellis, J. Kleijn, and G. Rozenberg, Team Automata for Spatial Access Control. In *Proceedings of the Seventh European Conference on Computer Supported Cooperative Work* (W. Prinz, M. Jarke, Y. Rogers, K. Schmidt, V. Wulf, eds.) Kluwer Academic Publishers, Dordrecht, 2001, 59 – 77. 296, 317
- [8] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM* 31, 3 (1984), 560 – 599. 300
- [9] V. Diekert and G. Rozenberg, *Book of Traces*, World Scientific, Singapore, 1995. 316
- [10] C. Duboc, Mixed Product and Asynchronous Automata. *Theoretical Computer Science* 42 (1986), 183 – 199. 309
- [11] C. A. Ellis, Team Automata for Groupware Systems. In *Proceedings of GROUP'97* (J. Clifford, B. Lindsday, and D. Maier, eds.), ACM Press, 1997, 415 – 424. 295, 297, 299, 300, 309, 311, 317, 318
- [12] R. Gawlick, R. Segala, F. F. Søgaaard-Andersen, and N. Lynch, Liveness in Timed and Untimed Systems. In *Proceedings of ICALP'94* (S. Abiteboul and E. Shamir, eds.), *Lecture Notes in Computer Science* 820, Springer-Verlag, Berlin, 1994, 166 – 177. 310
- [13] R. Janicki and P. E. Laurer, *Specification and Analysis of Concurrent Systems, The COSY Approach. EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, Berlin, 1992. 300, 309, 315
- [14] N. W. Keesmaat and H. C. M. Kleijn, Net-based Control versus Rational Control: The Relation between ITNC Vector Languages and Rational Relations. *Acta Informatica* 34 (1997), 23 – 57. 316
- [15] N. W. Keesmaat, H. C. M. Kleijn, and G. Rozenberg, Vector Controlled Concurrent Systems, Part I: Basic Classes. *Fundamenta Informaticae* 13 (1990), 275 – 316. 314, 316
- [16] N. W. Keesmaat, H. C. M. Kleijn, and G. Rozenberg, Vector Controlled Concurrent Systems, Part II: Comparisons. *Fundamenta Informaticae* 14 (1991), 1 – 38. 314
- [17] N. A. Lynch, *Distributed Algorithms*, Morgan Kaufmann Publishers, San Mateo, California, 1996. 310
- [18] N. A. Lynch and M. R. Tuttle, Hierarchical Correctness Proofs for Distributed Algorithms. In *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1987, 137 – 151. 310

- [19] N. A. Lynch and M. R. Tuttle, An Introduction to Input/Output Automata. *CWI Quarterly* 2,3 (1989), 219 – 246. (Also Technical Memo MIT/LCS/TM-373, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1988.) 295, 300, 309, 310
- [20] A. Mazurkiewicz, Basic Notions of Trace Theory. In *Lecture Notes in Computer Science* 354, Springer-Verlag, Berlin, 1989, 285 – 363. 316
- [21] R. Morin, Decompositions of Asynchronous Systems. In *Proceedings of CONCUR'98* (D. Sangiorgi and R. De Simone, eds), *Lecture Notes in Computer Science* 1466, Springer-Verlag, Berlin, 1998, 549 – 564. 309, 318
- [22] O. Müller, *A Verification Environment for I/O Automata Based on Formalized Meta-Theory*. Ph.D. thesis, Technische Universität München, 1998. 310
- [23] *Lectures on Petri Nets I: Basic Models* (W. Reisig and G. Rozenberg, eds.), *Lecture Notes in Computer Science* 1491, Springer-Verlag, Berlin, 1998. 315
- [24] *Lectures on Petri Nets II: Applications* (W. Reisig and G. Rozenberg, eds.), *Lecture Notes in Computer Science* 1492, Springer-Verlag, Berlin, 1998. 315
- [25] M. R. Tuttle, *Hierarchical Correctness Proofs for Distributed Algorithms*. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1987. (Also Technical Report MIT/LCS/TR-387, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1987.) 310, 312
- [26] W. Zielonka, Notes on Finite Asynchronous Automata. *R. A. I. R. O. — Inform. Théor. Appl.* 21 (1987), 99 – 135. 316, 318

Comparing Petri Net and Activity Diagram Variants for Workflow Modelling – A Quest for Reactive Petri Nets

Rik Eshuis* and Roel Wieringa

Department of Computer Science, University of Twente
P.O.Box 217, NL-7500 AE, Enschede, The Netherlands
`{eshuis,roelw}@cs.utwente.nl`

Abstract. Petri net variants are widely used as a workflow modelling technique. Recently, UML activity diagrams have been used for the same purpose, even though the syntax and semantics of activity diagrams has not been yet fully worked out. Nevertheless, activity diagrams seem very similar to Petri nets and on the surface, one may think that they are variants of each other. To substantiate or deny this claim, we need to formalise the intended semantics of activity diagrams and then compare this with various Petri net semantics. In previous papers we have defined two formal semantics for UML activity diagrams that are intended for workflow modelling. In this paper, we discuss the design choices that underlie these two semantics and investigate whether these design choices can be met in low-level and high-level Petri net semantics. We argue that the main difference between the Petri net semantics and our semantics of UML activity diagrams is that the Petri net semantics models resource usage of closed, active systems that are non-reactive, whereas our semantics of UML activity diagrams models open, reactive systems. Since workflow systems are open, reactive systems, we conclude that Petri nets cannot model workflows accurately, unless they are extended with a syntax and semantics for reactivity.

1 Introduction

Petri nets are a popular technique for modelling the control flow dimension of workflows. When modelling workflows, people tend to draw nodes that represent tasks or activities, and arrows between the nodes that represent sequencing of activities. The resulting diagrams look like Petri nets, and so Petri nets seem a natural technique for modelling workflows [2, 22]. The following arguments are often used to support this: Petri nets are graphical, they have a formal semantics, they can express most of the desirable routing constructs, there is an abundance of analysis techniques for proving properties about them, and finally they are vendor-independent. Most of these arguments do not refer to the domain

* Supported by NWO/SION, grant nr. 612-62-02 (DAEMON).

of workflow modelling (only the routing argument does) and point out advantages of Petri nets in general. Moreover, since Petri nets already existed before workflow management systems were invented, their semantics is not specifically intended for workflow modelling. So none of these arguments state why and how Petri nets are useful for workflow modelling. This is unsatisfactory for analysis purposes, since analysing a Petri net workflow model presupposes that the Petri net models the real workflow accurately.

Recently, UML activity diagrams [56] have also been used for workflow modelling. They too are graphical, use bubbles and arrows, are vendor-independent, and can express most desirable routing constructs. Unfortunately, the OMG semantics [56] is not formal (nor precise), and it is not intended for workflow modelling [28]. We therefore defined two semantics for UML activity diagrams that are intended for workflow modelling [27, 28]. The goal is to use these semantics for analysing workflow models in activity diagram notation by means of model checking [16, 25]. The first semantics is a high-level semantics, based upon the STATEMATE semantics of statecharts [37], that is easy to analyse (both for a computer and for a person) but somewhat abstract. By contrast, the second semantics is low-level and resembles both the behaviour of an abstract workflow system and the informal OMG semantics of UML state machines, but it is more difficult to analyse than the first semantics. We have implemented verification support using model checking for the first semantics in our diagram editing tool TCM [17, 25].

In this paper we discuss the design choices that underlie both our formal execution semantics. Since our purpose is to make analysis of activity diagram workflow models possible, the semantics must be an accurate representation of workflow behaviour. Our design choices are therefore motivated in terms of the domain of workflow modelling. Using these choices as a yard stick, we investigate how well Petri nets can model some important aspects of workflow modelling. We hope this provides relevant arguments for and against the claim that Petri nets are useful for workflow modelling.

This approach may seem subjective, since other persons might make other design choices, and consequently they might draw other conclusions about the suitability of Petri nets for workflow modelling. However, we think that the choices we have made in our semantics are reasonable, because they are motivated by the domain of workflow modelling. Even if one does not agree with the choices we made, our discussion gives – we hope – more insight in possible answers to the question what actually is a Petri net [20].

Our most important design choice is that the semantics for activity diagrams must be reactive. The token-game semantics, which is characteristic for Petri nets, does not represent reactivity, which is characteristic of workflow systems. A Petri net transition can fire if all its input places are in the current marking [48, 51]. But in a reactive system a transition can be taken (fired) if all its source nodes (input places) are in the current configuration (marking) *and* its trigger event occurs [37, 56]. This trigger event is an event in the environment of the system, that the system will react to by taking the transition. Although Petri

nets in our view are not reactive, we will study different ways of *simulating* reactive behaviour in different Petri net variants.

In the sequel, we presuppose some basic knowledge of Petri nets and high-level Petri nets (see e.g. [40, 48, 51, 53, 54]). We have looked at Petri net variants that are traditionally used to model and analyse workflows, namely Workflow Nets [2, 3], Information Control Nets [22], INCOME/WF [49], FunSoft nets [18], MILANO WFMS [9]. Next, we have looked at Petri net variants that are not specifically tailored towards workflow modelling but nevertheless can be useful: Open Nets [10], Petri nets with synchronous communication [15], Signal-Event Nets [36, 29], Contextual Nets [47, 30], Zero-Safe nets [12], and several variants of Object-Oriented Petri Nets [8, 46]. More information about some of these references can be found in recent overviews and collections about the use of Petri nets for workflow modelling [1, 55]. A comparison of our semantics with other formal modelling techniques (in particular STATEMATE [37, 59]) can be found elsewhere [26].

Structure. We start by explaining some characteristics of workflows and workflow systems in more detail. In Section 3 we discuss our two activity diagram semantics and the design choices that underlie these semantics. We also discuss the properties of every basic statechart step semantics, used in both STATEMATE [37] and UML [56]. We have adopted these properties as well in both our formal semantics. In Section 4 we study whether and how our semantics can be simulated in Petri nets. In particular, we discuss whether and how the statechart step semantics can be modelled in Petri nets. We end with conclusions.

2 Workflow

This section is based on literature (amongst others [2, 45, 60]) and several case studies that we did. A *workflow* is a set of business activities that are ordered according to a set of procedural rules to deliver a service. A workflow model (also known as workflow specification) is the definition of a workflow. An instance of a workflow is called a *case*. In a case, *work items* are passed and manipulated. An example of a case is the process that handles the insurance claim of John Smith. An example of a work item is the claim form of John Smith. The definition, creation, and management of workflow instances is done by a workflow management system (WFMS), on the basis of workflow models.

In general, two important dimensions of workflows are the control-flow dimension and the resource dimension [2, 45]. The control-flow dimension concerns the ordering of activities (or tasks) in time (what has to be done). The resource dimension concerns the organisational structure (who has to do it). Since both Petri nets and UML activity diagrams only model the control-flow dimension, we here focus on modelling the control-flow dimension of workflows. When we use the term workflow model, we refer to a model that describes the control-flow dimension.

Activities are done by *actors*. An *activity* is an amount of work that is uninterruptible and that is performed in a non-zero span of time by an actor. In an

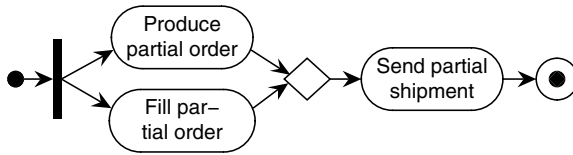


Fig. 1. Multiple simultaneous instantiations of Send partial shipment

activity, *case attributes* are updated. Case attributes are work items and other relevant data. The case may be distributed over several actors. Each distributed part of the case has a *local state*. There are two kinds of local state.

- In an *activity state* an actor is executing an activity in a part of the case. For every activity there should be at least one activity state, but different activity states can represent execution of the same activity.
- In a *wait state*, the case is waiting for some external event or temporal event. A special event is that the actor who has to do the next activity becomes available.

We allow multiple instances of states to be active at the same time. For example, the activity diagram in Fig. 1 shows two parallel activities *Produce partial order* and *Fill partial order* that each trigger an instance of *Send partial shipment* (the notation is explained in the next section). The result is that two instances of *Send partial shipment* may be active at the same time. The *global state* of the case is therefore a multiset (rather than a set) of the local states of the distributed parts of the case.

Actors are people or machines. Actors are grouped according to roles. A *role* is a set of characteristics of actors. A role can refer to skills, responsibility, or authority for people, and it can refer to computing capabilities for machines [45, 61]. Roles link actors and activities. The modelling of actors and roles, and the connection with workflow models falls outside the scope of this paper.

The *effect* of an activity can be constrained declaratively with a pre- and post-condition. The effect cannot be specified fully since execution of the activity falls outside the scope of the WFMS. The pre-condition also functions as guard: as long as it is false, the activity cannot be performed.

The WFMC [61] specifies four possible ordering relationships between activities: *sequence*, *choice*, *parallelism* and *iteration*. Van der Aalst et al. identified more ordering relationships [7]. And, to facilitate readability and re-use of workflow definitions, an ordered set of activities can be grouped into one *compound activity*. A compound activity can be used in other workflow definitions. A non-compound activity is called an *atomic activity*.

Architecture (Fig. 2). The following architecture is based upon amongst others [14, 34, 45, 60]. A workflow system (WFS), which is a WFMS instantiated with one or more workflow models, connects a database system and several applications that are used by actors to do work for the cases. In this paper we

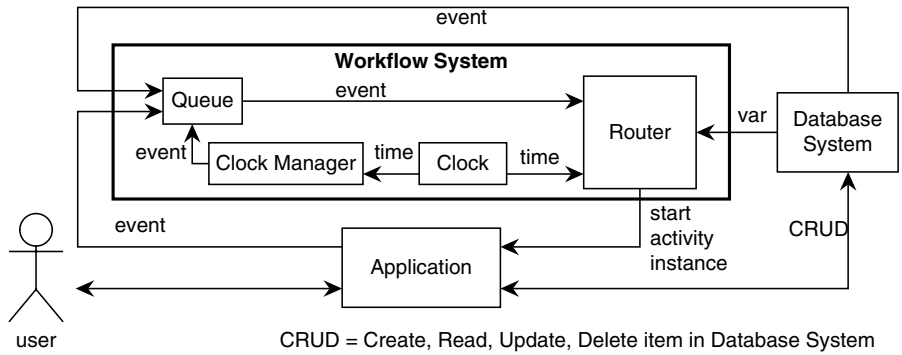


Fig. 2. Abstract workflow system architecture

assume that the WFS controls a single case (a generalisation to a WFS that controls multiple cases is straightforward). The main components of the WFS are the queue and the router. The environment interacts with the WFS by putting events in the queue. On basis of these events and the current state of the case, the router component of the WFS routes the case as prescribed by the workflow model of the case. As a result, some new activity instances can be started. Note that the case attributes are updated during an activity by the actors, not by the WFS. For example, an actor may update a work item by editing it with a word processor. The transitions between the states (active or waiting), on the other hand, are performed by the WFS, not by an actor. By taking these transitions the WFS routes the case. All attributes of a case are stored in the database. The state of the case is maintained by the WFS itself. Scheduled timeouts are maintained and raised by the clock manager on basis of the internal clock.

3 UML Activity Diagrams

Syntax. We explain the syntax by means of a small example. In Fig. 3 the workflow of “Processing Complaints” is shown (converted from a Petri net model in [2]; see Fig. 7 below). Ovals represent activity states and rounded rectangles represent wait states. In an activity state, some activity is busy executing whereas in a wait state, an external event is waited for, e.g. a deadline must occur, or some third party must send some information. Wait states are also used if the current parallel branch needs to synchronise with another parallel branch. An activity state is called an action state in UML [56]. The workflow starts in the black dot (the initial state) and ends at the bull’s eye (the end state). A bar represents a fork (more than one outgoing edge) or a join (more than one incoming edge). A diamond represents a choice (more than one outgoing edge) or a merging of different choices (more than one incoming edge).

State nodes are linked by directed edges, that express sequence. An edge can be labelled by $e[g]/a$ where e is an event expression, g a guard expression,

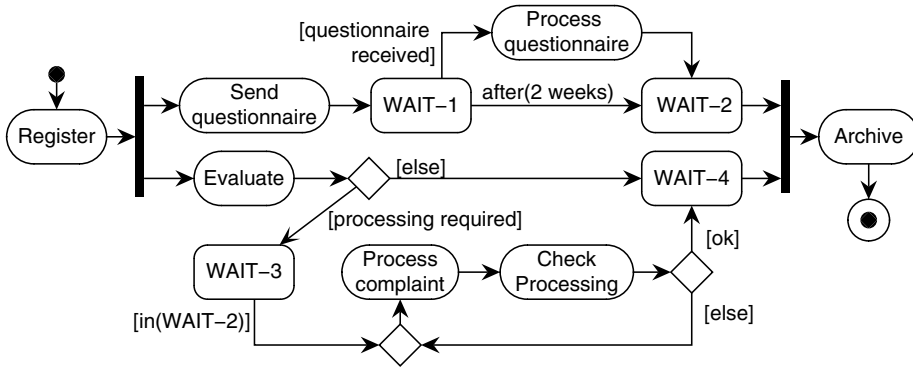


Fig. 3. Activity diagram for “Processing Complaints” workflow

and a an action expression. Each of these three components is optional. The meaning of these labels is that in order for the edge to be taken, event e must have occurred and guard g is true. When the edge is taken, the system performs action a . An edge that leaves an activity state is implicitly labelled with the completion event of the corresponding activity. We forbid that an edge that leaves an activity state has any other event expression in its label, since that would denote an interrupt, whereas an activity cannot be interrupted, since it is atomic. A special group of event expressions are the temporal event expressions, e.g. `after(2 weeks)` in Fig. 3 (which means that 2 weeks after state `WAIT-1` is entered the corresponding edge can be taken). A guard expression can refer to variables of the activity graph. The variables of an activity graph are booleans, integers and strings. Special guard expressions are the `in` and `else` predicates. Predicate `in(node name)` is true if and only if the system is in state `node name`. Predicate `else` can only be used to label an edge that leaves a choice state node (represented by diamond). It abbreviates the negation of the disjunction of the guard labels of the other edges that leave the choice node. For example, the `else` predicate used immediately after `Check processing` abbreviates `not ok`. The only action expressions we allow are (sequences of) send event actions to specify event generation. We do not allow other action expressions in an edge label since these would change the case attributes, which we do not want, since case attributes are changed by actors, not by the WFS.

Semantics. Our semantics is based upon the following line of reasoning. We use a UML activity diagram as a workflow model. A workflow model prescribes how a workflow system should behave. Hence, a UML activity diagram prescribes how a WFS should behave. We therefore motivate and define our execution semantics in terms of workflow systems.

In our opinion, workflow systems have the following characteristics.

1. **A WFS Is Reactive.** A *reactive* system runs in parallel with its environment and responds or reacts to input events by creating certain desirable effects in the environment [38, 58]. For a WFS, characteristic input events are activity termination events, in Fig. 3 for example that the **Register** activity terminates. And characteristic desirable effects for a WFS are the enabling of new activity instances.
2. **A WFS Has Coordination Functionality.** A WFS does not execute the activities themselves, but it merely coordinates the execution of the activities by the actors (people or machines). For example, in Fig. 3 the WFS does not register the complaint itself, but merely tells the relevant actors that one of them can start registering the complaint. Case attributes are only changed in activities by actors, not by the WFS.

Each semantics is a mapping of a syntactic domain into a semantic domain. The semantic domain we use in this paper is that of a run (to be precise, a set of runs). A *run* (or a trace) is a sequence of states connected by state changes. We assume that state changes are instantaneous. So time can only elapse in a state.

Since a run is a possible behaviour of a WFS, states of the run are states of the WFS. Components of a state of a run are:

- the state of the case (i.e., which states in the activity diagram are active, possibly multiple times),
- the queue of input events of the WFS,
- the case attributes and their values, and
- the scheduled time-outs and the value of the global clock.

A queue of input events is needed because of the first characteristic: a WFS is a reactive system. In a reactive system state changes are caused by input events. This means that the WFS must have some interface with the environment to observe the input events. We therefore use an input queue in which events are kept. The case attributes are needed to evaluate the guard conditions on the edges, i.e., they are only used for routing the case.

The second WFS characteristic, coordination, has several implications. First, activities are done by the environment in states of the WFS, i.e., during an activity state the WFS waits for an activity to complete (see Section 2). Second, an activity is specified declaratively, in particular its postcondition. An imperative specification would imply that the WFS does the activity. But the outcome of an activity is not computed by the WFS. Third, in a reaction case attributes are not changed. Instead, changing (updating) of case attributes is done by the environment. Also, the WFS does not maintain the case attributes, this is done by the environment (database). But the WFS must ensure that no two interfering activities are active simultaneously. Two activities interfere with each other if they update the same case attribute. The WFS ensures non-interference by not routing to a state in which some activities interfere with each other.

Within this general picture, still a wide variety of semantics for activity diagrams can be chosen. In previous work, we have defined two semantics. The first

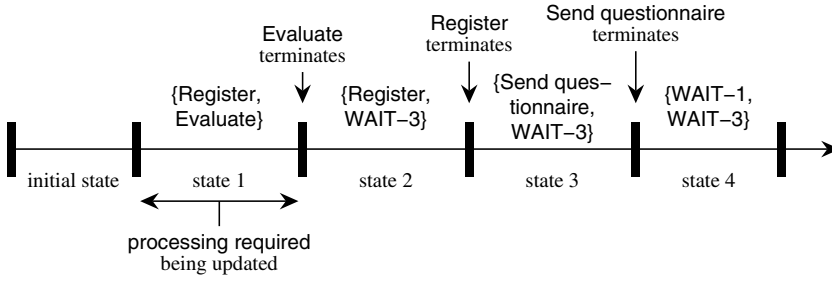


Fig. 4. Example run in requirements-level semantics

one is a *requirements-level semantics* [27] that is based upon the STATEMATE semantics of statecharts [37]. In the requirements-level semantics, the WFS is considered as a black box. In specifying requirements for the WFS, we are interested in qualitative requirements (what should be done), but not in quantitative requirements (how well it should be done), e.g. how fast a response is. We therefore abstract away from internal implementation details of the WFS. The best way to do this is to adopt the perfect synchrony hypothesis [11]. For a WFS, this hypothesis states that the WFS starts reacting to events immediately when it receives them, and also that the WFS reacts infinitely fast to these events. In a reaction, therefore, the whole input queue is read and the case is immediately routed, i.e., the state of the case is updated, and the input queue is reset again. Note that in this semantics the queue is actually a set of input events. Figure 4 shows an example run of the activity diagram in Fig. 3 under this semantics. In each state, the set of busy activities is shown.

In the second semantics [28] the perfect synchrony hypothesis is dropped. So a reaction of the WFS takes time, and input events are not immediately reacted to. We call this semantics the *implementation-level semantics*. This semantics stays close to the informal UML definition of state machines [56] (underlying UML statecharts) and the architecture of workflow systems [14, 45, 60, 34]. In the implementation-level semantics, the WFS is considered as a white box, consisting of the components shown in Fig. 2. The Router component is responsible for producing the desired reaction: routing the case to the new state, enabling some new activity instances to start. The Router component, however, has limited capacity. It processes one event at a time (rather than arbitrarily many as in the requirements-level semantics) and it takes time to process an event (whereas in the requirements-level semantics the WFS is infinitely fast). When the Router starts routing, it picks some input event from the queue. When it finishes routing the case, it updates the state of the case, it enables some new activities to start, it schedules some new timeouts and it removes some scheduled timeouts, because they have become irrelevant in the new state of the case. Next, the Router starts processing the next event from the queue. Since the next input events might have arrived while the Router was busy with a reaction, the content of

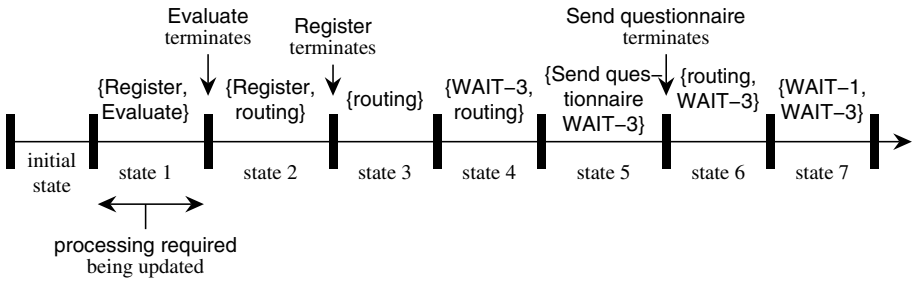


Fig. 5. Example run in implementation-level semantics

the queue might have changed during routing (whereas this is impossible in the requirements-level semantics, since there routing is instantaneous). Figure 5 shows an example run of the activity diagram in Fig. 3 under this semantics. In this figure, the term *routing* denotes that the Router is busy.

Evaluating the two semantics, the requirements-level semantics is easy to analyse, but somewhat abstract, whereas the implementation-level semantics is more concrete, but more difficult to analyse, both for a workflow designer as for a verification tool. The reason for this is that in the implementation-level semantics there is a delay between the occurrence of an event and the subsequent reaction of the WFS to that event occurrence, whereas in the requirements-level semantics, there is no such delay. For example, the run in Fig. 5 is harder to match with the activity diagram than the run in Fig. 4. Moreover, the implementation-level semantics is more difficult to analyse for a verification tool, because there will be more states due to the delay in response to event occurrences.

In future work, we intend to focus on analysis of functional (logical) properties of workflow models, for example the absence of deadlock. For such properties, it does not matter whether the requirements-level semantics or implementation-level semantics is chosen: if a workflow model contains for example a deadlock in one semantics, it also will have the deadlock in the other semantics and vice versa. So we can use the requirements-level semantics for analysis of such properties with the assurance that the analysis result will also hold when the workflow model is executed under the implementation-level semantics.

Note that we do not specify how the environment behaves. In general, the exact behaviour of the environment is unknown. In our model checking semantics, we have simply assumed that the environment can behave in every possible way, i.e., chaotically, but it must respect the dependencies between value change input events (see Section 4 for more details). For analysis purposes, we assume that the environment behaves in a fair way [25].

Step Semantics. The key part in both of the previous semantics is the execution of a step. A step is a collection of edges that are enabled in a certain state. By taking a step, the system reacts to events and routes the case to a new state.

In both semantics, we have adopted the same step semantics of statecharts. We here give a brief introduction to and motivation for this step semantics. More details can be found elsewhere [27, 28, 26].

We consider two cases. The first one is the basic case, in which action expressions on edges are not considered. Although statecharts are (in)famous for the numerous semantics invented for them, all semantics agree upon the definition of a step for the basic case that we present below. In the second case, event generation is taken into account. Adding event generation to the basic case, divides the group of statechart step semantics in two.

The Basic Case. In the basic case, (1) we do not consider internal event generation by the system in transitions, (2) an edge can be labelled with a single event only, and (3) we do not consider priority of transitions. Below we will add the first feature. The second feature is enforced by the UML definition [56]. The third feature is omitted from the current semantics of activity diagram, but can be added without a problem.

In the basic case, all the statechart step semantics exhibit the following three properties. This includes the most well-known ones of Harel and Naamad [37], implemented in the STATEMATE toolset as well as the different UML statechart step semantics [56], and the fixpoint semantics by Pnueli and Shalev [52].

First we list two of the three properties present in every statechart step semantics.

- **Events Can Occur Simultaneously.** The alternative would be to assume that no two events can occur at the same time. There are two reasons for rejecting this alternative. First, although the chance of two events occurring simultaneously is rather small, it is not equal to zero. Second, the reactive system (WFS) will respond to events by inspecting the contents of the queue. If no two events can occur simultaneously, the rate at which events occur in the environment must be slower than the rate at which the WFS reads input events (sampling rate). We do not want to impose such a restriction upon the environment and therefore do not make such an assumption. From this choice, it follows that two event occurrences are either simultaneous, or some time elapses between them. Note that this assumption is also made in our implementation-level semantics, although the Router will process only one event at a time.
- **Events Live for the Duration of a Step Only.** During execution of the system, the event queue is filled with events. The system reads the events from the event queue and reacts to them. There should however be some removal policy. If an event is not removed after it is processed, it would continue to have an effect, which is undesirable. Since the result of the event occurrences is the taking of a step, the events should be removed after this response has finished.

If the system processes events from the event queue, it reacts to these events by taking a *step*. In every statechart step semantics, a step is a bag of enabled edges that must be consistent and maximal. We explain these notions shortly.

First, the *configuration* of a system is the bag of nodes that are currently active (in Petri net terminology: a marking). An edge is *enabled* iff its sources are contained in the current configuration, its trigger events is being processed now, and its guard condition is true. A bag of enabled edges is *consistent* iff all edges can be taken simultaneously, i.e., the union of their sources is contained in the current configuration. Finally, the bag must be *maximal*, i.e., adding another enabled edge makes the resulting bag inconsistent.

The above definition of a step is a generalisation of the statechart step semantics, since in statechart the configuration is always a set, rather than a bag, and consequently, steps in a statechart are always sets, rather than bags.

In STATEMATE and in UML there is in addition a priority constraint, stating that edges with higher priority should be added first to a step. The precise definition of when an edge has priority over another one differs [42].

We now explain the third property of basic statechart steps in the basic case.

- **Steps Are Maximal.** Not imposing this constraint would imply that some edges that are enabled would not have to be part of the step, so would not have to be taken. Since an event is removed from the input after the subsequent step has been taken, this would mean that some input events would not cause all their effects, although, according to the workflow model, they *should* have these effect (namely all enabled edges should be taken). In other words, then the WFS would not react fully to these input events. That is why we require that a step be maximal.

Note that the maximality constraint is motivated in terms of reactive systems. For active systems, such a motivation would no longer hold, since then the system can decide itself what to do and does not have to do as much as possible. This gives an explanation for the fact that in Petri nets the maximality constraint is usually not adopted.

We emphasise that statechart steps in the STATEMATE semantics, UML semantics and the fixpoint semantics all share these three properties.

Event Generation. Next, we extend the basic case by allowing for *event generation* by the system. An event can be generated by taking an edge. Figure 6 gives an example (send actions follow the slash). Some of the edges have an identifier for ease of reference. The label on the edge t1 states that if the edge is taken (because e occurs and WAIT-1 is active), then event f is generated. Please note that communication by means of event-generation is always asynchronously, not synchronously (rendez-vous or hand-shaking).

There are two different ways of interpreting event generation. The first one is to let the generated events have an effect in the current step (chosen in the fixpoint semantics [52]), the second one is to let the generated events have an effect in the next step (chosen in the STATEMATE semantics [37]). To illustrate the differences between these two options, suppose in Fig. 6 the current configuration is [WAIT-1, WAIT-2] and event e occurs. If edge t1 is taken, then according to the fixpoint semantics event f is immediately available and consequently edge t3 can be taken simultaneously with edge t1. Whereas in the STATEMATE semantics,

event f can only be sensed *after* the step in which it is generated is taken, so *after* edge $t1$ is taken. Consequently, if the current configuration is $\{\text{WAIT-1}, \text{WAIT-2}\}$, and event e occurs, in the fixpoint semantics either step $\{t1, t2\}$ or $\{t1, t3\}$ is taken, but in the STATEMATE semantics, step $\{t1, t2\}$ is taken. The step $\{t1, t3\}$ is counter intuitive here, since it seems that event e is ignored in node WAIT-2. So there are circumstances in which the fixpoint semantics computes a counter intuitive step (this was first pointed out by Leveson et al. [43] using a similar example, but they mistakenly attribute the fixpoint semantics to STATEMATE). That is why in practice the STATEMATE approach is taken, even in the UML, where events are called signals. (We do not have operation calls in our activity diagram semantics.) We have adopted the STATEMATE interpretation for event generation as well, since it is also adopted by UML. As an aside, note in this interpretation too, there are anomalies. One may for example get infinite loops in which events are generated for ever, because some events cause each other to occur [43].

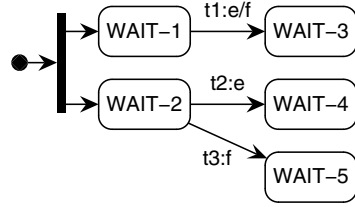


Fig. 6. Event generation

The State of the Practice. We do not know of any commercial WFMS that allows for the specification of workflow models using UML activity diagrams. But few of the current commercial workflow systems offer some support for modelling events [13]. We therefore expect that the constructs of our semantics related to events will be hard to express in workflow models of existing commercial WFMSs. On the other hand, our event broadcast semantics, in which one event can trigger more one edge, is similar to the publish-subscribe notification mechanism used in middleware application and recently adopted in the industry standard for workflow interoperability [50], defined by OMG and WfMC. Also, in active database systems [57] an execution semantics for rules is adopted that is similar to the semantics for edges that we use. In particular, in active databases a generated event has an effect in the next step, not in the current step. So in active databases also the STATEMATE interpretation is chosen (although the link with statecharts is not made by for example Widom and Ceri [57]). Also, the possibility of nontermination of the rule processing algorithm due to rules that trigger each other, is a well known feature of active databases [57].

Recently, UML activity diagrams have been proposed to model e-business services in e-business standards like ebXML [21]. We expect that process management tools that support e-business services will use UML activity diagrams. In for example ebXML, the event features of activity diagrams are used quite extensively: events are the standard means of communication between different business partners. Events are also used quite extensively in business modelling, especially in UML-based approaches, for example [24]. In academia, several WFMS research prototypes use event-based workflow models (e.g. [13, 33, 35, 59]), often inspired by active databases [57].

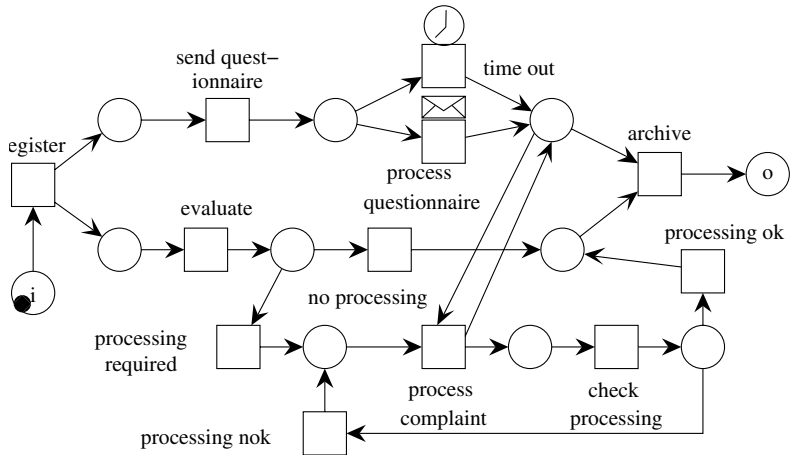


Fig. 7. Petri net for “Processing Complaints” workflow [2]

4 Modelling Workflows with Petri Nets

We now investigate how several – what we think are – important aspects of workflow models are modelled in Petri nets. We will compare the Petri net workflow models with our two semantics of UML activity diagrams. We take our requirements-level semantics as point of comparison, since it most resembles the Petri net semantics. At the end of this Section, we will discuss how the implementation-level semantics can be modelled in Petri nets. In order to make a fair comparison we assume that a Petri net models a WFS too.

Remark on terminology: from now on, we will use the standard Petri net terminology of place (corresponds to state node), transition (corresponds to edge), and marking (corresponds to configuration). By “step” we mean a statechart step, unless stated otherwise.

4.1 Modelling Events

Several researchers that use Petri nets for workflow modelling have recognised the importance of input events for workflow modelling ([2, 41]), even though using the a different name: ‘trigger’. Figure 7, taken from Van der Aalst [2], presents a typically example of the use of input events in a Petri net. (This Petri net models the same workflow as the activity diagram in Fig. 3.) The envelope and the clock denote external and temporal trigger events respectively.

Unfortunately, although the importance of input events is recognised, hardly ever a semantics is given for them. Van der Aalst [3] gives an interesting motivation for abstracting from events for analysis purposes, that we will discuss in Section 4.6. But first we study two approaches to model events in ordinary Petri nets and compare both approaches with our semantics of input events.

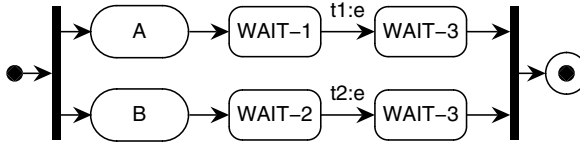


Fig. 8. Event broadcasting

Event as Token. For each input event a place is defined. The place represents a kind of interface with the environment. If the interface place is filled with a token, the input event occurs, otherwise it does not occur. The interface place is connected with all transitions that are triggered by the input event.

This is the approach taken in Trigger Modelling [41]; it is also suggested as an appropriate semantics for trigger events in Workflow Nets [2]. In these approaches, the environment is not specified, but the suggestion is made that the environment fills the interface places spontaneously, but no formal semantics is presented. Open nets [10] gives a formal semantics for nets with interface places, which could be used for Trigger Modelling and Workflow Nets.

One important difference of the event-as-token approach with our semantics of events is that in the event-as-token approach one event occurrence triggers at most one transition whereas in our semantics one event can trigger more than one transition (edge). This is because we have event broadcasting in our semantics, rather than point to point communication. For example, in Fig. 6 one occurrence of event e can trigger the two edges $t1$ and $t2$ simultaneously. Since in the standard Petri net semantics, firing a transition implies that its input tokens are consumed, in the standard Petri net semantics only one transition can fire because of one event occurrence.

One might wonder whether event broadcasting is desirable. In other words, isn't the standard Petri net interpretation of consuming events, so having an event trigger at most one transition, better? We think event broadcasting is desirable for the following reasons. First, if an event would trigger a single transition only, the event would not have all the effect that is specified in the workflow model. For example, a cancel event that stops a workflow would be awkward to model. In the event-as-token approach, a cancel event could only stop one parallel branch, whereas in our semantics an event is global for the whole workflow and there a cancel event can stop the whole workflow. To cancel a workflow in the event-as-token approach, for every parallel branch a separate cancel event needs to be generated.

Second, the broadcast mechanism is used quite extensively in the field of workflow systems. Several non-Petri net based WFMS prototypes [13, 33, 35, 59]) also use a broadcast semantics in their workflow models. The industry standard for workflow interoperability [50], defined by OMG and WfMC, uses a so-called publish-subscribe notification mechanism, which is similar to our broadcast semantics. An exception are XML and EDI based workflow specifications, which

currently only use point-to-point communication between business partners. However, some of these approaches [21] will adopt publish-subscribe notification in the near future. Also, these approaches do not specify what communication mechanism are used within an organisation, since it falls outside the scope of these frameworks. So even in these approaches, a broadcast mechanism can be used for intra-organisational communication.

Third, we observe that our broadcast semantics is equivalent to a point-to-point semantics if all the used event names in the activity diagram are unique. But, as we will explain next, it is not possible to fully capture the broadcast semantics with point-to-point communication, since the exact addressee is not always known at design time and may depend upon the current state of the case.

There are several ways to simulate the effect of event broadcasting in Petri nets. The most obvious one is to use transition fusion and glue the edges with the same event label together. Although this would work for the example in Fig. 6, this is only a partial solution, for two reasons. The first one is that it depends upon the current configuration (marking) of the activity diagram whether or not two edges are taken simultaneously. For example, in Fig. 8 the two edges are only taken simultaneously if the current configuration is [WAIT-1, WAIT-2]. Otherwise, if for example the configuration is [WAIT-1, B] and e occurs, then only t_1 is taken and configuration [WAIT-3, B] is reached. So, only at run-time it is known which edges need to be fused together, whereas transition fusing is applied at design time. Second, applying transition fusion at design time does not solve this problem, since the original edges cannot be left out. For example, if in Fig. 8 edges t_1 and t_2 are fused together into t_{12} , then edges t_1 and t_2 must remain in the model, since it is possible that either one of them is taken separately from the other. Consequently, if the current configuration is [WAIT-1, WAIT-2] and event e occurs, it still might be possible that only say t_1 is taken, and not the fused edge t_{12} .

Another possible way to simulate event broadcasting is to fill the interface place with as many tokens as needed to prevent that a transition cannot fire because of a lack of tokens. But the exact number of tokens that is needed is not known beforehand, since the number of transitions to be fired depends upon the current WFS state. Consequently, a lot of spare tokens would have to be introduced. This blurs the difference between two occurrences of the same event at different times and two copies of the same event occurrence. Although this could be resolved by time stamping tokens, the resulting semantics would be overly complex and more involved than the statechart step semantics.

A better alternative is to model the control flow between an interface place and a transition that it triggers as a read arc [47], also known as context relation. A read arc from a place to a transition means that although a token must be present in the place to let the transition fire, this token is not consumed. (A read arc from a transition to a place is impossible.) Technically, a flow relation is added to the usual Petri net semantics that specifies the read arcs [47].

But even using read arcs, the event semantics we give to value change events cannot be adequately modelled. A value change event is the event that a boolean

condition on some variable x becomes true, e.g. $[x > 10]$. In our semantics, if there are two value change events referring to the same variable, e.g. $[x > 10]$ and $[x > 15]$ then there *might* be a dependency between them, e.g. if $[x > 15]$ occurs then $[x > 10]$ *might* also occur, but not necessarily. For example, if x changes from 11 to 16, then $[x > 15]$ occurs but $[x > 10]$ does not (since x already was greater than 10). But if x changes from 9 to 16, both events do occur simultaneously. This cannot be modelled faithfully in the event-as-token approach (even a dependency relation between interface places, stating something like “if that interface place is filled, this one must be filled as well” does not solve this problem).

A final drawback of the event-as-token approach is that the resulting Petri net looks like ravioli, since the place where the input token e resides must be connected to all transitions that are triggered by e .

Event as Transition. In Petri nets, one can simulate an event by labelling a transition with the event name and interpret the firing of the transition as the event occurrence. By specifying synchronisation constraints [15, 29] between the event transition and the system transitions, it can be specified that an event occurrence triggers a system transition. Note, however, that then the environment is being modelled explicitly, rather than implicitly as in the event-as-token approach. In other words, the whole Petri net is now a model of both the environment and the WFS, rather than of the WFS only.

One advantage of this semantics is that it is very easy to specify that one event occurrence can trigger more than one system transition, since the synchronisation constraint is specified as just a relation between transitions.

But, as in the event-as-token approach, the dependency between value change events cannot be modelled faithfully, because the presence of the dependency depends upon the previous value of the corresponding variable.

Conclusion. We conclude that in both the event-as-token and the event-as-transition approach, we cannot model our full event semantics, since the dependency between value change events cannot be modelled completely. But both the combination of open nets with read arcs, and nets with synchronisation between transition are certainly steps in the right direction. In the next subsection, we will study how well the statechart step semantics can be modelled using these two approaches.

4.2 Modelling Steps

In the previous subsection we identified the two ways, open nets with read arcs, and nets with synchronisation constraints between transitions, that come closest to our event semantics (but nevertheless they are still different from it). We now study whether and how well the statechart step semantics can be modelled in these approaches. We focus both on the basic statechart step semantics and the event generation semantics. For each approach, we study whether the three properties of the basic statechart step semantics are met or not. And if they are not met, we study whether they *can* be met.

Event as Token. If we take the Petri net step semantics, it is no problem in this approach to model that events can occur simultaneously. But in the event-as-token approach, it is difficult to specify that events live for the duration of one step only, without changing the semantics of open nets at this point. The removal of an event occurrence has to be modelled by a separate transition that removes the token from the interface place. But the sequence ‘event occurrence-system reaction-event removal’ which is key part of the basic statechart step semantics, is not part of the standard Petri net semantics. It seems to us that it is impossible to model this sequence using standard Petri net semantics, since in this semantics any sequence of transitions, obeying the firing rules, is allowed. So, it could be possible that under the standard Petri semantics an event lives longer than a step.

Recently, a new Petri net variant, called zero safe nets [12], has been proposed that seems a good starting point for modelling the statechart step semantics. In zero safe nets, some places, called zero places, represent unobservable system states. A marking in which one or more zero places are filled is unstable, otherwise it is stable. During execution, the system moves from one stable marking (in which zero places are not filled) to another stable marking via a sequence of unstable markings. By modelling event places as zero places, the statechart step semantics can be simulated to some extent. Still there is a difference: zero safe nets have a constraint that all stable tokens present at the begin stable marking must be consumed during the sequence. For the statechart step semantics, this would mean that relevant transitions must fire immediately, which is of course not true.

Finally, the constraint that steps are maximal is not present in standard Petri net semantics. Rather, steps in the Petri net semantics can be any consistent subset of the bag of enabled transitions. Of course, the maximality constraint could be added without a problem (like incidentally done by some authors, e.g. [29]), but it does not seem very intuitive for the standard Petri net semantics. In fact, Foremniak and Starke [29] have considerably changed the standard Petri net semantics. We discuss their approach in more detail below.

Event as Transition. There are several Petri net variants that have incorporated synchronisation between transitions in their models [15, 8, 29]. The work of Christensen and Hansen [15] introduces the concept of synchronous transitions. They focus on symmetric synchronisation. Object-oriented Petri nets [8, 46] use both symmetric and asymmetric synchronisation between transitions, i.e., one transition has the initiative, the other one follows. All these references stick to the standard interleaving semantics, which differs considerably from the statechart step semantics, amongst others because the maximality constraint is not required.

Finally, in signal-event nets [36, 29] the standard Petri net step semantics is abandoned in favour of a semantics in which also a maximality constraint is adopted. Signal-event nets are introduced by Hanisch and Lüder [36] in order to model discrete event systems. To model a discrete event system, both the behaviour of an uncontrolled plant and of a controller that guides the behaviour

of the plant is modelled. Hanisch and Lüder argue that discrete event systems cannot be faithfully modelled using ordinary Petri nets. Discrete event systems are an excellent example of reactive systems: the controller must react to the behaviour of the plant and it does this in order to maintain the plant in a desired state. It is therefore interesting to note the similarities (and differences) between the execution semantics of signal-event nets and that of statecharts. Foremniak and Starke [29] introduce an execution semantics for signal-event nets. The key part of the execution of a signal-event net is a step. Before we discuss their definition of a step in more detail, we fix some terminology [29]. A transition is *forced* if it is triggered by another transition; otherwise it is *spontaneous*. (So in standard Petri nets, every transition is spontaneous.) A transition t can be forced by more than one transition. There are two options in that case: either all trigger transitions must occur simultaneously to trigger t (AND), or only one trigger transition has to occur in order to trigger t (XOR). We only consider the XOR interpretation here.

A set s of transitions is *signal complete* iff

- i. if s only contains spontaneous transitions, it is signal complete.
- ii. if s is signal complete, $t \notin s$ is forced and t is triggered by a transition $t' \in s$, then $s \cup \{t\}$ is signal complete.

A step s must satisfy the following constraints:

1. s contains transitions that are fired spontaneously, i.e., without being triggered by another transition,
2. the input places and input conditions (for read arcs) contain sufficient tokens for all transitions in the step to fire,
3. s is signal-complete,
4. for every non-spontaneous transition t' that is not in s , set $s \cup \{t'\}$ does not satisfy 1-3.

We can easily see the correspondence with our semantics: spontaneous transitions are transitions in the environments, representing events, whereas forced transitions are transitions in the workflow model (so done by the WFS). Constraint 1 states that every step must be triggered by at least one event. Constraint 2 states that all the transitions in the step must be enabled and consistent. Constraint 3 says that a transition that is triggered by an input event e can only be part of the step if e occurs. Constraint 4 states that the step be maximal. It is not difficult to see, that these constraints are indeed equivalent to the constraints we discussed in Section 3 for the basic statechart step semantics.

But, this definition differs with our semantics w.r.t. the generation of events during a step, since it assumes that events generated by the system are sensed immediately in the same step (as in the statechart fixpoint semantics [52]), whereas we assume that they are sensed after the current step has been taken (as in STATEMATE [37] and UML [56]). To illustrate this, we translate the activity diagram in Fig. 6, shown in the left-hand side of Fig. 9, into a signal-event net, shown as the right-hand side of Fig. 9. The interrupt arcs represent triggering;

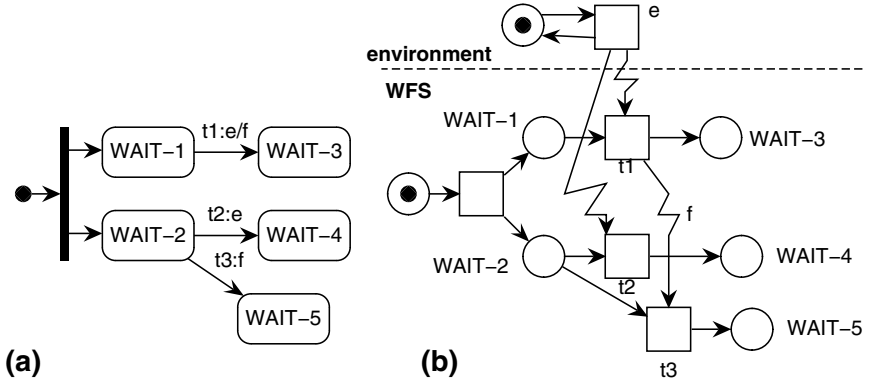


Fig. 9. Event generation modelled in signal-event nets

the triggered (forced) transition is pointed at. Suppose the current marking of the signal-event net is $[WAIT-1, WAIT-2]$ and transition e occurs. Then both $[e, t1, t2]$ as $[e, t1, t3]$ are valid steps, according to the constraints listed above. This is similar to the behaviour of the corresponding activity diagram (statechart) under the statechart fixpoint semantics of Pnueli and Shalev [52], as we explained in Section 3. But in our semantics [27, 28], only $[t1, t2]$ would be possible. As explained in Section 3, we regard the fixpoint semantics (and thus the signal-event step semantics) as counter intuitive here, since it seems that e is ignored in state node $WAIT-2$ if step $[t1, t3]$ is taken.

It is easy to show that the signal-event net execution semantics is a strict subset of the fixpoint statechart semantics (strict because in the statechart variant on which the fixpoint semantics is defined, an edge can be labelled with a negative event ($\neg e$), which is true iff the event does not occur. Negative events cannot be defined in signal-event nets).

A more intricate example is presented in Fig. 10. The predicate $in(x)$ that is used in the activity diagram, evaluates to *true* iff node x is contained in the current configuration. It can be translated into a Petri net construct using read arcs and inhibitor arcs. Inhibitor arcs are necessary to model $\text{not } in(x)$. In Fig. 10, read arcs are lines, so do not have an arrow; inhibitor arcs are lines with a circle at the transition end. If the current configuration is $[A, WAIT-1]$ and at the same time activity A terminates and event e occurs, then in our semantics a sequence of steps is taken such that finally configuration $[final, B]$ is reached. But in the corresponding signal-event net, configuration $[final, WAIT-2]$ is reached and event *done* is not responded to and is lost! Consequently, the final configuration will never be reached in that case. This is clearly undesirable. We therefore prefer our semantics of event generation.

Of course, now the question arises whether our semantics of event generation can be simulated in signal-event nets. We think that this is impossible, since in

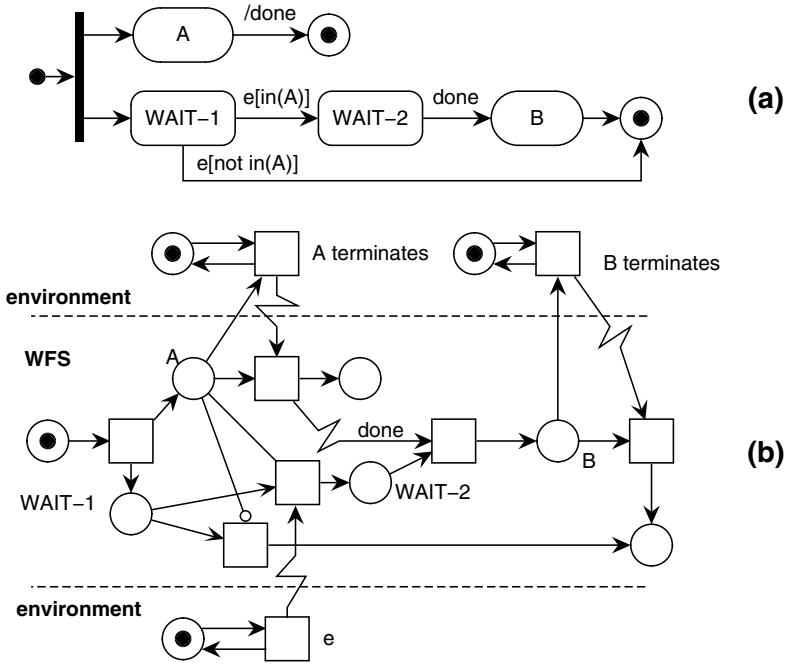


Fig. 10. Activity diagram and a similar signal-event net

both our semantics [27, 28] the input set acts as a kind of registry in which the events that are generated during a step are stored. This can only be simulated by treating events as tokens; if events are treated as transitions, events get lost after the step in which the events are generated completes. But above, we discussed the inadequacy of the event-as-token approach to model our step semantics. We come back to this issue in the conclusion of this subsection below.

Another point is that in general, statecharts have a priority constraint to deal with certain forms of nondeterminism. These are not present in the definition signal-event nets. Other differences between activity diagrams and signal-event nets are that (1) steps in signal-event nets are *sets* of transitions, rather than bags (but we could not find a compelling reason in Foremniak and Starke [29] why this is the case; probably the extension to bags is easily made), and (2) the environment must be modelled explicitly in signal-event nets, but not in activity diagrams.

Conclusion. Both in the event-as-token approach as in the event-as-transition approach, the statechart step semantics we have adopted cannot be modelled. However, in the event-as-transition approach, the signal-event net semantics resembles the statechart step semantics we use closely. But the signal-event semantics has a fixpoint semantics of generated events, whereas we have not. Our

semantics of event-generation can only be modelled using the event-as-token approach. It might therefore be worthwhile to try to incorporate the concepts used in signal-event nets into the event-as-token approach. Especially the concept of a forced transition seems promising. This concepts seems to be present in zero safe nets as well.

4.3 Modelling Data

The standard way to incorporate data in Petri nets is to use coloured tokens [40]. Coloured tokens are tokens that have attribute values. These attributes values are modified in/by transitions. Another way is to interpret places as predicates [31]. But then instances of the predicates can be seen as tokens that can change value when a transition consumes them. So, in both approaches, tokens carry data.

Therefore, the straightforward way to model case attributes in Petri nets is to attach these attributes to tokens. But attaching case attributes tokens suffers from the following problems.

Who Updates case Attributes. If case attributes are updated in some transition, than this transition cannot be part of the workflow model, because the WFS who executes the workflow model does not update case attributes, it only routes the case. (See Section 3). In other words, the environment (the actor) must be specified explicitly by a transition in order to let the case attributes change value.

Data Integrity. Several tokens may represent the same case attributes. Ideally, this situation should be prohibited, since an attribute may then inconsistently have several different values (i.e., the different tokens may assign different values to the same attribute). In terms of transaction theory, the isolation property fails to hold, since activities that update the tokens are not isolated from the other executing activities.

One possible solution is to represent each case attribute by a single coloured token. Then each transition that reads or writes the attributes must have this token as input and outputs the token when it finishes. Although the isolation property is then ensured, in standard Petri nets two read activities cannot be simultaneously active, since both consume the same token. That is not what we want, because the concurrency of the WFS is then reduced. In addition, the resulting net would look like ravioli if there are many case attributes.

To circumvent this, read arcs [47] can be used for read access. Interestingly, apparently read arcs have been proposed just to solve this problem of simultaneously access to shared data [30]. But unfortunately, read arcs do not solve the problem satisfactorily. To illustrate this, consider the Petri net with read arcs in Fig. 11. Data item x is updated and read by activity A and read only by B. In this net, although x cannot be updated and read simultaneously, it is possible that A reads a value of x that is subsequently changed by B. So, A and B are not isolated from each other (viewing both activities A and B as separate transactions.) Therefore, this solution does not satisfy our needs. (De Francesco et

the properties of an activity and the properties of a transition. First, a transition takes no time to execute, whereas an activity does. There are two ways to solve this problem. The first solution is to decompose the transition into a “begin activity transition” and “end activity” transition that are connected by a place representing “activity busy executing”. This solution results in a Petri net that is quite similar to an activity diagram. Then the execution of an activity is actually represented by a place. This approach is taken by for example Van der Aalst, Van Hee and Houben [6] and Desel and Erwin [19]. See the next item below for a discussion of this approach.

The second solution is to use timed or stochastic Petri nets, in which a transition can have a duration. In most timed and stochastic Petri net variants a transition still fires instantaneously, but it takes time before a transition is enabled. The transition in that case actually represents the starting or ending of an activity rather than the complete execution of the activity. This is not harmful for analysis purposes, but it gives a slightly awkward model of WFS reality. Analysis of timed and stochastic Petri nets is far more complex and involved than analysis of simple low-level nets.

However, our main objection against modelling an activity as a transition is the following. In Petri nets, a transition is executed by the system that the Petri net models. Hence, if a transition models an activity, this implies that the WFS does the activity. This approach violates the WFS characteristic that an activity is performed by an actor in the environment of the WFS, not by the WFS itself. *And it is this characteristic that creates the need for reactivity in a WFS.* By contrast, in our semantics the WFS does not do activities; it merely routes cases. In Petri nets that model activities as transitions, the routing is not modelled at all. Therefore, such Petri nets do not model a WFS.

As an aside, note that in some variant of Workflow Nets [5], some transitions can be labelled with a silent action that is not observable for the environment. Van der Aalst [4] suggests to use the silent step to model routing transitions. Transitions labelled with an observable action then represent workflow tasks. However, in that semantics, the silent action can be abstracted away from sometimes. For example, a sequential workflow model with two tasks a and b and a routing transition from a to b is equal to a model in which a is directly followed by b . It is unclear how this abstraction can be related to the execution of real workflow models: a WFS always routes a case after an activity completes. In our view, routing cannot be abstracted away from.

Of course, one could model the environment also in the Petri net workflow model, and let the activity be performed by the environment part of the Petri net model. But then the relationship with the corresponding part of the workflow is unclear, i.e., what should the WFS do while the environment is busy performing some activity?

Activity is Place. To the best of our knowledge, this interpretation is never chosen in Petri nets. Most people modelling a workflow in Petri nets probably would find this interpretation counter-intuitive since (as they argue) during an activity the case is changed, whereas a place is static (the local part of the case

is not changed). We disagree, however, with this argument, since for a WFS an activity state *does* represent something static, namely the WFS waits for an actor to complete the activity. The only dynamic behaviour of the WFS is when some activity completes and the case must be routed to a new state.

Nevertheless, the Petri net people who find this interpretation counter-intuitive are right to some degree. Whether we represent activities as places or as transitions, in Petri nets case attributes can only be changed in transitions, not in places. This corresponds to the fact that Petri nets model closed, active systems, in which the environment, i.e., that which is outside the Petri net, does not play any role. Any model of an open, reactive system, on the other hand, does allow for a change of case attributes during a state (place), namely if the change is initiated by the environment! (These changes are implicitly modelled and not explicitly represented by an edge in the diagram.) And this is exactly what happens during an activity: the environment (i.e., an actor using an application) updates case attributes, whereas the WFS waits for the activity to terminate. Consequently, to model change of case attributes in a Petri net, we must model the environment explicitly in the Petri net as well.

Conclusion. As stated above, in most Petri nets workflow models the first option is adopted. The only motivation that is given for this interpretation is that this is “straightforward”. We think that the real, underlying motivation is based upon the following properties of Petri nets: (1) a transition represents some change by the system, whereas a place represents a static condition on the Petri net model, and (2) a Petri net model can only change state by firing transitions. The two properties imply that all changes are caused by behaviour of the system itself. In other words, changes cannot occur due to the environment of the system. Consequently, any modelling language having these properties cannot faithfully model open, reactive systems; instead, they are more suitable for modelling closed, active systems. Petri nets are useful for example to represent (scarce) resource usage, e.g. the allocation of actors to activities.

4.5 Modelling the Implementation-Level Semantics

In the implementation-level semantics, only a single event can be processed at a time. Therefore, a queue is needed to store events that occur while the Router is busy processing some event. We now discuss whether this can be simulated using the event-as-token and event-as-transition approaches.

In the event-as-token approach, a queue can be modelled straightforwardly by switching to Petri nets with integers (counters) as is done in the FunSoft approach [23]. And a special place can be introduced to store the event that is currently being processed by the Router. But still, since in the event-as-token approach the statechart step semantics we use, cannot be simulated very well, the implementation-level semantics cannot be simulated very well either.

In the event-as-transition approach (signal-event nets), queues cannot be modelled, since the effect of an event is lost after the step in which it occurs is completed. So, if in the environment an event occurs (a spontaneous transition

fires), the WFS must react immediately, since otherwise the event will be lost. So, in the event-as-transition approach, the implementation-level semantics cannot be simulated at all.

We conclude that the implementation-level semantics cannot be modelled satisfactorily using Petri nets.

4.6 Petri Nets for Workflow Modelling

We conclude this Section by discussing the Petri net models we found in literature that are used to model and analyse workflows. Van der Aalst, Van Hee and Houben [6] use high-level nets to model and analyse Petri net based workflow models that also model resources. Van der Aalst [3] uses Workflow nets, low-level Petri nets with a single start and a single end place, to verify proper termination of a workflow model. Although Van der Aalst recognises the need for modelling input events, he abstracts away from them for analysis purposes for two reasons [3]. His first argument is that the environment cannot be modelled completely; from the point of view of the WFS it behaves nondeterministically. This is best modelled, he says, by leaving the events out. His second argument is that if an abstracted workflow is correct, the concrete one will also be. But as we showed above both arguments fail to hold if there is a dependency between different events in the same workflow model and if one event can trigger more than one transition. In these cases, abstracting from events will lead to different behaviour in the abstract model, when compared to the concrete model. Consequently, the verification results obtained for the abstract net might not be reliable anymore.

FunSoft nets [18] are high-level nets for software process modelling, but they can also be used for workflow modelling. Their semantics is defined in terms of Predicate/Transition nets [31]. FunSoft nets focus on the flow of resources (objects), like business documents, through an organisation and do not focus on modelling events. Some shorthands are defined to model for example FIFO queues. Several analysis techniques, including verification have been developed for FunSoft nets [18].

INCOME/WF [49] is a workflow management system based on high-level Petri nets where the tokens are nested relations. Nested relations are introduced to increase the concurrency of the net: the actual transitions are defined on the basic elements of the relation, not on the relation itself. This means that still for the basic elements, no concurrency exists, since the standard Petri net firing rule is employed, in which a transition consumes all tokens it reads.

Information/Control Nets [22] are a high-level Petri net variant for workflow modelling. The focus is on the modelling of resources, like documents, not on the modelling of events. The standard Petri net semantics for high-level nets is used.

Milano [9] is a research prototype to investigate the issue of flexible workflow models. The Petri nets that are used do not contain events, loops, data, real-time. Moreover, these nets must be safe.

It is interesting to notice, from this brief overview, that most Petri net workflow models provide little support for modelling events. And if a notation for events is suggested, no formal semantics for them is given. Most approaches interpret tokens as resources that are being used by activities in transitions to deliver a requested service for a customer. However, the assumption seems to be implicitly made that resources are *scarce*, since no two transitions can consume the same token simultaneously. This assumption is questionable: certainly, some resources are scarce, but also a wide variety of resources, especially information carriers, are not. (None of these Petri net variants uses read arcs.)

5 Discussion and Conclusion

From our comparison of our semantics with Petri net semantics, we draw the following conclusions. First, Petri nets model closed systems. All changes in Petri nets occur because of the firing of some transitions in the net that represent activity of some part of the system itself, rather than some activity in the system's environment. Second, standard Petri nets model active systems, rather than reactive ones. A transition is enabled if its input places are filled. Also, an enabled transition does not have to fire immediately. Our semantics is reactive. An edge in an activity diagram is enabled if its source state nodes are in the current configuration and its trigger event occurs in the environment. And an enabled edge *must* fire immediately. That is why we impose a maximality constraint on steps in our semantics. This constraint is lacking for standard Petri nets.

In Petri nets, reactivity can be simulated to some extent by modelling the environment in the Petri net as well. This is done, for example, in a recently proposed variant of Petri nets, called signal-event nets. These nets are also motivated by the domain of reactive systems. Signal-event nets have a complex semantics that differs considerably from the standard Petri net token-game semantics (amongst others, a maximality constraint is imposed on steps). Since their semantics is so different from the token-game semantics, it is questionable whether these are Petri nets at all. We showed that signal-event nets behave similar to the fixpoint step semantics for statecharts, defined ten years earlier by Pnueli and Shalev [52]. We are convinced that it is impossible to simulate in Petri nets the STATEMATE interpretation of event generation, that is also used in UML. However, it might be worthwhile to try to incorporate the concepts of signal-event nets into Petri net variants that model events as tokens, for example open nets. The resulting Petri net variants would likely be closer to the STATEMATE interpretation of event generation than any of the currently existing Petri net variants, but we expect such variants will still be different.

Third, Petri nets in general model scarce resources, rather than unscarce ones. A transition can only fire if there are enough input tokens present, i.e., enough scarce resources are available. Using read arcs this can be circumvented, because with a read arc a token can be tested without being consumed. Thus, read arcs allow for elegant specification of concurrent access to shared data. However, if an activity is seen as a transaction, as is usually done in workflow

modelling, read arcs must be combined with mutex places to enforce isolation between activities. We prefer our own approach using local variables, since it is more simple.

Fourth, the Petri nets that came closest to the requirements-level semantics we gave to activity diagrams contained inhibitor arcs, read arcs, synchronisation between transitions, and coloured tokens with timestamps. These nets had to contain both a description of the workflow and of the environment. Roughly speaking, the net had twice as many nodes compared to the corresponding activity diagram. Such Petri nets are truly gargantuan and difficult to analyse, both for a workflow modeller and a verification tool. (For example, a lot of the analysis results for standard Petri nets do not carry over to signal-event nets [29].) And even with these Petri nets, the possible dependencies between value change events could not be modelled. Moreover, these Petri nets still stay far away from our implementation-level semantics. They do not resemble it at all.

Also, one of the acclaimed advantages of Petri nets, that there is an abundance of analysis techniques available for them, is only true for low-level nets; its applies to a lesser extent to high-level nets. As we showed, not every desirable construct can be modelled in low-level and high-level nets; read arcs, inhibitor arcs and synchronisation constraints are needed as well. But for these latter net variants, there are only very few analysis techniques available.

Of course, these conclusions are based upon our assumption that a workflow model describes the behaviour of a WFS. One could argue whether this assumption is valid. In fact, does not a Petri net workflow model describe an organisation, rather than a computerised system? But even then, the interaction between the organisation and its environment must be modelled (customers, government, suppliers,...), since the organisation is a reactive system as well. Consequently, Petri net models of organisational behaviour suffer from the same problems as Petri net models of WFS behaviour: they cannot model reactivity.

From the above, it follows that if a Petri net is used to model a reactive system, the reactivity of that system is abstracted away from. But we consider reactivity to be one of the most important aspects of workflow modelling. If reactivity is abstracted away from, then at least some justification should be given that assures that the analysis results on the Petri net model will also carry over to a reactive setting. We do not think such justification has been given yet. If no justification is given, it is unclear what the relationship is between a Petri net model and the actual execution of a similar workflow model by a WFS. We conclude that Petri nets cannot faithfully model reactive systems, unless they are extended with a syntax and semantics for reactivity

Finally, Petri net variants are sometimes used as workflow modelling language in workflow management systems, for example in Cosa [44] and MQSeries [39, 45]. This does not mean, however, that these WMFSs also use the Petri net semantics! For example, in Cosa, activities are modelled as transitions. In the token game semantics of Petri nets, a transition fires instantaneously whereas in real life an activity will not be performed instantaneously. So, although the Petri net syntax is used, it is very doubtful whether the Petri net semantics

is used. We think that this observation holds for other workflow management tools as well, since most adhere to the reference model of the Workflow Management Coalition [60] that, as we do, views workflow systems as reactive systems (cf. Fig. 2) that have coordination functionality.

Acknowledgements

The authors would like to thank Wil van der Aalst and Jörg Desel for their critical comments on a previous version of this paper.

References

- [1] W. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management*. LNCS 1806. Springer, 2000. 323, 348, 349
- [2] W.M.P. van der Aalst. The application of Petri nets to workflow management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998. 321, 323, 325, 333, 334
- [3] W.M.P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In Aalst et al. [1], pages 161–183. 323, 333, 345
- [4] W.M.P. van der Aalst. Personal communication, 2001. 343
- [5] W.M.P. van der Aalst and T. Basten. Inheritance of workflows: An approach to tackling problems related to change. *Theor. Comp. Sci.*, 270(1-2):125–203, 2001. 343
- [6] W.M.P. van der Aalst, K.M. van Hee, and G.J. Houben. Modelling workflow management systems with high-level Petri nets. In G. De Michelis, C. Ellis, and G. Memmi, editors, *Proc. 2nd Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31–50, 1994. 343, 345
- [7] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros. Advanced workflow patterns. In O. Etzion and P. Scheuermann, editors, *Proc. CoopIS 2000*, LNCS 1901, pages 18–29. Springer, 2000. 324
- [8] G. Agha, F. Decindio, and G. Rozenberg. *Concurrent Object-Oriented Programming and Petri Nets*. LNCS 2001. Springer, 2001. 323, 337
- [9] A. Agostini and G. de Michelis. A light workflow management system using simple process models. *Computer Supported Cooperative Work*, 9(3-4):335–363, 2000. 323, 345
- [10] P. Baldan, A. Corradini, H. Ehrig, and R. Heckel. Compositional modeling of reactive systems using open nets. In K.G. Larsen and M. Nielsen, editors, *Proc. CONCUR 2001*, LNCS 2154, pages 502–518, 2001. 323, 334
- [11] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Sci. of Comp. Prog.*, 19(2):87–152, 1992. 328
- [12] R. Bruni and U. Montanari. Zero-safe nets: Comparing the collective and individual token approaches. *Information and Computation*, 156(1-2):46–89, 2000. 323, 337
- [13] F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Transactions on Database Systems*, 24(3):405–451, 1999. 332, 334

- [14] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual Modeling of Workflows. In M. P. Papazoglou, editor, *Proc. 14th Int. Object-Oriented and Entity-Relationship Modelling Conference (OOER'95)*, LNCS 1021, pages 341–354. Springer, 1995. 324, 328
- [15] S. Christensen and N.D. Hansen. Coloured Petri nets extended with channels for synchronous communication. Technical Report PB-390, Aarhus University, 1992. 323, 336, 337
- [16] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999. 322
- [17] F. Dehne, R. Wieringa, and H. van de Zandschulp. Toolkit for conceptual modeling (TCM) — user's guide and reference. Technical report, University of Twente, 2000. Available at <http://www.cs.utwente.nl/~tcm>. 322
- [18] W. Deiters and V. Gruhn. Process management in practice applying the FUNSOFT net approach to large-scale processes. *Autom. Softw. Eng.*, 5(1):7–25, 1998. 323, 345
- [19] J. Desel and T. Erwin. Modeling, simulation and analysis of business processes. In Aalst et al. [1], pages 129–141. 343
- [20] J. Desel and G. Juhás. What is a Petri net? Informal answers for the informed reader. In H. Ehrig, G. Juhás, J. Padberg, and G. Rozenberg, editors, *Unifying Petri Nets*, LNCS 2128, pages 1–27. Springer, 2001. 322
- [21] ebXML. <http://www.ebxml.org>. 332, 335
- [22] C. A. Ellis and G. J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, LNCS 691, pages 1–16. Springer, 1993. 321, 323, 345
- [23] W. Emmerich and V. Gruhn. Software process modelling with FUNSOFT nets. Technical Report 47, Dept. of Computer Science, University of Dortmund, 1990. 344
- [24] H.-E. Eriksson and M. Penker. *Business Modeling With UML: Business Patterns at Work*. Wiley Computer Publishing, 2000. 332
- [25] R. Eshuis and R. Wieringa. Verification support for workflow design with UML activity graphs. In *Proc. Int. Conf. on Software Eng. (ICSE)*, 2002. To appear. 322, 329
- [26] R. Eshuis and R. Wieringa. A formal semantics for UML activity diagrams. Technical Report TR-CTIT-01-04, University of Twente, 2001. 323, 330
- [27] R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In H. Hussmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE 2001)*, LNCS 2029, pages 76–90. Springer, 2001. 322, 328, 330, 339, 340, 342
- [28] R. Eshuis and R. Wieringa. An execution algorithm for UML activity graphs. In M. Gogolla and C. Kobryn, editors, *Proc. <<UML>> 2001*, LNCS 2185, pages 47–61. Springer, 2001. 322, 328, 330, 339, 340
- [29] A. Foremniak and P.H. Starke. Analyzing and reducing simultaneous firing in signal-event nets. *Fundamenta Informaticae*, 43:81–104, 2000. 323, 336, 337, 338, 340, 347
- [30] N. De Francesco, U. Montanari, and G. Ristori. Modelling concurrent accesses to shared data via Petri nets. In U. Montanari and E. R. Olderog, editors, *Proc. PROCOMET'94*, pages 489–508. North-Holland, 1994. 323, 341, 342
- [31] H. J. Genrich. Predicate/transition nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties*, LNCS 254, pages 207–247. Springer, 1987. 341, 345

- [32] H. J. Genrich and P. S. Thiagarajan. A theory of bipolar synchronization schemes. *Theoretical Computer Science*, 30(3):241–318, 1984. 342
- [33] A. Geppert, D. Tombros, and K. R. Dittrich. Defining the semantics of reactive components in event-driven workflow execution with event histories. *Information Systems*, 23(3-4):235–252, 1998. 332, 334
- [34] P. Grefen and R. Remmerts de Vries. A reference architecture for workflow management systems. *Journal of Data & Knowledge Engineering*, 27(1):31–57, 1998. 324, 328
- [35] C. Hagen and G. Alonso. Beyond the black box: Event-based inter-process communication in process support systems. In *Proc. 19th Int. Conf. on Distributed Computing Systems (ICDCS '99)*, pages 450–457. IEEE, 1999. 332, 334
- [36] H.-M. Hanisch and A. Lüder. A signal extension for Petri nets and its use in controller design. *Fundamenta Informaticae*, 41(4):415–431, 2000. 323, 337
- [37] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996. 322, 323, 328, 330, 331, 338
- [38] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer, 1985. 327
- [39] IBM. MQ Series Workflow (Websphere). <http://www.ibm.com>. 347
- [40] K. Jensen. *Coloured Petri Nets. Basic concepts, analysis methods and practical use*. EATCS monographs on Theoretical Computer Science. Springer-Verlag, 1992. 323, 341
- [41] S. Joosten. Trigger Modelling for Workflow Analysis. In G. Chroust and A. Benczur, editors, *Proceedings CON'94*, pages 236–247, Vienna, October 1994. 333, 334
- [42] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *Proc. FMOODS'99, IFIP TC6/WG6.1*, pages 331–347. Kluwer, 1999. 331
- [43] N. L. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994. 332
- [44] Software Ley. Cosa. <http://www.cosa.de>. 347
- [45] F. Leymann and D. Roller. *Production Workflow — Concepts and Techniques*. Prentice Hall, 2000. 323, 324, 328, 347
- [46] M. Löwe, D. Wikarski, and Y. Han. Higher-order object nets and their application to workflow modeling. Technical Report 95-34, Informatik, Technical University Berlin, 1995. 323, 337
- [47] U. Montanari and F. Rossi. Contextual nets. *Acta Inf.*, 32(6):545–596, 1995. 323, 335, 341
- [48] T. Murata. Petri nets: Properties, analysis, and applications. *Proc. of the IEEE*, 77(4):541–580, 1989. 322, 323
- [49] A. Oberweis, R. Schätzle, W. Stucky, W. Weitz, and G. Zimmermann. IN-COME/WF: A Petri net based approach to workflow management. In H. Krallmann, editor, *Wirtschaftsinformatik '97*, pages 557–580. Springer, 1997. 323, 345
- [50] OMG. Workflow management facility specification. OMG Document Number formal/00-05-02, 2000. Available at <http://www.omg.org>. 332, 334
- [51] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall, Englewood Cliffs, 1981. 322, 323

- [52] A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, LNCS 526, pages 244–265. Springer, 1991. 330, 331, 338, 339, 346
- [53] W. Reisig. *Petri Nets: An Introduction*. Number 4 in EATCS Monographs on Theoretical Computer Science. Springer, 1985. 323
- [54] W. Reisig and G. Rozenberg, editors. *Lectures on Petri nets I: Advances in Petri nets*, LNCS 1491. Springer, 1998. 323
- [55] K. Salimifard and M. Wright. Petri net-based modelling of workflow systems: An overview. *European Journal of Operational Research*, 134(3):218–230, 2001. 323
- [56] UML Revision Taskforce. *OMG UML Specification v. 1.4*. Object Management Group, 2001. Available at <http://www.omg.org>. 322, 323, 325, 328, 330, 338
- [57] J. Widom and S. Ceri, editors. *Active Database Systems – Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers, 1996. 332
- [58] R. J. Wieringa. *Design Methods for Software Systems: Yourdon, Statestate and the UML*. Morgan Kaufmann, 2002. To be published. 327
- [59] D. Wodtke and G. Weikum. A formal foundation for distributed workflow execution based on state charts. In F. N. Afrati and P. Kolaitis, editors, *6th International Conference on Database Theory (ICDT)*, LNCS 1186. Springer, 1997. 323, 332, 334
- [60] Workflow Management Coalition. The workflow reference model (WFMC-TC-1003), 1995. Available at www.wfmc.org. 323, 324, 328, 348
- [61] Workflow Management Coalition. Workflow management coalition specification — terminology & glossary (WFMC-TC-1011), 1999. Available at www.wfmc.org. 324

BASYSNET – An Integrated Approach for Automated Control System Development

Eckehard Schnieder, Mourad Chouikha, Stefan Einer, and
Michael Meyer zu Hörste

Institut für Regelungs- und Automatisierungstechnik
Technische Universität Braunschweig
`sek@ifra.ing.tu-bs.de`

Abstract. This paper shows an universal approach to the design and development of discrete or hybrid discrete-continuous control systems. As means of description Petri nets are used and as method the general approach called BASYSNET. Three main steps have to be fulfilled to create a control system: firstly, if it is possible, the model of the uncontrolled plant and the process objectives, which determine the requirements for the control system, have to be formalized. The second step is the development of the formal controller model. For these tasks two different ways are possible. If the process model is formalizable, both - the formal process model and the formal process objective serve as inputs for the algorithm automated synthesis providing the formal controller model. If the process model can not be transformed in a formal description, e. g. due to a reasonable not to handle state-space, a heuristic approach has to be used. In the third step the partitioning of the control model into Hard- and Software will be done.

1 Introduction

The development of safety-critical systems and also multipurpose automation systems leads to more and more complex control structures especially for discrete and hybrid systems. One approach to solve the problem and to improve the results is to use so-called formal approaches.

The combination of well structured and concise expressed methodological procedures in combination with formal means of description for both the development procedures and the resulting system models provide lean productive and robust systems engineering. Our approach reported here incorporates this design philosophy by a suited combination of the three components:

- means of description
- methods for design and structuring
- tools for advanced system engineering to support methods and description

Following this principles the approach called BASYSNET (The Braunschweig Description, Analysis and SYnthesis Method based on Petri NETs) has been developed at the Institut für Regelungs- und Automatisierungstechnik (Institute

of Control and Automation Engineering) of the Technische Universität Braunschweig (Technical University of Braunschweig) since several years [7, 11, 15].

1.1 Structure of this Chapter

Three development stages of the framework BASYSNET are described here: the formalization of the process objectives in section 2.1, the formal synthesis of the controller in section 2.2 and the heuristic approach for the modelling of process and controller in section 2.3. Some application examples for communication based systems are introduced in the section 3.

2 Development Stages

The basis of the development of control systems for a given physical process in the sense of system behaviour are two models with this adjunct descriptions:

- the model description of the uncontrolled physical process (process model) and
- the model description of the control respective process specification (process objective).

Both their formulations are normally informal like charts or text. Wherever the target is to get a suitable Hardware and Software combination for the controller. This is a kind of a formal description. It is created from the process objectives under the prerequisites given by the plant process concept. There are (at least) three different ways to reach the goal:

- The common way is to understand the process concept and the process objectives, to create the control objective and the control model. This concept has to be transformed into the controller model which is an more formal description. In the last step the control program will be developed by humans. This way is not outlined in this paper.
- If the process description is formalizable the formal synthesis approach is possible. Both - the plant process and the derived process objective - are transformed via the model level into a formal description. Result is the formal process model and the formal process objective. The second step is the automatical synthesis of the formal controller model, which will be partitioned in the final third step in Hard- and Software for the control program.
- In some specific cases the formalization of the process concept is not possible because e.g. the behavior of the uncontrolled process is not predictable or the state-space is reasonable not possible to handle. In this case the synthesis approach is not suitable, but a heuristic model-based (or intuitive) approach is still open. The process objectives can be formalized like in the model-based approach, but the process concept stays in the informal state. Hence the definition of the formal controller model has to be done in a manual way. The last step - the partitioning into Hard- and Software - will be done like in the model-based approach.

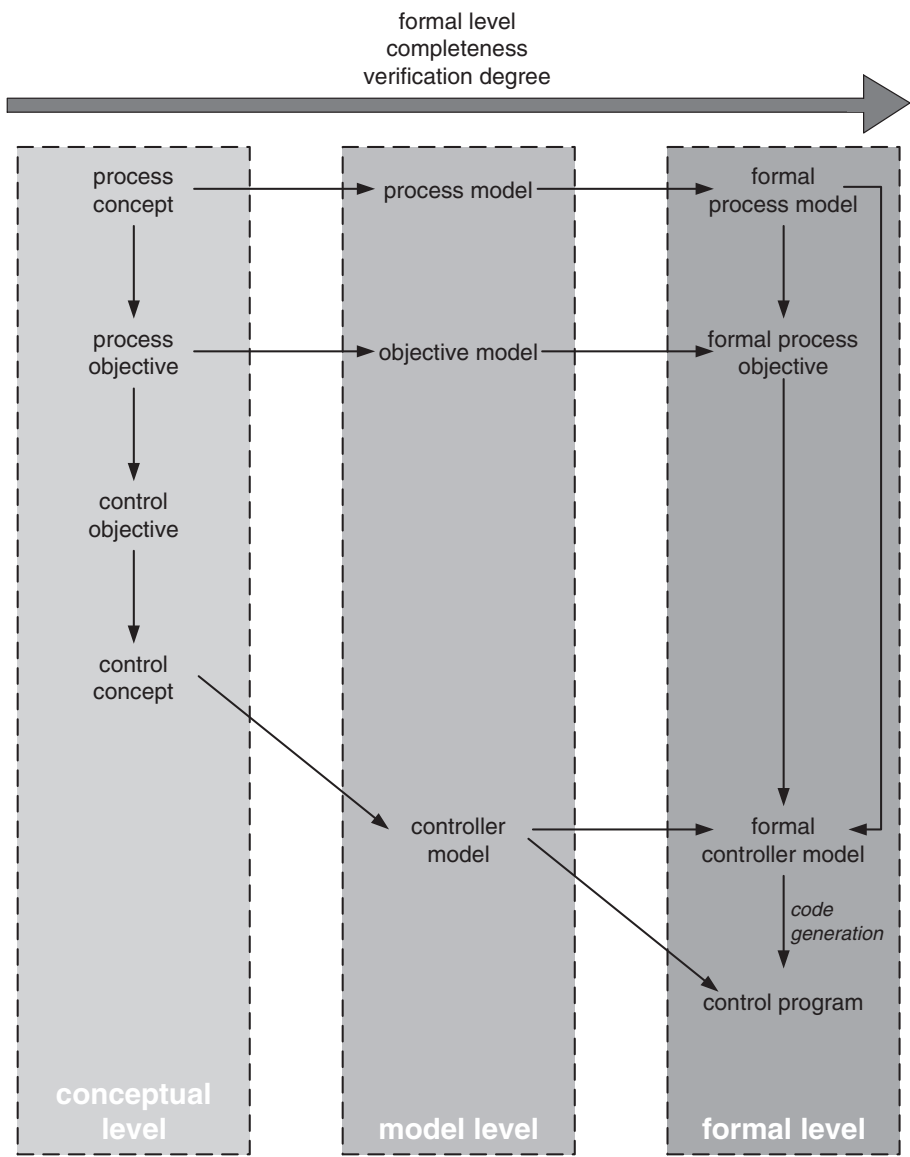


Fig. 1. Representation of the different approaches for control development

The three different ways are shown in Figure 1. The main parts of the three different approaches are discussed in more detail in the following sections and examples are attached for the explanation of both formal approaches.

2.1 Formalizing Process Objectives

Both methods of formal system design introduced by this paper are based on the concept of separation between the behavior of the plant process and control system to be specified. Each method has also to take the operational process into account. The specification of the operational process describes the allowed behavior of the whole plant process. The real system behavior must not differ from the part of process described by the operational view. Therefore, the operational process view is equivalent to the objectives of the process [5].

Operational Processes In complex systems, more than one scenario i.e. the regular procedure of the operational process is usual, because an automation system should be able to react on different events from the outside at the same process state. These events differ by values of parameters, which are not controlled by the system itself. In addition most automation systems have to be fault-tolerant. Therefore, not every fault has to be forbidden, but the operational process must provide the reaction of those permissive faults. Considering that in most states of process such permissive faults may appear, the number of possible sequences describing the operational process increases. In fact complex systems are resulting in complex operational processes, too.

Describing the operational process only in natural language prevents to specify unequivocally the connections between the possible scenarios. Therefore, no validation of system behavior is possible. This problem can be solved by using formal methods already for the specification of the operational process as an early step in the system development process. However, an explicit specification of all scenarios by means of a state-machine is too extensive because of the complexity. Hence, it is not practical for modeling the operational process and, in addition, for later extensions. An operational concept was proposed for describing the operational process by behavior modeling with modular concurrent components. Petri nets are a suitable means of description for solving this task. Modeling the system behaviour with Petri nets allows to derive the occurrence graph and thus the derivation of all procedural sequences of the operational process. The next section will introduce an approach for modeling the operational process with Petri nets.

Modeling Approach The plant process can be conceived independently from the technical implementation of the control system. Each relevant physical and technical state of one component is called a local state. A global process state is represented by a set of local states from different components. The change of a global process state can be a synchronous transition of local states of different components. This is called an event (see Fig. 2).

Using Petri nets for the description, places represents local states and transitions represent events. In the example in Figure 2 it is shown that same physical process states can be changed by different events. This can happen for two reasons. The first reason is the occurring of an event that depends not only on the

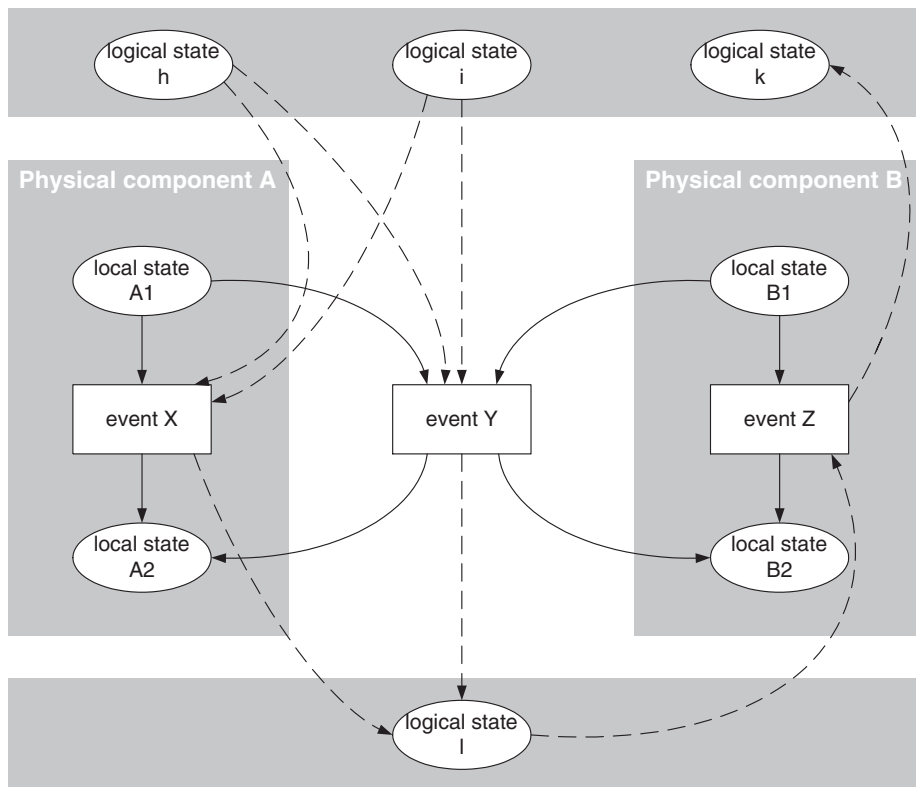


Fig. 2. Process Modelling by Petri nets

internal process state but also on the environment of the system - not especially modeled here. In this case, considering only the physical states, the events are in real conflict (e.g. X and Z or Y and Z). It can not be controlled by the system during this modeling status, which event will occur. Normally only one of these events in a conflict is a regular one. Others are fault approaches and reactions on these faults, which are also part of the operational process. The second reason for different events enabled by the same physical process state is that physical process states normally do not consider the history of a process. In this case, the events must not be in real conflict. Therefore, it is not sufficient to model only local states on the physical layer. In order to solve the problem additional logical states are provided. Logical states are not states of physical components but “memory” states of abstract components, which have to be derived from the contextual domain knowledge and objectives of the process [14]. Logical states are represented by places, too. The set of all logical states at one time instant is called process situation. It considers the process history. Now, each occurring event is determined as well by physical as logical states and conflicts only happen for reasons from outside of the system.

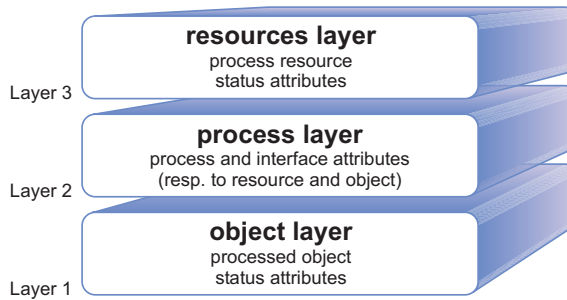


Fig. 3. The three layers holistic system model

2.2 Formal Synthesis Approach

Starting point of the model-based control design is a formal model of the specification as well as a formal model of the plant process which has to be controlled. Based on these two components, a control synthesis can be carried out automatically by applying different synthesis algorithms. Consequently, a modeling methodology and formal synthesis algorithms are required to synthesize controllers. The specifications can be formulated according to subsection 2.1.

In order to simplify and optimize the modeling process of discrete event systems we exploit the advantages of the object-oriented methods, such as easy comprehensibility and completeness as well as automatically resulting documentation [16]. As shown in [3] the object-oriented analysis and modeling allows to structure the relevant information for any system. An important result of this analysis are moreover the identified inheritance structures, which inform us about similar behaviors and characteristics of objects. The dynamic behaviors of each object are represented by Petri nets. This information can be exploited to generate object libraries. Indeed, these libraries comprise pre-defined objects, which can be inserted, combined and specified (define parameter) during modeling with a design tool.

The plant model is composed of 3 layers as shown in Figure 3, which represent different aspects of the plant. The first layer describes the product, e.g. a workpiece. This layer contains the properties and the states of the processed object. The second layer represents the processing flow (material flow and the interactions between products and resource). Here, the possible locations and the corresponding transportation processes of the processed object are described. The third layer represents the dynamics of the process resources (active objects) which influence the state and the location of the processed product.

If a formal model exists, e.g. by means of a Petri net description of the system behavior, a controller can be algorithmically designed on this base in closed methodology, which is called control synthesis. Differently than continuous control systems, where usually the observance or consequence of desired values - also with presence of disturbances - is to be obtained, the control objective of discrete event systems is different and determines thereby certainly and substan-

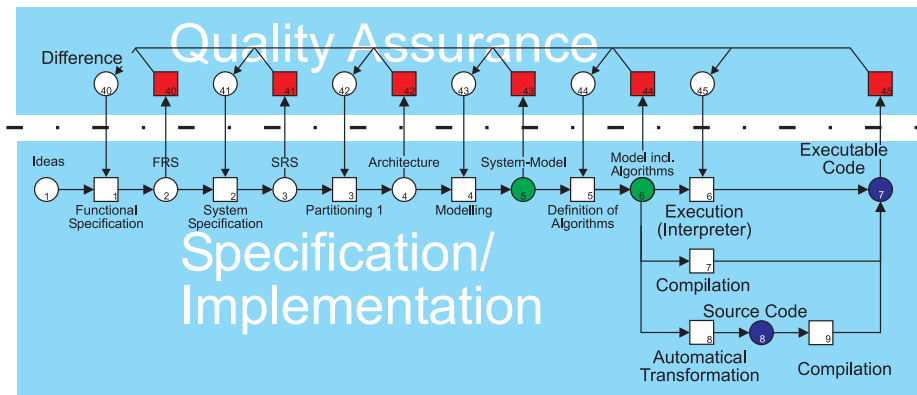


Fig. 4. The cascade of automated system development

tially the structure of the controller. As already mentioned, control objectives can be specified e.g. in the form of scenarios.

The main task of the control synthesis is to calculate a transition firing sequence from a given start state to a likewise given end state considering all the stated system specifications. A controller net, which represents the control algorithm, is then derived from the calculated sequence. The automatic generation of the controller net guarantees a high semantic correctness of the control algorithm. For generating a control algorithm there are many possibilities (at least three). The methods which are implemented in the tool COSYNET, which is developed in our institute, are an occurrence graph based search and an optimized calculation and search. Applying the first method, the complete occurrence graph must be calculated. A firing sequence is then searched considering the specifications. This method is suitable for small and simple processes because the calculation of the occurrence graph of large processes require enormous processing time and memory space.

The second method is based on an optimized calculation of occurrence graph parts simultaneously (during the calculation) considering the specifications. In the worst case the complete occurrence graph is calculated, but only the right sequence with the corresponding states is stored.

2.3 Heuristic Modelling Approach

For general conditions the above model-based synthesis is sometimes not applicable. E.g. it is (reasonable) impossible to model explicitly for several trains in a wide area the resulting state-space of the uncontrolled process. A less effort-consuming solution is to describe the achievable behavior in a manual way. A specific step-by-step approach has been developed in our institute [12, 8, 10].

The Figure 4 shows the loop cascade for the complete heuristic approach. In this figure the squares represent activities which are to be carried out, circles represent inputs or results. Both are marked with a number and in the following text squares are referred with their number in square brackets and circles with round parenthesis. The sequence from the ideas (1) to the executable code (7) is the process of specification and implementation of the system, which is related to the waterfall model [17]. The local feedback loops are added in relation to the spiral model [1]. The specific knowledge about the system context is needed in addition to the results from the previous step in all steps of the realization ([1] to [5]). In different phases of the cascade different parts of the contextual knowledge are used.

The general structure of the control system has been developed according to the step [4] of the development cascade. This results in a first layer of the Petri net model, which can be simulated in a symbolic way, that is with unstructured tokens. A detailed simulation is first possible when the algorithms (6) are defined, which leads to step [5] of the life cycle. The model of the controller can be made by using different paradigms, because the use of a means of description is independent from a specific method.

In the definition of the Algorithms [5] the structure of the model has to be refined by a new decomposition level. Functions which are represented in the step [4] in a single transition have to be refined by a subnet. The complexity of this net can lead to a structure which contains itself some more subnets.

To ensure that the system will perform the intended functions, all assumptions about the system context have to be consistent over all development phases.

The feedback cascades in the upper part of the development cycle incorporate the quality assurance like control loops likewise this corresponds to the reviews of the spiral model [1]. With each step, the results are checked for compliance with the results of the previous. Should this check-up produce any deviations, which are inadmissible, it is inevitable to repeat the current or an earlier step. In this context, three model based methods of quality assurance are mentioned: test, validation by means of simulations, and verification using the mathematical methods of formal analysis. Refining the grey squares in Figure 4, this can be represented in the way shown in Figure 5. As indicated through the additional arrows which point to the circle “differences”, there are even more methods to identify differences and to check if they are admissible or not.

Verification, Validation and Test are three methods of quality assurance which can be used in several phases of the system development. Validation by means of simulation can be used in the early phases of the life cycle, when the level of detail is abstract and the main focus is on the behavior of the system. In the following phases, which add more and more details to the system, simulation can still be used to check, that the behavior also on the detailed level is correct. Formal analysis methods can be used as well, to verify specific properties of the system. Initially, the results do not have a direct relationship with the specified system. To transform them into specific system properties, the abstract results have to be interpreted in the context of the system. Tests can be applied to

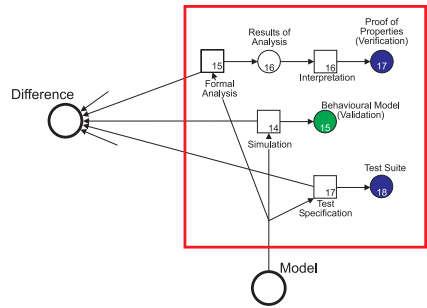


Fig. 5. Model-based quality assurance

implementations and they can show that the system behavior in a specific case corresponds to a test reference. In this case, the formal model could serve as the test reference.

3 Examples

Some Examples for the application of BASYSNET for communication based systems will be shown in this section. Both examples are related to the domain of railway transport systems. The first one is the control of a autonomous level crossing and the second one is a train control system.

3.1 Transportation Systems: Level Crossings

The approach in formalizing process objectives as described in section 2.1 is validated by applying it on an official case study offered by a priority programme of DFG¹. This case study [9] deals with an realistic level crossing system based on radio communication between train and track. More details in applying the introduced approach can be found in [2] and [6].

The approach of formal synthesis has been also tested on this same example. The requirements and the process object of the level crossing have been described with Petri nets and the controller for this system has been generated. Details can be found in [2].

In order to show the separation between both approaches the level crossing example has been modelled and simulated according to the heuristic approach. The results are also shown in [2].

Another application example for the formal synthesis applied to discrete and continuous processes can be found in [4].

¹ The DFG is the central autonomous (selfgoverned) academic institution in Germany serving all fields of science and the humanities by financing research projects.

3.2 Transportation Systems: Train Control Systems

Train control systems have been used as an example for the heuristic approach. Normally the number of trains and the track layout can not be used for the development of a train control system, because it is in general not known. This is a kind of an open environment, which leads to the constraint to use the heuristic approach.

The construction of the model takes place in four different levels of Petri nets: Context, Process, Scenario and Function level. The system context is depicted on the uppermost level. This net comprises the interfaces between the modelled sub-systems, normally on-board subsystem, trackside subsystem and the systems environment. All the other subsystems are comprised in the environment, they may, however, be further refined during a later phase of modelling work. This net corresponds to the formal representation of the system architecture. On this level, all the interfaces are defined as uni-directional channels. The next level is formed by the nets of process visualization. This level defines the operational scenarios and their sequence of operation. Every operational scenario is represented by one transition in this level and refined as a net in the next level. The different scenarios are refined with two more aspects, namely the interfaces are disintegrated until individual messages are singled out. In the nets themselves, the sequence of events is shown. It is possible to simulate parallel event sequences as a function of specific initial situations and stimulations. The functions, if explicitly modelled, form a net level of their own, and are visualized with a much simpler structure: they only show the transition refinements. This is why they do not have any interfaces with external systems, even if the transition itself has such interfaces. Communication is safeguarded by the higher (scenario) level net. The details of this approach are shown in [18] and [13].

4 Summary

This paper shows an approach for the development of control systems. Three steps lead to a control program: the formalization of the process concept and the process objectives, the development of the formal controller model and the translation to the control program. The second step can be performed in two different ways: if the control concept is formalizable, the model-based approach can be chosen, if it is not, the heuristic approach is applied.

References

- [1] B. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, pages 61–72, May 1988. 359
- [2] M. Chouikha, S. Einer, M. Meyer zu Hörste, and E. Schnieder. Ansätze für die Entwicklung von Eisenbahnleitsystemen auf der Basis von Petrinetzen. In E. Schnieder, editor, *FORMS 2000 - Formale Techniken für die Eisenbahnsicherung*, pages 57 – 84, Düsseldorf, 2000. VDI-Verlag. (In German). 360

- [3] M. Chouikha, B. Ober, and E. Schnieder. Model-based control synthesis for discrete event systems. In M. Hamza, editor, *IASTED Conference on Modelling and Simulation*, pages 276–280, Pittsburgh PA, May 1998. 357
- [4] M. Chouikha, E. Schnieder, and B. Ober. Automatisierter Steuerungsentwurf für diskrete und kontinuierlich-diskrete Systeme. *AT - Automatisierungstechnik*, 49(6):280 – 286, Juni 2001. 360
- [5] S. Einer and E. Schnieder. A petri net based modelling approach for operating methods in train control systems. In *IFAC Conference on Control Systems Design (CSD2000)*, Bratislava (Slovak Republik), 2000. 355
- [6] S. Einer, R. Slovak, and E. Schnieder. Modelling train control systems with petrinets - an operational model. In *Systems, Man and Cybernetics*, Nashville, USA, 10 2000. 360
- [7] GMA Unterausschuß 1.51 Standardisierte Beschreibungsmittel in der Automatisierungstechnik. *Glossar*. www.ifra.ing.tu-bs.de/gma181/glossar.htm, October 1996. (in German). 353
- [8] A. Janhsen, K. Lemmer, B. Ptok, and E. Schnieder. Modelling and simulation of new european train control system. In *IFAC Symposium on Transportation Systems*, Chania (Greece), 1997. 358
- [9] L. Jansen. *Referenzfallstudie Verkehrsleittechnik*. Institut für Regelungs- und Automatisierungstechnik, Technischen Universität Braunschweig, Braunschweig (Germany), 1999. (in German), see: www.ifra.ing.tu-bs.de/~m33/spezi/. 360
- [10] L. Jansen, M. Meyer zu Hörste, and E. Schnieder. Technical Issues in Modelling the European Train Control System (etcs) using coloured Petrinets and the Design CPN Tools. In *Design CPN Users Conference*, pages 103 –115, Aarhus (Denmark), 1998. 358
- [11] M. Meyer zu Hörste, S. Parthasarathy, and E. Schnieder. Notation, method, tool: A conceptual framework for the application of formal methods. In *9th IFAC Symposium on Control in Transportation Systems*, Braunschweig (Germany), June 2000. 353
- [12] M. Meyer zu Hörste, B. Ptok, E. Schnieder, and H. Schrom. A case study for the automated system design: the satellite-based train control system. In *IFAC Conference on Control Systems Design (CSD2000)*, Bratislava (Slovak Republik), 2000. 358
- [13] M. Meyer zu Hörste and E. Schnieder. Modelling train control systems with petrinets - a functional model. In *Systems, Man and Cybernetics*, Nashville, USA, 10 2000. 361
- [14] S. Parthasarathy and E. Schnieder. The explication problem: the achilles heel of formal methods. In *Entwicklung und Betrieb komplexer Automatisierungssysteme, EKA 99*, pages 93 – 103, Braunschweig (Germany), May 1999. 356
- [15] E. Schnieder. *Methoden der Automatisierung*. Vieweg, Braunschweig (Germany), 1999. (in German). 353
- [16] W. Stein. *Object-Oriented Analysis Methods - Comparison, Evaluation and Choice*. BI-Verlag, Mannheim, 1993. (in German). 357
- [17] E. Yourdon. *Modern Structured Analysis*. Prentice-Hall, 1989. 359
- [18] M. Meyer zu Hörste. Modelling and simulation of train control systems with petri nets. In Dines Bjorner, editor, *FMERail Workshop 3*, St. Pölten (Austria), February 1999. www.ifad.dk/Projects/FMERail/proceedings3.htm. 361

The Electronic Tool Integration Platform (ETI) and the Petri Net Technology

Volker Braun, Tiziana Margaria, Bernhard Steffen

Chair of Programming Systems, University of Dortmund, Germany
{Volker.Braun, Tiziana.Margaria, Steffen}@cs.uni-dortmund.de

Abstract. The paper presents the Electronic Tool Integration platform (ETI), a community platform designed for project-oriented, domain-specific, public or private interactive experimentation with heterogeneous tools. ETI managers, developers, and users are assisted by an advanced, personalized Online Service guiding experimentation, coordination and simple browsing of the available tool repository according to their degree of experience. This enables even newcomers to master the wealth of existing tools in a short time span, and to identify the most appropriate collection of tools to solve their own application-specific tasks. Concepts and current realization are illustrated along a concrete instantiation: the ETI community associated with the International Journal on Software Tools for Technology Transfer (STTT).

The ETI technology is application-independent, and we are convinced that it can offer a useful service to thematic communities like the Petri net tool developers and users by means of its underlying interoperability platform. Conversely, we envisage that some tools, initially located in the ETI repository for online experimentation will eventually migrate into the algorithmic kernel of the platform, as means e.g. for generalizing the model for execution.

1 ETI's Goals

The need as well as the availability of software tools is steadily increasing. Unfortunately, the uptake is significantly hindered because of the lack of transparency. It is hardly possible to orient oneself in the jungle of tools, modelling paradigms and promises. What is needed are trusted reference sites, that provide qualified information about tool profiles, integrability, usability and other important properties. Even more, these reference sites should be linked in order to allow a more global orientation. Thus, ideally, a network of reference sites should be aimed at, with cross referencing and a community support on different levels of expertise. Technically, this requires a common approach for building and setting up such sites in a way that by construction supports the network of sites architecture.

The Electronic Tool Integration platform (ETI) is designed for the support of communities wishing to set up sites for the project-specific, domain-specific, public or private interactive experimentation with heterogeneous tools¹. ETI

¹ The ETI platform is realized on top of the **METAFrame** environment [46, 37, 44].

managers, developers, and users are assisted by an advanced, personalized Online Service guiding experimentation, coordination and simple browsing of the available tool repository according to their degree of experience. In particular, this allows even newcomers to orient themselves in the wealth of existing tools and to identify the most appropriate collection of tools to solve their own application-specific tasks. Thus the ETI project can be seen as a concrete initiative that stimulates the communication between tool providers and tool users as well as between academia and industrial practice, supporting the transfer of tool-related technology.

ETI, born in 1996, is a Web-based, open communication platform, where tool providers can publish their tools and get valuable feedback from the end users, and users can compare different tools within their application domain. In particular, they can combine functionalities of tools of different application domains to solve problems a single tool never would be able to. A more detailed exposition, including background and related work can be found in [45], which describes the original instantiation for the ETI community associated to the International Journal on Software Tools for Technology Transfer (STTT) [24].

The ETI community service is an online service which allows users to create their own application-specific or community-specific Electronic Tool Integration sites. These sites are meant to constitute a meeting point for distributed communities for tool-related discussion and experimentation. In particular they are intended to spread new technologies at the tool-level and to convince and teach potential users via an easy experimentation. Thus ETI-sites may well be used for tool-related teaching and as a platform for potential customers looking for tools solving their current problem. Central is here the ETI-sites' unique support for high-level tool coordination and for personalization and online-discussion.²

As an example, the ETI Online Service for STTT currently comprises verification tools for real time systems and model checkers. The integration of programming language tools like type checkers, optimizers and code generators is on the way. The STTT ETI Service can be accessed via its homepage, <http://eti.cs.uni-dortmund.de>. From there, users can

1. **access online information** on the tools via hyperlinks to each tool's home site.
2. **access online a stand-alone version** of each tool, centrally located at the ETI service sites
3. **access the ETI repository** of integrated tools. It contains a collection of functionalities offered by individual tool, classified for ease of retrieval according to behavioural and interfacing criteria.
4. **experiment** at ease with the integrated tools and functionalities, by
 - (a) *running the* (stand-alone or integrated) *tools* on libraries of examples, case studies, and benchmarks made available on the ETI platform,

² Details about the ETI Online Community Service can be found under <http://www.eti-service.org>

- (b) *testing and running single tool functionalities*, capturing specific features offered by the integrated tools, on the same examples, from within a uniform graphical user interface provided by ETI,
 - (c) *constructing own application-specific heterogenous tools* through combination of single functionalities coming from different tools within the ETI platform,
 - (d) *loosely specifying coordination tasks*, which can be then automatically completed by means of ETI's coordination support. This, in particular, takes care of data format incompatibilities, as detailed in [36].
5. **experiment with own sets of data**, to be deployed in user-specific, protected home areas.

We are optimistic that this service-oriented and interactive approach can help overcoming the typical hesitation to try out new technologies: serious hurdles, like installation of the tools, getting acquainted with new user interfaces, lack of direct comparability of the results and of performances, are eliminated. Moreover, the intended collaborative effort of the ETI user community can be of inestimable help for everybody in need of tool support: to provide easily accessible information about fair, application-specific evaluations of various competing tools on the basis of predefined benchmarks is a clearly perceived valuable contribution. This form of remote assistance and advice is particularly important in geographical areas where support and distribution are still rare, and in areas of application outside ones own core competence.

In the following, we sketch some related approaches (Sect. 2), and a brief history of the initiative (Sect. 3). We then describe the ETI community Online Service (Sect. 4), present the meta-model for ETI sites (Sect. 5) and illustrate on a simple example its coordination features (Sect. 6). We then present the platform's multilayered software architecture (Sect. 7). The potential for synergy and mutual benefits in the cooperation with the Petri net researchers, tool builders, and users is briefly sketched in Sect. 8. Finally we draw our conclusions and perspectives (Sect. 9).

2 Related Approaches

We are not aware of any project which combines the features provided by the ETI platform. Roughly, the approaches and tools coming closest to our platform can be classified as follows:

2.1 Approaches Focussing on Tool Access

- **Link collections:** There are a lot of Web sites available which provide links to software tools in a special application domain. Examples are the Petri Nets Tool Database [42] or the Formal Methods Europe [21] database. The ETI platform covers this aspect of tool access by providing links to the home page of any tool available in the tool repository.

- **Software archives:** In contrast to software archives, like collections of Java Beans, ETI is not a distribution platform: tool providers retain all legal rights to, and responsibilities for their software. The user cannot download the software. He can only experiment with the tools installed on one of the ETI application servers. To obtain a license of the chosen tool, the user can contact the tool provider via the information supplied by the platform.
- **Web sites providing execution facilities of a single tool:** Some Web sites like the HyTech home page [25] or the smv guided tour [43] give access to tool functionality via an HTML form. In most cases simple CGI [9] scripts are used to access a single functionality of the tool and to deliver the results to the user in text form.

This kind of tool access is also available within the ETI platform. In addition to that, users can combine tool functionalities to sequential programs and then execute them.

2.2 Approaches Focussing on Tool-specific Communication

Many more sites exist which do not specifically address tool execution, but focus on information and community building around a specific tool (e.g for the Petri Net Kernel at <http://www.informatik.hu-berlin.de/top/pnk/index.html> or for the CADP toolsuite at <http://www.inrialpes.fr/vasy/cadp/>). More elaborate are

- **News Groups and Mailing Lists:**
Mailing list like the Uppaal [34] user's list, or news groups provide a means for tool-specific communication, however, without any tool related functionality. They only address one aspect of the ETI platform, i.e. the communication functionality. Precondition to participate in the discussion are email and a news client.
- **Community Sites:**
Community sites like the Java Developer Connection [27] offer discussion groups, email notification, and information in a special application field. Like in the previous item, these community sites only cover the community-specific communication functionality, however, in a more integrated fashion. They do not provide Web-based experimentation support.

2.3 Approaches Focussing on Tool Integration and Coordination

- **PROSPER:**
The PROSPER (Proof and Specification Assisted Design Environments) [16] toolkit provides an infrastructure where existing verification tools can be integrated into nearly any application (like CAD and CASE tools). This provides end users with applications enriched with new validation features, but it does not provide any user-level handle to coordinate verification tools. Coordination is exclusively is done by the PROSPER expert.

- **ToolBus:**

Closest to ETI is TOOLBUS [6, 32], a very general environment for tool coordination. However, being based on a process algebra, its coordination specifications are much more general and complicated than in ETI. Thus the user community is rather restricted.

In contrast, the ETI project provides an open platform, where the tool coordination facility is designed to be available to a very wide community. To coordinate different tool features, experienced users may use ETI's coordination language HLL (High-Level Language). Unexperienced users are additionally supported by ETI's synthesis component. Here, the user can define *what* he wants to achieve, instead of specifying *how* he wants to achieve his goal. ETI's automated synthesis component then generates sequential programs that implement the intended task. Generated programs can be executed via the Internet.

3 A Brief History of the ETI Project and its Roots

- **1993**

Start of the METAFrame project [46, 37] at the University of Passau, Germany. In the initial phase an interpreter for the Pascal-like coordination language HLL (High-Level Language) and a graph library was implemented by four students.

- **1994**

The initial version of today's ETI's synthesis component was added to the METAFrame environment. It offers sophisticated support for the systematic and structured computer aided generation of application-specific complex objects from collections of reusable components.

- **03.1995 - 10.1995**

The first version of the Service Definition Environment was implemented in a cooperation between the University of Passau and Siemens Nixdorf Informationssysteme AG in Munich.

- **1996**

The ETI project was founded, in the context of the Springer International Journal on Software Tools for Technology Transfer (STTT) [24], with the goal to provide platform independent, Web-based access to METAFrame's coordination and synthesis functionality. It was intended to establish a virtual meeting point to organize the ETI community for flexible tool evaluation and coordination.

- **31.03. - 2.4.1998**

First official presentation of STTT/ETI at ETAPS' 98, the first European Joint Conferences on Theory and Practice of Software, in Lisbon, Portugal, including online ETI demonstrations.

- **1999**

Start of the platform re-engineering process, motivated by feedback of the platform users.

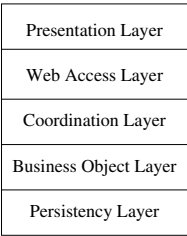


Fig. 1. The Layers of a Web Application

- **06.2000**
Launch of the ETI community site (<http://www.eti-service.org>) and of the current version of the ETI platform.
- **2001**
Installation at Kansas State University, as support platform for the Bandera project. Installation at the International School of Advanced Studies in Dortmund.
- **2002**
Installation at Carnegie Mellon University in support of distributed educational activities.

4 The ETI Community Online Service

The ETI Online Service is an environment designed for users wanting to create their own application-specific or community-specific ETI sites: These sites are intended to constitute a meeting point for distributed communities for tool-related discussion and experimentation. In particular they are intended to spread new technologies at the tool-level and to convince and teach potential users via an easy experimentation.

In the following, we will give some information about the community aspect. The subsequent sections will then discuss the central requirements and ETI-concepts while focussing on the coordination aspects.

4.1 The ETI Community Infrastructure

The ETI platform offers a virtual meeting point, where people involved in the development, use, and maintenance of the platform (called ETI community) may exchange information, and discuss on topics of their interest. This virtual meeting point is implemented as a Web application that provides access to mailing lists, discussion groups, frequently asked questions, software updates, documentation, etc. To ensure that everyone gets access to the right information in a fast and easy way, the application is customizable by the end users. Additionally, a profile-based permission concept controls the access to the offered functionality and

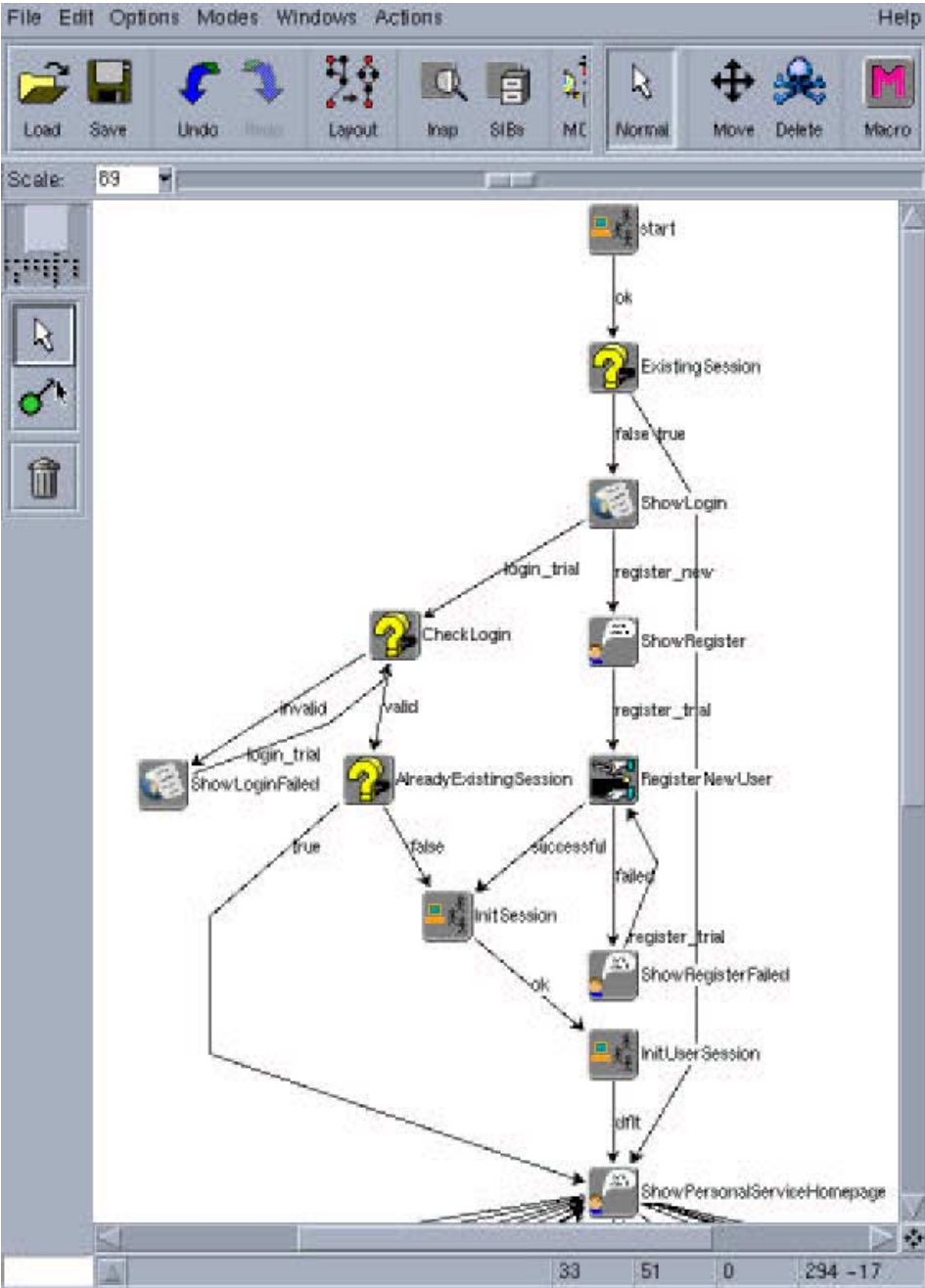


Fig. 2. The Service Logic Graph Editor

data. Here we document the development process and the tools which have been used to build this component of the platform.

The ETI community application has been implemented and is continuously being extended, as mentioned in the brief project history, using the Service Definition Environment (SDE) [48, 47] of the METAFrame project. The SDE is a workflow design tool and it offers a compiler which generates a Java Servlet [23] out of the workflow specifications (see [8] for more information). The main difference between this technology and other widespread techniques for Internet site development (like e.g. Allaire's Cold Fusion Scripts [10] and Server Side Includes [13]) is the strict architectural separation of different layers with different purposes: as shown in Figure 1, we distinguish the application's GUI (*presentation layer*), the application logic (*coordination layer*), and the base functionality implemented by the *business object layer*.

This clean structure has clear advantages wrt. ease of instantiation and of maintenance. To this aim, we introduce an explicit representation of the application logic as a directed graph, called Service Logic Graph, like the one shown in Figure 2, instead of embedding database queries or scripting fragments into the HTML code. SLGs are the way we design the ETI service itself: within the Service Logic Graph, nodes represent functional entities called Service Independent Building Blocks (SIBs) and edges represent the flow of control between the SIBs. This graph can be easily edited and modified without programming, simply using the service logic graph editor of the Service Definition Environment, as shown for a small service graph example in Figure 2.

As consequence, the service logic graph provides the developer with a *global* instead of a *local* (page-oriented) view of the application. This makes the development and the maintenance of the application easier. Using the service logic graph editor, the application can be configured without any programming effort and a new GUI may be designed without touching the application's code.

In contrast to other known tools, the application can be analyzed already during its design using the validation features offered by the Service Definition Environment (see [44] for more details). These features are used to validate the application on the service logic level, which summarizes the application in a workflow-oriented fashion. Service logic graphs do not consider the implementation code of the SIBs, which are considered atomic entities and assumed to be correct. This has two reasons. Since a typical end user of the Service Definition Environment has little or no programming skills, he would be unable to interpret error reports on the implementation level. On the other hand, verification of complex Web applications on the code level is unfeasible if not undecidable.

In addition to the tool, we introduced a process which supports the development of reliable Web applications. One of the main aspects of this process is the collaborative organization of different user groups having different skills: HTML specialists build the GUI of the application, application experts configure the application using the Service Definition Environment, SIB integrators provide the SIBs and specialists having skills in object-oriented techniques design and implement the base functionality required to realize the SIBs. In combina-

tion with the validation features offered by the Service Definition Environment, this division of labor is our key concept for developing complex, reliable Web applications with a cooperative distributed effort.

4.2 Organizational Meeting Point

The ETI community site <http://www.eti-service.org> organizes the collaborative effort to enhance and application-specifically instantiate the platform by coordinating the quite heterogeneous and worldwide distributed ETI community.

The ETI online community functions as an informal, virtual meeting point for the exchange of experiences and opinions: end users, tool providers, tool integrators, platform developers and site managers are invited to comment, to query, and to provide feedback about the platform, in particular concerning its potential to support ETI-instantiations. The focus of this site is the generic platform which can be instantiated using tools from a chosen application domain. This application or community-specific instantiation is typically done at one of the ETI-sites. These instantiations may then well be used for tool-related teaching and as a platform for potential customers looking for tools solving their current problem. After setup, users can access the software tools provided by a site using the *ToolZone* client software, available on the site.

The functionalities offered by the platform cover two distinct needs:

- features used to organize the ETI community like (tool-specific) discussion groups, newsletters, a developer’s corner etc., and
- features that give direct access to the tools contained in the ETI site’s tool repositories.

Together, these features allow a user to retrieve information on the available tools, execute single tool functionalities, combine tool functionalities, execute tool combinations, and exchange and discuss information and experiences with tool providers and other ETI users in a cooperative virtual space.

The way we model and conceptualize the treatment of complex tools is explained in the following section.

5 The ETI Meta Model

The ETI meta model introduces the terms used in modelling and designing the repository of each ETI site’s application domain, and defines the relations between them. Figure 3 shows a UML class diagram of this model, whose key notions are detailed in the following sections.

5.1 Activities

In general, tools are not integrated into the tool repository as monolithic blocks. Rather, single tool features are identified and prepared to be accessed via the

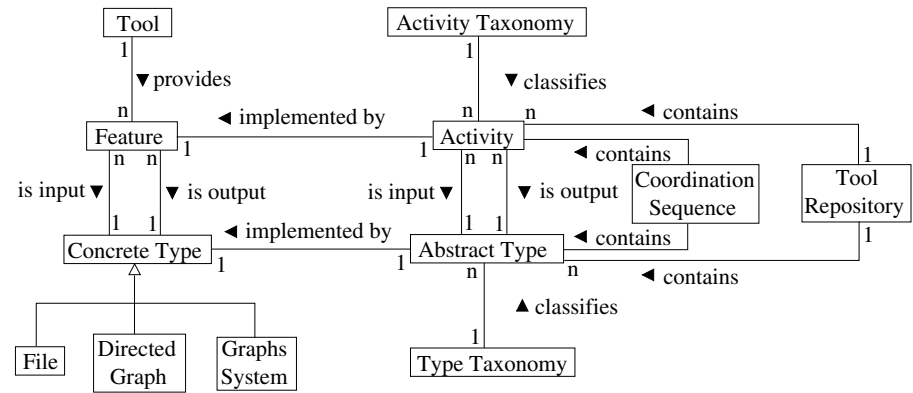


Fig. 3. The ETI Meta Model

platform. To this aim each tool feature is wrapped into an ETI-specific component called ETI-Activity, which constitute the elementary components used later for the tool coordination. An ETI-Activity models a tool feature as a *transformational* entity. This means that a tool feature can be seen as a component taking an object of type T_1 as input and delivering an object of type T_2 as output. Tool features beyond this kind of modelling can also be integrated but are formally treated as activities during our analyses.

Each activity provides an interface, which defines the view of the platform on this component. In contrast to other component models, like DCOM [51] or JavaBeans [26], the structure of the interface of an activity is very simple: it contains only one operation reflecting the transformational character of the activity. Thus the signature of this operation consists only of an input type and an output type.

As an example, Table 1 shows a set of simple activities identified in the world of the UNIX operating system, including the associated text processing commands their implementation is based on.

Activity Name	Input Type	Output Type	Unix Cmd.	Description
latex	TEXFile	DVIFile	latex	Structured text formatting and type setting program.
dvips	DVIFile	PSFile	dvips	Converts a TeX DVI file to PostScript.
gv	PSFile	Display	gv	A PostScript viewer

Table 1. Activities Table for text editing-related commands

Up to this point, activities are just specified by one operation defining their input/output behavior, and types are just abstract names. This declarative view on an activity is sufficient for ETI's synthesis facility which combines single

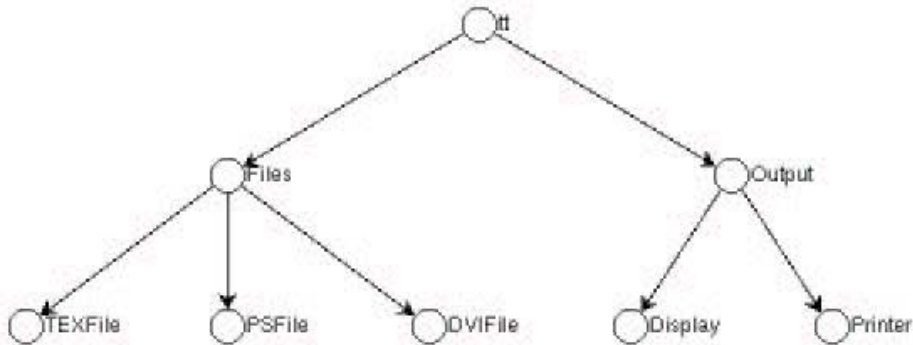


Fig. 4. A UNIX-based Type Taxonomy

activities to sequential programs (see Section 6.1). However, in order to execute an activity, an implementation of the entities contained in the tool repository (i.e. the activities and the types) is required.

The implementation of a type is very simple: the (abstract) types found in the activity signatures are mapped to (concrete) data types used by the chosen tool feature. In the current status of the project, three generic data types are available: files, directed graphs and graph systems. These generic types cover the most important data types required in ETI's first application domain: the analysis and verification of distributed (real-time) systems.

Activities can be executed in two modes. In the stand-alone execution mode a single activity is run by the user. Activities can also be executed in the context of a coordination program. In this case, the activity is run in tool-coordination execution mode. Consequently, the implementation of an activity is done by providing two code fragments, each of them specifying the behavior of the activity in one execution mode.

5.2 Taxonomies

For a flexible handling (retrieval, loose object specification, abstract views), activities and types are classified by means of *Activity and Type Taxonomies*. Formally, a taxonomy (defined over a set of objects S) is a projection of the power-set lattice over S onto elements with a particular profile, which are identified by names.

Taxonomies can be represented as directed acyclic graphs (DAGs) (see Fig. 4), where each leaf represents an atomic entity (here activity or type) and each intermediate node represents a set of entities. Conceptually, edges reflect an *is-a* relation between their target and source nodes. From the semantics point of view, an intermediate node represents all atomic entities which can be reached from this node within the taxonomy DAG. With respect to this model, edges reflect the standard set inclusion.

Fig. 4 shows a simple classification of the types introduced in Table 1. Here, the type group `tt` represents all types which are available in the tool repository, i.e. `TEXFile`, `PSFile`, `DVIFile`, `Display`, and `Printer`. In contrast, the type group `Files` only represents the types `TEXFile`, `PSFile`, and `DVIFile`.

6 Coordination Sequences

On the basis of the available activities, users can build and then execute sequential programs called *coordination sequences*, which are finite paths of the form

$$T_1 \xrightarrow{a_1} T_2 \xrightarrow{a_2} T_3 \cdots T_{n-1} \xrightarrow{a_{n-1}} t_n$$

where T_i is a type and a_i is an activity which transforms an object of type T_i into an object of T_{i+1} . For illustration, we provide the concrete example of a UNIX-based coordination sequence, which uses the activities introduced in Table 1.

Example: A UNIX-based Coordination Sequence

The following coordination sequence takes a tex-file as input, runs the `latex` command on it, transforms the DVI result into a PostScript file and displays the PostScript file on the screen using the PostScript viewer `gv`.

$$TEXFile \xrightarrow{latex} DVIFile \xrightarrow{dvips} PSFile \xrightarrow{gv} Display$$

6.1 Loose Specifications

Building a coordination sequence manually is a non-trivial task: the user must know the activities contained in the tool repository, and must solve tool interfacing issues like finding the right data type transformer to connect features possibly provided by different tools. To ease this task, the ETI platform offers a synthesis component which automatically generates coordination sequences from abstract descriptions. These abstract descriptions are loose in two orthogonal dimensions:

- *Local Looseness*: Characterization of types and activities at the abstract level of taxonomies, instead of enumerating them explicitly. Here names contained in the taxonomies are interpreted as propositional predicates. They can be combined by means of the Boolean operators $\&$ (*and*), $|$ (*or*), and $\sim$ (*not*) to specify sets of activities and types. With respect to the type taxonomy presented in Fig. 4, the formula $tt \ \& \ \sim \ Files$ characterizes the types `Display` and `Printer`.
- *Global/Temporal Looseness*: The characterization of whole coordination sequences is done in terms of abstract constraints specifying precedences, eventuality, and conditional occurrence of single taxonomy entities, rather than specifying the precise sequence of the types and activities. This kind of looseness is based on Semantic Linear-time Temporal Logic (SLTL) [49].

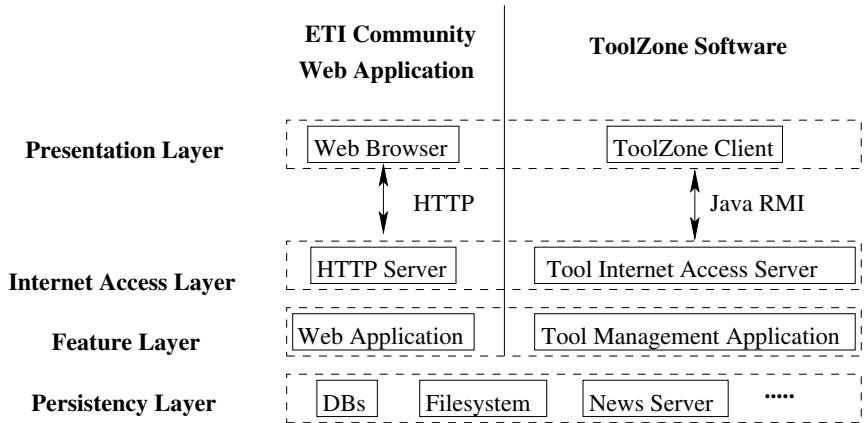


Fig. 5. The logical Platform Architecture

With respect to the activities shown in Table 1 and the type taxonomy presented in Fig. 4, the user could write a loose specification of the form of

$$TEXFile > Output$$

to query for all coordination sequences able to display a `TEXFile` on some `Output` device. In this example, ETI's synthesis component delivers the coordination sequence shown in the above Example as result. Note that the specification does not contain the exact description of the output device: we use the type group `Output` instead of one of the atomic types `Display` or `Printer` (local looseness). Additionally, the formula only specifies that an `Output` object must follow a `TEXFile` object without fixing when (as direct successor or anywhere later in the future) this occurrence should take place (temporal looseness). This is specified using the *before* operator $>$.

Using loose specifications, the user characterizes what he wants to achieve instead of how to achieve it. This goal-oriented aspect is the main difference between ETI's coordination facility and other coordination approaches like UNIX piped commands, scripting languages (e.g. Perl [52], Python [35]) or the ToolBus [6, 32]: there the user is forced to specify the exact coordination process.

7 The Platform's Software Architecture

Logically, each ETI-site's platform is built using a four layered software architecture (see Fig. 5).

The *persistency* (database) layer stores the tool repository as well as the discussion groups, mailing list archives, and user data. On top of this layer, the *feature* layer provides the real functionality offered by the ETI platform. This layer contains a Java-based Web application which implements the community services as well as a C++-based tool management application that gives access

to the tool-related functionality. In the terminology of Fig. 1, it realizes both the coordination layer and the business object layer, which are there distinctly mentioned since that subdivision is peculiar of working with the Agent Building Center. The *Internet access* layer makes the functionality offered by the platform accessible via the Internet. For its implementation we use a standard HTTP server (like Apache [3]) to make the community services available to the public. The functionality provided by the tool management software is encapsulated by the tool Web access server which can be connected via Java RMI [20]. The *presentation* layer contains software components that enable the site users to access the functionality provided by the feature layer via an intuitive graphical user interface. Here, a standard Web browser can be used to participate in the community infrastructure. In contrast, a Java client (called ToolZone client) is required to gain access to the tools contained in the tool repository. This ToolZone client can be launched from the Web browser using the Java Web Start technology [28].

Usability, security, openness, and maintainability are the key requirements for a successful Internet service. In the ETI platform they are addressed as follows:

7.1 Usability

- Intuitive graphical user interface (GUI) for the client application: ETI provides an elaborate GUI with help functionality and hypertext support.
- Easy access to the features provided by the platform: The taxonomy-based retrieval mechanism together with the synthesis of coordination sequences from loose specifications enables a goal-oriented use of the platform without requiring any knowledge about the repositories' content and the interface-constraints between the integrated functionalities.
- Minimal set of requirements for client machines: The community features of the ETI platform can be used with a standard Web browser, and the access to the tool repository is provided by a Java-based software which requires a standard Java Runtime Environment to be installed on the client machine.
- Availability and performance: ETI distributes its functionality over a server farm. A load-balancing component ensures a performant access to the provided functionality.

7.2 Security

The ETI platform offers protected home areas and a profile-based security concept to control the access to personal data and functionalities. In addition, sensitive parts of the community features are only accessible using Secure Socket Layer (SSL) based encryption.

7.3 Openness

"A-posteriori"-integration is a key requirement for the ETI platform: every tool should be integrable into the tool repository as an ETI-activity. In fact, ETI

makes no assumptions on the design, the availability (e.g. communication via source code or binary version, operating system), and the accessibility (e.g. system calls, Java RMI [20], CORBA [38]) of the tool functionalities.

7.4 Maintainability

End users want easy, fast and reliable access to the functionality provided by the ETI platform. In particular, they want to

- browse through the tool repository looking for available activities,
- combine activities to coordination sequences and execute them using the ToolZone client,
- access tool-related information and discussion groups via a standard Web browser.

The corresponding maintenance of the ETI platform, which we support by means of the ETI community site, requires cooperation by people of different skills and profiles. We distinguish four groups of experts involved in different ways with the platform: tool providers, tool integrators, platform developers, and site managers.

Tool Providers Tool providers are end users with special focus. Whereas the standard end user looks for candidate tools or a combination of tool functionalities to solve a certain problem, the tool provider is mainly interested in publishing his tool and getting valuable feedback by the end users. In addition to the tool itself, the tool provider supplies benchmarks, examples, and descriptions defining the tool's profile.

Tool Integrators Tool integrators make new activities and types available to the ETI tool repository. They investigate the tools to be integrated, identify new activities, and establish connections between the new activities/types and already available ones. Thus they need a good understanding of the tools to be integrated and of the application domain modeled in the tool repository. In addition, they must be expert in C++, the language of the tool management application. We support the integration task by providing a tailored process together with some utility tools (see [8]).

Platform Developers We distinguish two groups of platform developers. One group providing new SIBs, which can then be used to extend the ETI community application. An another group adding new functionality to the ToolZone software. Whereas SIB developers are only required to have Java skills, ToolZone software developers must have Java and C++ knowledge. In addition, ToolZone software developers need a good understanding of the platform's architecture.

Site Managers Site managers build up and maintain an ETI site. This includes setting up the servers, installing the required software and customizing the site-specific version of the community application. They need expertise about UNIX-based operating system (e.g. Linux or SUN's Solaris) for the installation process. The customization of the Web application, which requires some fundamental HTML knowledge in order to adapt the look and feel of the Web site, is done with our Service Definition Environment.

8 ETI and the Petri Net Technology

As will be explained in the following two subsections, ETI and the Petri Net Technology can mutually profit from each other: whereas ETI delivers an infrastructure which may provide significant support for the development and dissimulation of the Petri Net Technology, Petri nets generalize ETI's current transition system-based modelling of tool execution by allowing an explicit modelling of concurrency.

8.1 ETI for the Petri Net Technology

Petri net technology comprises a wealth of variants, each distinguished by its expressive power: There are extensions of Petri nets for capturing complex programming languages, as well as performance, probability, and real time, which have been adopted by industry in particular for safety critical systems. Prominent application examples include the application-specific usage of the application-independent technologies, implemented in tools for the development and validation of traffic control systems [39], for workflow and business process modelling (e.g. [1, 2, 19, 14, 17, 22] or for synthesis of asynchronous controllers (like Petrify [11]) to name just a few. On their turn, Petri Nets have been object of studies from several points of view, e.g. concerning their formal definition as visual languages [4, 5].

This success is, last but not least, due to the wealth of supporting tools for the analysis, synthesis, optimization, transformation and validation of nets, many of which can be found via the Petri Nets Tool Database [42].

For example, Design/CPN ([33]) is a tool package supporting the use of Coloured Petri Nets [29]. It includes several tools, each offering many functionalities: an editor for the construction, modification and syntax check of CPN models, a simulator supports interactive and automatic simulation, algorithms for the construction and analysis of state spaces or reachability graphs/trees, and a performance tool that supports simulation based performance analysis.

PEP (Programming Environment Based on Petri Nets) [40] is conceived for the design and verification of parallel systems, which can be modelled as parallel programs, processes in a process algebra, or Petri nets. It offers verification algorithms that use partial order methods to avoid the state explosion problem, and interfaces from SDL, and to INA, SMV, and SPIN.

Petrify [41] includes analysis, transformation, and optimization algorithms: it is able to obtain Petri nets with some specific properties (e.g. pure, free choice, unique choice, place irredundant)[12].

The Petri Net Kernel (PNK)[31] is itself a toolkit, currently available in Python and Java. It provides tool programmers with an infrastructure for writing Petri Net algorithms. It relieves the programmer from implementing standard functions on Petri Nets, provides a graphical user interface e.g. for simulation and animation purposes, and it support extensibility to new kinds of Petri Nets by means of PN Type definitions. The PNK can be regarded as a means to implement new activities and types as a basis for the ETI coordination.

Moreover, there is work on interchange formats, e.g. the Petri Net Markup Language (PNML)[30], and compositionality issues, e.g. the box calculus [7], addressing scalability and reuse. The ‘Petri Net Baukasten [15] aims at organizing this diversity for a wide community of users. Thus we are envisaging a fruitful cooperation between the ETI initiative and the Petri Net Baukasten team. Here both the community service infrastructure as well as the interoperability platform may beneficially come into play.

8.2 Petri Net Technology for ETI

The ETI synthesis component bases on transition systems as a model for execution of sequences of tools. This captures quite a number of applications and it is easy to understand. Moreover, there are numerous formal methods which can be directly applied on this level without any preprocessing. Petri nets are more general and expressive. They allow a direct modelling of concurrency, which can be beneficially exploited for specifying distribution potential for the execution process. We are in particular considering a Petri net-like modelling for realizing in ETI a ‘data flow’-oriented execution paradigm.

9 Conclusions and Perspectives

We have presented the ETI approach, which aims at overcoming the typical hesitation to try out new technologies: serious hurdles, like installation of the tools, getting acquainted with new user interfaces, lack of direct comparability of the results and of performances, are eliminated. In particular, it is intended to support fair, application-specific evaluation of various competing tools on the basis of predefined benchmarks. Together with the ETI community online service, which provides topic-specific discussion fora, this infrastructure would ideally complement the discussed Petri Net initiatives of Sect. 8: not only can the Petri Net tools profit from the ETI infrastructure, but also the ETI project may well profit from the results of the Petri net research and corresponding tools. Thus we envisage Petri Net tools not only in the ETI repository for online experimentation, but also in the algorithmic kernel of the ETI platform as means for generalizing the model for execution.

References

1. W. van der Aalst, K. van Hee: *Workflow Management: Models, Methods, and Systems*, MIT Press, 368 pp., ISBN 0-262-01189-1.
2. W.M.P. van der Aalst: *The Application of Petri Nets to Workflow Management*, Journal of Circuits, Systems and Computers (8), 1998, pp.21–66.
3. *The Apache Project*: <http://www.apache.org>.
4. R. Bardohl, H. Ehrig: *Conceptual Model of the Graphical Editor GENGED for the Visual Definition of Visual Languages*, 6th Int. Workshop on Theory and Applications of Graph Transformation, TAGT'98 Selected Papers (Ehrig, Engels, Krewski, Rozenberg Eds.) Springer Verlag, LNCS N. 1764 (2000), pp. 252-266
5. R. Bardohl, H. Ehrig, C. Ermel: *Generic Description, Behavior and Animation of Visual Modeling Languages*, Proc. IDPT 2000, Dallas, USA, June 2000.
6. J. A. Bergstra, P. Klint. *The ToolBus Coordination Architecture*, Proc. Int. Conf. on Coordination Models and Languages, LNCS 1061, pp. 75-88, Springer Verlag, 1996.
7. Eike Best and Maciej Koutny: *A Refined View of the Box Algebra*. Application and Theory of Petri Nets'95
8. V. Braun, T. Margaria, C. Weise: *Integrating Tools in the ETI Platform*, [50], pp. 31-48.
9. *CGI: Common Gateway Interface*, <http://www.w3.org/CGI/>.
10. *Cold Fusion, a cross-platform Web Application Server*, <http://www.allaire.com/products/coldfusion/>.
11. J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev: *Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers*, IEICE Transactions on Information and Systems, Vol. E80-D, No. 3, March 1997, pages 315-325.
12. J. Cortadella, M. Kishinevsky, L. Lavagno and A. Yakovlev. Deriving Petri Nets from Finite Transition Systems, IEEE Transactions on Computers, Vol. 47, No. 8, pages 859-882, August 1998.
13. T. David, *Server Side Includes (SSI) Tutorial*, <http://www.tdscripsts.com/ssi.html>.
14. J. Dehnert: *Four Systematic Steps Towards Sound Business Process Models*, Proc. 2nd Intern. Colloquium on Petri Net Technologies for Modelling Communication Based Systems, Fraunhofer Gesellschaft ISST, Berlin 2001, pp. 55-64
15. J. Dehnert, M. Gajewsky, S. Lembke and H. Weber: *The "Petri Net Baukasten": 2nd Installment*. Proc. Uniform Approaches to Graphical Process Specification Techniques(UniGra), Satellite workshop to European Joint Conferences on Theory and Practice of Software (ETAPS) , Electronic Notes in Theoretical Computer Science (ENTCS), Vol. 44, No. 4, Genova, Italy, 2001.
16. L. A. Dennis, G. Collins, M. Norrish, R. Boulton, K. Slind, G. Robinson, M. Gordon, T. Melham: *The PROSPER Toolkit*, Proc. TACAS 2000, Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 1785, pp. 78-92, Springer Verlag.
17. Desel, J., Erwin, T.: *Hybrid specifications: looking at workflows from a run-time perspective*. International Journal of Computer System Science & Engineering 15 Nr. 5. S. 291 - 302, 2000.
18. Design CPN Website: <http://www.daimi.au.dk/designCPN/>.
19. G. Dinkhoff, V. Gruhn, A. Saalmann, M. Zielonka. *Business Process Modeling in the Workflow Management Environment LEU. Proc. 13. Int. Conf. on the Entity-Relationship Approach*, pp.46-63, Manchester (UK), Dec.1994, Springer, LNCS 881.

20. T. Downing: *Java RMI, Remote Method Invocation*, IDG Books Worldwide Inc., 1998.
21. *Formal Methods Europe*, <http://www.fmeurope.org>.
22. R. Heckel: *Open Petri Nets as Semantic Model for Business Process Integration* Proc. 2nd Intern. Colloquium on Petri Net Technologies for Modelling Communication Based Systems, Fraunhofer Gesellschaft ISST, Berlin 2001, pp. 129 - 134.
23. J. Hunter, W. Crawford: *Java Servlet Programming*, O'Reilly, 1998.
24. *International Journal on Software Tools for Technology Transfer*, <http://sttt.cs.uni-dortmund.de>.
25. *HyTech Home Page*, <http://www-cad.EECS.Berkeley.EDU/~tah/HyTech/>.
26. *JavaBeans: The only Component Architecture for Java Technology*, <http://java.sun.com/products/javabeans/>.
27. *Java Developer Connection*, <http://developer.java.sun.com/developer/>.
28. *Java Web Start*, <http://java.sun.com/products/javawebstart/>.
29. K. Jensen: *A Brief Introduction to Coloured Petri Nets*. Proc. of the TACAS'97 Workshop, Enschede, The Netherlands 1997, Lecture Notes in Computer Science Vol. 1217, Springer-Verlag 1997, 203-208.
30. M. Jünger, E. Kindler, M. Weber: *Towards a Generic Interchange Format for Petri Nets*. In: R. Bastide, J. Billington, E. Kindler, F. Kordon, and K. H. Mortensen (eds.): *Meeting on XML/SGML based Interchange Formats for Petri Nets*. pages 1-5, Aarhus, Denmark, 21st ICATPN. June 2000.
31. E. Kindler, M. Weber: *The Petri Net Kernel - An Infrastructure for building Petri Net Tools*. *International Journal on Software Tools for Technology Transfer*, 3 (2001), Springer Verlag, pp. 486-497.
32. P. Klint, P. Olivier: *The TOOLBUS Coordination Architecture: A Demonstration*, Proc. of 5th Int. Conference on Algebraic Methodology and Software Technology (AMAST'96), LNCS 1101, pp. 575-578, Springer Verlag, 1996.
33. L.M. Kristensen, S. Christensen, K. Jensen: *The Practitioner's Guide to Coloured Petri Nets*. *International Journal on Software Tools for Technology Transfer*, 2 (1998), Springer Verlag, 98-132.
34. K. G. Larsen, P. Petterson, W. Yi: *Uppaal in a nutshell*, *Int. Journal on Software Tools for Technology Transfer*, Vol. 1, pp. 134-152, Springer Verlag, 1997.
35. M. Lutz: *Programming Python*, O'Reilly, 2001.
36. T. Margaria, V. Braun, J. Kreiler: *Interacting with ETI: A User Session*, [50], pp. 49-63.
37. T. Margaria, B. Steffen: *Coarse-grain Component Based Software Development: The METAFrame Approach*, Proc. STJA'97, "Smalltalk und Java in Industrie und Ausbildung", 10.-11. September 1997, Erfurt (D), ISBN 3-00-001828-X, pp. 29-34.
38. OMG: *The Common Object Request Broker: Architecture and Specification Revision 2.0*, Technical Document ptc/96-08-04.
39. J. Padberg, L. Jansen, H. Ehrig, E. Schnieder, R. Heckel: *Cooperability in Train Control Systems: Specification of Scenarios using open Nets*, In SDPS (2000)
40. PEP Website: <http://theoretica.informatik.uni-oldenburg.de/~pep/>.
41. Petrify Website: <http://www.lsi.upc.es/~jordic/petrify/>.
42. *Petri Nets Tool Database*, <http://www.daimi.aau.dk/PetriNets/tools/db.html>.
43. *SMV guided tour*, <http://www.cs.cmu.edu/~modelcheck/tour.html>.
44. B. Steffen, T. Margaria: *METAFrame in Practice: Design of Intelligent Network Services*, in LNCS State-of-the-Art Survey *Correct System Design - Recent Insights and Advances*, Lecture Notes in Computer Science 1710, pp. 390 - 415, 1999.
45. B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration platform: concepts and design*, [50], pp. 9-30.

46. B. Steffen, T. Margaria, A. Claßen, V. Braun: *The METAFrame'95 Environment*, Proc. CAV'96, Int. Conf. on Computer-Aided Verification - Juli-Aug. 1996, New Brunswick, NJ, USA, LNCS 1102, pp. 450-453, Springer Verlag.
47. B. Steffen, T. Margaria, A. Claßen, V. Braun, N. Kalt: *Hierarchical Service Definition*, Annual Review of Communications, Int. Engineering Consortium (IEC), pp. 847-856, 1997.
48. B. Steffen, T. Margaria, A. Claßen, V. Braun, M. Reitenspieß, M. Wendler: *Service Creation: Formal Verification and Abstract Views*, 4th Int. Conf. on Intelligent Networks (ICIN'96), pp. 96-101, 1996.
49. B. Steffen, T. Margaria, B. Freitag: *Module Configuration by Minimal Model Construction*, Techn. Rep. MIP-9313, Fak. für Mathematik und Informatik, Universität Passau (Germany), 1993.
50. *Special section on the Electronic Tool Integration Platform*, Int. Journal on Software Tools for Technology Transfer, Vol. 1, Springer Verlag, November 1997
51. T. Thai: *Learning DCOM*, O'Reilly, 1999.
52. L. Wall, T. Christiansen, J. Orwant: *Programming Perl*, 3rd Edition, O'Reilly, 2000.

Inheritance of Business Processes: A Journey Visiting Four Notorious Problems

W.M.P. van der Aalst

Eindhoven University of Technology

Faculty of Technology and Management, Department of Information and Technology

P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands

w.m.p.v.d.aalst@tm.tue.nl

Abstract. The new millennium is characterized by speed and distribution in every aspect of most business and organization undertaking. Organizations are challenged to bring ideas and concepts to products and services in an ever-increasing pace. Companies distributed by space, time and capabilities come together to deliver products and solutions for which there is any need in the global marketplace. This creates new opportunities but also poses new problems. In this paper we will address four problems directly related to changing business processes in the networked economy. Two of the problems addressed surface when existing workflow specifications are changed: the *dynamic change problem* and the *management information problem*. The third problem refers to *coordination problems in inter-organizational workflows*. The fourth problem becomes relevant when predefined workflow specifications are *customized* for a particular business situation. This paper will show that these problems have common characteristics. Moreover, we will point out that an approach based on inheritance of dynamic behavior provides a partial solution for each of the problems.

1 Introduction

Inheritance is one of the cornerstones of objectoriented programming and object-oriented design. The basic idea of inheritance is to provide mechanisms which allow for constructing *subclasses* that inherit certain properties of a given *super-class*. This paper focuses on workflow processes [1, 24, 31, 43]. Therefore, a *class* corresponds to a *workflow process definition* (i.e., a routing diagram) and *objects* (i.e., instances of the class) correspond to *cases*. In most object-oriented methods a class is characterized by a set of *attributes* and a set of *methods*. Attributes are used to describe properties of an object (i.e., an instance of the class). Methods symbolize operations on objects (e.g., create, destroy, and change attribute). The structure of a class is specified by the attributes and methods of that class. Note that the structure only refers to the static aspects of the interface. The dynamic behavior of a class is either hidden inside the methods or modeled explicitly (in UML the life-cycle of a class is modeled in terms of statecharts [50]). Although the dynamic behavior is an intrinsic part of the class description (either

explicit or implicit), inheritance of dynamic behavior is not well-understood. (See [13, 14] for an elaborate discussion on this topic and pointers to related work. Examples of alternative approaches are given in [38, 57, 58].) Given the widespread use of inheritance concepts/mechanisms for the static aspects, this is remarkable. Every object-oriented programming language supports inheritance with respect to the static structure of a class (i.e., the interface consisting of attributes and methods). Since workflow management aims at supporting business processes, these results are not very useful in this context. However, the work presented in [4, 6, 13, 14] deals with inheritance of dynamic behavior in a comprehensive manner. This work is based on a particular class of Petri nets: the so-called sound workflow nets defined in [1]. This class of Petri nets corresponds to workflow processes without deadlocks, livelocks, and other anomalies. Other inheritance-based approaches abstract from the causal relations between tasks/methods, i.e., the control or routing structure is not taken into account. Some of the workflow management systems available claim to be object-oriented and thus provide some support for inheritance. For example, the workflow management system InConcert [30] allows for building workflow class hierarchies. Unfortunately, inheritance is restricted to the attributes and the structure of the process is not taken into account. Many workflow management systems have been implemented using object-oriented programming languages. However, these systems do not offer object-oriented mechanisms such as inheritance to the workflow designer or the designer has to program code to benefit from the object-oriented features provided by the host language. Nevertheless, we think that inheritance is a very useful concept for workflow management. Therefore, we advocate the use of the inheritance notions presented in [4, 6, 13, 14] for the four workflow related problems already mentioned in the abstract:

1. How to deal with the *dynamic change problem*? (Avoiding consistency problems when migrating cases from one process to another.)
2. How to deal with the *management information problem*? (Providing aggregate management information of work in progress in the presence of many versions/variants of the same workflow process.)
3. How to handle *coordination problems* in *inter-organizational* workflows by enforcing local consistency rules?
4. How measure the *difference* between two processes? (Delta analysis to estimate the effort required to customize a process.)

The goal of this paper is to demonstrate that several problems related to workflow design and analysis can be addressed using the principle of inheritance. This paper does not provide new scientific results but integrates existing results which have been published in several papers [3, 5, 6, 7, 10, 14]. This way we hope to reveal commonalities and show the relationships between four practical problems and our work on inheritance.

2 Problems

In this paper, we focus on business processes which can be characterized as workflow processes. Workflows are *case-based*, i.e., every piece of work is executed for a specific case: an order, an insurance claim, a tax declaration, etc. The objective of a workflow management system is to handle these cases (by executing *tasks*) as efficiently and effectively as possible. The *workflow process definition* specifies which tasks need to be executed and in what order. When a task is executed for a case, this is usually done by using one or more resources, e.g., a machine, an employee, etc. In this paper, we use Petri nets [48] to represent workflow process definitions as indicated in [1]. For more papers on the application of Petri nets to workflow management we refer to [8, 11, 12, 17, 19, 22, 23, 29, 42, 49].

A Petri net represents a workflow if and only if it has exactly one starting place (*source*) and exactly one end place (*sink*) and the net obtained by adding a transition with the sink as the only input place and the source as the only output place is strongly connected. The latter is the case if for every two nodes x and y in the net, there is a path from x to y . Petri nets with this particular structure are called *workflow nets*. Each task is modeled by a transition. Tasks are connected by places (represented by circles) to specify the ordering of tasks. Places may contain tokens (represented by black dots). The *state*, often called *marking*, is the distribution of tokens over places. A transition, i.e., a task, is enabled if and only if each of the input places contains a token. Enabled transitions can fire while removing tokens from the input places and putting tokens on the output places. A detailed description of the class of workflow nets is beyond the scope of this paper and not needed for the remainder. However, some basic knowledge of Petri nets is needed to fully understand the concepts.

2.1 Problem 1: Dynamic Change

At the moment, there are more than 200 workflow products commercially available [24] and many organisations are introducing workflow technology to support their business processes. It is widely recognised that workflow management systems should provide flexibility [7, 8, 9, 12, 16, 21, 28, 33, 36, 37, 47, 56]. However, today's workflow management systems have problems dealing with *changes*, e.g., new technology, new laws, and new market requirements may lead to (structural) modifications of the workflow process definition at hand. In addition, ad-hoc changes may be necessary, e.g., because of exceptions. The inability to deal with various changes limits the application of today's workflow management systems.

In this paper, we restrict ourselves to changes in the process perspective. In the process perspective there are basically two types of changes:

- *Individual (ad-hoc) changes*, i.e., ad-hoc adaptation of the workflow process: A single case (or a limited set of cases) is affected. A good example is that of a hospital: If someone enters the hospital with serious heart problems, you are not going to ask him for his ID, although the workflow process may prescribe this. For these ad-hoc changes one can distinguish between entry

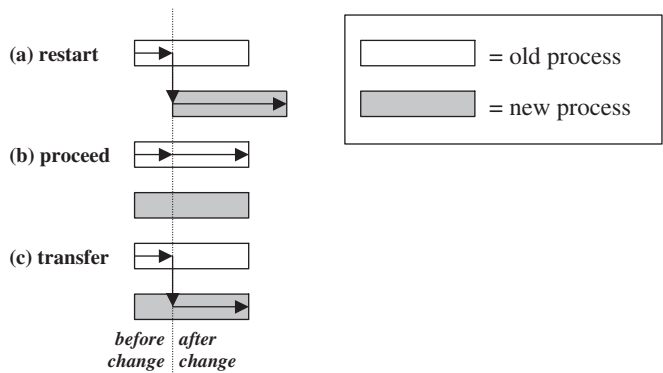


Fig. 1. How to handle running cases?

time changes (changes that occur when a case is not yet in the system) and on-the-fly changes (while in the system, the process definition for a case changes).

- *Structural (evolutionary) changes*, i.e., evolution of the workflow process: All new cases benefit from the adaptation. A structural change is typically the result of a BPR effort. An example of such a change is the change of a 4-year curriculum at a university to a 5-year one.

While executing a change there are typically running cases in the system. Figure 1 shows three ways to deal with these active workflow instances: (a) *restart*: running cases are rolled back and restarted at the beginning of the new process, (b) *proceed*: changes do not affect running cases by allowing for multiple versions of the process, and (c) *transfer*: a case is transferred to the new process. The term *dynamic change* [21] is used to refer to the latter policy. Restarting cases causes no real difficulties except that it is often difficult to rollback the tasks that have already been executed. The proceed policy also causes hardly any problems. In fact, it is the only policy truly supported by today’s commercial workflow management systems. The only policy that causes serious theoretical and practical problems is the transfer of cases. The term *dynamic change* refers to the problem of handling old cases in a new process, e.g., how to transfer cases to a new, i.e., improved, version of the process.

Figure 2 illustrates the dynamic change problem. If the sequential workflow process (left) is changed into the workflow process where tasks *B* and *C* can be executed in parallel (right) there are no problems, i.e., it is always possible to transfer a case from the left to the right. The sequential process has five possible states and each of these states corresponds to a state in the parallel process. For example, the state with a token in *s3* is mapped onto the state with a token in *p3* and *p4*. In both cases, tasks *A* and *B* have been executed and *C* and *D* still need to be executed.

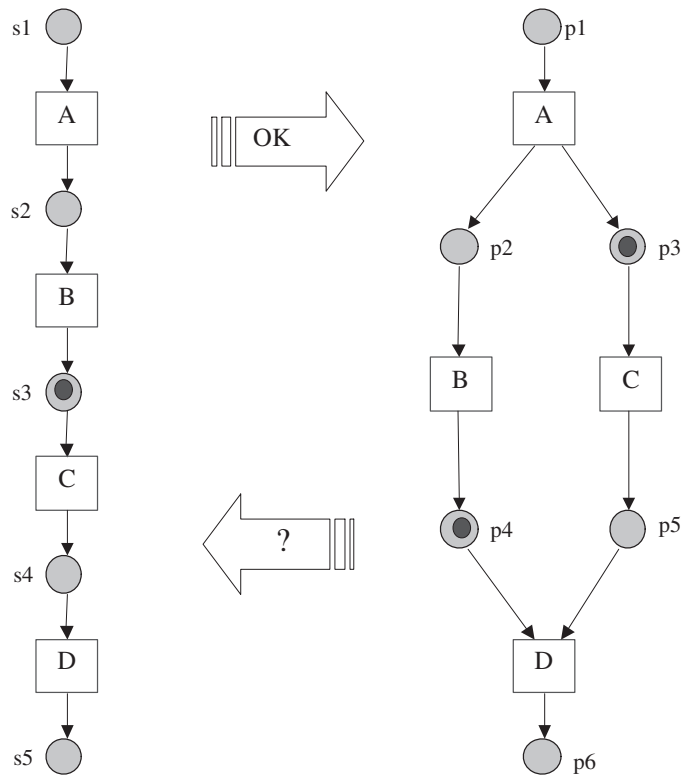


Fig. 2. The dynamic change problem

Now consider the situation where the parallel process is changed into the sequential one, i.e., a case is moved from the right-hand-side process to the left-hand-side process. For most of the states of the right-hand-side process this is no problem, e.g., a token in $p1$ is moved to $s1$, a token in $p3$ and a token in $p4$ are mapped onto a single token in $s3$, and a token in $p4$ and a token in $p5$ are mapped onto a single token in $s4$. However, the state with a token in $p2$ and $p5$ (A and C have been executed) causes problems because there is no corresponding state in the sequential process (it is not possible to execute C before B). This simple example shows that it is not straightforward to migrate old cases to the new process after a change. Some authors have proposed a solution for the dynamic change problem [12, 18, 20]. However, these solutions either require human intervention or are restricted to workflows with a particular structure.

2.2 Problem 2: Management Information

Another problem of change is that it typically leads to multiple variants of the same process. For evolutionary (i.e., structural) change the number of variants

is limited. Ad-hoc changes may lead to the situation where the number of variants may be of the same order of magnitude as the number of cases. To manage a workflow process with different variants it is desirable to have an aggregated view of the work in progress. Note that in a manufacturing process the manager can get a good impression of the work in progress by walking through the factory. For a workflow process handling digitized information, this is not possible. Therefore, it is of the utmost importance to supply the manager with tools to obtain a condensed but accurate view of the workflow processes. Figure 3 shows a workflow process with two variants: a sequential one (left) and a parallel one (middle). The numbers in the places indicate the number of cases in a specific state, e.g., in the sequential process there are 3 cases in-between task *B* and task *C*, and in the parallel process there are 2 cases in-between *A* and *B*. Since the manager requires an aggregated view rather than a view for every variant of the workflow process, the cases need to be mapped onto a generalized version of the different processes. A solution is to find the Greatest Common Divisor (GCD) or the Least Common Multiple (LCM) for the two processes shown. Finding the GCD or LCM of a set of processes is difficult and different definitions can be used (cf. [5]). Since all the states of the sequential process can be represented in the parallel process, we may choose the parallel process to present management information. Other choice are possible. However, let us assume that the GCD of the sequential process and the parallel process is indeed the parallel one. Figure 3 shows the aggregated view of the two workflow processes (right). For all places in the right-hand-side process except *m3*, it is quite straightforward to verify that the numbers are correct. The number of tokens in place *m3* corresponds to the number of cases in-between *A* and *C*. In the sequential process, there are $1+3=4$ cases in-between *A* and *C*. In the parallel process, there are also 4 cases in-between *A* and *C*, which brings the total to 8. For this small example, it may seem trivial to obtain this information. However, in general there are many variants of processes which may have up to 100 tasks and it is far from trivial to present aggregated information to the manager. The topic of generating management information was addressed in [5, 6, 7, 55, 56]. Despite its relevance for the next generation of workflow management systems only few researchers seem to be working on this topic.

2.3 Problem 3: Inter-organizational Interface Agreements

E-commerce refers to the enabling of purchasing and selling of goods and services through a communications network [32, 59]. The ability to conduct business activities involved in marketing, finance, manufacturing, selling and negotiation, electronically, is what E-commerce is all about. One major objective of adopting E-commerce strategies is to reduce costs and improve the efficiency of business processes, by replacing paper business with electronic alternatives. E-commerce, in its earliest incarnation known as *Electronic Data Interchange* (EDI), has been traditionally used by larger corporations to share and exchange information between business partners and suppliers using private networks. EDI enables the

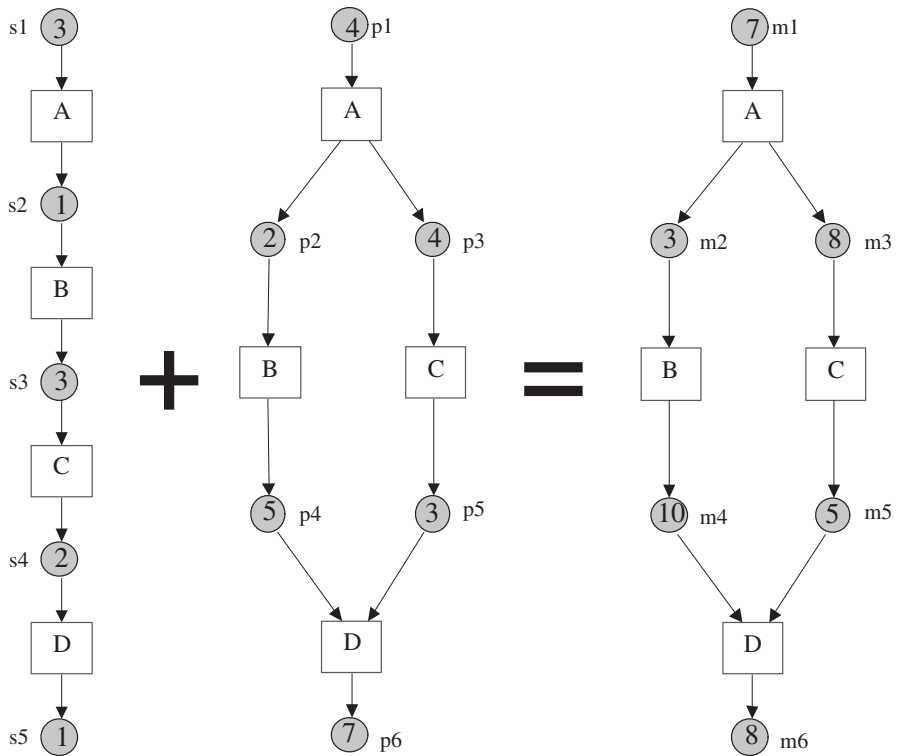


Fig. 3. Mapping cases in different processes onto one workflow process

exchange of business data from one computer to another computer. It eliminates the need to re-key information from documents or messages by supporting the creation of electronic versions of documents or messages using public standard formats, which can then be transmitted, received and interpreted by other systems. Typical applications were supply-chain management processes like order placement and processing. However with the explosive growth of the Internet in the last couple of years, E-commerce is now able to offer solutions for a much broader range of business processes than EDI previously addressed. Also, the extensive availability of the Internet has enabled smaller companies, hindered previously by the large financial investment required for these private networks, to conduct business electronically. Technologies like bar coding, automatic teller machines, email, fax, video-conferencing, workflow and the world-wide-web have continued to impact the success of E-commerce. Although the term E-commerce frequently refers to on-line retailing involving businesses and consumers, experts predict that as E-commerce continues to grow, business-to-business E-commerce will continue to enjoy the lion share of the revenue. Business-to-business E-commerce has seen tremendous growth due to the glob-

alization of the worldwide economy, which in turn is enabled in large part by the omnipresence of the Internet. Many corporations are extending their markets by mergers and strategic alliances with other companies throughout the world. Business processes of each of the business partners now become coupled in some way, creating inter-organizational workflow processes. Workflow systems enable the automated management and coordination of tasks, people, and resources involved in performing a business process, in a way that streamlines and improves the efficiency of the business process. They provide tools for modeling, enactment, administration, and monitoring of business processes. They, therefore, could be very useful in managing complex workflow processes such as those that involve multiple organizations, i.e., *inter-organizational workflows*. In particular the design of such workflow processes is often very complex and presents some challenges [15, 25, 26, 35, 39, 40, 41, 42, 44, 45].

The third problem we introduce in this paper, is a problem caused by a malfunctioning of the coordination of two business partners participating in one common process. To describe this problem we use the following terms [10]:

- The *total workflow* is the whole inter-organizational workflow as it is actually executed.
- The *public workflow*, also referred to as *contract workflow*, is the business process the partners agreed on. This workflow only comprises tasks which are of interest to all business partners involved, i.e., it is an abstraction of the real workflow.
- A *private workflow* is a part of the total workflow executed by a specific business partner (also called *domain*).

One of the characteristics of inter-organizational workflow is that the companies involved do not know about the processes enacted inside the other domains. The only view they have of the others is the abstraction specified in the public workflow. Each of the business partners involved is responsible for a part of the public workflow. Tasks which are only of local interest are added without informing the other business partners. Tasks related to quality control, internal bookkeeping, and storage management are typical examples of tasks which are only of local interest. Local extensions of the workflow may cause serious problems at the global level. Reordering tasks and adding or removing causal relations may cause deadlocks, livelocks, confusion, and other anomalies. Consider for example the public workflow shown in Figure 4. This workflow is partitioned over two domains: tasks *A*, *B*, *C*, and *D* are mapped onto domain *L* (left) and tasks *E*, *F*, *G*, and *H* are mapped onto domain *R* (right). The input place of the private workflow of domain *R* is *p8* and *p11* is the output place of this domain.

Both private workflows are modified to accommodate local needs. In domain *L* a causal relation is added between task *C* and task *B* to make sure that *C* is executed before *B*. In domain *R* the order in which *F* and *G* are executed is reversed. Figure 5 shows the two modified private workflows. Place *p12* has been added to force the execution of *C* before *B*. From a local point of view, this change does not cause any problems. For domain *L* it is perfectly acceptable to reduce the degree of parallelism. However, this change leads to

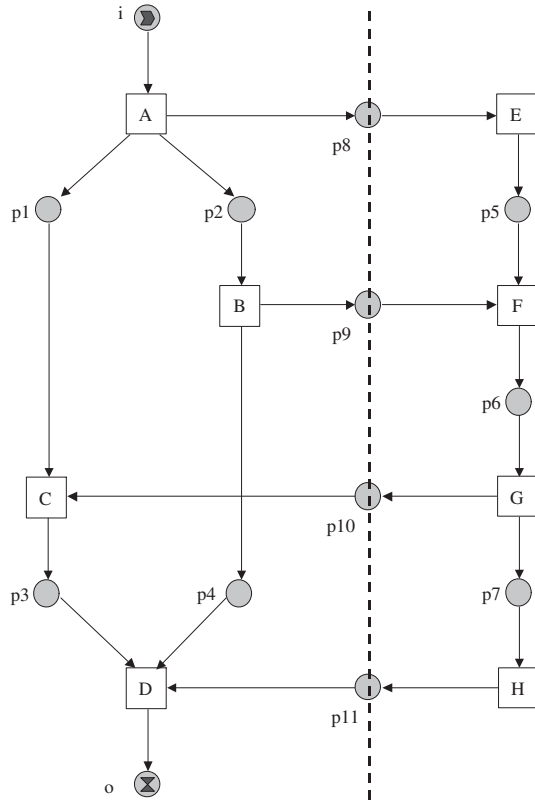


Fig. 4. The public workflow used to illustrate problems resulting from local change

a global deadlock. On the one hand C should be executed before B (place $p12$) and on the other hand B should be executed before C (causal relation via $p9$, $p6$, and $p10$). The circular dependency involving tasks C , B , F , and G causes the overall workflow process to deadlock in the state marking places $p1$, $p2$, and $p5$. Figure 5 also shows the modification of the private workflow of domain R : task F and task G are reversed. This change alleviates the problem caused by the addition of place $p12$. If both private workflows are modified as indicated in Figure 5 there would be no deadlock. However, the modification of the private workflow of domain R is also not acceptable. Based to the public workflow both partners agreed upon, the execution of task C may depend on the results of task F . By reversing the order of task F and task G there can be problems (e.g., missing data) because task C may be executed before task F . These examples show that local changes may cause global errors, i.e., modifications which are harmless from a local perspective may cause deadlocks, livelocks, missing data,

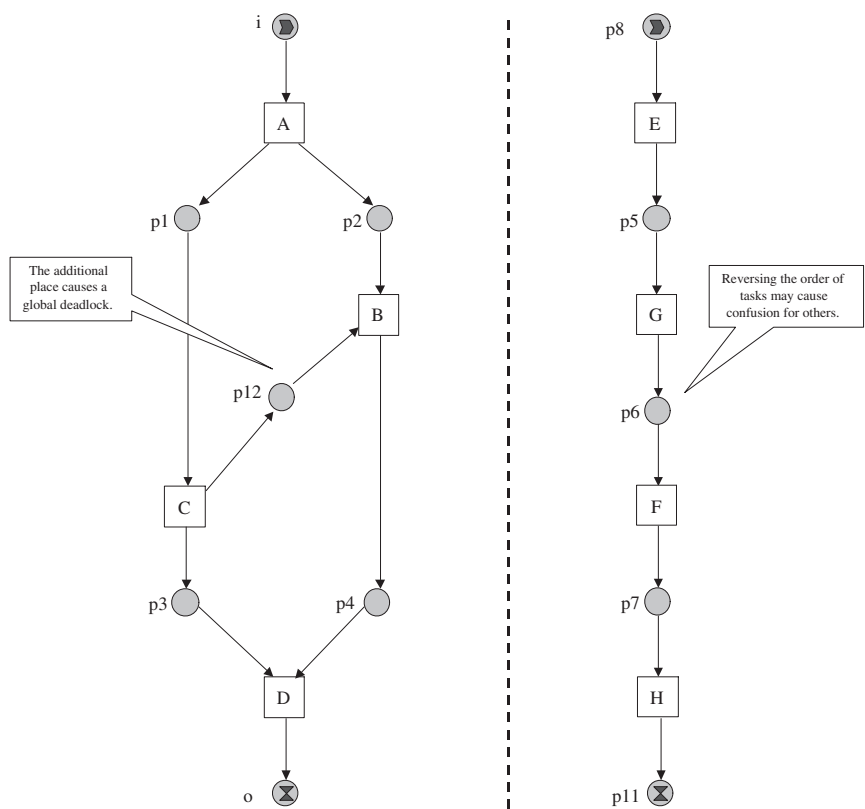


Fig. 5. Both private workflows have been modified

etc. Note that even if both business partners know about each others fragment of the public workflow and take this knowledge into account, the anomalies such as described can occur (both make conflicting changes, e.g., serializing tasks differently).

2.4 Problem 4: Customizing Business Processes

Today’s information systems support a variety of business processes. These information systems are based on general purpose software packages (e.g., workflow management systems), domain specific support systems (e.g., ERP - Enterprise Resource Planning - systems), software for a specific type of businesses (e.g., call centers, or hospitals), and company specific solutions. To avoid ‘re-inventing the wheel’ companies are ‘sharing’ software by using standard solutions. As a result, the business process that is supported may differ from the business process actually desired. Consider for example today’s generation of ERP systems, e.g., SAP R/3 and BaanERP. These systems are based on industry’s ‘best practices’.

However, the ‘best practices solution’ may not apply to a specific company. Such a company, assuming that it already made the choice to use a particular ERP, has two choices: Either the best practices solution is used despite its shortcomings or the software is *customized* by reconfiguring or reprogramming parts of the functionality. To make this choice the costs of customization need to be assessed. Moreover, if the company is in the process of selecting a standard system (e.g., an ERP system), then the costs of customization amongst different systems need to be compared.

Suppose a company interested in evaluating the costs of customization has both a concise specification of the desired business processes (e.g., the current process) and a model of the business processes supported by the standard system. In this case, the challenge is to determine (automatically) the difference between both models. We use the term *delta analysis* for such an investigation [27]. However, delta analysis is not as simple as it may seem. Consider for example the two workflow processes shown in Figure 2. What is the difference between both processes? Just considering the tasks used/supported is not sufficient. If the routing differs, then the processes also differ. A sequential process consisting of 50 tasks has little in common with a process where the same 50 tasks are executed in parallel or where only one of these tasks is selected. For delta analysis it is necessary to determine the commonality of both processes. Only if it is clear where both processes “agree on”, it is possible to point out differences. Consider the two workflow processes shown in Figure 6. What do they have in common? In a way we are again looking for the *Greatest Common Divisor* (GCD) of the two workflows (cf. Section 2.2).

The GCD of two workflow processes is the part where both processes agree on [5]. Therefore, only tasks which are used in both processes can appear in the GCD. However, it is not clear whether all tasks which are used in both processes should appear in the GCD. The two workflow processes shown in Figure 6 agree on the role of task *A* and task *D*. Therefore, the GCD should at least comprise these two tasks. For the other two tasks this is less clear. Figure 7 shows four possibilities. If both *B* and *C* are present in the GCD, then it is reasonable to put them in parallel: All execution sequences possible in the sequential process are also possible in the parallel process (alternative (A) in Figure 7). However, one might argue that alternative (A) is not the GCD since both processes do not agree on the ordering of *B* and *C*. Since both workflow processes agree on the role of task *B* relative to task *A* and task *D*, we may just add task *B* to the GCD (alternative (B) in Figure 7). The same holds for task *C* (alternative (C) in Figure 7). Since the choice between alternative (B) and alternative (C) is rather arbitrary one might argue that none of these tasks should be present (alternative (D) in Figure 7). This small example shows that the selection of a GCD is far from trivial. Note that the deliberations used when selecting a GCD are the same as when establishing the differences between two workflow processes. Therefore, a good definition of GCD is crucial for a structured delta analysis.

Note that in order to do a delta analysis there have to be descriptions of both the desired (or current) process and the process as supported by the standard

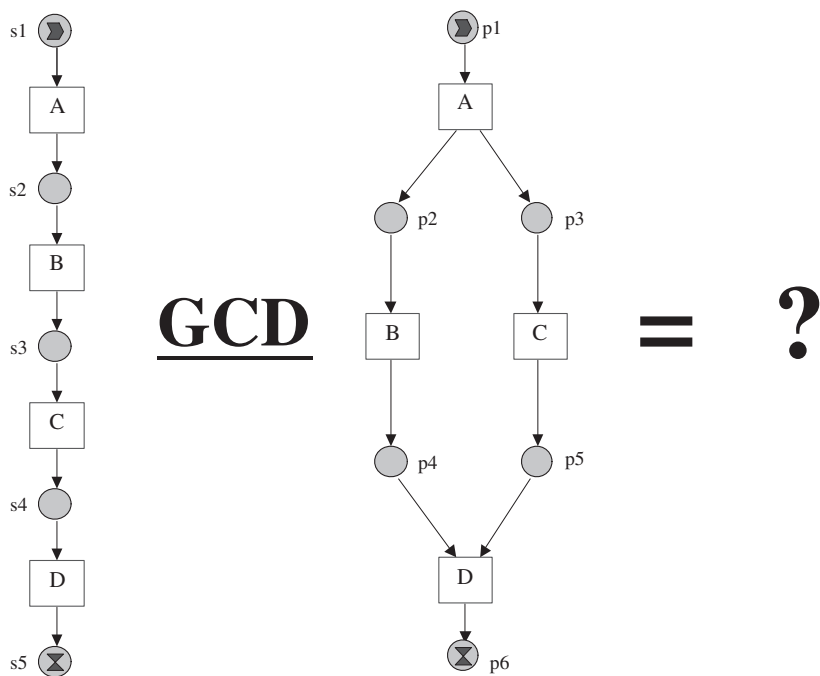


Fig. 6. What is the GCD of both processes?

software. Given this observation it is interesting to point out the role of reference models in today’s ERP systems. A good example of such a reference model is the ARIS (IDS Prof. Scheer) reference model of the SAP R/3 functionality [51, 34]. This reference model describes the business processes as they are (or could be) supported by SAP R/3. This model can be used to clarify and configure the SAP R/3 system. The Event-driven Process Chains (EPC’s) used for specifying the business processes in ARIS and SAP R/3 are similar to Petri nets [2]. If a tool such as ARIS is also used to model the desired (or current) workflow process, then all the ingredients are there to do a delta analysis. Note that apart from issues such as determining the GCD, there are other problems such as the naming of tasks. Therefore, a good ontology is a prerequisite for any form of delta analysis.

3 Inheritance of Dynamic Behavior

To tackle the four problems identified in the previous section, we propose a solution based on inheritance. First we define four inheritance notions for workflow processes (i.e., workflows specified by workflow nets). Consider two workflow processes x and y . When is x a subclass of y ? x is a subclass of superclass y

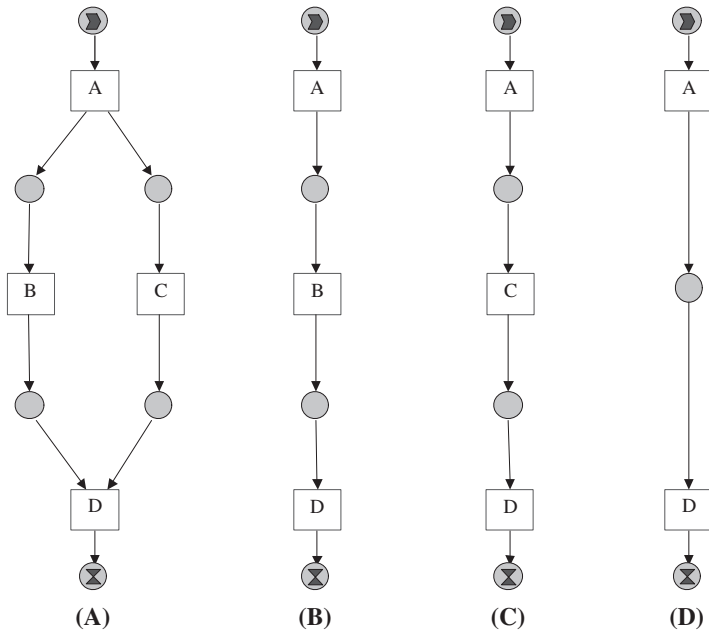


Fig. 7. Alternative GCD's?

if x inherits certain features of y . Intuitively, one could say that x is a subclass of y if and only if x can do what y can do. Clearly, all tasks present in y should also be present in x . Moreover, x will typically add new tasks. Therefore, it is reasonable to demand that x can do what y can do with respect to the tasks present in y . In fact, the behavior with respect to the existing tasks should be identical.

In [4, 14] we have identified four different notions of inheritance: *protocol inheritance*, *projection inheritance*, *protocol/projection inheritance*, and *life-cycle inheritance*. Protocol/projection inheritance is the most restrictive form of inheritance. If x is a subclass of y with respect to protocol/projection inheritance, then x is a subclass of y with respect to protocol inheritance *and* projection inheritance. Life-cycle inheritance is the most liberal form of inheritance, i.e., protocol and/or projection inheritance implies life-cycle inheritance.

The notion of projection inheritance is based on *abstraction*: *If it is not possible to distinguish x and y when arbitrary tasks of x are executed, but when only the effects of tasks that are also present in y are considered, then x is a subclass of y with respect to projection inheritance.*

For distinguishing x and y under projection inheritance we only consider the tasks present in both nets (i.e., in y). All other tasks in x are renamed to τ . One can think of these tasks as silent, internal, or not observable. Since branching

bisimulation [13] is used as an equivalence notion, we abstract from transitions with a τ label, i.e., for deciding whether x is a subclass of y only the tasks with a label different from τ are considered. The behavior with respect to these tasks is called the *observable behavior*. Added tasks (i.e., tasks present in x but not in y) can be executed but are not observable by the outside world, i.e., projection inheritance conforms to hiding or abstracting from tasks new in x .

The notion of protocol inheritance is based on *encapsulation* (i.e., blocking of new tasks): *If it is not possible to distinguish x and y when only tasks of x that are also present in y are executed, then x is a subclass of y .*

For distinguishing x and y under protocol inheritance all tasks present in x but not in y are blocked. The new tasks are simply disallowed to be executed.

A formal definition of the four forms of inheritance is beyond the scope of this paper. (The definition builds on branching bisimulation equivalence and an abstraction operator which renames a given set of tasks to τ .) The interested reader is referred to [4, 6, 13, 14].

Figure 8 shows five workflow processes modeled in terms of workflow nets. Special symbols are used to indicate the source place (i) and the sink place (o). Workflow process (A) consists of three sequential tasks: *register*, *handle*, and *archive*. Each of the other workflow processes extends this process with one additional task: *check*. In workflow process (B) task *check* can be executed arbitrarily many times between *register* and *handle*. Workflow process (B) is a subclass of workflow process (A) with respect to projection inheritance: If task *check* is abstracted from, then the two processes behave equivalently (i.e., are branching bisimilar). Workflow process (B) is also a subclass of workflow process (A) with respect to protocol inheritance: Blocking task *check* yields two equivalent processes. Workflow process (C) is not a subclass with respect to projection inheritance: Hiding task *check* introduces the possibility to skip task *handle* and thus change the actual behavior. However, (C) is a subclass of (A) with respect to protocol inheritance. Workflow process (D) is a subclass of workflow process (A) with respect to projection inheritance: Hiding this task results in two equivalent processes. However, (D) is not a subclass of (A) with respect to protocol inheritance: Blocking task *check* results in a deadlock. Workflow process (E) is a subclass of workflow process (A) with respect to projection inheritance: The detour via task *check* can be hidden thus yielding an observable behavior identical to (A). Workflow process (E) is not a subclass of workflow process (A) with respect to protocol inheritance. All workflow processes are a subclass of (A) with respect to life-cycle inheritance. For life-cycle inheritance some of the new tasks are blocked and others are hidden to obtain two equivalent processes. Only workflow process (B) is a subclass with respect to protocol/projection inheritance.

In [4, 6, 13, 14] we proposed a number *inheritance-preserving transformation rules*. These rules correspond to frequently used design constructs and preserve one or more of the four inheritance notions. A detailed description of these rules is beyond the scope of this paper. Therefore, we give an informal description of four inheritance rules: PP, PT, PJ, and PJ3. Protocol/projection inheritance-preserving transformation rule PP is illustrated by Figure 9. New transitions

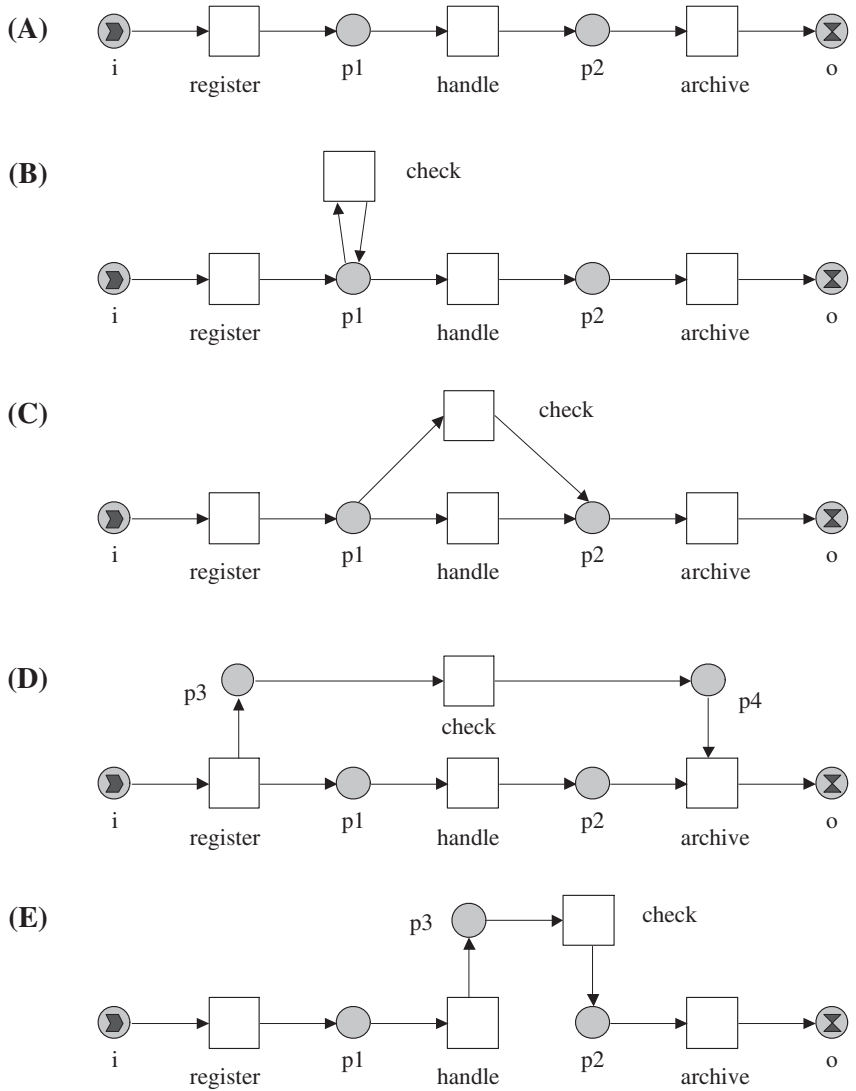


Fig. 8. Five routing diagrams describing variants of a simple workflow process

(i.e., tasks) and places (i.e., conditions) are added to the original workflow net such that tokens are only temporarily removed from a place in the original net. The added subnet may have any structure as long as it is guaranteed that any token taken from place p will be returned eventually and no tokens are left in the subnet. Since the subnet only postpones behavior, the extended workflow net (right) is a subclass of the original workflow net (left) under both projection

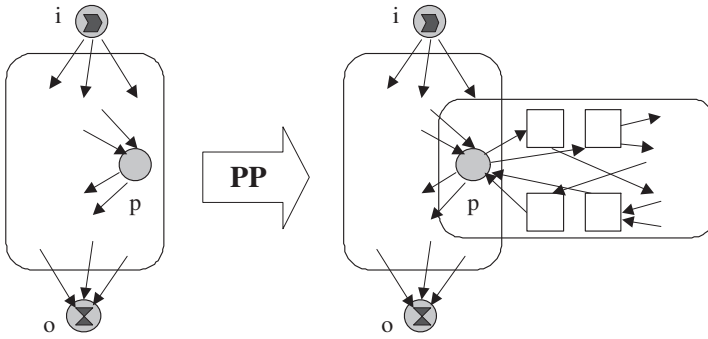


Fig. 9. Inheritance-preserving transformation rule PP

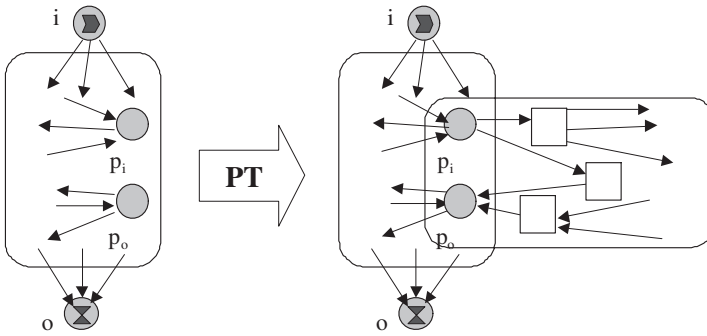


Fig. 10. Inheritance-preserving transformation rule PT

and protocol inheritance. If one abstracts from the newly added tasks or blocks the newly added tasks, one cannot find any differences between both nets.

Figure 10 illustrates transformation rule PT. PT preserves protocol inheritance and adds alternative behavior. The added subnet removes tokens from the original net to execute tasks not present in the original net. The added subnet may have any structure as long as any token taken from place p_i (input place) will be returned in place p_o (output place) eventually and no tokens are left in the subnet. Other requirements are that new tasks consuming tokens from p_i should not appear in the original net and that the routes via the subnet do not create new states in the original net. It is easy to see that PT preserves protocol inheritance: The added subnet is never activated if all new tasks are blocked. Note that PT does not preserve projection inheritance since the subnet can be used to bypass parts of the original net.

Figure 11 shows inheritance-preserving transformation rule PJ. Rule PJ inserts new tasks in-between a task t_p and a place p in the original workflow net.

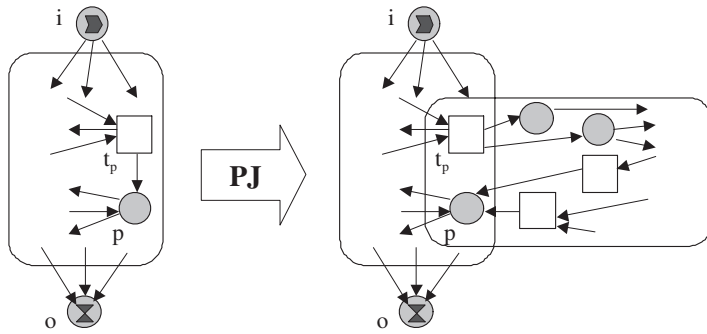


Fig. 11. Inheritance-preserving transformation rule PJ

In fact, rule PJ can be used to insert an arbitrary subflow in-between t_p and p . The added subnet may have any structure as long as it is guaranteed that once the subnet is activated by firing t_p eventually a token is put in place p and no tokens are left behind. It is easy to see that the extended workflow net (right) is a subclass of the original workflow net (left) under projection inheritance: by abstracting from the newly added tasks the observable behaviors coincide.

Projection inheritance-preserving transformation rule PJ3 can be used to add parallel behavior. (The rule is named PJ3 for historical reasons.) Figure 12 illustrates this rule. The execution of task t_i activates the subnet containing new tasks to be executed in parallel. Task t_o synchronizes the original workflow net and the added subnet. The added subnet may use arbitrary routing constructs as long as (1) the execution of t_i is always followed by the execution of t_o in the original net and t_o is always preceded by t_i , (2) activation of the subnet via firing t_i is always followed by a state which marks the input places of t_o in the subnet, and (3) no tokens are left behind in the subnet after firing t_o . If these three requirements are guaranteed, then the extended workflow net (right) is a subclass of the original workflow net (left) under projection inheritance.

For a formal definition of these rules we refer to [4, 6, 13, 14]. Details and subtle requirements are omitted to simplify the presentation of the main ideas. The workflow nets shown in Figure 8 illustrate the four rules. Rule PP introduces new tasks which only postpone behavior. Workflow process (B) shown in Figure 8 can be constructed from (A) by applying this rule; task *check* only postpones the execution of *handle*. Rule PT introduces alternative behavior. Workflow process (C) shown in Figure 8 can be constructed from (A) by applying PT. Rule PJ inserts new tasks in-between existing tasks. Workflow process (A) shown in Figure 8 can be extended to workflow process (E) using this rule. The extension can be a single task but also a complex subflow containing many tasks and all kinds of causality relations. Rule PJ3 adds parallel behavior. Workflow process (A) shown in Figure 8 can be extended to workflow process (D) using this rule. The four rules (PP, PT, PJ, and PJ3) correspond to design constructs that are often

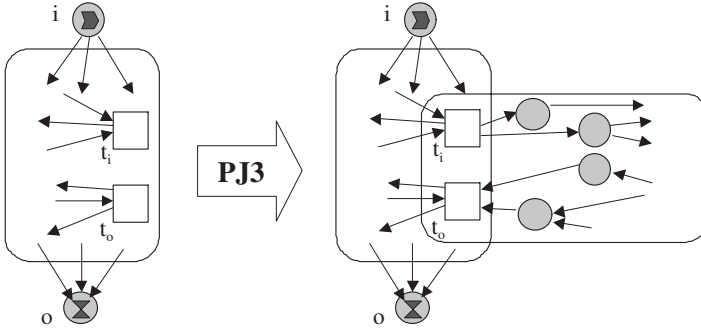


Fig. 12. Inheritance-preserving transformation rule PJ3

used in practice, namely iteration, choice, sequential composition, and parallel composition. If the designer sticks to these rules, inheritance is guaranteed!

4 Solution?

In the remainder we sketch in what way inheritance can assist in tackling the problems identified Section 2. We will not provide detailed solutions: For some of the problems we refer to other publications and for others we are exploring alternative solutions.

4.1 Problem 1: Dynamic Change

The inheritance-preserving rules can be used to *avoid* the first problem indicated in this paper. Moreover, the four rules enable the designer to establish syntactic and semantic correctness in a compositional manner. For example, soundness can be verified by analyzing the original part and the extension separately. To tackle the dynamic change problem we use the *transfer rules* presented in [6]. Suppose that x is a subclass of y constructed using the rules PP, PT, PJ, and PJ3. For any state in workflow process y it is possible to transfer a case to x such that the transfer is instantaneous (i.e., no postponements needed) and does not introduce syntactic errors (e.g., deadlocks, livelocks, and improper termination) nor semantic errors (e.g., the double execution of tasks or unnecessary skipping of tasks). Moreover, it is also possible to transfer cases from subclass x to superclass y without any problems. Note that the transfer rules are derived from the transformation rules introduced earlier. The transfer rules to move a case to a subclass are: r_{PT} , r_{PP} , r_{PJ} , $r_{PJ3,C}$ and $r_{PJ3,P}$ (see [6]). Transfer rules r_{PT} , r_{PP} , and r_{PJ} are rather trivial because additional behavior (i.e., alternative branches or parts inserted in-between existing parts) is introduced without eliminating existing states. The transfer rule corresponding to transformation rule PJ3 is more complex because PJ3 adds parallel behavior rather than

additional behavior. When adding parallel behavior, it may be necessary to mark places in the newly added parts. If this is the case, there is a choice to put the tokens in the beginning of the parallel part (conservative approach $r_{PJ3,C}$) or to put the tokens at the end of the parallel part (progressive approach $r_{PJ3,P}$). This choice depends of the desired policy. The transfer rules to move a case to a superclass are: $r_{PT,C}^{-1}$, $r_{PT,P}^{-1}$, r_{PP}^{-1} , r_{PJ}^{-1} , and r_{PJ3}^{-1} . The transfer rule corresponding to transformation rule PJ3 is simple: Simply remove the parallel parts. Transfer rules r_{PP}^{-1} and r_{PJ}^{-1} move tokens from the extended part to the superclass part. For the transfer of cases to the superclass under transformation rule PT there are again two choices: a conservative approach ($r_{PT,C}^{-1}$) and a progressive approach ($r_{PT,P}^{-1}$). Again the choice depends of the desired transfer policy. Note that as long as the designer sticks to the inheritance preserving transformation rules, the transfer rules can be generated automatically. This means that there is no need to design complicated migration schemes.

To illustrate the transfer rules, we use the five workflow processes shown in Figure 8. Suppose a case is in variant (A) in the state corresponding to $p1$. If the case is migrated to variant (B), (C), or (E), then there is no need to modify the state (i.e., the state in the new process is just a token in $p1$). If the case is migrated to variant (D), the new state is either $p1 + p3$ (conservative mapping, $r_{PJ3,C}$) or $p1 + p4$ (progressive mapping, $r_{PJ3,P}$). A case in variant (E) in the state corresponding to $p3$, is mapped onto state $p2$ if it is transferred to (A). A case in variant (D) in the state corresponding to $p2 + p4$, is mapped onto state $p2$ if it is transferred to (A). These examples show that, as long as the designer sticks to the inheritance-preserving transformation rules, it is indeed possible to migrate cases from a superclass to a subclass and vice versa, i.e., the inheritance-preserving transformation rules can be used to avoid the dynamic change bug illustrated by Figure 2.

4.2 Problem 2: Management Information

The transfer rules r_{PT} , r_{PP} , r_{PJ} , $r_{PJ3,C}$, $r_{PJ3,P}$, $r_{PT,C}^{-1}$, $r_{PT,P}^{-1}$, r_{PP}^{-1} , r_{PJ}^{-1} , and r_{PJ3}^{-1} can also be used to construct management information. Construct or select an appropriate workflow process such that each of the variants is a subclass or superclass of this processes via one or more inheritance-preserving transformation rules (applied in either direction). This workflow process is called the *management information net*. Any case residing in any of the variants can be mapped onto the management information net using the transfer rules. Therefore, the aggregate management information can be obtained automatically. Consider for example the following situation: There is one process template and every ad-hoc variant is a subclass of this variant constructed using the rules PP, PT, PJ, and PJ3. The template can also be used as a management information net and all cases can be projected onto this net using $r_{PT,C}^{-1}$, $r_{PT,P}^{-1}$, r_{PP}^{-1} , r_{PJ}^{-1} , and r_{PJ3}^{-1} . See [6] for technical details and more scenarios showing that the inheritance concepts can really help to provide aggregate management information.

4.3 Problem 3: Inter-organizational Interface Agreements

In general, modeling workflow processes that span multiple organizations can be complex. Many workflow-modeling tools require that the complete details of the entire process be fixed at the design stage. This approach is often the bottleneck in workflow design, because it requires each participating business partner to understand the nature of their partners' local processes which makes things even more complicated. This is neither necessary nor desirable. Partners should be able to agree on their business process at a level that abstracts from irrelevant and confusing details. Furthermore, organizations should be able to participate towards the completion of a business process, while at the same time be at liberty to construct their private processes in any way that places them at best advantage.

The approach we propose here for designing interorganizational workflows, is a four-step process that involves creation of a public process, partitioning the public process amongst the partners and allowing for modification by the individual partners of their parts of the process to create private processes. This approach is based on the inheritance-preserving transformation rules and is described in more detail in [3, 10]. The steps are:

1. *Design public workflow process.*

In the first step, the partners agree on the overall structure of the common business process. The key tasks are identified as well as the interfaces between them. The interfaces are specified as control and data dependencies. Process repositories may also be used at this stage to aid the design process. The workflow process model resulting from this stage will not contain organization specific information like role assignment, resources assignment, etc. We refer to the workflow process represented by this model as the *public process*.

2. *Partition the public workflow process definition amongst business partners.*

In this step, the partners are assigned to be responsible for completing parts of the process. The workflow net is then partitioned along organizational lines. The partitioning creates a set of workflow net fragments for each business partner. Each set contains fragments of the public workflow net that the partner is responsible for.

3. *Create a private workflow for each business partner.*

The partitioning stage creates sets that are not necessarily workflow nets, but rather fragments of a workflow net. A fragment may have multiple input or output places, or no input or output places at all. Moreover, a fragment may contain disjunct subsets (i.e. is unconnected). In other words, the fragment may not be a workflow net (i.e., the requirements mentioned earlier are violated). This step automatically creates a workflow net from each fragment set for each of the partners to manage locally in their organization. Note that in this step we abstract from places solely dedicated to message exchange between fragments, i.e., we remove all source places except the initial one and remove all sink places except the last one in the fragment. If after this abstraction we do not obtain a workflow net, additional modifications by

connecting the fragments by introducing arcs and implicit places are needed. The result is a sound workflow net whose behavior preserves the interface of the original public process. We will refer to each of the local workflows as *private* processes

4. *Modify private processes using inheritance preserving transformation rules.*
Here, each business partner will modify their private process created in the previous step to accurately reflect their own business process and incorporate details of how the tasks will be implemented at their organization. Some of the details that will be added at this stage are task implementation, assignment of roles to the tasks, data objects to be used and any mapping of global data objects to these local data objects, business rules, deadline specification etc. These extensions or modifications to the minimal workflow must also preserve the behavior of the global business process. To guarantee that local extensions and modifications do not disturb the behavior of the public process, we use a notion of inheritance of dynamic behavior. For this purpose, we bring *projection inheritance* into play. Private workflow processes are modified while preserving projection inheritance, i.e., one is allowed to transform the local workflow into an arbitrary subclass under projection inheritance. For this purpose three of the four transformation rules can be used: PP, PJ, and PJ3.

The result of this approach is a set of private workflows which together form the actual total workflow. This total workflow is a sound workflow net, i.e., anomalies such as deadlocks and dangling references cannot occur. Moreover, the total workflow is a subclass of the private workflow under projection inheritance. This means that the workflow actually executed is consistent with the abstract workflow the business partners agreed on in the first place. These strong results show that the notion of inheritance can be used to tackle Problem 3. See [3, 10] for more details.

4.4 Problem 4: Customizing Business Processes

The fourth problem to be tackled by the inheritance notions defined earlier is the problem of delta analysis, i.e., given two workflow processes: What is the difference between those processes and how much does it cost to customize a process such that it coincides with the other? One of the core problems is to decide where both processes agree on, i.e., to determine the GCD (Greatest Common Divisor). In Figure 7, we showed four possible candidates for the GCD of the two workflow processes shown in Figure 6. Based on the inheritance relations we can experiment with various definitions of the GCD. The most straightforward definition of the GCD of two workflow processes x and y is the “smallest” superclass of both x and y under life-cycle inheritance, i.e., the x is a subclass of the GCD, y is a subclass of the GCD, and there is no workflow process z such that z is a real subclass of the GCD and a superclass of both x and y . Based on this definition it can be shown that for any set of workflow processes there exists a GCD. However, in some cases there may be multiple GCD’s. Consider

for example the two workflow processes shown in Figure 6. Workflow process (B) is a GCD of these two processes: Process (B) is a superclass of both processes shown in Figure 6 and there is no “smaller” workflow process satisfying the same conditions. However, for similar reasons, process (C) is also a GCD. If we closely observe the two processes shown in Figure 6, it may seem reasonable to have two GCD’s. Both processes agree on the fact that tasks B and C are executed in-between tasks A and D . However, they do not agree on the ordering of B and C . Therefore, either B or C is added to the GCD.

The concept of GCD was introduced in [6] and a detailed analysis of this concept is given in [5]. Since none of the inheritance relations is a lattice, there is a trade-off between “uniqueness” and “existence”. By using a weak notion of GCD existence is guaranteed but there may be multiple GCD’s. By using a stronger notion, existence is no longer guaranteed but if the GCD exists, it is unique. Given this tradeoff, we define the notion of Maximal Common Divisor (MCD). An MCD is a “smallest” superclass of both x and y under life-cycle inheritance. Both (B) and (C) (Figure 7) are MCD’s of the two workflow processes shown in Figure 6. Given any set there is at least one MCD. In [5] we reserve the term GCD for a stronger notion: z is the GCD of x and y if and only if x is a subclass of z , y is a subclass of z , and for any v such that x is a subclass of v and y is a subclass of v : z is a subclass of v . If there is a GCD using this stronger notion, it is unique.

In [5, 6] conditions are given such that the MCD of a set of workflow processes is a GCD, and therefore, unique. Further research is needed to evaluate the usefulness of this particular notion of GCD for delta analysis. Given the fact that inheritance defines a relation on processes, it is useful for comparing workflows. Note that each of the four inheritance relations defines a partial order, i.e., each relation is reflexive, anti-symmetric, and transitive [14]. Therefore, inheritance is a sound basis for defining a notion of GCD. If a set of workflow processes is related under inheritance via subclass-superclass relationships, it is generally quite easy to find the GCD. If this is not the case (such as in Figure 6), the computation of a GCD is more involved and there are typically multiple candidates (i.e., MCD’s). As pointed out when defining the problem of delta analysis, computing the GCD is just one of the ingredients needed to calculate the costs of customization. For example, the naming of tasks is important (i.e., a good ontology is a necessity) and it may be difficult to establish a suitable cost structure (What is the cost of adding/deleting a causal relation compared to adding/deleting/changing a task?). See [27] for a more elaborate discussion on these other aspects.

5 Conclusion and Tool Support

In this paper we visited four notorious problems related to the design and analysis of workflow processes:

- The problem of *dynamic change*: How to migrate workflow instances from the old to the new process?

- The problem of *management information*: How to provide aggregate information in the context of multiple versions/variants of a given process?
- The problem of *inter-organizational interface agreements*: How to align workflow processes of different organizations while keeping local autonomy?
- The problem of *customizing*: How to measure the differences and commonalities of different processes (i.e., delta analysis)?

The goal of this journey has been the integration of previous work in this area and to demonstrate that a diverse set of problems can be addressed using the principle of inheritance.

Our tool Woflan [54] supports the four notions of inheritance used in this paper. Given two workflow processes, Woflan can decide whether one process is a subclass of another process. These processes may be created using the BPR tool Protos [46] or workflow management systems such as Staffware [53] and COSA [52]. Clearly, this does not provide a complete solution for the four problems presented in this paper. However, the fact that Woflan supports our inheritance notions demonstrates that tool support is possible. In the future, we hope to integrate our ideas into commercial run-time systems (i.e., workflow management systems, enterprise resource planning systems, and e-business systems).

References

- [1] W. M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998. 383, 384, 385
- [2] W. M. P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999. 394
- [3] W. M. P. van der Aalst and K. Anyanwu. Inheritance of Interorganizational Workflows to Enable Business-to-Business E-commerce. In A. Dognac, E. van Heck, T. Saarinen, and et. al., editors, *Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC'99)*, pages 141–157, Nashville, Tennessee, October 1999. 384, 402, 403
- [4] W. M. P. van der Aalst and T. Basten. Life-cycle Inheritance: A Petri-net-based Approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, Berlin, 1997. 384, 395, 396, 399
- [5] W. M. P. van der Aalst and T. Basten. Identifying Commonalities and Differences in Object Life Cycles using Behavioral Inheritance. In J. M. Colom and M. Koutny, editors, *Application and Theory of Petri Nets 2001*, volume 2075 of *Lecture Notes in Computer Science*, pages 32–52. Springer-Verlag, Berlin, 2001. 384, 388, 393, 404
- [6] W. M. P. van der Aalst and T. Basten. Inheritance of Workflows: An Approach to Tackling Problems Related to Change. *Theoretical Computer Science*, 270(1-2):125–203, 2002. 384, 388, 396, 399, 400, 401, 404
- [7] W. M. P. van der Aalst, T. Basten, H. M. W. Verbeek, P. A. C. Verkoulen, and M. Voorhoeve. Adaptive Workflow: On the Interplay between Flexibility and Support. In J. Filipe, editor, *Enterprise Information Systems*, pages 63–70. Kluwer Academic Publishers, Norwell, 2000. 384, 385, 388

- [8] W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors. *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2000. 385
- [9] W. M. P. van der Aalst, G. De Michelis, and C. A. Ellis, editors. *Proceedings of Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM'98)*, Lisbon, Portugal, June 1998. UNINOVA, Lisbon. 385
- [10] W. M. P. van der Aalst and M. Weske. The P2P approach to Interorganizational Workflows. In K. R. Dittrich, A. Geppert, and M. C. Norrie, editors, *Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01)*, volume 2068 of *Lecture Notes in Computer Science*, pages 140–156. Springer-Verlag, Berlin, 2001. 384, 390, 402, 403
- [11] N. R. Adam, V. Atluri, and W. Huang. Modeling and Analysis of Workflows using Petri Nets. *Journal of Intelligent Information Systems*, 10(2):131–158, 1998. 385
- [12] A. Agostini and G. De Michelis. Improving Flexibility of Workflow Management Systems. In W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 218–234. Springer-Verlag, Berlin, 2000. 385, 387
- [13] T. Basten. *In Terms of Nets: System Design with Petri Nets and Process Algebra*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, December 1998. 384, 396, 399
- [14] T. Basten and W. M. P. van der Aalst. Inheritance of Behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001. 384, 395, 396, 399, 404
- [15] R. W. H. Bons, R. M. Lee, and R. W. Wagenaar. Designing trustworthy interorganizational trade procedures for open electronic commerce. *International Journal of Electronic Commerce*, 2(3):61–83, 1998. 390
- [16] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Workflow Evolution. *Data and Knowledge Engineering*, 24(3):211–238, 1998. 385
- [17] J. Dehnert. Four Steps Towards Sound Process Models. (See this volume). 385
- [18] C. Ellis and K. Keddara. ML-DEWS: Modeling Language to Support Dynamic Evolution within Workflow Systems. *Computer Supported Cooperative Work*, 9(3/4):293–333, 2000. 387
- [19] C. A. Ellis. Information Control Nets: A Mathematical Model of Office Information Flow. In *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, pages 225–240, Boulder, Colorado, 1979. ACM Press. 385
- [20] C. A. Ellis and K. Keddara. A Workflow Change Is a Workflow. In W. M. P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, Berlin, 2000. 387
- [21] C. A. Ellis, K. Keddara, and G. Rozenberg. Dynamic change within workflow systems. In N. Comstock, C. Ellis, R. Kling, J. Mylopoulos, and S. Kaplan, editors, *Proceedings of the Conference on Organizational Computing Systems*, pages 10 – 21, Milpitas, California, August 1995. ACM SIGOIS, ACM Press, New York. 385, 386
- [22] C. A. Ellis and G. J. Nutt. Modelling and Enactment of Workflow Systems. In M. Ajmone Marsan, editor, *Application and Theory of Petri Nets 1993*, volume 691 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, Berlin, 1993. 385
- [23] R. Eshuis and R. Wieringa. Comparing Petri Net and Activity Diagram Variants for Workflow Modeling: A Quest for Reactive Petri Nets. (See this volume). 385

- [24] L. Fischer, editor. *Workflow Handbook 2001, Workflow Management Coalition. Future Strategies*, Lighthouse Point, Florida, 2001. 383, 385
- [25] P. Grefen, K. Aberer, Y. Hoffner, and H. Ludwig. CrossFlow: Cross-organizational Workflow Management in Dynamic Virtual Enterprises. *International Journal of Computer Systems, Science, and Engineering*, 15(5):277–290, 2001. 390
- [26] P. Grefen and S. Angelov. Three-Level Process Specification for Dynamic Service Outsourcing: From Petri Nets to ebXML and WFPDL. (See this volume). 390
- [27] V. Guth and A. Oberweis. Delta-Analysis of Petri Net based Models for Business Processes. In E. Kovacs, Z. Kovacs, B. Cserto, and L. Pepei, editors, *Proceedings of the 3rd International Conference on Applied Informatics*, pages 23–32, 1997. 393, 404
- [28] Y. Han and A. Sheth. On Adaptive Workflow Modeling. In *Proceedings of the 4th International Conference on Information Systems Analysis and Synthesis*, pages 108–116, Orlando, Florida, July 1998. 385
- [29] R. Heckel. Open Petri Nets as Semantic Model for Workflow Integration. (See this volume). 385
- [30] InConcert. *InConcert Process Designer's Guide*. InConcert Inc., Cambridge, Massachusetts, 1997. 384
- [31] S. Jablonski and C. Bussler. *Workflow Management: Modeling Concepts, Architecture, and Implementation*. International Thomson Computer Press, London, UK, 1996. 383
- [32] R. Kalakota and A.B. Whinston. *Frontiers of Electronic Commerce*. Addison-Wesley, Reading, Massachusetts, 1996. 388
- [33] K. Keddara. *Dynamic Evolution of Workflow Systems*. PhD thesis, University of Colorado, Boulder, Colorado, USA, 1999. 385
- [34] G. Keller and T. Teufel. *SAP R/3 Process Oriented Implementation*. Addison-Wesley, Reading MA, 1998. 394
- [35] E. Kindler, A. Martens, and W. Reisig. Inter-Operability of Workflow Applications: Local Criteria for Global Soundness. In W.M.P. van der Aalst, J. Desel, and A. Oberweis, editors, *Business Process Management: Models, Techniques, and Empirical Studies*, volume 1806 of *Lecture Notes in Computer Science*, pages 235–253. Springer-Verlag, Berlin, 2000. 390
- [36] M. Klein, C. Dellarocas, and A. Bernstein, editors. *Proceedings of the CSCW-98 Workshop Towards Adaptive Workflow Systems*, Seattle, Washington, November 1998. 385
- [37] M. Klein, C. Dellarocas, and A. Bernstein, editors. *Adaptive Workflow Systems*, Special Issue of Computer Supported Cooperative Work, 2000. 385
- [38] C. Lakos. Composing Abstractions of Coloured Petri Nets. In M. Nielsen and D. Simpson, editors, *Application and Theory of Petri Nets 2000*, volume 1825 of *Lecture Notes in Computer Science*, pages 323–345. Springer-Verlag, Berlin, 2000. 384
- [39] A. Lazcano, G. Alonso, H. Schuldt, and C. Schuler. The WISE Approach to Electronic Commerce. *International Journal of Computer Systems, Science, and Engineering*, 15(5):345–357, 2001. 390
- [40] R.M. Lee. Distributed Electronic Trade Scenarios: Representation, Design, Prototyping. *International Journal of Electronic Commerce*, 3(2):105–120, 1999. 390
- [41] R.M. Lee and R.W.H. Bons. Soft-Coded Trade Procedures for Open-edi. *International Journal of Electronic Commerce*, 1(1):27–49, 1996. 390
- [42] K. Lenz and A. Oberweis. Inter-Organizational Business Process Management With XML Nets. (See this volume). 385, 390

- [43] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice-Hall PTR, Upper Saddle River, New Jersey, USA, 1999. 383
- [44] M. Merz, B. Liberman, and W. Lamersdorf. Using Mobile Agents to Support Interorganizational Workflow-Management. *International Journal on Applied Artificial Intelligence*, 11(6):551–572, 1997. 390
- [45] M. Merz, B. Liberman, and W. Lamersdorf. Crossing Organisational Boundaries with Mobile Agents in Electronic Service Markets. *Integrated Computer-Aided Engineering*, 6(2):91–104, 1999. 390
- [46] Pallas Athena. *Protos User Manual*. Pallas Athena BV, Plasmolen, The Netherlands, 1999. 405
- [47] M. Reichert and P. Dadam. ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998. 385
- [48] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998. 385
- [49] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets II: Applications*, volume 1492 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998. 385
- [50] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, Reading, MA, USA, 1998. 383
- [51] A. W. Scheer. *Business Process Engineering, Reference Models for Industrial Enterprises*. Springer-Verlag, Berlin, 1994. 394
- [52] Software-Ley. *COSA User Manual*. Software-Ley GmbH, Pullheim, Germany, 1998. 405
- [53] Staffware. *Staffware 2000 / GWD User Manual*. Staffware plc, Berkshire, United Kingdom, 1999. 405
- [54] H. M. W. Verbeek, T. Basten, and W. M. P. van der Aalst. Diagnosing Workflow Processes using Woflan. *The Computer Journal*, 44(4):246–279, 2001. 405
- [55] M. Voorhoeve and W. M. P. van der Aalst. Conservative Adaption of Workflow. In M. Wolf and U. Reimer, editors, *Proceedings of the International Conference on Practical Aspects of Knowledge Management (PAKM'96), Workshop on Adaptive Workflow*, pages 1–15, Basel, Switzerland, October 1996. 388
- [56] M. Voorhoeve and W. M. P. van der Aalst. Ad-hoc Workflow: Problems and Solutions. In R. Wagner, editor, *Proceedings of the 8th DEXA International Workshop on Database and Expert Systems Applications*, pages 36–40, Toulouse, France, September 1997. IEEE Computer Society Press, Los Alamitos, California, 1997. 385, 388
- [57] H. Wehrheim. Behavioural Subtyping and Property Preservation. In S. Smith and C. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2000)*, pages 213–232. Kluwer, 2000. 384
- [58] H. Wehrheim. Subtyping Patterns for Active Objects. In H. Giese and S. Philippi, editors, *Proceedings 8ter Workshop des GI Arbeitskreises GROOM (Grundlagen objekt-orientierter Modellierung)*, volume 24/00, Münster, Germany, 2000. University of Münster. 384
- [59] V. Zwass. Electronic commerce: structures and issues . *International Journal of Electronic Commerce*, 1(1):3–23, 1996. 388

A Formal Framework for Petri Net Class Transformations

Francesco Parisi-Presicce

Dipartimento di Informatica, Università di Roma "La Sapienza"
Via Salaria 113, 00198 Rome, Italy
`parisi@dsi.uniroma1.it`

Abstract. A formalization of transformations of Petri Nets is presented in a categorical framework. It uses rule based modifications for the transformation of models within one Petri net class (replacement of one subnet by another one) and functors for the enrichment of the models to a different class (addition of features previously ignored). Net class transformations are defined between elementary nets, place/transition nets and algebraic high level nets. These class transformations preserve the rules and the derivation sequences. This compatibility guarantees that the order of application of a net class transformation and a model transformation does not affect the result. The net class transformations are then shown to "lift" providing a form of stepwise enrichment of net classes.

1 Introduction

Almost 40 years of Petri nets have led to an enormously rich theory and a large number of successful practical applications, see e.g. [9, 24, 23]. The application domains range from production systems, traffic control, to workflow management etc. Several variants of Petri nets with different complexity have been developed in order to express specific features in these domains, and a variety of Petri net-based process models have been proposed in the literature. These provide a methodology for the use of Petri nets within the software development process, mainly in the analysis and design phase. In general, a simple Petri net is initially proposed and then it is incrementally enhanced by, e.g. , time, roles, exception handling, initial state etc. In different development steps, different Petri net models are produced, each enhancing the initial model by additional features and details.

This paper presents a summary of a formal framework for the transformations of Petri net models and classes, based on the the theory of high-level replacement systems (HLR systems), originally introduced in [13] as a generalization of graph grammars. A distinction is made between *net model transformation* and *net class transformation*, introduced on a conceptual level in [18]. Net model transformations, i.e., changes of the model within one Petri net class, can be realized by a rule that replaces a subnet by another one. Net class transformations, instead, are used to change the underlying Petri net formalism. Certain aspects of systems like data, initial state, etc. are to be added in the modeling process,

which requires the existence of these components in the Petri net formalism. Net class transformations are used as a syntactical enrichment of the formalism, and once the underlying net class is transformed, the newly added features can be further specified.

The framework supports an incremental approach to system development, where enhancements can be performed concurrently on different abstraction layers. Therefore, it is essential to translate net model transformations to different abstraction layers in order to reflect the modifications. This satisfies the fundamental requirement of consistency among models of different abstraction layers. In this paper we discuss consistency, which means that each model is a proper enhancement of the starting model. With respect to the two different levels of transformation, this consistency can be considered as a form of confluence: compatibility of net class transformations and net model transformations demands that net model transformations be propagated across Petri net classes. Moreover, the order of application of the different transformations should not affect the result. This requirement has some impact on the formalization of net class transformations. We formalize net class transformations as categorical functors, similar to the categorical formalization of net model transformations in the context of high-level replacement systems. In our main Theorem 1 (Preservation of Derivations) we present an important compatibility result for these different transformations, which ensures consistency among models in different development steps. Moreover, we define and investigate net class transformations between elementary nets [36], place/transition nets, marked nets [11] and a new variant of algebraic high-level nets, similar to [38, 37, 32]. The new definition [21] comprises explicit variables and a new definition of morphism that allows substituting variables with terms and thus a wider applicability of rules. This paper deals only with the problem of consistency among models at different levels of abstraction. For a broader perspective on Petri Net transformations and their role for the Expert View and the Application Developer View, see [2].

The paper is structured as follows: In Section 2 an example motivates our approach of relating different Petri nets via net class and net model transformations. In Section 3 different low-level nets and our new notion of algebraic high-level nets are reviewed. Net class transformations between these Petri net classes are shown in Section 4. Finally, the lifting of these net class transformations is presented in Section 5 and some concluding remarks are made in Section 6. All proofs are omitted and can be found in [20] and in [21].

2 Motivation and Example

For a structured use of Petri nets in the development of software systems, several Petri net based process models have been proposed in literature. These process models start with an abstract model, which is then enhanced by additional features and details of the system in subsequent development steps. Each of these Petri net based process models follows one of two opposing paradigms. Either one complex Petri net class is fixed throughout the development process (usually,

certain aspects like data or time constraints are left undefined in early phases and specified later on, as in [28, 25, 5, 6]), or different models based on several Petri net classes are developed, each of the classes adequate in expressiveness for one development step [7, 26, 27]. While the first paradigm avoids changes of the Petri net class, the second paradigm is preferable with respect to readability, understandability, and analysis techniques.

Enhancement by details means a modification of the model in order to incorporate details like exception handling, business rules etc. This kind of enhancement, that we call *net model transformation*, is formalized with rule-based modifications, based on the theory of high-level replacement systems, as the formal basis for model transformations. A high-level replacement system is given by a category and a distinguished class $\mathcal{M}$ of morphisms used to form rules. The rules are given by $r = (L \xleftarrow{u} K \xrightarrow{v} R)$, where u and v are morphisms in the class $\mathcal{M}$, and L represents the part to be deleted, R the part to be added and the interface K the part to be preserved. The application of a rule is free of side-effects and a rule may be applied more than once to a given model performing the same modification to different subnets. The rule-based approach has been successfully applied in [16] for the development of a large case study of a medical information system and for numerous smaller examples [33, 34, 35].

Enhancement by a feature depends on the underlying paradigm of the Petri net based process model. When only one Petri net class is employed throughout the development process, the addition of a feature is performed by assigning a description which had yet not been specified. This corresponds to a modification of the net and can be captured by net model transformations as described above. In [7, 26, 27] the underlying paradigm is to use the most adequate Petri net class for each development step. Adequate here means that all aspects important in that phase can be modeled but not more. The complexity of the underlying Petri net classes increases as the development proceeds. This approach leads to more understandable models in earlier phases when analysis can be more easily performed. In order to change the underlying net class between the development steps, a *net class transformation* has to be performed. Since in many Petri net based process models enhancement by features (net class transformation) and enhancement by details (model transformation) are intertwined, the compatibility of the different transformations is crucial. The consistency of models on different abstraction layers is a major requirement: each model should be a proper enhancement of the initial model. In this sense, the formalization of net class transformations as functors between different categories of Petri nets enables an investigation of the compatibility between net class and model transformations. Since it is necessary to transform not only Petri Nets, but also the morphisms in the rules, a functor must be defined.

We illustrate net class and net model transformations by means of a simple example (already presented in [21]) of the development of a producer/consumer system, starting from the abstract and simple model given in Figure 1. The elementary net $N_{0_{EN}}$ consists of two separate systems of a producer (on the

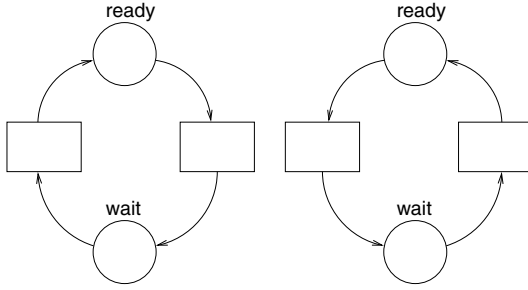


Fig. 1. Initial model $N_{0_{EN}}$ of producer and consumer

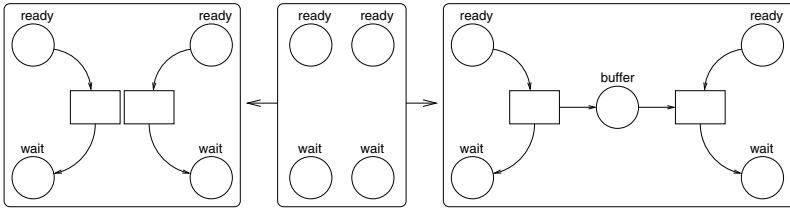


Fig. 2. Rule $buffer_{EN}$ for cooperation between producer and consumer

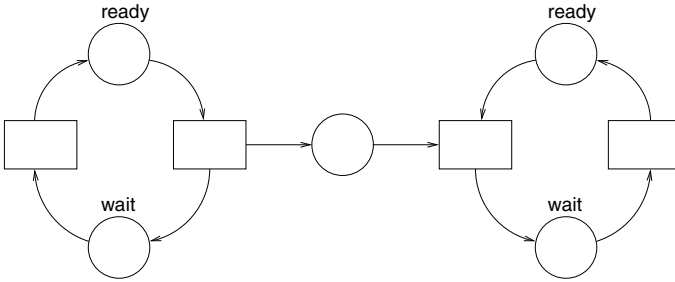


Fig. 3. Producer and consumer system $N_{1_{EN}}$

left-hand side), cycling between idling and producing, and a consumer (on the right-hand side) cycling between idling and consuming.

To synchronize the two isolated systems, the initial model is modified by applying the rule $buffer_{EN}$ in Figure 2 that adds a buffer where the producer puts a produced item from which the consumer takes it.

The result of the application of the rule $buffer_{EN}$ to the initial net is described in Figure 3. The model transformation is described as $N_{0_{EN}} \xrightarrow{buffer_{EN}} N_{1_{EN}}$.

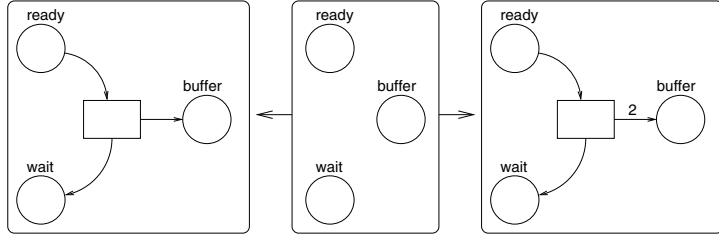


Fig. 4. Rule $quick_{PT}$ to change the arc weight

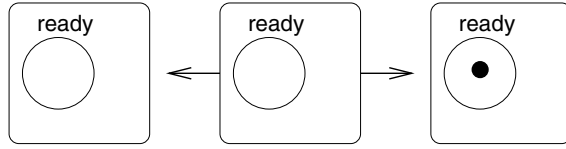


Fig. 5. Rule $initialize_{mEN}$ to add the marking

2.1 Place/Transition Nets

The change from elementary nets to place/transition nets uses a net class transformation, with which the initial model $N_{0_{EN}}$ is syntactically transformed into a place/transition net by the functor $Weight$ that weighs all arcs with 1. The resulting net $N_{0_{PT}}$ is like $N_{0_{EN}}$ in Figure 1, because the arc weight of 1 is omitted by convention. The functor also transforms the rule $buffer_{EN}$ into the rule $buffer_{PT}$ which operates on place/transition nets. Its application yields a place/transition $N_{1_{PT}}$ net corresponding to the net $N_{1_{EN}}$. Another modification by the rule $quick_{PT}$ changes the pace of the producer, that now produces two items instead of one in a single production step.

This new rule $quick_{PT}$ can be translated to **EN**, the category of elementary nets, where $quick_{EN}$ is the identity rule.

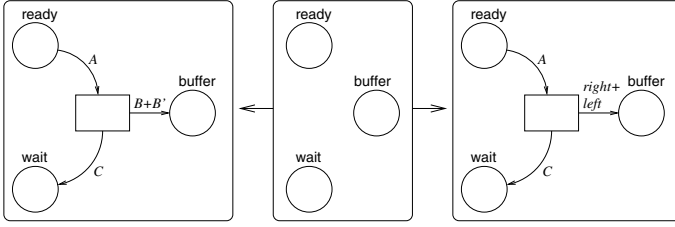
2.2 Marked Elementary Nets

To add the feature of the initial state of the system $N_{0_{EN}}$, the net class transformation $Mark$ is used. The translation of the initial elementary net of Figure 1 leads to the marked elementary net $N_{0_{mEN}}$, whose marking is empty. Again the rule $buffer_{EN}$ is translated to a corresponding rule for marked elementary nets $buffer_{mEN}$, resulting in $N_{1_{mEN}}$.

By applying the rule $initialize_{mEN}$ twice, we obtain the initial state where both consumer and producer are ready.

2.3 Algebraic High-Level Nets

The transformation from the place/transition net $N_{0_{PT}}$ into an algebraic high level net $N_{0_{AHL}}$ is performed by the functor $Data$ defined in Definition 15. The

Fig. 6. Rule $items_{AHL}$

empty algebraic specification and empty algebra are added, and for each arc with weight n , we create n unique variables, which are attached to the arc in the algebraic high level net. The firing conditions are left empty by this transformation. The rule $buffer_{PT}$ is translated into the rule $buffer_{AHL}$ in **AHL**, the category of algebraic high level nets. A subsequent model transformation via $buffer_{AHL}$ leads to the net N_{1AHL} . Finally, the algebraic specification and the arc inscriptions can be modified by specifying the items produced or consumed by some corresponding terms.

3 Rule-Based Transformations of Petri Nets

In this section we briefly review the different version of Petri Nets analyzed and the main notions of rule transformations and high-level replacement system.

3.1 Categories of Petri Nets

In this subsection we define the categories **PT**, **EN**, **marked PT**, and **AHL** of place/transition nets, elementary nets, marked place/transition nets, and algebraic high level nets, respectively, and mention their cocompleteness properties.

Definition 1 (Place/Transition Net). A place/transition net is given by $N = (P, T, pre, post)$ where P is the set of places, T is the set of transitions, $pre, post : T \rightarrow P^\oplus$ are the pre- and postdomain of transitions.

Given two place/transition nets $N_i = (P_i, T_i, pre_i, post_i)$, $i = 1, 2$, a morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ with functions $f_P : P_1 \rightarrow P_2$, and $f_T : T_1 \rightarrow T_2$ is called **transition preserving** iff $f_P^\oplus \circ pre_1 = pre_2 \circ f_T$ and $f_P^\oplus \circ post_1 = post_2 \circ f_T$. **strict** if it is transition preserving and f_P and f_T are injective.

Transition preserving morphisms and place/transition nets give rise to the category **PT**.

Definition 2 (Elementary Net). An elementary net is given by $N = (P, T, pre, post)$ with P the set of places, T the set of transitions, $pre, post : T \rightarrow \mathcal{P}(P)$ the pre- and post-domain of the transitions.

Given two elementary nets $N_i = (P_i, T_i, pre_i, post_i)$, $i = 1, 2$, a morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ with functions $f_P : P_1 \rightarrow P_2$, and $f_T : T_1 \rightarrow T_2$ is called

transition preserving iff $\mathcal{P}(f_P) \circ pre_1 = pre_2 \circ f_T$ and $\mathcal{P}(f_P) \circ post_1 = post_2 \circ f_T$
strict if it is transition preserving and f_P and f_T are injective.

Transition preserving morphisms and elementary nets give rise to the category **EN**.

Next, we briefly review marked place/transition nets, studied in [19, 34].

Definition 3 (Marked Place/Transition Net). A marked place/transition net is given by $N = (P, T, pre, post, \widehat{m})$ with P the set of places, T the set of transitions, $pre, post : T \rightarrow P^\oplus$ the pre- and postdomain of transitions, and $\widehat{m} \in P^\oplus$ the initial marking.

The definition of morphism is based on functions between the sets of places and transitions. These functions have to be compatible with pre- and postfunctions and with the initial marking. We define morphisms that employ placewise comparison of markings based on the notion of restriction.

Definition 4 (Restriction). Given $M \in P^\oplus$ and $P' \subseteq P$, then there is $M' \in P'^\oplus$ and $M'' \in (P \setminus P')^\oplus$ such that $M = M' \oplus M''$.

We define $M|_{P'} = M'$ (read “ M restricted to P' ”).

For $P' = \{p\}$ we denote $M|_p = M|_{P'}$. For a function $f_P : P_0 \rightarrow P$ we write $M|_{f_P} = M|_{f_P(P_0)}$

Definition 5 (Marked Place/Transition Net Morphisms). Given two marked place/transition nets $N_i = (P_i, T_i, pre_i, post_i, \widehat{m}_i), i \in \{1, 2\}$, a morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ with functions $f_P : P_1 \rightarrow P_2$, and $f_T : T_1 \rightarrow T_2$ is called

transition preserving if the following conditions hold:

- $f_P^\oplus \circ pre_1 = pre_2 \circ f_T$ and $f_P^\oplus \circ post_1 = post_2 \circ f_T$
- $f_P^\oplus(\widehat{m}_1|_p) \leq \widehat{m}_2|_{f_P(p)}$

marking strict if it is transition preserving and $f_P^\oplus(\widehat{m}_1|_p) = \widehat{m}_2|_{f_P(p)}$

strict if it is marking strict and f_P and f_T are injective.

Marked place/transition nets and transition preserving morphisms give rise to the category **marked PT**.

We now review a new notion of algebraic high level nets, introduced in [21]. Unlike the classical notion ([32]), the new definition comprises explicit variables that may be substituted with terms in our new notion of morphisms. Hence, these morphisms establish a more flexible relationship between algebraic high level nets and can be employed in the context of model transformations. The laxer notion of morphism allows an occurrence from an algebraic high level net inscribed with variables to a more specific one inscribed by terms and thus a relaxed applicability of rules. Therefore, a stepwise specification of the model can be performed by rules.

Definition 6 (Algebraic High Level Net). An algebraic high level net is given by

$$N = (X, A, SPEC, P, T, pre, post, cond)$$

where

- X is a set of variables,
- $A \in |\mathbf{Alg}(\mathbf{SPEC})|$ is a SPEC algebra,
- $SPEC \in |\mathbf{SPEC}|$ is an algebraic specification with $SPEC = (\Sigma, E)$,
- P is a set of places,
- T is a set of transitions,
- $pre, post : T \rightarrow (T_{OP}(X) \times P)^\oplus$
are the pre- and postcondition functions of T , defining, for each transition with adjacent arcs, the arc inscriptions and the weight
- $cond : T \rightarrow \mathcal{P}_{fin}(EQNS(\Sigma))$
are the function that maps each transition to a finite set of conditions over the signature, with variables representing the firing conditions.

Definition 7 (Algebraic High Level Net Morphisms). Given two algebraic high level nets $N_i = (X_i, A_i, SPEC_i, P_i, T_i, pre_i, post_i, cond_i)$ for $i = 1, 2$, an algebraic high level net morphism $f : N_1 \rightarrow N_2$ is given by $f = (f_X, f_{SPEC}, f_A, f_P, f_T)$ where

- $f_X : X_1 \rightarrow T_{OP_2}(X_2)$ maps variables to terms,
- $f_{SPEC} : SPEC_1 \rightarrow SPEC_2$ is a specification morphism,
- $f_A : A_1 \rightarrow V_{f_\Sigma}(A_2)$ is a homomorphism in $\mathbf{Alg}(\mathbf{SPEC}_1)$,
- $f_P : P_1 \rightarrow P_2$ maps places to places in **Set**,
- $f_T : T_1 \rightarrow T_2$ maps transitions to transitions in **Set**

such that the following diagram, in which $(f_{SPEC}, f_X)^\sharp$ denotes the function $T_{OP_1}(X_1) \rightarrow T_{OP_2}(X_2)$ uniquely induced by (f_{SPEC}, f_X) , commutes componentwise

$$\begin{array}{ccc}
 \mathcal{P}_{fin}(EQNS(SIG_1)) & \xleftarrow{cond_1} T_1 & \xrightleftharpoons[post_1]{pre_1} (T_{OP_1}(X_1) \times P_1)^\oplus \\
 \downarrow \mathcal{P}_{fin}(f_X^\sharp) & \downarrow f_T & \downarrow ((f_{SPEC}, f_X)^\sharp \times f_P)^\oplus \\
 \mathcal{P}_{fin}(EQNS(SIG_2)) & \xleftarrow{cond_2} T_2 & \xrightleftharpoons[post_2]{pre_2} (T_{OP_2}(X_2) \times P_2)^\oplus
 \end{array}
 \quad (1) \qquad (2)$$

An algebraic high level net morphism is strict if in addition:

- $f_X : X_1 \rightarrow X_2$ is an injective mapping of variables,
- $f_{SPEC} : SPEC_1 \rightarrow SPEC_2$ is a strict specification morphism (see [12]),
- $f_A : A_1 \rightarrow V_{f_\Sigma}(A_2)$ is an isomorphism in $\mathbf{Alg}(\mathbf{SPEC}_1)$,
- f_P and f_T are injective.

Algebraic high level nets and algebraic high level net morphisms give rise to the category **AHL**.

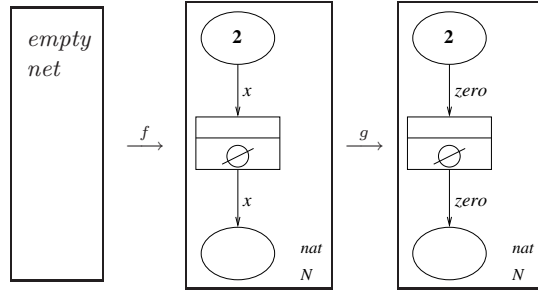


Fig. 7. Firing and Algebraic High Level Net Morphisms

Remark 1 (Firing and Algebraic High Level Net Morphism). Unlike the case of strict algebraic high level net morphisms, the firing behaviour is not preserved by algebraic high level net morphisms in general. As an example, consider the algebraic high level nets and algebraic high level net morphisms in Fig. 7: the transition in the intermediate algebraic high level net can fire, but there is no transition in the left-hand net which is enabled, i. e. the morphism f does not reflect the firing behaviour. Similarly, the transition in the right-hand net cannot fire, i. e. the morphism g , with $g_X(x) = \text{zero}$ and identity components otherwise, does not preserve the firing behaviour.

The following proposition summarizes the cocompleteness properties of the petri net categories defined. Proofs and details in [30, 17], [30], [19], and [20, 21].

Proposition 1 (Finite cocompleteness).

1. The category **PT** as defined in Def. 1 is finitely cocomplete.
2. The category **EN** as defined in Def. 2 is finitely cocomplete.
3. The category **marked PT** as defined in Def. 3 is finitely cocomplete.
4. The category **AHL** as defined in Def. 7 is not finitely cocomplete in general.
5. In the category **AHL** pushouts exist if one of the two morphisms is strict.

3.2 High Level Replacement Systems

In this subsection we briefly review the concepts of high-level replacement (HLR) systems in the sense of [14], a categorical generalization of graph grammars. High-level replacement systems are formulated for an arbitrary category $\mathcal{C}$ with a distinguished class $\mathcal{M}$ of morphisms. High-level replacement systems allow a great variety of interpretations of the concept of rules. In [12] they have been instantiated by algebraic specifications and Petri nets and used in the context of software development.

In general, rules are split into a part $L \setminus K$ to be deleted, an part $R \setminus K$ to be added and an interface K which is required for the application of the rule and is preserved.

Definition 8 (Rules and Transformations). A rule $r = (L \xleftarrow{u} K \xrightarrow{v} R)$ in $\mathcal{C}$ consists of the objects L , K and R , called left-hand side, interface (or gluing object) and right-hand side respectively, and two morphisms $K \xrightarrow{u} L$ and $K \xrightarrow{v} R$, from a distinguished class $\mathcal{M}$ of morphisms in $\mathcal{C}$.

Given a rule $r = (L \xleftarrow{u} K \xrightarrow{v} R)$, a direct transformation $G \xRightarrow{r} H$, from an object G to an object H is given by two pushout diagrams (1) and (2) in the category $\mathcal{C}$.

$$\begin{array}{ccccc}
 L & \xleftarrow{u} & K & \xrightarrow{v} & R \\
 g_1 \downarrow & & (1) \ g_2 \downarrow & & (2) \ g_3 \downarrow \\
 G & \xleftarrow{c_1} & C & \xrightarrow{c_2} & H
 \end{array}$$

The morphisms $L \xrightarrow{g_1} G$ and $R \xrightarrow{g_3} H$ are called occurrences of L in G and R in H , respectively. By an occurrence of a rule $r = (L \xleftarrow{u} K \xrightarrow{v} R)$ in a structure G is meant an occurrence of the left-hand side L in G .

A transformation sequence $G \xRightarrow{*} H$, indicates that either G is isomorphic to H or there is a sequence of direct transformations: $G = G_0 \xRightarrow{r_1} G_1 \xRightarrow{r_2} \dots \xRightarrow{r_n} G_n = H$.

Definition 9 (High-Level Replacement System). Given a category $\mathcal{C}$ together with a distinguished class of morphisms $\mathcal{M}$ the pair $(\mathcal{C}, \mathcal{M})$ is called a HLR-category if $(\mathcal{C}, \mathcal{M})$ satisfies the following HLR-Conditions

1. Existence of semi- $\mathcal{M}$ pushouts
2. Existence of $\mathcal{M}$ -pullbacks
3. Inheritance of $\mathcal{M}$ under pushouts and under pullbacks
4. Existence of binary coproducts and compatibility with $\mathcal{M}$
5. Pushout squares of $\mathcal{M}$ -morphisms are pullback squares.
6. $\mathcal{M}$ -pushout-pullback-decomposition

Transition preserving place/transition net morphisms, with strict morphisms in the distinguished class $\mathcal{M}$, give rise to a high-level replacement system. Similarly for elementary nets, marked place/transition nets and algebraic high level nets.

Proposition 2 (Net based HLR categories).

1. The category **PT** as defined in Def. 1 together with strict morphisms forms a high-level replacement system.
2. The category **EN** as defined in Def. 2 together with strict morphisms forms a high-level replacement system.
3. The category **marked PT** as defined in Def. 3 together with strict morphisms forms a high-level replacement system.
4. The category **AHL** as defined in Def. 7 together with strict morphisms forms a high-level replacement system.

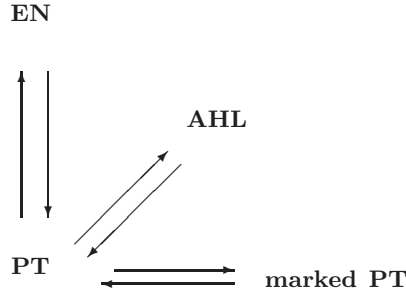


Fig. 8. Functors between Categories

4 Relating Different Categories

In this section we define functors between the categories in Subsection 3.1, which are represented graphically in Fig. 8.

These functors are examined with respect to freeness, preservation of colimits, and other properties. Then Theorem 1 states sufficient conditions for preservation of derivations by functors. One of the conditions is $\mathcal{M}$ -compatibility which is defined below and holds for each of the functors.

Definition 10 ($\mathcal{M}$ -compatible Functors). *Given two high-level replacement systems $(\mathcal{C}_1, \mathcal{M}_1)$ and $(\mathcal{C}_2, \mathcal{M}_2)$ and a functor $F : \mathcal{C}_1 \rightarrow \mathcal{C}_2$.*

F is $\mathcal{M}$ -compatible, if and only if for each $m_1 \in \mathcal{M}_1$ we have $F(m_1) \in \mathcal{M}_2$.

4.1 Functors between PT and EN

The first pair of functors relates place/transition nets and elementary nets. To obtain an elementary net from a place/transition net, weights are discarded. Viceversa, an elementary net can be viewed as a place/transition net with all arc weights equal to 1.

Definition 11 ($Causality : \mathbf{PT} \rightarrow \mathbf{EN}$). *The functor $Causality : \mathbf{PT} \rightarrow \mathbf{EN}$ between the categories $\mathbf{PT}$ and $\mathbf{EN}$ is defined as follows:*

on objects *For a place/transition net $N = (P_N, T_N, pre_N, post_N)$ define $Causality(N) = (P_{N'}, T_{N'}, pre_{N'}, post_{N'})$ with $P_{N'} = P_N$, $T_{N'} = T_N$, $pre_{N'}(t) = \{p \mid p \leq pre_N(t)\}$, and $post_{N'}(t) = \{p \mid p \leq post_N(t)\}$.*

on morphisms *For a place/transition net morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ define $Causality(f) = f : Causality(N_1) \rightarrow Causality(N_2)$.*

Definition 12 ($Weight : \mathbf{EN} \rightarrow \mathbf{PT}$). *The functor $Weight : \mathbf{EN} \rightarrow \mathbf{PT}$ between the categories $\mathbf{PT}$ and $\mathbf{EN}$ is defined as follows:*

on objects For an elementary net $N = (P_N, T_N, pre_N, post_N)$ define
 $Weight(N) = (P_{N'}, T_{N'}, pre_{N'}, post_{N'})$ with $P_{N'} = P_N$, $T_{N'} = T_N$, $pre_{N'}(t) = \bigoplus_{p \in pre_N(t)} p$, and $post_{N'}(t) = \bigoplus_{p \in post_N(t)} p$.
on morphisms For an elementary net morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ define $Weight(f) = f : Weight(N_1) \rightarrow Weight(N_2)$.

Proposition 3. *The functors $Weight$ and $Causality$ are $\mathcal{M}$ -compatible and preserve finite colimits.*

4.2 Functors between PT and Marked PT

Functors in this subsection relate place/transition nets and marked place/transition nets by adding empty markings (with the functor $Mark$) and by removing markings (with the functor $Unmark$).

Definition 13 ($Mark : \mathbf{PT} \rightarrow \mathbf{marked PT}$). *The functor $Mark : \mathbf{PT} \rightarrow \mathbf{marked PT}$ between the categories $\mathbf{PT}$ and $\mathbf{marked PT}$ is defined as follows:*

on objects For a place/transition net $N = (P_N, T_N, pre_N, post_N)$ define
 $Mark(N) = (P_{N'}, T_{N'}, pre_{N'}, post_{N'}, \widehat{m_{N'}})$ with $P_{N'} = P_N$, $T_{N'} = T_N$, $pre_{N'} = pre_N$, $post_{N'} = post_N$, and $\widehat{m_{N'}} = \epsilon$ the empty initial marking.
on morphisms For a place/transition net morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ define $Mark(f) = f : Mark(N_1) \rightarrow Mark(N_2)$.

Definition 14 ($Unmark : \mathbf{marked PT} \rightarrow \mathbf{PT}$). *The functor $Unmark : \mathbf{marked PT} \rightarrow \mathbf{PT}$ between the categories $\mathbf{PT}$ and $\mathbf{marked PT}$ is defined as follows:*

on objects For a marked place/transition net $N = (P_N, T_N, pre_N, post_N, \widehat{m_N})$ define $Unmark(N) = (P_{N'}, T_{N'}, pre_{N'}, post_{N'})$ with $P_{N'} = P_N$, $T_{N'} = T_N$, $pre_{N'} = pre_N$, $post_{N'} = post_N$.
on morphisms For a marked place/transition net morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ define $Unmark(f) = f : Unmark(N_1) \rightarrow Unmark(N_2)$.

Proposition 4. *The functors $Mark$ and $Unmark$ are $\mathcal{M}$ -compatible and preserve finite colimits. Furthermore the functor $Unmark$ is left adjoint to $Mark$.*

4.3 Functors between PT and AHL

Here functors between place/transition nets and algebraic high level nets are defined to forget the algebraic structure (with the functor $Skeleton$) or to add a trivial one (with the functor $Data$).

Definition 15 ($Data : \mathbf{PT} \rightarrow \mathbf{AHL}$). *The functor $Data : \mathbf{PT} \rightarrow \mathbf{AHL}$ between the categories $\mathbf{PT}$ and $\mathbf{AHL}$ is defined as follows:*

on objects For a place/transition net $N = (P_N, T_N, pre_N, post_N)$ define
 $Data(N) = (X_{N'}, A_{N'}, SPEC_{N'}, P_{N'}, T_{N'}, pre_{N'}, post_{N'}, cond_{N'})$ where:

- The set of variables (of the form (t, p)) is given by the set $X_{N'} = X_{pre} \uplus X_{post} = inc \circ (id_T, pre_N)(T) \uplus inc \circ (id_T, post_N)(T)$ (where $inc(t, w)$ is the join $(\uplus)$ of λ_i copies of (t, p_i) for each i , if $w = \lambda_1 \cdot p_1 \oplus \dots \oplus \lambda_n \cdot p_n \in P^\oplus$),
- $SPEC_{N'} = \emptyset$ is the empty algebraic specification,
- $A_{N'} = \emptyset$ is the empty algebra,
- $P_{N'} = P_N$,
- $T_{N'} = T_N$,
- $pre_{N'}(t) = \bigoplus_{(t,p) \in inc \circ (id_T, pre_N)(t)} ((t, p), p)$
- $post_{N'}(t) = \bigoplus_{(t,p) \in inc \circ (id_T, post_N)(t)} ((t, p), p)$
- $cond_{N'}(t) = \emptyset$ for each $t \in T_{N'}$.

on morphisms For a place/transition net morphism $f = (f_P, f_T) : N_1 \rightarrow N_2$ define $Data(f) = (f_X, f_{SPEC}, f_A, f_P, f_T) : Data(N_1) \rightarrow Data(N_2)$ with

- f_X given by $(f_T, f_P)^\oplus$,
- $f_{SPEC} = \emptyset$ the initial SPEC morphism,
- $f_A = \emptyset$ the initial algebra morphism.

Definition 16 (*Skeleton : AHL $\rightarrow$ PT*). The functor *Skeleton* : **AHL** $\rightarrow$ **PT** between the categories **PT** and **AHL** is defined as follows:

on objects For an algebraic high level net $N = (X_N, A_N, SPEC_N, P_N, T_N, pre_N, post_N, cond_N)$ define $Skeleton(N) = (P_{N'}, T_{N'}, pre_{N'}, post_{N'})$ with:

- $P_{N'} = P_N$
- $T_{N'} = T_N$
- $pre_{N'}(t) = pr_{P^\oplus} \circ pre_N(t)$ with $pr_{P^\oplus} : (TOP(X_N) \times P_N)^\oplus \rightarrow P_N^\oplus$ the projection
- $post_{N'}(t) = pr_{P^\oplus} \circ post_N(t)$ with $pr_{P^\oplus} : (TOP(X_N) \times P_N)^\oplus \rightarrow P_N^\oplus$ the projection

on morphisms For a morphism $f = (f_X, f_{SPEC}, f_A, f_P, f_T) : N_1 \rightarrow N_2$ define $Skeleton(f) = (f_P, f_T) : Skeleton(N_1) \rightarrow Skeleton(N_2)$.

Proposition 5. The functors *Data* and *Skeleton* are $\mathcal{M}$ -compatible and preserve finite colimits.

4.4 General Results

In this section we state results concerning the preservation of rules and derivations by functors. Then we claim that all of the functors of the previous sections preserve derivations. Proofs can be found in [20].

Definition 17 (Preservation of Derivations). Given two high-level replacement systems $(\mathcal{C}_1, \mathcal{M}_1)$ and $(\mathcal{C}_2, \mathcal{M}_2)$ and a functor $F : \mathcal{C}_1 \rightarrow \mathcal{C}_2$.

The functor F preserves derivations iff for each rule $r : L \xleftarrow{l} K \xrightarrow{r} R$ and derivation $G \xrightarrow{r} H$ in $\mathcal{C}_1$ there is a rule $F(r) : F(L) \xleftarrow{F(l)} F(K) \xrightarrow{F(r)} F(R)$ and a derivation $F(G) \xrightarrow{F(r)} F(H)$ in $\mathcal{C}_2$.

Theorem 1. *Given two high-level replacement systems $(\mathcal{C}_1, \mathcal{M}_1)$ and $(\mathcal{C}_2, \mathcal{M}_2)$ and a functor $F : \mathcal{C}_1 \rightarrow \mathcal{C}_2$.*

If F preserves pushouts and is $\mathcal{M}$ -compatible, then it preserves derivations.

Proposition 6 (Functors Preserving Derivations). *The following functors preserve derivations:*

- *Functor Causality : $\mathbf{PT} \rightarrow \mathbf{EN}$ as defined in Def. 11*
- *Functor Weight : $\mathbf{EN} \rightarrow \mathbf{PT}$ as defined in Def. 12*
- *Functor Mark : $\mathbf{PT} \rightarrow \mathbf{marked PT}$ as defined in Def. 13*
- *Functor Unmark : $\mathbf{marked PT} \rightarrow \mathbf{PT}$ as defined in Def. 14*
- *Functor Data : $\mathbf{PT} \rightarrow \mathbf{AHL}$ as defined in Def. 15*
- *Functor Skeleton : $\mathbf{AHL} \rightarrow \mathbf{PT}$ as defined in Def. 16*

5 Around the Cube

The functors presented in Section 4 operate on the net structure (*Causality* and *Weight*), on the initial marking (*Mark* and *Unmark*) and on the data of a Petri net (*Data* and *Skeleton*). They add one of these components or ‘remove’ it and the components are basically independent of each other. We now discuss the lifting of these functors as described in the cube in Figure 9, where the directed arcs represent functors to and from the corresponding categories.

The cube is based on the following three dimensions:

- net structure (vertical lines)
- data (diagonal lines)
- marking (horizontal lines)

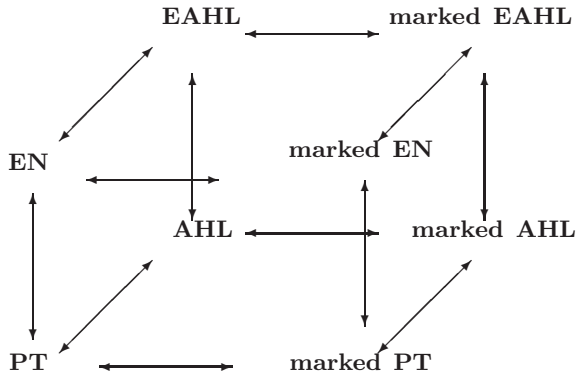


Fig. 9. Cube of net class transformations

The categories in the lower left corners and the pairs of functors between them have been presented in the previous section. The category **EAHL** consists of a new class of elementary algebraic high level nets, which can be regarded as a combination of elementary nets and algebraic high level nets: they comprise data, but arc weight and marking are restricted to one. Similarly, objects of **marked EN** are marked elementary nets and objects of **marked AHL** are marked algebraic high level nets. The category **marked EAH** in the right upper corner consists of objects combining all of these features.

The theory of abstract Petri nets, introduced in [30] and reviewed next, yields a suitable frame for a uniform lifting of the low-level functors between **PT**, **EN**, and **marked PT**. Then this approach is used to uniformly define functors operating on the net structure and marking with different domains and codomains as in the cube above.

5.1 Theory of Abstract Petri Nets

In [30] different classes of low-level Petri nets are defined uniformly by a net structure functor Net , which is the composition of a pair of adjoint functors.

Definition 18 (Net Structure Functor). *Given the category **Set**, a subcategory **Struct** of commutative semigroups **CSGroup**, and functors $F \dashv G : \mathbf{Struct} \rightarrow \mathbf{Set}$, where F is left adjoint to G with universal morphisms $\gamma^S : S \rightarrow G \circ F(S)$ for all sets S in **Set**, the composition*

$$Net = G \circ F : \mathbf{Set} \rightarrow \mathbf{Set}$$

is called a net structure functor.

Definition 19 (Low-Level Abstract Petri Net). *A low-level abstract Petri net $N = (P, T, pre, post)$ is given by sets T and P , called transitions and places, and functions*

$$T \begin{array}{c} \xrightarrow{pre} \\ \xrightarrow{post} \end{array} Net(P)$$

called pre- and postcondition of T , where Net is a net structure functor.

Although (low-level) Abstract Petri nets do not comprise an initial marking, the notion of marking can be defined and used to specify the firing behaviour. Given a low-level abstract Petri net $N = (P, T, pre, post)$, where $pre, post : T \rightarrow Net(P)$, its *marking* is given by $m \in Net(P)$, where F is as in Def. 18. The marking here is given by $m \in Net(P)$ instead of $m \in F(P)$ as we do not treat the behaviour of abstract nets. For defining deletion and production, the operation of the commutative group is used in [30] in order to state that morphisms preserve firing behaviour.

To obtain place/transition nets as instances of low-level abstract Petri nets, we can choose, for the net structure functor Net , the category of commutative monoids **CMon** as **Struct**, and as $F : \mathbf{Set} \rightarrow \mathbf{CMon}$ the free functor assigning

the free commutative monoid to all sets and extensions of set based functions to morphisms.

$$\mathbf{Set} \xrightarrow{F} \mathbf{FCMon} \xrightarrow{\mathcal{I}} \mathbf{CMon} \xrightarrow{G} \mathbf{Set}$$

(Unsafe) elementary nets are obtained by choosing $\mathbf{PSet}$, the category of power sets, as $\mathbf{Struct}$, as F the usual power set functor $\mathcal{P}$, and as G the inclusion $\mathcal{I} : \mathbf{PSet} \rightarrow \mathbf{Set}$.

$$\mathbf{Set} \xrightarrow{\mathcal{P}} \mathbf{PSet} \xrightarrow{\mathcal{I}} \mathbf{Set}$$

5.2 Lifting of Functors

A change of the net structure can be realized by a functor transformation and, in turn, this net class transformation can be induced by a natural transformation. Proofs in [20].

Proposition 7. *Given two net structure functors $Net, Net' : \mathbf{Set} \rightarrow \mathbf{Set}$ yielding the Petri net categories $\mathbf{PNC}$ and $\mathbf{PNC}'$, a natural transformation $\nu : Net \rightarrow Net'$ uniquely defines a net class transformation $\hat{\nu} : \mathbf{PNC} \rightarrow \mathbf{PNC}'$ such that $pre'_{\hat{\nu}(N)} = \nu_{Net(P)} \circ pre_N$ and $post'_{\hat{\nu}(N)} = \nu_{Net(P)} \circ post_N$*

The functors *Causality* and *Weight* of Section 4 are induced by the following natural transformations.

Definition 20 (Natural Transformations *MonPow* and *PowMon*). *Given net structure functors $Net = G \circ F$, where $F : \mathbf{Set} \rightarrow \mathbf{CMon}$, and $Net' = G' \circ F' : \mathbf{Set} \rightarrow \mathbf{Set}$, with $F' = \mathcal{P}$ as above, the natural transformation *MonPow* : $Net \rightarrow Net'$ is defined by*

$$MonPow_X : Net(X) \rightarrow Net'(X)$$

with

$$MonPow_X(w) = \bigcup_{x \leq w} x \in X^\oplus$$

for $w \in X^\oplus = Net(X)$.

The natural transformation *PowMon* : $Net' \rightarrow Net$ is defined by

$$PowMon_X : Net'(X) \rightarrow Net(X)$$

with

$$PowMon_X(X') = \bigoplus_{x \in X'} x \in \mathcal{P}(X)$$

for $X' \in \mathcal{P}(X) = Net'(X)$.

Proposition 8. *The functors *Causality* and *Weight* are induced by the natural transformations *PowMon* and *MonPow*, respectively.*

We can use the natural transformations above also for net class transformations between other Petri net classes, provided that they are based on a net structure functor. This prerequisite is satisfied for the low-level nets in **EN**, **marked PT**, and **marked EN**, as we will show, and also satisfied for high-level abstract Petri nets [30]. Although the instantiation there differs slightly from our notion of algebraic high level nets, we can still adopt the idea and generalize our notion of algebraic high level nets by replacing the concrete net structure functor $_{-}^{\oplus}$, used for defining the *pre*- and *post*-functions, by a generic net structure functor. Instantiation of the generic algebraic high level net by $Net = G \circ I \circ F : \mathbf{Set} \xrightarrow{F} \mathbf{FCMon} \xrightarrow{\mathcal{I}} \mathbf{CMon} \xrightarrow{G} \mathbf{Set}$ yields the algebraic high level nets of Def. 6. Applying the natural transformation *MonPow* to the above net structure functor yields the new net class of elementary algebraic high level nets **EAHL**. Viceversa, the application of *PowMon* to these nets yields again algebraic high level nets.

Definition 21 (Generic Algebraic High Level Net). *Given a net structure functor Net , a generic algebraic high level net is given by an algebraic high level net where the *pre*- and *post* functions are replaced by*

$$pre, post : T \rightarrow Net(T_{OP}(X) \times P)$$

The lifting of *Causality* and *Weight* to marked nets requires a generalization of those nets in the spirit of abstract Petri nets [30]. Given a net structure functor Net , a *generalized marked place/transition net* is given by a marked place/transition net where the *pre*- and *post* functions and the initial marking are replaced by

$$pre, post : T \rightarrow Net(P) \quad \text{and} \quad m \in Net(P)$$

The instantiation of the net structure functor by

$$Net = G \circ I \circ F : \mathbf{Set} \xrightarrow{F} \mathbf{FCMon} \xrightarrow{\mathcal{I}} \mathbf{CMon} \xrightarrow{G} \mathbf{Set}$$

yield the marked place/transition nets of Def. 3.

Proposition 9 (Lifting of Causality and Weight). *The natural transformation *PowMon* and *MonPow* defined in Def. 20 induce the following net class transformations.*

- *Causality* : **PT** → **EN**
- *Weight* : **EN** → **PT**
- *Causality*_{AHL} : **AHL** → **EAHL**
- *Weight*_{EAHL} : **EAHL** → **AHL**
- *Causality*_{m, PT} : **marked PT** → **marked EN**
- *Weight*_{m, EN} : **marked EN** → **marked PT**
- *Causality*_{m, AHL} : **marked AHL** → **marked EAHL**
- *Weight*_{m, EAHL} : **marked EAHL** → **marked AHL**

5.3 Lifting of *Mark*

The dynamic behaviour of Petri nets is based on the notion of an (initial) marking and the firing of transitions for a change of the marking. Transitions delete tokens in their predomain and produce tokens on the postdomain, as described by the *pre*- and *post*-functions of the net. Typically, the structure of marking and codomain of *pre*- and *post*- functions coincide for low-level nets (e. g. in place/transition nets the marking is given by $m \in P^\oplus$ and the *pre* (*post*) function by $pre : T \rightarrow P^\oplus$ ($post : T \rightarrow P^\oplus$). Deletion and production of tokens during firing of a transition are represented as subtraction and addition in the underlying structure (e. g. commutative monoids for place/transition nets). For a uniform lifting of the functor *Mark* of Def. 13 we generalize this principle, and define, given $pre, post : T \rightarrow Y$ where the set Y designates the codomain of the functions, a *marking* m of the net as an element $m \in Y$.

We can now give a uniform construction of adding initial marking to a net. In fact, this construction can be lifted to a functor. We choose a distinguished initial marking to be added, where no places are marked. This choice is consistent with the morphisms on marked place/transition nets already defined. For simplicity, when adding marking for the other categories, we do not change the notions of morphisms, which, however, can be done in a manner similar to the definition of **marked PT** in Def. 3. The functors that forget the initial marking can be defined in a similar way (compare *Unmark* in Def. 14).

Summarizing we have a uniform lifting of the functor *Mark*, adding initial marking to a place/transition net and the inverse functor *Unmark* defined in Def. 13 and 14. Finally, the lifting of *Data* uses the previously discussed liftings of *Weight/Causality* and *Mark/Unmark*. This gives the following result:

Definition 22 (Lifting of *Data*). *The functors adding data are defined as follows*

- $Data : \mathbf{PT} \rightarrow \mathbf{AHL}$ is defined as in Def. 15
- $Data_{EN} : \mathbf{EN} \rightarrow \mathbf{EAHL}$ is defined by

$$Data_{EN} = Causality_{m.AHL} \circ Data \circ Weight$$
- $Data_{m.PT} : \mathbf{marked PT} \rightarrow \mathbf{marked AHL}$ is defined by

$$Data_{m.PT} = Mark_{AHL} \circ Data \circ Unmark$$
- $Data_{m.EN} : \mathbf{marked EN} \rightarrow \mathbf{marked AHL}$ is defined by

$$Data_{m.EN} = Causality_{m.AHL} \circ Data_{m.PT} \circ Weight_{m.EN}$$

So the cube described in Fig. 9 commutes.

6 Concluding Remarks

In this paper we have presented an incremental approach to the stepwise development of models based on net model changes and on net class enrichments. The approach is placed in a categorical setting based on high level replacement systems.

To model net class enrichments and abstractions, functors have been defined that preserve the rules and the derivations, and that add features in an "independent" way. This allows them to be lifted so that the addition and removal of markings and data and changes in the net structure can be described in a uniform way independently of the net classes.

Further research can follow several directions. One is to investigate other net classes such as colored or timed Petri nets and to define the appropriate functors to model these net class enrichments. Another direction is the study of the preservation of properties such as safety, liveness and deadlock freeness (among others) by model and net class transformations. Some results in this direction can already be found in [33, 34, 35, 22].

Acknowledgements

Most of the research reported here has been carried out in collaboration with M. Gajewsky, who was supported by the German Research Council (DFG) within the "DFG- Forschergruppe PETRINETZ-TECHNOLOGIE" This work was also partially supported by the TMR network "General Theory of Graph Transformation Systems" (GETGRATS) and by the Esprit WG APPLIGRAPH. Thanks to the anonymous referee for several suggestions that helped improve the presentation.

References

- [1] J. Adamek, H. Herrlich, and G. Strecker. *Abstract and Concrete Categories*. Series in Pure and Applied Mathematics. John Wiley and Sons, 1990.
- [2] B. Braatz, H. Ehrig and M. Urbasek. Petri Net Transformations in the Petri Net Baukasten. (this volume). 410
- [3] C. Brown and D. Gurr. Refinement and Simulation of Nets – a categorical characterization. In K. Jensen, editor, *13th International Conference on Application and Theory of Petri Nets*, volume 616, pages 76–92. Lecture Notes in Computer Science, Springer Verlag, Jun 1992.
- [4] W. Brauer, R. Gold, and W. Vogler. A Survey of Behaviour and Equivalence Preserving Refinements of Petri Nets. *Advances in Petri Nets*, Lecture Notes in Computer Science 483:1–46, 1990.
- [5] A. Borusan. A Petri Net Based Modelling Technology for Manufacturing System Engineering. In *Proceedings Cars And FOF'93 International Conference on CAD, CAM, Robotics and Factories of the Future*, pages 41–50, New Jersey, USA, 1993. 411
- [6] A. Borusan. Modelling and design of complex systems with Petri nets. In E. Schnieder, editor, *Entwurf komplexer Automatisierungssysteme*, pages 237–247. TU Braunschweig, June 1995. 411
- [7] Kamran R. Damavandi. Eine Methodenkette zur integrierten Software-Entwicklung: Von der Problemstellung zum Programmsystem mit Petrinetze. Technical Report N91–15769, Technische Universität Berlin, Fachbereich 20, Informatik — Washington, DC, USA: National Aeronautics and Space Administration (/3/XAD, ETN–91–98497), 1990. Dissertation. 411

- [8] J. Desel. On Abstraction of Nets. In *Advances in Petri Nets*, pages 78–92. Springer Verlag LNCS 524, 1991.
- [9] W. Deiters and V. Gruhn. Software Process Model Analysis Based on FUNSOFT Nets. *Mathematical Modelling and Simulation*, 8, May 1991. 409
- [10] J. Desel and A. Merceron. Vicinity Respecting Net Morphisms. In *Advances in Petri Nets*, pages 165–185. Springer Verlag, 1990. Lecture Notes in Computer Science 483.
- [11] J. Desel and W. Reisig. Place/transition Petri nets. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets: Basic Models*, pages 122–173. Springer Verlag, LNCS 1491, 1998. 410
- [12] H. Ehrig, M. Gajewsky, and F. Parisi-Presicce. *High-Level Replacement Systems with Applications to Algebraic Specifications and Petri Nets*, volume 3: Concurrency, Parallelism, and Distribution, chapter 6, pages 341–400. World Scientific, Handbook of Graph Grammars and Computing by Graph Transformations edition, 1999. 416, 417
- [13] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. From graph grammars to high level replacement systems. In *4th Int. Workshop on Graph Grammars and their Application to Computer Science*, LNCS 532, pages 269–291. Springer Verlag, 1991. 409
- [14] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991. 417
- [15] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer Verlag, Berlin, 1985.
- [16] C. Ermel, J. Padberg, and H. Ehrig. Requirements Engineering of a Medical Information System Using Rule-Based Refinement of Petri Nets. In D. Cooke, B. J. Krämer, P. C-Y. Sheu, J. P. Tsai, and R. Mittermeir, editors, *Proc. Integrated Design and Process Technology*, pages 186–193. Society for Design and Process Science, 1996. Vol.1. 411
- [17] H. Ehrig, J. Padberg, C. Ermel, and L. Grigoriu. Petrinetze: Modellierung, Strukturierung und Kompositionalität. Skript zur Lehrveranstaltung, SS99, Technische Universität Berlin, 1999. 417
- [18] M. Gajewsky. Concepts and Requirements for Transformations within Petri Net Based Process Models. In A. Ertas, editor, *5th World Conference on Integrated Design and Process Technology, Special Session on Model Integration*, 2000. CD-ROM, 8 pages. 409
- [19] M. Gajewsky, K. Hoffmann, and J. Padberg. Place Preserving and Transition Gluing Morphisms in Rule-Based Refinement of Place/Transition Systems. Technical Report 99-14, Technical University Berlin, 1999. 415, 417
- [20] M. Gajewsky and F. Parisi-Presicce. Formal Transformations of Petri Nets. Technical Report 2000-12, Technical University Berlin, 2000. 410, 417, 421, 424
- [21] M. Gajewsky and F. Parisi-Presicce. On the Compatibility of Model and Model-Class Transformations In M. Cerioli, G. Reggio editors, *Recent Trends in Algebraic Development Techniques (WADT'01). Selected Papers*. pages 109–127, Lecture Notes in Computer Science 2267, Springer Verlag 2001. 410, 411, 415, 417
- [22] M. Gajewsky, J. Padberg and M. Urbasek. Rule-Based Refinement of Place/Transition Systems: Preserving liveness properties. Technical Report 2001-8, Technical University Berlin, 2001. 427

- [23] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 3: Practical Use. Springer Verlag, EATCS Monographs in Theoretical Computer Science edition, 1997. 409
- [24] A. Janhsen, K. Lemmer, B. Ptok, and E. Schnieder. Formal specifications of the European Train Control System. In *IFAC Transportation Systems, 8th Symposium on Transportation Systems*, 1997. 409
- [25] Sabine Lembke. Anwendungsentwicklung in der LION-Entwicklungsumgebung (LEU). In Hartmut Ehrig, Wolfgang Reisig, and Herbert Weber, editors, *Kick-Off-Workshop der DFG-Forscherguppe Petrinetz-Technologie*, number 73 in Informatik-Berichte der Humboldt-Universität zu Berlin, pages 72–81, July 1996. 411
- [26] Axel Martens. Software-Engineering von Workflow-Applikationen mit Petrinetzen. Diplomarbeit, Humboldt-Universität zu Berlin, Institut für Informatik, 1997. 411
- [27] Susanne Müller. *Ablaufmodellierung als Analyse-, Entwurfs- und Realisierungsmethodik im Softwareentwicklungsprozeß*. Reihe Wirtschaftsinformatik, Band 15. Verlag Josef Eul, Bergisch Gladbach, Köln, 1995. 411
- [28] A. Oberweis. *Modellierung und Ausführung von Workflows mit Petri-Netzen*. Teubner-Reihe Wirtschaftsinformatik. B. G. Teubner Verlagsgesellschaft, Stuttgart Leipzig, 1996. 411
- [29] J. Padberg. Survey of high-level replacement systems. Technical Report 93-8, Technical University of Berlin, 1993.
- [30] J. Padberg. *Abstract Petri Nets: A Uniform Approach and Rule-Based Refinement*. PhD thesis, Technical University Berlin, 1996. Shaker Verlag. 417, 423, 425
- [31] Julia Padberg. Categorical Approach to Horizontal Structuring and Refinement of High-Level Replacement Systems. *Applied Categorical Structures*, 7(4):371–403, December 1999.
- [32] J. Padberg, H. Ehrig, and L. Ribeiro. Algebraic high-level net transformation systems. *Mathematical Structures in Computer Science*, 5:217–256, 1995. 410, 415
- [33] J. Padberg, M. Gajewsky, and C. Ermel. Rule-Based Refinement of High-Level Nets Preserving Safety Properties. In E. Astesiano, editor, *Fundamental Approaches to Software Engineering*, pages 221–238. Springer Verlag, Lecture Notes in Computer Science 1382, 1998. 411, 427
- [34] Julia Padberg, Maïke Gajewsky, and Kathrin Hoffmann. Incremental Development of Safety Properties in Petri Net Transformations. In G. Engels and G. Rozenberg, editors, *Theory and Application of Graph Transformations (TAGT'98)*, *Lecture Notes in Computer Science* 1764, pages 410–425. Springer Verlag, 1999. 411, 415, 427
- [35] Julia Padberg, Kathrin Hoffmann, and Maïke Gajewsky. Stepwise Introduction and Preservation of Safety Properties in Algebraic High-Level Net Systems. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, pages 249–265. Springer Verlag, Lecture Notes in Computer Science 1783, 2000. 411, 427
- [36] G. Rozenberg and J. Engelfriet. Elementary net systems. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets: Basic Models*, pages 12–121. Springer Verlag, LNCS 1491, 1998. 410
- [37] W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80:1–34, 1991. 410

- [38] J. Vautherin. Parallel System Specification with Coloured Petri Nets. In G. Rozenberg, editor, *Advances in Petri Nets 87*, pages 293–308. Springer Verlag, 1987. Lecture Notes in Computer Science 266. [410](#)

Concurrency Based Transition Refinement for the Verification of Distributed Algorithms

Sibylle Peuker *

Software Verification Research Centre
The University of Queensland
Brisbane, Australia
sibylle@svrc.uq.edu.au

Abstract. We suggest a new notion of behaviour preserving transition refinement based on partial order semantics. This notion is called *transition refinement*. We introduced transition refinement for elementary (low-level) Petri Nets earlier [Peu01a]. For modelling and verifying complex distributed algorithms, high-level (Algebraic) Petri nets are usually used.

In this paper, we define transition refinement for Algebraic Petri Nets. This notion is more powerful than transition refinement for elementary Petri nets because it corresponds to the simultaneous refinement of several transitions in an elementary Petri net.

Transition refinement is particularly suitable for refinement steps that increase the degree of distribution of an algorithm, e.g. when synchronous communication is replaced by asynchronous message passing. We study how to prove that a replacement of a transition is a transition refinement.

Keywords: Distributed Algorithms; Concurrency; Action Refinement; Partial Order Semantics; Verification; Algebraic Petri Nets.

1 Introduction

Appropriate refinement techniques are crucial in the design and verification of distributed algorithms. The derivation of a distributed algorithm by stepwise refinement starts with a simple algorithm, which satisfies some desired properties but contains declarative, non-implementable parts. In a series of refinement steps, that algorithm is transformed into a fully distributed, implementable algorithm such that the desired properties are preserved in each step.

In the refinement process, we are often faced with the problem that a synchronous action of several agents has to be distributed into asynchronous actions of the agents, e.g. message-passing actions.

In Fig. 1, we have two agents a and b that perform a state change. Agent a changes from $a1$ to $a2$, agent b changes from $b1$ to $b2$. On the left-hand side, this

* Supported by the German Research Foundation (DFG) funded project: Kompositionale Verifikation, and the Australian Research Council Large Grant A49937045, Effective Real-Time Program Analysis.

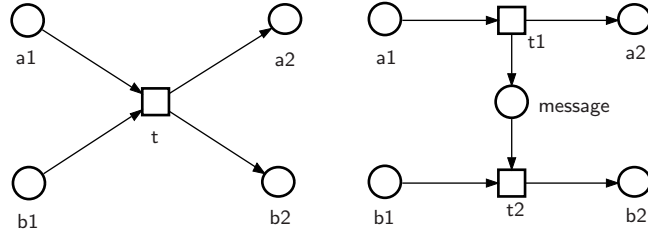


Fig. 1. A synchronous action of two agents and message passing between the agents

is done by a synchronous action of both agents; on the right-hand side agent *a* changes its state first and sends a message to *b*. Agent *b* performs the state change after receiving the message.

This situation is inconsistent with many notions of refinement, which was discussed earlier [Peu01a, Peu02]. In particular, it depends on the environment of the replaced transition whether a replacement of a synchronous action by asynchronous message passing is a refinement, i.e. if it indeed preserves properties of the system. We assume that transition *t* is a part of a system in which there is persistently a token on *a1* but *t* is never enabled because there is never a token on *b1*. After replacing *t* by *t1* and *t2*, transition *t1* may fire. If we assume that the token on *a1* is reproduced after firing of *t1*, then *t1* may even fire infinitely often but *t2* is never enabled. The replacement changes the behaviour of the system considerably and cannot be regarded as a property preserving refinement.

To deal with a situation like this, we proposed the concept of (*elementary*) *transition refinement* [Peu01a]. Whether a replacement of a transition is a transition refinement, does not only depend on the replacement but also on the vicinity of *t*. Transition refinement is based on the causal order of actions in an algorithm. Because of their natural partial order semantics, Petri nets are well-suited to define transition refinement. In [Peu01a], we give the formal definition of transition refinement for elementary Petri nets (P/T-nets without arc weights), and we discuss how to prove that a replacement of a transition is a transition refinement.

Behaviour preserving action refinement was already advocated by Wirth [Wir71] and developed in many subsequent papers. Valette already defined a behaviour preserving transition refinement for Petri nets in 1979 [Val79]. In this paper, a refinement of a transition is defined to be independent of the environment. Vogler extends the work of Valette [Vog87]. He defines property preserving refinement for Petri nets by means of *modules*. A module is a replacement for a transition that is a property preserving refinement in each possible environment. For this reason, a module is much more restrictive than the replacements allowed in our setting. In particular, modules cannot introduce new concurrency. In [vGG90], van Glabbeek and Goltz introduce a refinement operator, which transforms the environment such that introducing new concurrency

is possible. In our approach, we assume, that the environment (the distributed system) is fixed.

There are other approaches for property preserving refinement with Petri nets [PHG00, KP99] where, however, only safety properties are preserved. With algebraic transition refinement, we focus on liveness properties¹ of the system. Basically all liveness properties of a system are preserved by transition refinement. Safety properties of the refined system, such as S-invariants, may be composed of safety properties of the original system and the properties of the replacement. In particular, if each run of the original system terminates then each run of the refined system terminates as well and the final states of both systems coincide. Property preservation with transition refinement is discussed in more detail elsewhere [Peu01a].

For the modelling and verification of complex distributed algorithms, elementary Petri nets are not sufficiently expressive. Usually, the use of Algebraic Petri nets [Rei91, KV01] is more adequate. For a set of examples see [Rei98].

In this paper, we define transition refinement for Algebraic Petri nets. We lead the notion back to transition refinement in elementary Petri nets. The refinement of a transition in an Algebraic Petri net corresponds to the simultaneous refinement of a set of transitions in an elementary Petri net. We show that simultaneous refinement of a set of transitions in an elementary Petri net is not the same as refining these transitions one after another. Therefore, algebraic transition refinement is more powerful than elementary transition refinement.

Furthermore, we discuss how we can, nevertheless, use the proof criteria for elementary transition refinement in the case of algebraic transition refinement.

The paper is organised as follows: In Section 2, we define basic notions for elementary systems and their concurrent runs. In Section 3, we formally define transition refinement for elementary systems. In Section 4, we discuss how to prove that a replacement for a transition is a refinement. In Section 5, we define the simultaneous refinement of a set of transitions, and we discuss proof criteria in Section 6. In Section 7, we define basic notions for algebraic systems, their concurrent runs, and their unfoldings. After all this preparation, we finally define algebraic transition refinement in Section 8, and we discuss proof criteria in Section 9.

2 Elementary Systems – Basic Definitions

In this section, we define syntax and semantics of elementary systems. The reader familiar with elementary Petri nets and concurrent runs may skip this section (We use P/T-nets without arc weights).

¹ Considering distributed algorithms, we use the notions liveness and safety properties in the sense of [AS85]. A liveness property states that *something good* eventually happens, and a safety property states that *something bad* never happens. These notions are not to be confused with the notions of liveness and safety usually used in Petri nets.

Definition 1. (Petri Net) A Petri net (net for short) $N = (P, T, F)$ consists of two disjoint sets P and T and a relation $F \subseteq (T \times P) \cup (P \times T)$ such that for each $t \in T$ the sets $\bullet t = \{p \in P \mid (p, t) \in F\}$ and $t^\bullet = \{p \in P \mid (t, p) \in F\}$ are non-empty and finite.

The elements of P , T and F are called places, transitions and arcs of the net, respectively. For each $x \in P \cup T$ the set $\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$ is called the *preset* of x and the set $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$ is called the *postset* of x .

A *marking* of the Petri net N is a function $M : P \rightarrow \mathbb{N}$, that assigns a natural number to each place p such that there are only finitely many places which are assigned a number not equal to zero. We visualise a marking by black tokens; each place carries the number of tokens indicated by the marking. To each transition t , we assign two markings t^- and t^+ , which are defined by

$$t^-(p) = \begin{cases} 1 & \text{for } p \in \bullet t \\ 0 & \text{otherwise} \end{cases} \quad t^+(p) = \begin{cases} 1 & \text{for } p \in t^\bullet \\ 0 & \text{otherwise} \end{cases}$$

We define *addition of markings* pointwise, i.e. for two markings M_1 and M_2 of the net N , the sum $M_1 + M_2 : P \rightarrow \mathbb{N}$ is defined by $(M_1 + M_2)(p) = M_1(p) + M_2(p)$ for all $p \in P$.

A marking is changed by firing a transition. A transition t is *enabled* in a marking M if each place in the preset of t has at least one token, i.e. $M(p) \geq 1$ for each $p \in \bullet t$. An enabled transition may fire. This leads to a marking M' , which is given by $M' + t^- = M + t^+$. We denote this by $M \xrightarrow{t} M'$ and call it a *step*. A transition is *disabled* in M if it is not enabled. Two transitions t_1 and t_2 are in *conflict* in M if both transitions are enabled in M but their presets are not disjoint, i.e. $\bullet t_1 \cap \bullet t_2 \neq \emptyset$.

To describe executions of a Petri net, we need an initial state to start with. Let M_0 be a marking of the Petri net N . We call $\Sigma = (N, M_0)$ a *system* and we call M_0 the initial marking of N . As mentioned earlier, we consider concurrent runs of a system. For all systems, we assume, that an enabled transition eventually fires or a conflicting transition fires. This *progress assumption* is formalised in the definition of a concurrent run.

A concurrent run of a system is modelled by a special kind of Petri net with non-branching places that is called an *occurrence net*. The places of an occurrence net are called *conditions* and the transitions are called *events*.

Definition 2. (Occurrence Net) A net $K = (B, E, \leq)$ is an occurrence net if

- (i) $|\bullet b| \leq 1$ and $|b^\bullet| \leq 1$ for each condition $b \in B$.
- (ii) The set ${}^\circ K = \{b \in B \mid \bullet b = \emptyset\}$ is non-empty and finite.
- (iii) The relation $\leq$ is acyclic and the reflexive-transitive closure of $\leq$ is a partial order, which we denote by $\leq$.
- (iv) Each element of $B \cup E$ has only finitely many predecessors w.r.t. $\leq$.

The next definition provides some important notions for reasoning about an occurrence net. A cut is a maximal but finite set of independent conditions w.r.t. the flow relation $\leq$ of the net. Furthermore, independent events are called concurrent.

Definition 3. (Concurrent Set, Cut, Concurrent Events) *A set $C \subseteq B$ is a concurrent set of the occurrence net $K = (B, E, \leq)$ if for each two $b_1, b_2 \in C$, we have $b_1 \not\leq b_2$ or $b_1 = b_2$. C is a cut if it is a maximal finite concurrent set.*

Two events $e_1, e_2 \in E$ are concurrent if $e_1 \not\leq e_2$ and $e_1 \not\geq e_2$. This is denoted by $e_1 \text{ co } e_2$.

The set ${}^\circ K = \{b \in B \mid \bullet b = \emptyset\}$ is a cut. We call it the *initial cut* of K . The set $K^\circ = \{b \in B \mid b^\bullet = \emptyset\}$ is a concurrent set. K° is a cut if and only if K is finite. For each event $e \in E$, the sets $\bullet e$ and $e^\bullet$ are concurrent sets. For two cuts C_1 and C_2 of K we define $C_1 \leq C_2$ if and only if for all $b_1 \in C_1$ and $b_2 \in C_2$ holds $b_2 \not\leq b_1$.

If we want to model a concurrent run of a system, we need (besides an occurrence net) a mapping which identifies each event of the occurrence net with the occurrence of a transition of the system and each condition with a place. This is usually done by a Σ -inscription:

Definition 4. (Σ -Inscription) *Let $\Sigma = ((P, T, F), M_0)$ be a system and $K = (B, E, \leq)$ an occurrence net. The mapping $r : B \cup E \rightarrow P \cup T$ is a Σ -inscription of K if r maps conditions to places and events to transitions.*

By a Σ -inscription, each cut C of K can be associated with a marking of Σ . We denote this marking by $r(C)$ and define $r(C) : P \rightarrow \mathbb{N}$ with $r(C)(p) = |\{b \in C \mid r(b) = p\}|$.

Definition 5. (Process) *Let $\Sigma = ((P, T, F), M_0)$ be a system. A pair $\rho = (K, r)$ is a process of Σ if $K = (B, E, \leq)$ is an occurrence net and $r : B \cup E \rightarrow P \cup T$ is a Σ -inscription such that*

- (i) $r({}^\circ K) = M_0$
- (ii) *for each event $e \in E$ there is a transition $t \in T$ such that*

$$r(e) = t, \quad r(\bullet e) = t^- \quad \text{and} \quad r(e^\bullet) = t^+.$$

Condition (i) requires that the initial cut corresponds to the initial marking of the Petri net. In condition (ii), it is formalised that each event e is associated with the firing of a transition t of the Petri net, such that the pre- and postsets of e and t match. In this case, e is called an *occurrence* of transition t in the process. Concurrent runs are the maximal processes. We define this by the progress assumption that every enabled transition will eventually fire or it will be disabled because a conflicting transition fires or with other words: no transition of Σ is enabled in K° .

Definition 6. (Concurrent Run) Let $\Sigma = ((P, T, F), M_0)$ be a system. A pair $\rho = (K, r)$ is a concurrent run (or short: a run) of Σ if ρ is a process of Σ and for all transitions t in T holds

$$\bullet t \not\subseteq \{r(b) \mid b \in K^\circ\}$$

We denote the set of all runs of Σ by $\mathcal{R}(\Sigma)$.

A marking M is a *reachable marking* of Σ if there is a concurrent run which contains a cut C such that $r(C) = M$.

Definition 7. (Safe System) A system is *safe*, if no place ever contains more than one token; i.e. for each reachable marking M and for each place $p \in P$ holds: $M(p) \leq 1$.

3 Elementary Transition Refinement

In this section, we will define transition refinement for elementary systems. We use the notion *elementary transition refinement* to distinguish it from simultaneous transition refinement and transition refinement in algebraic systems. We skip the adjective *elementary* if it is clear from the context that a single transition is refined in an elementary system.

As an example, we consider the system in Fig. 2. This system consists of two agents a and b which are initially *quiet*. Both agents can synchronously change their states from *quiet* to *busy* (Transition t). A busy agent can change its state back to *quiet* without synchronising with the other agent (Transitions t_1 and t_2).

Transition t in system Σ is a synchronous action of the two agents. We want to replace that transition by asynchronous message passing between the agents. An example for a *replacement* that achieves this is shown in Fig. 3.

A replacement for a transition is a net which has the same input/output behaviour as the transition. Such a net includes the pre- and postset of the transition that shall be replaced. We consider the system, which consists of this net and the marking that assigns one token to each place of the preset of t and

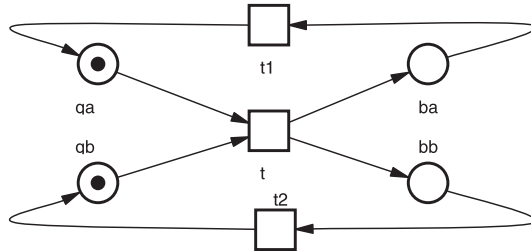


Fig. 2. System Σ with transition t

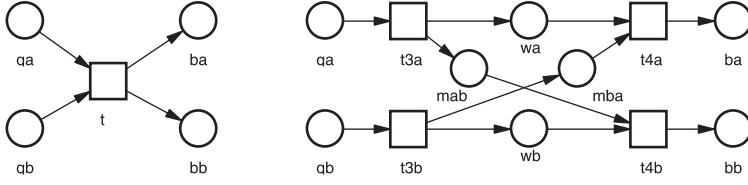


Fig. 3. Transition t and a replacement N_E for t

leaves all other places unmarked. We require, that each run of this system is finite and ends in a state where only the places of the postset of the transition are marked with one token.

In the replacement N_E , the change from **quiet** to **busy** is more complex and not longer a single synchronous action of both agents. A **quiet** agent sends a message to the other agent and changes its state to **waiting**. When a **waiting** agent receives a message, it changes its state from **waiting** to **busy**. Firing of transition t changes the marking from **quiet.a** and **quiet.b** to **busy.a** and **busy.b**.

Technically, the transition t and the replacement N_E have the same input output behaviour: If N_E starts with only the places **quiet.a** and **quiet.b** marked with one token it ends in a state where only **busy.a** and **busy.b** are marked.

Definition 8. (Replacement) Let $N = (P, T, F)$ be a net and let $t \in T$ be a transition of the net. The net $N_E = (P_E, T_E, F_E)$ is a replacement for t , if

- (i) $P \cap P_E = \bullet t \cup t^\bullet$,
- (ii) $T \cap T_E = \emptyset$ and
- (iii) each run $\rho = (K, r)$ of (N_E, t^-) is finite² and ends with $r(K^\circ) = t^+$.

Note, that we chose a canonical initial marking for the replacement to formalise, that a transition and its replacement have the same input/output behaviour: all places of the replacement, which belong to the preset of t carry one token, all other places are empty. By this definition, we do not allow new places of the replacement to have an initial marking other than zero. We discuss a weakening of this requirement in [Peu01b].

It strongly depends on the environment, if this replacement is also a refinement, i.e. preserves the important properties of the system. We will formally define, what we mean by this, and we will see that in our example N_E is indeed an transition refinement of t in the system Σ . Before we do this, we have to formalise how a transition technically is replaced by a replacement, which in fact is obvious.

Definition 9. (Extended Net) Let $N = (P, T, F)$ be a net, let t be a transition of the net and let $N_E = (P_E, T_E, F_E)$ be a replacement for t . The extended

² Strictly speaking, t^- is a marking of N , not of N_E , but of course we mean $t^-(p) = 1$ for $p \in \bullet t$ in N and $t^-(p) = 0$ for all other $p \in P_E$. The same holds for t^+ in this definition.

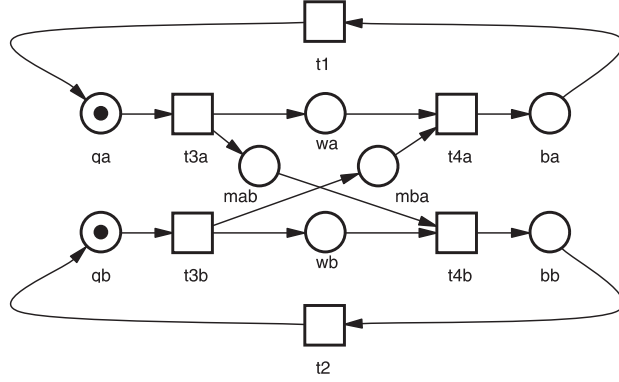


Fig. 4. The extended system $\Sigma[t \rightarrow N_E]$

net $N[t \rightarrow N_E]$ is the Petri net (P', T', F') defined by $P' = P \cup P_E$, $T' = (T \setminus \{t\}) \cup T_E$ and $F' = (F \cup F_E) \cap (P' \times T' \cup T' \times P')$.

For a system $\Sigma = (N, M_0)$ we denote by $\Sigma[t \rightarrow N_E]$ the extended system $(N[t \rightarrow N_E], M'_0)$ with $M'_0(p) = M_0(p)$ for $p \in P$ and $M'_0(p) = 0$ for all other p .

We refer to the system Σ without transition t as the *environment* of t or the *environment* of N_E , depending on the context.

Before we can explain when a replacement is a refinement, we need another notion: Let ρ be a concurrent run of Σ . Similar to the system itself, we replace each occurrence of the transition t in ρ by a concurrent run of N_E (see Fig. 5). We call the result a $[t \rightarrow N_E]$ -*expansion* of ρ . A replacement N_E of a transition t in Σ is a transition refinement if each $[t \rightarrow N_E]$ -expansion of a run of Σ is a run of $\Sigma[t \rightarrow N_E]$ and vice versa, if each run of $\Sigma[t \rightarrow N_E]$ may be obtained by expanding a run of Σ as described.

Although, it is intuitively easy to replace the occurrences of a transition t by runs of the replacement (N_E, t^-) , the definition is lengthy because of technicalities. The definition does not involve a trick or profound insight. The difficulty of the formal definition against the informal definition is the careful denotation of conditions and events such that the runs of the replacement can be easily inserted into the whole run.

Definition 10. ($[t \rightarrow N_E]$ -Expansion) Let $\Sigma = ((P, T, F), M_0)$ be a system, t be a transition of Σ , $N_E = (P_E, T_E, F_E)$ be a replacement for t and let $\rho = ((B, E, \prec), r)$ be a run of Σ . Let furthermore

- (i) $E(t) = \{a \in E \mid r(a) = t\}$ be the set occurrences of t in ρ ,
- (ii) for each $a \in E(t)$ the pair (K_a, r_a) be a run of (N_E, t^-) such that ${}^\circ K_a = {}^\bullet a$ and $K_a^\circ = a^\bullet$ and for all $b \in {}^\bullet a \cup a^\bullet$ holds $r(b) = r_a(b)$,
- (iii) the sets $E_a \cup B_a \setminus ({}^\circ K_a \cup K_a^\circ)$ be pairwise disjoint for different a and additionally pairwise disjoint with $E \cup B$

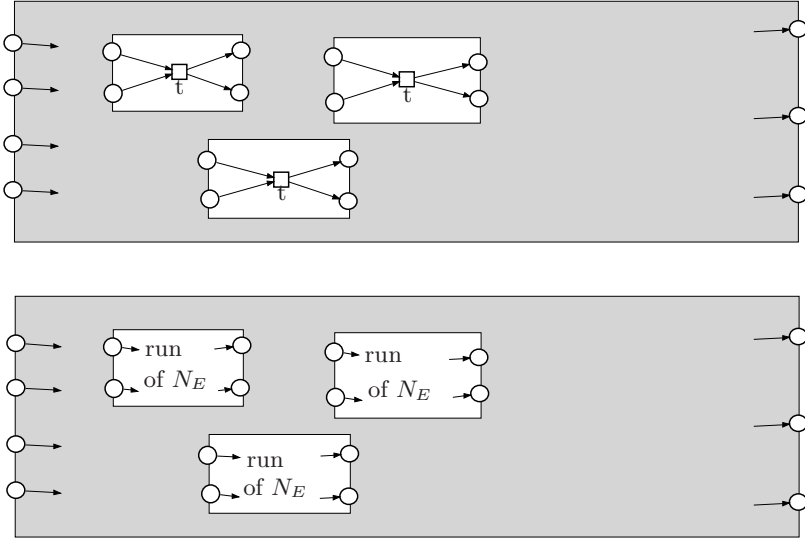


Fig. 5. A run and an expansion of the run

(iv) $(B', E', \leq')$ be the occurrence net defined by

$$\begin{aligned}
 B' &= B \cup \bigcup \{B_a \mid a \in E(t)\}, \\
 E' &= (E \setminus E(t)) \cup \bigcup \{E_a \mid a \in E(t)\}, \\
 \leq' &= (\leq \setminus (E(t) \times B \cup B \times E(t))) \cup \bigcup \{\leq_a \mid a \in E(t)\},
 \end{aligned}$$

(v) r' be the $\Sigma[t \rightarrow N_E]$ -inscription defined by

$$r'(x) = \begin{cases} r(x) & \text{for } x \in B \cup E \setminus E(t), \\ r_a(x) & \text{for } x \in B_a \cup E_a. \end{cases}$$

Then we call $\rho' = ((B', E', \leq'), r')$ a $[t \rightarrow N_E]$ -expansion of ρ .

Remark 1. By this definition, a $[t \rightarrow N_E]$ -expansion of a run ρ of Σ is an occurrence net with a $\Sigma[t \rightarrow N_E]$ -inscription. In [Peu01b], we show, that it is even a process of $\Sigma[t \rightarrow N_E]$ but it does not need to be a run of $\Sigma[t \rightarrow N_E]$. There may only actions of N_E be enabled in the end of the process.

Definition 11. (Elementary Transition Refinement) Let Σ be a system and let t be a transition of Σ . N_E is an elementary transition refinement of the transition t in the system Σ , if N_E is a replacement and the set of all runs of $\Sigma[t \rightarrow N_E]$ equals the set of all $[t \rightarrow N_E]$ -expansions of runs of Σ .

We can visualise the definition with the following commuting diagram that says: We start at a system Σ . No matter if we first replace the transition t and then construct the runs of the extended system, or first construct the runs of Σ and then replace each occurrence of t in a run by a run of the replacement, we always obtain the set $\mathcal{R}(\Sigma[t \rightarrow N_E])$ of the concurrent runs of the extended system.

$$\begin{array}{ccc}
 \Sigma & \xrightarrow{\text{form runs}} & \mathcal{R}(\Sigma) \\
 \downarrow \text{replace } t & & \downarrow \text{replace } t \\
 \Sigma[t \rightarrow N_E] & \xrightarrow{\text{form runs}} & \mathcal{R}(\Sigma[t \rightarrow N_E])
 \end{array}$$

4 Proof Criteria for Elementary Transition Refinement

Because of the semantic nature of the definition, it is often quite difficult to prove that a replacement is a transition refinement. To make this proof easier, we gave some sufficient proof criteria in [Peu01a]. First, we gave an equivalent formulation of transition refinement which is still semantic. Furthermore, we gave a more syntactic criterion that is applicable for a large class of algorithms.

For an exemplary use of proof criteria, we give now a simple criterion that is not covered in [Peu01a]. It is only applicable for a restricted class of algorithms.

Theorem 1. *Let Σ be a system, t be a transition of Σ and N_E a replacement for t . We assume that both, Σ , and (N_E, t^-) are safe and have only non-branching places (i.e. for each place p holds $\bullet p \leq 1$ and $p \bullet \leq 1$). Furthermore, we assume that all tokens on places in the preset of t are eventually used in each run of Σ (i.e. no place in the preset of t is persistent). If $\Sigma[t \rightarrow N_E]$ is safe as well then N_E is a transition refinement of t in Σ .*

Proof. Each safe net with only non-branching places has (up to isomorphisms of the occurrence nets) only one concurrent run. Let ρ be the run of Σ . The run ρ has only one $[t \rightarrow N_E]$ -expansion ρ' because (N_E, t^-) has only one run either. By remark 1, this expansion is a process of $\Sigma[t \rightarrow N_E]$ and in its end at most transitions of N_E are enabled. By assumption there is no place of the preset of t marked in the end of ρ' . Hence, ρ' is even a run of $\Sigma[t \rightarrow N_E]$. The run ρ' is the only run of $\Sigma[t \rightarrow N_E]$ because this system is also safe and has (by construction) only non-branching places. Hence, N_E is a transition refinement. $\square$

We consider again the example from the last section. First, we note that Σ and N_E have only non-branching places (by just looking at the net structure). Furthermore, we note that by the progress assumption the tokens on `quiet.a` and `quiet.b` are always consumed, hence, no place of the preset of t is persistent in Σ . It is easy to see that Σ , and (N_E, t^-) are safe. We now have to show that $\Sigma[t \rightarrow N_E]$ is safe. This is usually done by S-invariants. An S-invariant

is an especially easy invariant that assigns a value (a natural number) to each marking, and this value does not change when firing a transition (see eg [KV01]). S-Invariants may be checked syntactically. In the extended system $\Sigma[t \rightarrow N_E]$ we have the S-invariants:

$$\begin{aligned}\square \text{quiet.a} + \text{waiting.a} + \text{busy.a} &= 1 \\ \square \text{quiet.b} + \text{waiting.b} + \text{busy.b} &= 1 \\ \square \text{quiet.a} + \text{message.ab} + \text{busy.b} &= 1 \\ \square \text{quiet.b} + \text{message.ba} + \text{busy.a} &= 1\end{aligned}$$

The first invariant means that in the places `quiet.a`, `waiting.a`, `busy.a` together is always exactly one token. From all invariants together we can deduce that in each place there is never more than one token, hence, the system is safe. By Theorem 1, we proved that N_E is a transition refinement of t in Σ .

5 Simultaneous Transition Refinement

In this section, we define the simultaneous refinement of a set of transitions. Simultaneous transition refinement is the basis for transition refinement in Algebraic Petri nets.

Replacing two transitions simultaneously is not the same as replacing these two transitions one after another. We explain this with the example in Fig. 6. After replacing t_1 by N_{t_1} in Σ it is not possible anymore to replace t_2 by N_{t_2} . This is due to the restrictions for a replacement. We required in the definition, that the only places that the replacement has in common with the system is the preset and postset of the replaced transition. After the replacement of t_1 by N_{t_1} , the place H belongs to the system and it is neither in the preset nor in the postset of t_2 in this system. Hence, we can not replace t_2 by N_{t_2} in the system $\Sigma[t_1 \rightarrow N_{t_1}]$. When we replace t_1 and t_2 simultaneously, we allow that their two replacements have common places or transitions. We call the set $\mathcal{N}_E$ a *set of replacements* for the set of transitions T' if $\mathcal{N}_E = \{N_t \mid t \in T'\}$ and for each $t \in T$ the net $N_t = (P_t, T_t, F_t)$ is a replacement for t .

Definition 12 (Extended Net). Let $N = (P, T, F)$ be a net, let $T' \subseteq T$ be a set of transitions of the net and let $\mathcal{N}_E = \{N_t \mid t \in T'\}$ be a set of replacements for T' . The extended net $N[T' \rightarrow \mathcal{N}_E]$ is the net $(\hat{P}, \hat{T}, \hat{F})$ which is given by $\hat{P} = P \cup \bigcup_{t \in T'} P_t$, $\hat{T} = (T \setminus T') \cup \bigcup_{t \in T'} T_t$ and $\hat{F} = (F \cup \bigcup_{t \in T'} F_t) \cap (\hat{P} \times \hat{T} \times \hat{P})$. For a system $\Sigma = (N, M_0)$, we denote the extended system by $\Sigma[T' \rightarrow \mathcal{N}_E]$ with $(N[T' \rightarrow \mathcal{N}_E], M'_0)$ and $M'_0(p) = M_0(p)$ for $p \in P$ and $M'_0(p) = 0$ otherwise.

The replacements for the transition are not necessarily disjoint, they may have common places or transitions as in the example in Fig. 6. Both replacements contain the place H .

It is now canonical how to expand a run by inserting runs of the different replacements:

Definition 13 (Simultaneous Expansion of a Run). Let Σ be a system, let $T' \subseteq T$ be a set of transitions of Σ and let $\mathcal{N}_E = \{N_t \mid t \in T'\}$ be a set of replacements for the transitions of T' . Furthermore, let ρ be a run of Σ . We call $\rho' = ((B', E', <'), r')$ a $[T' \rightarrow \mathcal{N}_E]$ -expansion of ρ , if we get ρ' from ρ by replacing each occurrence of each transition $t \in T'$ by a run of the replacement (N_t, t^-) (as described in Def. 10).

Definition 14 (Simultaneous Transition Refinement). Let Σ be a system and let $T' \subseteq T$ be a set of transitions of Σ . The set $\mathcal{N}_E = \{N_t \mid t \in T'\}$ is a simultaneous transition refinement of the transitions of T' in the system Σ if for each $t \in T'$ the net N_t is a replacement for t , and if the set of all runs of $\Sigma[T' \rightarrow \mathcal{N}_E]$ equals the set of all $[T' \rightarrow \mathcal{N}_E]$ -expansions of runs of Σ .

In the example in Fig. 6, N_{t1} is a transition refinement for $t1$ and N_{t2} is a transition refinement for $t2$. Nevertheless, $\{N_{t1}, N_{t2}\}$ is not a simultaneous transition refinement because the run of the extended system which consists of $t3$ and $t6$ in a row is no expansion of a run of Σ .

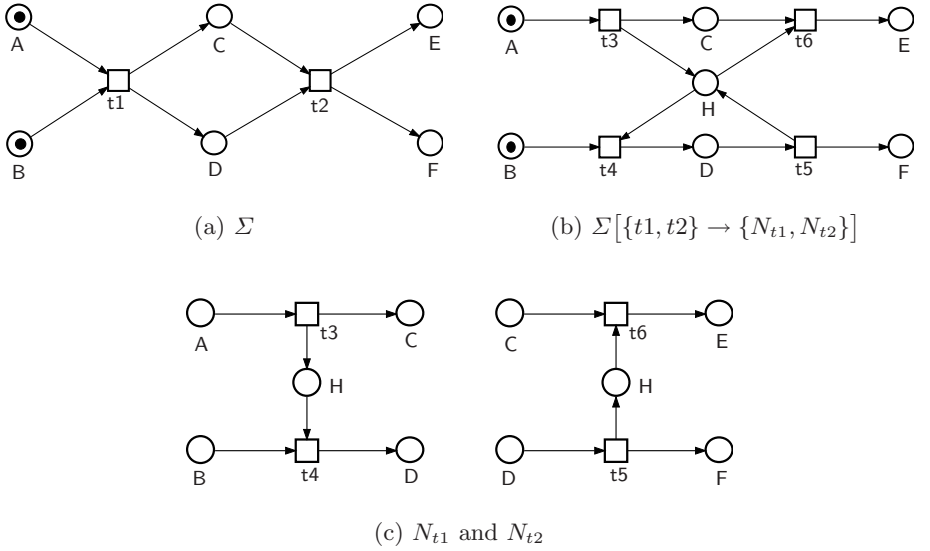


Fig. 6. A system, the extended system, and the replacements

6 Proof Criteria for Simultaneous Transition Refinement

In general, it is more difficult to find proof criteria for simultaneous transition refinement than for elementary transition refinement. Some of the proof criteria of [Peu01a] cannot be generalised to the simultaneous case.

We will now show that, however, we may use the criteria for elementary transition refinement if the replacement for the set of transitions consists of disjoint replacements for each transition. Two Petri nets $N_1 = (P_1, T_1, F_1)$ and $N_2 = (P_2, T_2, F_2)$ are disjoint if $(P_1 \cup T_1) \cap (P_2 \cup T_2) = \emptyset$.

Assume that we have disjoint replacements for t_1 and t_2 in the system Σ . First, we note that the simultaneous replacement of t_1 and t_2 yields the same result as the replacement of one transition after the other in either order. The following diagram commutes in this case:

$$\begin{array}{ccc}
 \Sigma & \xrightarrow{\text{replace } t_1} & \Sigma[t_1 \rightarrow N_1] \\
 \downarrow \text{replace } t_2 & & \downarrow \text{replace } t_2 \\
 \Sigma[t_2 \rightarrow N_2] & \xrightarrow{\text{replace } t_1} & \Sigma[\{t_1, t_2\} \rightarrow \{N_1, N_2\}]
 \end{array}$$

Simultaneous transition refinement with disjoint replacements does not seem too interesting at first sight. By looking more closely, it becomes interesting if we consider simultaneous transition refinement to be a preparation for algebraic transition refinement. By dealing with case studies, we observed that in algebraic Petri nets the unfolding of a transition into different modes often consists of disjoint nets. If such a transition is replaced, the replacement often (but not necessarily) consists of disjoint subnets (see also example in the next section).

Theorem 2. *Let Σ be a system, T' be a set of transitions of Σ and $\mathcal{N}_E = \{N_t \mid t \in T'\}$ be a set of replacements for T' . If each net N_t is a transition refinement for the transition t and if the nets N_t are pairwise disjoint then $\mathcal{N}_E$ is a simultaneous transition refinement of T' in Σ .*

Proof. This theorem can be deduced directly from the Definitions 11, and 14. The runs of the replacements cannot interfere with each other because they have no common places. $\square$

Not every transition of an algebraic Petri net unfolds into disjoint subnets for each mode. Usually, we can partition the set of modes into equivalence classes such that the replacement unfolds into disjoint subnets such that there is one subnet for each equivalence class. In this case, we need the following generalisation of Theorem 2:

Theorem 3. *Let I be an index set and let $(T_i)_{i \in I}$ be a family of pairwise disjoint subsets of transitions of Σ . Let further the set $\mathcal{N}_i = \{N_t \mid t \in T_i\}$ be a simultaneous transition refinement of T_i in Σ for each $i \in I$. If each two replacements $N_t, N_{t'}$ from different $\mathcal{N}_i$ (i.e. t and t' are from different T_i) are disjoint then $\bigcup_{i \in I} \mathcal{N}_i$ is a simultaneous transition refinement of $\bigcup_{i \in I} T_i$ in Σ .*

Proof. This theorem, as well as Theorem 2, is deduced directly from the Definitions 11, and 14. $\square$

7 Algebraic Systems – Basic Definitions

In this section, we formally define Algebraic Petri nets and their concurrent runs. Algebraic Petri nets were introduced in [Rei91]. They are related to Coloured Petri nets of [Jen92]. We use the notions and notations of [KR96, WWV⁺97, Rei98].

An *algebra* $\mathcal{A} = (U, Op)$ consists of a universe U and a set Op of operations of U . Each operation $f \in Op$ has a functionality $f : U_1 \times U_2 \times \dots \times U_n \rightarrow U_{n+1}$ such that each $U_i \subseteq U$.

For each set U , we denote the powerset (the set of all subsets of U) by 2^U . By 2_e^U , we denote the set of all *finite* subsets of U . A subset U' of the universe is a *sort*. A *sorted set of variables* for an algebra $\mathcal{A} = (U, Op)$ is a pair $\mathcal{X} = (X, dom)$ of a set of variables X and a mapping $dom : X \rightarrow 2^U$ that assigns to $x \in X$ a sort $dom(x)$.

The set $\mathcal{T}(\mathcal{A}, \mathcal{X}, U')$ of *terms* of the sort U' over the algebra $\mathcal{A}$ and the sorted set of variables $\mathcal{X}$ is defined inductively as follows:

- (i) $x \in \mathcal{T}(\mathcal{A}, \mathcal{X}, U')$ if $x \in X$ and $dom(x) \subseteq U'$.
- (ii) $f(u_1, \dots, u_n) \in \mathcal{T}(\mathcal{A}, \mathcal{X}, U')$ if $f : U_1 \times U_2 \times \dots \times U_n \rightarrow U_{n+1}$ and $u_i \in \mathcal{T}(\mathcal{A}, \mathcal{X}, U_i)$ for $i = 1, \dots, n$ and $U_{n+1} \subseteq U'$.

In this definition, we use f as the symbol that denotes the operation f . For better readability will not distinguish between an operation and the corresponding operation symbol. We denote the set of all terms of an algebra over $\mathcal{X}$ by $\mathcal{T}(\mathcal{A}, \mathcal{X})$.

An *assignment* for the sorted set of variables $\mathcal{X} = (X, dom)$ of the algebra $\mathcal{A}$ is a mapping $\beta : X \rightarrow U$ which assigns a value to each variable with respect to the sort of the variable, i.e. $\beta(x) \in dom(x)$ for all $x \in X$. The evaluation of a term with respect to an assignment β is as usual defined inductively. This canonical extension of an assignment to the set of all terms is also denoted by $\beta : \mathcal{T}(\mathcal{A}, \mathcal{X}) \rightarrow U$.

A *multiset* m over the set A is a mapping $m : A \rightarrow \mathbb{N}$ that assigns to each $x \in A$ a natural number $m[x]$. This number states the *multiplicity* of x in m . A multiset is finite if only finitely many elements have a multiplicity other than zero. We denote the set of all finite multisets over A by $\mathbb{N}_e^A$.

Definition 15 (Algebraic Petri Net). An Algebraic Petri net $N = (\tilde{N}, \mathcal{A}, \mathcal{X}, d, w, g)$ is given by

- (i) a net $\tilde{N} = (P, T, F)$,
- (ii) an algebra $\mathcal{A} = (U, Op)$ that contains the sort $\mathbb{B} = \{\text{true}, \text{false}\}$,
- (iii) a sorted set of variables $\mathcal{X} = (X, dom)$ for $\mathcal{A}$,
- (iv) a mapping $d : P \rightarrow 2^U$ that assigns to each p a sort $d(p)$ such that $\mathbb{N}_e^{d(p)} \subseteq U$ (i.e. U contains all finite multisets over $d(p)$),

- (v) a mapping $w : F \rightarrow \mathcal{T}(\mathcal{A}, \mathcal{X}, U)$ that assigns to each arc f a term $w(f)$ such that $\text{dom}(w(f)) \subseteq 2_e^{d(p)}$ if $f = (p, t) \in P \times T$ or $f = (t, p) \in T \times P$ (i.e. the sort of $w(f)$ consists of the finite subsets of the sort of the place p belonging to f .)
- (vi) a mapping $g : T \rightarrow \mathcal{T}(\mathcal{A}, \mathcal{X}, \mathbb{B})$ assigns to each transition a transition guard (a term of the sort $\{\text{true}, \text{false}\}$).

Each elementary Petri net may be considered as an algebraic Petri net with the algebra only consisting of the multisets over $\{\bullet\}$, with all arc weights equal to $\{\bullet\}$, and with all transition guards equal to true.

The elements of the sorts of places are called *tokens*. A *marking* of the algebraic net N is a mapping $M : P \rightarrow \mathbb{N}_e^U$ that assigns to each $p \in P$ a finite multiset of tokens from $d(p)$. A marking describes a global state of the net N . An algebraic Petri net N together with a marking M_0 is an *algebraic system* $\Sigma = (N, M_0)$ (or short: a system). M_0 is called the *initial marking* of Σ .

A marking is changed by firing of a transition. The change depends on the transition t and the assignment β of variables of $\mathcal{X}$. We call β a *mode* of the transition. A transition together with a mode (t, β) is an *action*.

Remark 2. The actual change of the marking does not depend on the whole assignment but only on the variables that occur at incoming or outgoing arcs of the transition. Therefore, we do not distinguish between two actions (t, β_1) and (t, β_2) if β_1 and β_2 are identical for all variables that occur at incoming or outgoing arcs of the transition. In this case (t, β_1) and (t, β_2) denote the same action.

For each action (t, β) , we define two markings $(t, \beta)^-$ and $(t, \beta)^+$ that describe the tokens consumed and produced by the action (similar to the elementary case):

$$\begin{aligned} (t, \beta)^-(p) &= \beta(w(p, t)) \\ (t, \beta)^+(p) &= \beta(w(t, p)) \quad \text{for all } p \in P \end{aligned}$$

An action (t, β) is *enabled* in the marking M if $(t, \beta)^- \leq M$ and if the transition guard holds, i.e. $\beta(g(t)) = \text{true}$. A transition t is *enabled* in the marking M if there is a mode β such that the action (t, β) is enabled in the marking M . An enabled action or transition may fire. The *firing* of the action (t, β) leads to a marking M' which is given by:

$$M' + (t, \beta)^- = M + (t, \beta)^+$$

We call this a step $M \xrightarrow{(t, \beta)} M'$.

Two different ³ actions (t_1, β_1) and (t_2, β_2) are in *conflict* in a marking M if they are both enabled but their presets are not disjoint, i.e. there is a place p and an element a of the sort of p such that $(t_1, \beta_1)^-(p)[a] > 0$ and $(t_2, \beta_2)^-(p)[a] > 0$.

³ Reminder: Two actions (t_1, β_1) and (t_2, β_2) are different if $t_1 \neq t_2$ or if there is a variable x occurring at an ingoing or outcoming arc or in the transition guard of t_1 such that $\beta_1(x) \neq \beta_2(x)$ (see Remark 2).

A marking M' is *reachable* from the marking M if there is a (possibly empty) sequence of steps $M_0 \xrightarrow{(t_1, \beta_1)} M_1 \dots M_{n-1} \xrightarrow{(t_n, \beta_n)} M_n$ such that $M = M_0$ and $M' = M_n$. A marking is reachable in the system Σ if it is reachable from the initial marking of the system.

A marking M is *safe* if there is no place that carries two identical tokens, i.e. $M(p)[x] \leq 1$ for all $p \in \mathcal{P}$ and all $x \in d(p)$. A system is safe if each reachable marking is safe. In safe systems, we may consider the multisets of tokens, that are assigned to each place in a marking, as sets.

We now define *concurrent runs* as partial order semantics for Algebraic Petri nets. Similar to the elementary case, we use labelled occurrence nets for the presentation of concurrent runs. We label each event of the occurrence net with an action of the system and each condition of the occurrence net with a token, i.e. a pair consisting of a place and an element of the sort of the place.

Definition 16 (Σ -Inscription). Let Σ be an algebraic system and $K = (B, E, \leq)$ an occurrence net. A Σ -inscription of K is a mapping r that assigns to each event $e \in E$ an action $r(e) = (t, \beta)$ of Σ , and that assigns to each condition $b \in B$ a pair $r(b) = (p, a)$ such that p is a place of Σ and $a \in d(p)$ is an element of the sort of p .

Each finite co-set C of K can be matched with a marking of Σ by a Σ -inscription. We denote this marking by $r(C)$ and define $r(C) : P \rightarrow \mathbb{N}_e^U$ with $r(C)(p)[a] = |\{b \in C \mid r(b) = (p, a)\}|$.

Definition 17 (Process). Let $\Sigma = (N, M_0)$ an algebraic system. A pair $\rho = (K, r)$ is a process of Σ if $K = (B, E, \leq)$ is an occurrence net and r is a Σ -inscription such that

- (i) $r(\circ K) = M_0$ and
- (ii) for each event $e \in E$ and each transition (t, β) holds: If $r(e) = (t, \beta)$ then

$$r(\bullet e) = (t, \beta)^- \quad \text{and} \quad r(e^\bullet) = (t, \beta)^+$$

Condition (i) ensures that the initial cut is matched with the initial marking of the system. In condition (ii), we formalise that each event e is associated with the firing of a transition t of the algebraic Petri net in a certain mode. As in the elementary case, we define a concurrent run to be a process that is maximal with respect to firing of transitions, i.e. no transition is enabled in the end of a concurrent run. In safe systems, this is equivalent to the assumption that each enabled transition eventually fires or a conflicting transition fires.

Definition 18 (Concurrent Run). Let $\Sigma = (N, M_0)$ be an algebraic system (with the set of transitions denoted by T). A pair $\rho = (K, r)$ is a concurrent run (short: run) of Σ if ρ is a process of Σ and all transitions $t \in T$ are disabled in $r(K^\circ)$, i.e. for each action (t, β) and each finite co-set $C \subseteq K^\circ$ holds $(t, \beta)^- \not\leq r(C)$ or $\beta(g(t)) = \text{false}$. We denote the set of all concurrent runs of Σ by $\mathcal{R}(\Sigma)$.

Remark 3. A marking M of Σ is reachable if and only if there is a concurrent run with a cut C such that $r(C) = M$.

We can transform each algebraic system Σ into an elementary system Σ^e such that both systems have the same concurrent runs. We call Σ^e the *unfolding* of Σ . The set of transitions of Σ^e equals the set of actions of Σ , and the set of places equals the set of all tokens, i.e. as in Def. 16 the set of all pairs (p, a) such that p is place of Σ , and $a \in d(p)$ is an element of the sort of p . The flow relation of Σ^e depends on arc inscriptions and transition guards of Σ .

We construct the unfolding as follows: we consider for each transition t of the algebraic system all possible modes. We draw a transition for each mode β in which the transition guard of t evaluates to true. We denote this transition by (t, β) . We draw a place for each place p of the algebraic system, and for each element z of the sort of p . We denote this place by $p.z$. If the action (t, β) consumes the token $p.z$ for firing in the algebraic system, we draw an arc from $p.z$ to t, β in the unfolding. If, vice versa, the token $p.z$ is produced by the action (t, β) , we draw an arc from the transition t, β to the place $p.z$. Finally, we mark the places according to the initial marking of the algebraic system.

Definition 19 (Unfolding). Let $N = ((P, T, F), \mathcal{A}, \mathcal{X}, d, w, g)$ be an algebraic Petri net. The elementary Petri net $N^e = (P^e, T^e, F^e)$ is the unfolding of N , if the following conditions are met:

- (i) $P^e = \{p.z \mid p \in P \text{ and } z \in d(p)\}$
- (ii) $T^e = \{t, \beta \mid t \in T, \beta \text{ is a mode of } \mathcal{X} \text{ with } \beta(g(t)) = \text{true}\}$ ⁴
- (iii) $((p, a), (t, \beta)) \in F^e$ if, and only if, $(t, \beta)^-(p)[a] \neq 0$ and $((t, \beta), (p, a)) \in F^e$ if, and only if, $(t, \beta)^+(p)[a] \neq 0$

Let $\Sigma = (N, M_0)$ be an algebraic system with $N = ((P, T, F), \mathcal{A}, \mathcal{X}, d, w, g)$. The elementary system $\Sigma^e = (N^e, M_0^e)$ with $N^e = (P^e, T^e, F^e)$ is the unfolding of Σ if N^e is the unfolding of N , and if the initial markings satisfy $M_0^e[(p, z)] = M_0(p)[z]$.

Theorem 4. An algebraic system Σ and its unfolding Σ^e have the same concurrent runs.

Proof. From the Definitions 19, and 16, we can deduce that each Σ -inscription of an occurrence net is a Σ^e -inscription. It is easy to check that the conditions from the Definitions 17, and 18 are satisfied for Σ if, and only if, they are satisfied for Σ^e .

Corollary 1. An algebraic system is safe if, and only if, its unfolding is safe.

⁴ Again, we use the convention of Remark 2 that we do not distinguish two actions of the same transition if the assignments to the variables are equal for all variables occurring at ingoing or outgoing arcs of the transition.

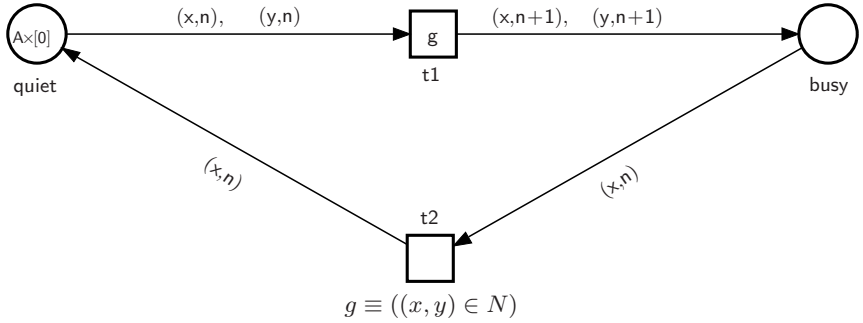


Fig. 7. The system Σ

8 Algebraic Transition Refinement

Before we introduce transition refinement for Algebraic Petri nets, we consider an example to explain the involved notions intuitively.

We consider an algorithm which is very similar to the example in Section 3. We want to use this algorithm not only for two agents but for more general networks of agents, and we also want to count the rounds of each agent. Let (A, N) be a communication network that consists of four agents $A = \{a, b, c, d\}$ and two bidirectional communication channels, one between a and b and one between c and d . Agents a and b are neighbours and c and d are neighbours. The channels are given by the neighbourhood relation $N = \{(a, b), (b, a), (c, d), (d, c)\}$: by one pair for each direction of each channel⁵. In Fig. 7, we define an algorithm for this network.

Initially, each agent is assigned the round number 0 and each agent is in the state **quiet**. Two neighbours with the same round number can synchronously change their states from **quiet** to the next round and the state **busy**. Each **busy** agent can finish its round by becoming **quiet** without changing its round number.

The unfolding of transition $t1$ consists of a transition for each mode that evaluates the transition guard of $t1$ to be true. This are four transitions for each round number k , namely for the modes $[x = a, y = b, n = k]$, $[x = c, y = d, n = k]$, $[x = b, y = a, n = k]$ and $[x = d, y = c, n = k]$. For all other assignments of variables, the transition guard g evaluates to false. Fig. 8(left) shows the unfolding of transition $t1$ for a fixed round number $n = k$ for two of the four modes: for $[x = a, y = b, n = k]$ and $[x = c, y = d, n = k]$. The unfolding for the mode $[x = b, y = a, n = k]$ coincides with the unfolding for $[x = a, y = b, n = k]$ besides the transition is inscribed $t1.ba$ instead of $t1.ab$. The right side of Fig. 8 shows an replacement for each of the two elementary Petri nets. The round number k actually occurs in each place and transition name. We omitted it because of space limitations.

⁵ For a more profound introduction to modelling network algorithms by Algebraic Petri nets see eg [Rei98, WWV⁺97]

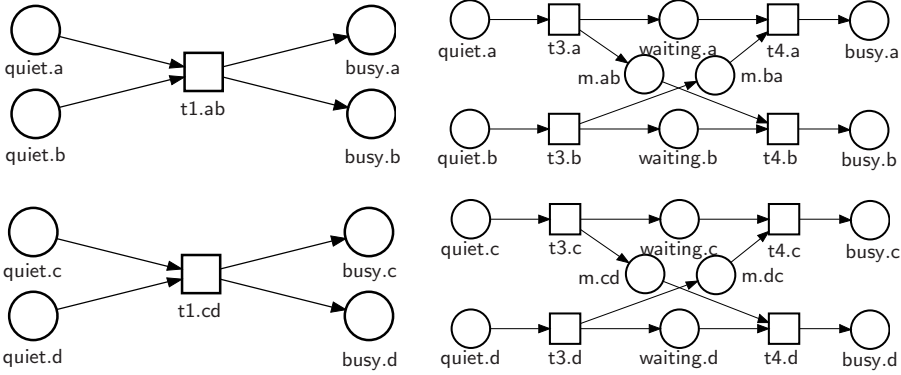


Fig. 8. The unfolding of $t1$ and replacements for the mode $[n = k]$

The replacements are disjoint for different round numbers. The Petri nets on the right are as well replacements for the unfoldings of the two omitted modes $[x = b, y = a, n = k]$, and $[x = d, y = c, n = k]$.

An algebraic replacement for transition $t1$ is an algebraic Petri net that, unfolded, equals the union of a set of replacements for the modes of the transition. We define the union of two nets (P_1, T_1, F_1) and (P_2, T_2, F_2) as $N_1 \cup N_2 = (P_1 \cup P_2, T_1 \cup T_2, F_1 \cup F_2)$. Furthermore, we require that the sorts of the places of the preset and postset of $t1$ coincide in the algebraic replacement and in Σ .

Fig. 9(a) shows again transition $t1$, and Fig. 9(b) shows an algebraic replacement N_E for $t1$. The unfolding of N_E is the union of the two replacements for each mode $[n = k]$ of Fig. 8.

We now consider a transition t of an arbitrary system Σ . If we replace the algebraic transition t in Σ , we simultaneously for each β replace all transitions (t, β) in the unfolding Σ^e of Σ . Let $\mathcal{N}_E = \{N_{(t, \beta)} \mid (t, \beta) \in T^e\}$ be a set of replacements for all transitions (t, β) in the unfolding Σ^e . The algebraic Petri net N_E is an *algebraic replacement* for t in Σ if the unfolding of N_E is the union $\bigcup_{(t, \beta) \in T^e} N_{(t, \beta)}$ of replacements of actions of t in the unfolding. Furthermore, there are some syntactic requirements for connecting N_E and Σ .

Definition 20 (Algebraic Replacement). Let $N = ((P, T, F), \mathcal{A}, \mathcal{X}, d, w, g)$ be an algebraic Petri net and t a transition of N . Furthermore, let N^e be the unfolding of N . The algebraic Petri net $N_E = ((P_E, T_E, F_E), \mathcal{A}_E, \mathcal{X}_E, d_E, w_E, g_E)$ is a algebraic replacement for t in N if there is a set $\mathcal{N}_E = \{N_{(t, \beta)} \mid (t, \beta) \in T^e\}$ of replacements for all transitions (t, β) in the unfolding N^e such that the unfolding of N_E is exactly the union $\bigcup_{(t, \beta) \in T^e} N_{(t, \beta)}$ of the replacements of the actions of t in the unfolding, and additionally we have:

- (i) $P \cap P_E = \bullet t \cup t \bullet$ and $T \cap T_E = \emptyset$,
- (ii) $\mathcal{A}_E = \mathcal{A}$ and $\mathcal{X}_E = \mathcal{X}$,

(iii) for all $p \in \bullet t \cup t^\bullet$ holds $d(p) = d_E(p)$,

In Algebraic Petri nets, it is more difficult to decide if a Petri net is a replacement for a transition than in elementary Petri nets. The following lemma states a sufficient criterion for an algebraic Petri net to be a replacement for a transition.

Lemma 1. *Let $N = ((P, T, F), \mathcal{A}, \mathcal{X}, d, w, g)$ be an Algebraic Petri net and t a transition of N . The algebraic Petri net $N_E = ((P_E, T_E, F_E), \mathcal{A}_E, \mathcal{X}_E, d_E, w_E, g_E)$ is an algebraic replacement for t in N if*

- (i) $P \cap P_E = \bullet t \cup t^\bullet$ and $T \cap T_E = \emptyset$,
- (ii) $\mathcal{A}_E = \mathcal{A}$ and $\mathcal{X}_E = \mathcal{X}$,
- (iii) for all $p \in \bullet t \cup t^\bullet$ holds $d(p) = d_E(p)$ and
- (iv) for each mode β each run (K, r) of $(N_E, (t, \beta)^-)$ is finite and $r(K^\circ) = (t, \beta)^+$.

Proof. We have to show that the unfolding of N_E is the union of a family of replacements for the unfolded modes of the transition. According to Theorem 4, the unfolding N_E^e of N_E is a replacement for each transition (t, β) of the the unfolding Σ^e of Σ . We choose for each mode β the net $N_{(t, \beta)} = N_E^e$ as a replacement. The union of these replacements is N_E^e itself. $\square$

We mark N_E with the preset of $t1$ for the assignment $[x = a, y = b, n = 0]$, i.e. with the tokens $(a, 0), (b, 0)$ in the place *quiet*. N_E has only one run with this initial marking. This run ends with the tokens $(a, 1), (b, 1)$ in the place *busy*.

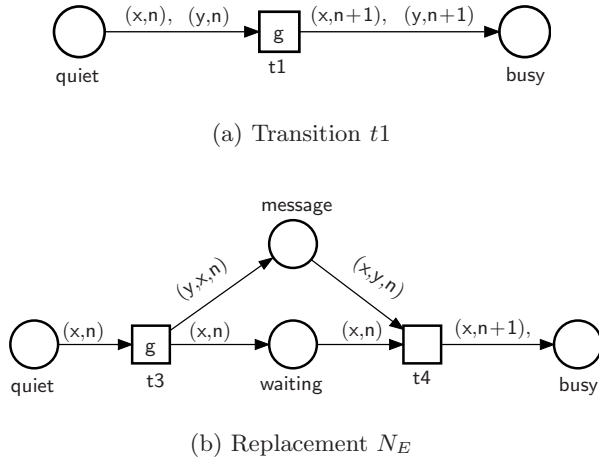


Fig. 9. Transition guard $g \equiv ((x, y) \in N)$

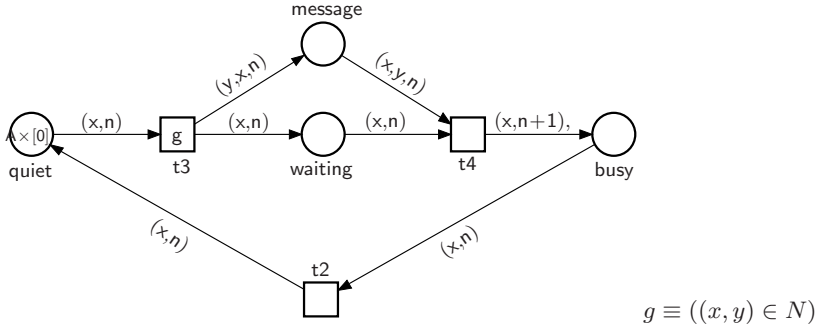


Fig. 10. The system $\Sigma[t1 \rightarrow N_E]$

This is the very postset of the transition $t1$ in the mode $[x = a, y = b, n = 0]$. We have to use this kind of reasoning for each mode to prove that N_E is an algebraic replacement of $t1$. Usually this is easier than the proof by using the definition.

The extended system, the expansion of a run, and algebraic transition refinement can now be defined canonically.

Definition 21 (Extended System). Let $N = ((P, T, F), \mathcal{A}, \mathcal{X}, d, w, g)$ be an Algebraic Petri net, t be a transition of N and $N_E = ((P_E, T_E, F_E), \mathcal{A}, \mathcal{X}, d_E, w_E, g_E)$ be an algebraic replacement for t . The extended Petri net $N[t \rightarrow N_E]$ is given by

- (i) the net (P', T', F') with $P' = P \cup P_E$, $T' = (T \setminus \{t\}) \cup T_E$ and $F' = (F \cup F_E) \cap (P' \times T' \cup T' \times P')$.
- (ii) the algebra $\mathcal{A}$,
- (iii) the sorted set of variables $\mathcal{X}$,
- (iv) the sort function $d' = d \cup d_E$
- (v) the weight function $w' = w|_{F \cap F'} \cup w_E$
- (vi) the transition guards $g' = g|_{T \setminus \{t\}} \cup g_E$.

For a system Σ , we denote the extended system by $\Sigma[t \rightarrow N_E] = (N[t \rightarrow N_E], M'_0)$ with $M'_0(p) = M_0(p)$ for $p \in P$ and $M'_0(p) = []$ for all other p .

Fig. 10 shows the extended system for the above example.

Definition 22 (Expansion of a Run). We consider a system $\Sigma = (N, M_0)$ with $N = ((P, T, F), \mathcal{A}, \mathcal{X}, d, w, g)$, a transition t of N and a replacement $N_E = ((P_E, T_E, F_E), \mathcal{A}, \mathcal{X}, d_E, w_E, g_E)$ for t . Let $\rho = ((B, E, \leq), r)$ be a run of Σ . We define

- (i) $E(t) = \{a \in E \mid \text{there is a mode } \beta \text{ with } r(a) = (t, \beta)\}$ the set of events of ρ that are associated with the firing of transition t ,

- (ii) for each $a \in E(t)$ a run (K_a, r_a) of $(N_E, r(a)^-)$ with ${}^\circ K_a = {}^\bullet a$ and $K_a^\circ = a^\bullet$ and for all $b \in {}^\bullet a \cup a^\bullet$ holds $r(b) = r_a(b)$, and such that the sets $E_a \cup B_a \setminus ({}^\circ K_a \cup K_a^\circ)$ are pairwise disjoint for different a and, furthermore, pairwise disjoint with $E \cup B$
- (iii) the occurrence net $(B', E', \leq')$ given by

$$\begin{aligned} B' &= B \cup \bigcup \{B_a \mid a \in E(t)\}, \\ E' &= (E \setminus E(t)) \cup \bigcup \{E_a \mid a \in E(t)\}, \\ \leq' &= (\leq \setminus (E(t) \times B \cup B \times E(t))) \cup \bigcup \{\leq_a \mid a \in E(t)\}, \end{aligned}$$

- (iv) the $\Sigma[t \rightarrow N_E]$ -inscription r' given by

$$r'(x) = \begin{cases} r(x) & \text{for } x \in B \cup E \setminus E(t), \\ r_a(x) & \text{for } x \in B_a \cup E_a. \end{cases}$$

Then we call $\rho' = ((B', E', \leq'), r')$ a $[t \rightarrow N_E]$ -expansion of ρ .

Definition 23 (Algebraic Transition Refinement). Let Σ a system and t a transition of Σ . N_E is an algebraic transition refinement of the transition t in the system Σ if N_E is an algebraic replacement and the set of concurrent runs of $\Sigma[t \rightarrow N_E]$ equals the set of the $[t \rightarrow N_E]$ -expansions of concurrent runs of Σ .

We omit the adjective *algebraic* if it is clear from the context.

Algebraic transition refinement, as well as elementary transition refinement, can be generalised to the simultaneous refinement of several algebraic transitions. We do not distinguish between algebraic and simultaneous algebraic transition refinement because both are equivalent to simultaneous elementary transition refinement.

9 Proof Criteria for Algebraic Transition Refinement

We defined an algebraic replacement in a way that allows us to transpose the proof criteria for simultaneous transition refinement to algebraic transition refinement. We consider again the example from the last section. We want to show that N_E is a transition refinement of $t1$ in Σ . To use Theorem 3, we have to partition the modes of $t1$ into equivalence classes such that the unfolding of the replacement is divided into disjoint nets: one net for each equivalence class. This is easy for our example. Transition $t1$ has four modes for each round number n and each two of them have the same replacement (see Fig. 8). We have two equivalence classes for each mode. One class consists of two modes with the same replacement.

We can now separate an arbitrary equivalence class of modes of the transition $t1$. Fig. 11 shows this exemplary for the modes $[x = a, y = b, n = 5]$ and $[x = b, y = a, n = 5]$. Transition $t12$ simulates transition $t1$ in the mode $[x =$

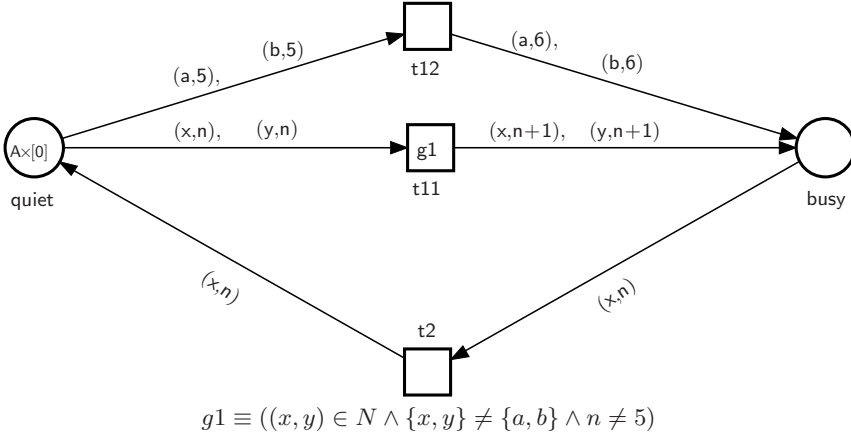


Fig. 11. The system Σ'

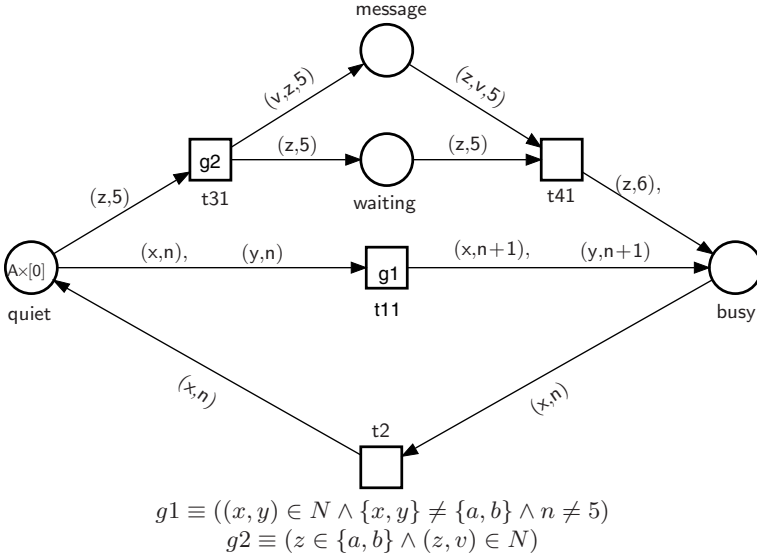


Fig. 12. The system $\Sigma'[t12 \rightarrow N'_E]$

$a, y = b, n = 5]$ and $[x = b, y = a, n = 5]$, transition $t11$ simulates transition $t1$ in all other modes. The systems Σ and Σ' have the same concurrent runs besides the labelling of transitions.

We replace transition $t12$ as shown in Fig. 12. We can deduce that the algebraic replacement N_E is an algebraic transition refinement of $t1$ in Σ with

Theorem 3 if we can show that the replacement in Fig. 12 is a transition refinement for arbitrary modes in Σ' .

This method is a bit fussy but not difficult. It has the following advantage: We do not have to show that the replacement is a transition refinement for all modes simultaneously but only for one class of modes at a time. This is in general much easier.

In our example, it is easy to show that transition t_{31} and transition t_{41} occur exactly once in the mode $[z = a]$ and once in the mode $[z = b]$ in each run of $\Sigma' [t_{12} \rightarrow N'_E]$, and that they together form a run of the replacement. We may assume that the system operates exactly as the system Σ until round 4 and we can use the properties of Σ in the proof.

10 Conclusion

We proposed a new property preserving transition refinement, which is based on partial order semantics. We used Algebraic Petri nets as a formal model. We showed that transition refinement in Algebraic Petri nets is more expressive than transition refinement in elementary Petri nets because refining a transition in an Algebraic Petri net corresponds to the refining several transitions in an elementary Petri net simultaneously.

A replacement for a transition is not universally a refinement that preserves properties of the system. A refinement is only defined in the context of an environment. We discussed, how to use results from elementary transition refinement for algebraic transition refinement.

We give a complex example of the application of algebraic transition refinement in another paper [Peu02]. There, we applied algebraic transition refinement as the major tool in a new correctness proof for the algorithm of Gallager, Humblet, and Spira (GHS-algorithm, [GHS83]). The GHS-algorithm is a distributed algorithm for computing the minimum spanning tree of a weighted graph.

In this paper, we only stated a simple syntactic proof criterion for a restricted class of algorithms. Interesting further work includes the search for syntactic proof criteria for special classes of distributed algorithms, e.g. self-stabilising algorithms.

References

- [AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, October 1985. 432
- [GHS83] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems*, 5(1):66–77, January 1983. 453
- [Jen92] Kurt Jensen. *Coloured Petri Nets*, volume 1 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1992. 443
- [KP99] Ekkart Kindler and Sibylle Peuker. Integrating distributed algorithms into distributed systems. *Fundamenta Informaticae*, 37:291–309, 1999. 432

- [KR96] Ekkart Kindler and Wolfgang Reisig. Algebraic system nets for modelling distributed algorithms. *Petri Net Newsletter*, 51:16–31, December 1996. 443
- [KV01] Ekkart Kindler and Hagen Völzer. Algebraic nets with flexible arcs. *Theoretical Computer Science*, 262(1-2):285–310, 2001. 432, 440
- [Peu01a] Sibylle Peuker. Property preserving transition refinement with concurrent runs: An example. In *International Conference on Application of Concurrency to System Design*, pages 77–86. IEEE Computer Society, 2001. 430, 431, 432, 439, 442
- [Peu01b] Sibylle Peuker. *Halbordnungsbasierte Verfeinerung zur Verifikation verteilter Algorithmen*. PhD thesis, Humboldt-Universität zu Berlin, available via <http://dochost.rz.hu-berlin.de/abstract.php3/dissertationen/peuker-sibylle-2001-07-03>, 2001. 436, 438
- [Peu02] Sibylle Peuker. Deriving a distributed minimum weight spanning tree algorithm with transition refinement for Algebraic Petri nets. In *Proceedings of Petri Nets 2002*, LNCS. Springer, 2002. To appear. 431, 453
- [PHG00] Julia Padberg, Kathrin Hoffman, and Maike Gajewski. Introduction and preservation of safety properties in algebraic high-level net systems. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, volume 1783 of LNCS, pages 249–265. Springer-Verlag, 2000. 432
- [Rei91] W. Reisig. Petri Nets and Algebraic Specifications. *Theoretical Computer Science*, 80:1–34, May 1991. 432, 443
- [Rei98] Wolfgang Reisig. *Elements of Distributed Algorithms: Modeling and Analysis with Petri Nets*. Springer, 1998. 432, 443, 447
- [Val79] R. Valette. Analysis of Petri Nets by stepwise refinement. *Journal of Computer and System Sciences*, 18:35–46, 1979. 431
- [vGG90] R. J. van Glabbeek and U. Goltz. Refinement of actions in causality based models. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness*, volume 430 of LNCS, pages 267–300. Springer-Verlag, 1990. 431
- [Vog87] Walter Vogler. Behaviour preserving refinements of Petri nets. In G. Tinhofer and G. Schmidt, editors, *Graph-Theoretic Concepts in Computer Science*, volume 246 of LNCS, pages 82–93. Springer Verlag, 1987. 431
- [Wir71] Niklaus Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971. 431
- [WWV⁺97] M. Weber, R. Walter, H. Völzer, T. Vesper, W. Reisig, S. Peuker, E. Kindler, J. Freiheit, and J. Desel. DAWN: Petrinetzmodelle zur Verifikation Verteilter Algorithmen. Informatik-Bericht 88, Humboldt-Universität zu Berlin, December 1997. 443, 447