Amy L. Murphy
Jan Vitek (Eds.)

LNCS 4467

Coordination Models and Languages

9th International Conference, COORDINATION 2007
Paphos, Cyprus, June 2007
Proceedings

Springer

Lecture Notes in Computer Science 4467

Amy L. Murphy Jan Vitek (Eds.)

Coordination Models and Languages

9th International Conference, COORDINATION 2007
Paphos, Cyprus, June 6-8, 2007
Proceedings

Springer

Volume Editors

Amy L. Murphy
FBK-IRST-SRA
Via Sommarive 18, 38050 Povo, TN, Italy
E-mail: murphy@itc.it

Jan Vitek
Purdue University
Department of Computer Sciences
West Lafayette, IN 47907, USA
E-mail: jv@cs.purdue.edu

Library of Congress Control Number: 2007927649

CR Subject Classification (1998): D.2.4, D.2, C.2.4, D.1.3, F.1.2, I.2.11

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-72793-0 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-72793-4 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12069552 06/3180 5 4 3 2 1 0

Preface

Modern information systems rely increasingly on combining concurrent, distributed, real-time, reconfigurable, and heterogeneous components. New models, architectures, languages, and verification techniques are necessary to cope with the complexity induced by the demands of today's software development. COORDINATION aims to explore the spectrum of languages, middleware, services, and algorithms that separate behavior from interaction, therefore increasing modularity, simplifying reasoning, and ultimately enhancing software development.

This volume contains the proceedings of the Ninth International Conference on Coordination Models and Languages, COORDINATION 2007, held in Paphos, Cyprus in June 2007. For the second time, it was part of the federated conference DisCoTec. COORDINATION itself is part of a series whose proceedings have been published in LNCS volumes 1061, 1282, 1594, 1906, 2315, 2949, 3454, and 4038. From the 51 submissions received from around the world, the Program Committee selected 17 papers for presentation and publication in this volume on the basis of originality, quality, and relevance to the topics of the conference. Each submission received at least three reviews. As with previous editions, the paper submission and selection processes were managed entirely electronically. This was accomplished using EasyChair, a free Web-based conference management system. In addition to the technical paper presentations, COORDINATION hosted an invited presentation by Rachid Guerraoui from Ecole Polytechnique Federale de Lausanne.

We are grateful to all the Program Committee members who devoted much effort and time to read and discuss the papers. Moreover, we acknowledge the help of additional external reviewers who evaluated submissions in their area of expertise.

Finally, we would like to thank the authors of all the submitted papers and the conference attendees, for keeping this research community lively and interactive, and ultimately ensuring the success of this conference series.

June 2007

Amy L. Murphy
Jan Vitek

Organization

Coordination 2007, the Ninth International Conference on Coordination Models and Languages, is part of the set of federated conference DisCoTec 2007, also including DAIS 2007, the Seventh IFIP International Conference on Distributed Applications and Interoperable Systems, and FMOODS 2006, the Eighth IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems.

Program Co-chairs

Amy L. Murphy	ITC-IRST, Italy and University of Lugano Switzerland
Jan Vitek	Purdue University, USA

Program Committee

Nadia Busi	University of Bologna, Italy
Vinny Cahill	Trinity College Dublin, Ireland
Paolo Ciancarini	University of Bologna, Italy
William Cook	University of Texas, Austin, USA
John Field	IBM, USA
Christopher Gill	Washington University in Saint Louis, USA
Aniruddha Gokhale	Vanderbilt University, USA
Chris Hankin	Imperial College, UK
Michael Hicks	University of Maryland, College Park, USA
Valérie Issarny	INRIA, France
Christoph Kirsch	University of Salzburg, Austria
Doug Lea	SUNY Oswego, USA
Toby Lehman	IBM, USA
Alberto Montresor	University of Trento, Italy
Oscar Nierstrasz	University of Bern, Switzerland
Anna Philippou	University of Cyprus, Cyprus
Ernesto Pimentel	University of Malaga, Spain
Giovanni Russello	Imperial College, UK
Jim Waldo	SUN Microsystems, USA
Herbert Wiklicky	Imperial College, UK

Steering Committee

Rocco De Nicola	University of Florence, Italy
Farhad Arbab	CWI, The Netherlands
Paolo Ciancarini	University of Bologna, Italy

Chris Hankin	Imperial College, UK
Jean-Marie Jacquet	University of Namur, Belgium
Antonio Porto	New University of Lisbon, Portugal
Gian Pietro Picco	University of Trento, Italy
Gruia-Catalin Roman	Washington University Saint Louis, USA
Carolyn Talcott	SRI International, USA
Herbert Wiklicky	Imperial College, UK

External Reviewers

Silvia Amaro
Carlos Canal
Siobhán Clarke
Silviu Craciunas
Alessandra Di Pierro
Matthew Fluet
Manel Fredj
Daniele Gorla
Claudio Guidi
Francisco Gutiérrez
Bernhard Kast
Amogh Kavimandan
David Kitchin
Ivan Lanese
Pablo López
Maria-Cristina Marinescu
Sebastian Nanz
Manuel Oriol
Luca Padovani
Pascal Poizat
Harald Röck
Letian Rong
Gwen Salaün
Ana Sokolova
Oleg Sokolsky
Roberto Speicys Cardoso
Robert Staudinger
Christian Stefansen
Nikhil Swamy
Ian Wehrman
Verena Wolf

Table of Contents

Session 1. Middleware

Session 2. Logic Programming

Session 3. Formal Approaches

Session 4. Concurrency

Session 5. Components and Services I

Session 6. Manet

Session 7. Components and Services II

A Coordination Model for Triplespace Computing

Elena Simperl[1,2], Reto Krummenacher[2], and Lyndon Nixon[1]

[1] Free University of Berlin, Germany
[2] DERI, University of Innsbruck, Austria
simperl@inf.fu-berlin.de, reto.krummenacher@deri.org,
nixon@inf.fu-berlin.de

Abstract. Recent advances in middleware technologies propose semantics-aware tuplespaces as an instrument for coping with the requirements of *scalability*, *heterogeneity* and *dynamism* arising in highly distributed environments such as the Web or the emerging Semantic Web. In particular, Semantic Web services have inherited the Web service communication model, which is based on synchronous message exchange, thus being incompatible with the REST architectural model of the Web. Analogously to the conventional Web, truly Web-compliant service communication should, however, be based on persistent publication instead of message passing. This paper reconsiders *"triplespace computing"*, a coordination middleware for the Semantic Web. We look at how a coordination model for triplespace systems could look like - in order to manage formal knowledge representations in a space and to support the interaction patterns characteristic for the Semantic Web and Semantic Web services - as a precursor to the design and implementation of a triplespace platform in the context of the TripCom project.[1]

1 Introduction

The Semantic Web will have a significant impact on the next generation of worldwide network information systems. In order to build semantic applications, some middleware is necessary to offer uniform access to distributed information. This middleware differs from similar technologies for Web-based systems in that it incorporates means to cope with the machine-processable semantics of Web resources. Moreover, Semantic Web services - a core building block of the Semantic Web - have inherited the Web service communication model, which is based on synchronous message exchange, thus being incompatible with the Representational State Transfer (REST) architectural model of the Web [15]. In order to ensure truly Web-compliant service communication, there is need for a middleware solution that is based, analogously to the conventional Web, on persistent information publication instead of message passing, thus allowing services to exchange information in a time and reference decoupled manner.

In this paper we reconsider *"triplespace computing"*, a coordination middleware for the Semantic Web. Triplespace systems are able to manage information formalized using Semantic Web representation languages and to coordinate information exchange

[1] TripCom (IST-4-027324-STP): http://www.tripcom.org

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 1–18, 2007.

among agents processing this information. However, this new application scenario imposes several revisions of the Linda model. New coordination primitives as well as new types of tuples and spaces are needed in order to enable tuplespaces to deal not only with plain data, but also with interpretable information with assigned truth values. Further on, the new approach has to provide optimal support for the interaction patterns characteristic for the communication among Semantic Web services. In this paper we will use the terms *"triplespaces"* and *"triplespace system"* interchangeably to refer to an implementation of the triplespace computing paradigm.

The rest of this paper is organized as follows: Sections 2 and 3 present the basic principles of the Semantic Web and analyze the requirements that need to be satisfied by a (space-based) middleware in order to enable the realization of Semantic Web applications. We elaborate on triplespace computing and the underlying coordination model in Section 4. Section 5 gives an overview of similar initiatives, while Section 6 concludes with a discussion of open issues and future work.

2 An Introduction to the Semantic Web

The Semantic Web is an extension of the current Web in which information is given well-defined, machine-processable meaning, better enabling computers and people to co-operate [2]. It is intended to complement the current World Wide Web with a *network of URI-addressable information*, which is represented and linked in such a way that it can be easily processed by machines both at a *syntactical* and at a *semantical* level. For this purpose Web resources should be annotated with machine-understandable metadata that is formalized by use of common vocabularies with predefined semantics, known as *"ontologies"*. Further on, semantically enriched Web services should be able to process and exchange this information.

The first step towards the realization of the Semantic Web have been made through the standardization of representation languages for Web knowledge like RDF [22],

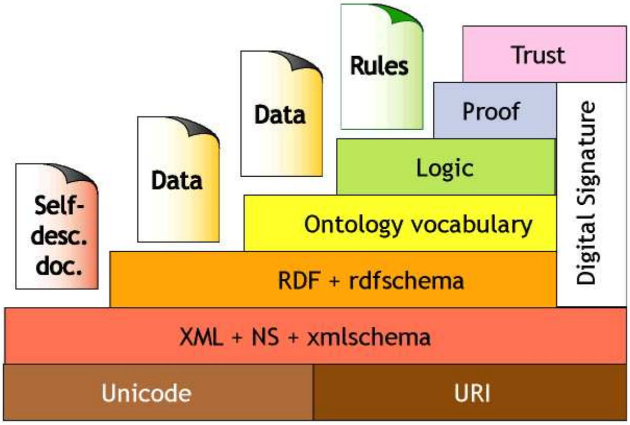


Fig. 1. The Semantic Web stack by Tim Berners-Lee [1]

RDFS [3] and OWL [26] and the increasing dissemination of ontologies that provide a common basis for annotations. Additionally, recent efforts in the area of Web services propose methods and tools to represent Web services in a Semantic Web compatible manner in order to improve tasks like automatic service discovery or composition [5,7,14,24].

The Semantic Web is built on XML-based syntaxes which use URIs to uniquely identify Web resources (cf. Figure 1). Resources can denote not only common Web documents, but any entity represented within a computer system (e.g. persons, physical objects, RDF statements) and are described by machine-processable metadata that provides data about their properties, capabilities, and requirements. Metadata is then a collection of RDF statements of the type ⟨*subject, predicate, object*⟩, where the three fields can be individually referenced by means of URIs. For exemplification purposes we annotate the present paper (if available as a Web document) with information concerning its author, content etc. Examples of such RDF statements could be:

The email address of Elena Simperl, one of the authors of this paper, is "simperl@inf.fu-berlin.de". This information can be represented in RDF in form of the following two triples:

⟨*x:Coordination07Paper, dc:contributor, http://userpage.fu-berlin.de/simperl*⟩
⟨*http://userpage.fu-berlin.de/simperl, adr:mail, "simperl@inf.fu-berlin.de"*⟩

The subject of the first statement is this paper, identified by an imaginary URI *x:Coordination07Paper*. The predicate *dc:contributor* is part of the Dublin Core (DC) metadata scheme and denotes a standardized property, while the object *http://userpage.fu-berlin.de/simperl*, the value of the property, represents the co-author Elena Simperl. The object in an RDF statement can be another RDF resource - represented by a URI, as in the first triple - a literal - represented through a simple XML datatype, as in the second triple - or a blank node. Further on, the example shows the way individual triples are naturally interlinked to RDF graphs: the object of the first statement can be used as subject in subsequent statements.

The contact address of the co-author is "Takustr. 9, 14195 Berlin, Germany". Structured information like addresses can be represented in RDF in aggregated form, by using so-called *"blank nodes"*, which denote anonymous resources, identified by application internal URIs:

⟨*http://userpage.fu-berlin.de/simperl, adr:contact, _:bn1*⟩
⟨*_:bn1, adr:street, "Takustr."*⟩
⟨*_:bn1, adr:number, "9"*⟩
⟨*_:bn1, adr:zip, "14195"*⟩
⟨*_:bn1, adr:city, "Berlin"*⟩

Here *_:bn1* identifies an anonymous resource which aggregates the address-related data, i.e. there is some entity that is characterized by an *adr:street*, an *adr:number*, etc.

RDF is considered to be the standard interchange format of the Semantic Web and is intended to be used as a simple yet powerful annotation language for Web resources.

The next layers on Figure 1 add logically more expressive languages for ontologies (e.g. OWL) and support for rules (e.g. SWRL [19]). RDFS and OWL are used to define common vocabularies for metadata which enable interoperability among applications. Besides, they re-define Web resources in terms of classes and properties with a well-founded semantics which can be exploited by reasoners to validate the models and to automatically generate implicit knowledge. We demonstrate the usage of ontologies and ontology representation languages by extending the previous examples with additional information formalized in RDFS and OWL. In that way we align the local terms to external vocabularies. This provides a more precise specification of the underlying domain, thus forming the basis for more powerful information retrieval in a fictive publication repository.

This paper is a special kind of publication. Specialization and generalization hierarchies can be built in RDFS and OWL. A *Paper* is a special type of *Publication*:

⟨*x:Paper, rdfs:subClassOf, x:Publication*⟩

Further on, one can link ontological information to concrete instance data; the statement

⟨*x:Coordination07Paper, rdf:type, x:Paper*⟩

specifies that the current document is a particular instance of the class *Paper* in our imaginary publications ontology. In this way a query looking for publications of a given author would match not only to the documents which are explicitly defined (through annotations) as publications, but also to specific publication types such as conference papers or journal articles.

The concept* Publication *in the local ontology with the namespace 'x' is equivalent to the concept* Text *in the Dublin Core vocabulary. Equivalence relationships between entities can be expressed in OWL:

⟨*x:Publication, owl:sameClassAs, dc:Text*⟩

Aligning the personal ontology to a standard model such as Dublin Core is the first step towards increased syntactic and semantic interoperability. Ensuring syntactic compatibility enables more flexible retrieval services, which are then able to go beyond simple string matching. The semantic interoperability allows applications handling the local ontology to better understand the meaning of the corresponding concept, since the equivalence between concepts implies also an inheritance relationship by means of which the local concept is enriched with the properties externally defined for its equivalent.

A paper consists of several sections. The domain model is further refined in order to define typical (physical or conceptual) parts of a scientific paper. As in the previous example, extending the model supplies machines with deeper background knowledge on the application domain, thus enabling advanced, domain-tailored behavior. From a modeling point of view we introduce a new property *partOf* which relates entities to their parts:

⟨*x:Section x:partOf x:Paper*⟩

In addition to specifying their domain and range, one can refine the semantics of the domain properties by specifying features such as transitivity, symmetry or cardinality constraints.

The remaining layers of the Semantic Web are still at a much more immature stage. However, issues of proof and trust are vital to the success of the Semantic Web since one needs techniques to verify the quality of the created metadata. The proof layer (cf. Figure 1) is intended to provide languages and tools to prove whether statements created by arbitrary authors are true or not. The trust layer addresses the same issue from a different perspective. It should define mechanisms which, in correlation with digital signatures, enable the definition of provenance and reputation information for resource metadata.

3 Requirements for a Semantics-Aware Middleware

With the growing importance of open distributed systems, in particular the World Wide Web, new requirements arose from the way services communicate and coordinate their access to shared information. The coordination between clients in such open environment is more complex as a system cannot know in advance which clients will use it, nor which characteristics the active clients might have. For example, clients may not necessarily agree in advance on shared models, protocols and types for the data exchanged. It may become the task of the middleware to (partially) resolve such heterogeneities. For open distributed systems these twin issues of dynamism and heterogeneity need to be taken into account and are furthermore tightly related to the problem of scalability. One answer to these issues is the application of semantic technologies to enhance the descriptions of clients, services, protocols and data for example in form of the aforementioned *Semantic Web services*.

The advantage of adding semantics to service, protocol and data descriptions is that the interfaces to the different agents, services and information sources can be dynamically generated, while the semantics of the exchanged information is formally defined. Consequently there are new means for the automatization of service and data discovery, mediation and service composition. This suggests that (Semantic) Web services provide a good communication model for distributed computing scenarios, as envisioned by service-oriented architectures [10]. However, Semantic Web service infrastructures do not provide support for persistent publication of data, nor for time decoupling of messages so that data can outlive the services publishing or consuming it [11].

These features however are the characteristic of Linda-based systems [17]. Yet, applying tuplespaces to the open global environment of the (Semantic) Web raises new requirements [11,20]:

- a reference mechanism. The Web uses URLs as a global mechanism to uniquely address resources. This offers means to address particular spaces or tuples independently of their tuple fields and field data.

- a separation mechanism. Distributed applications which have independent naming schemes may use the same names for their resources. On the Web, vocabularies can be kept separate - even when using the same terms - by help of a namespaces mechanism. The concept of namespaces should be handed down to tuplespace systems as well in order to inherit the Web functionality.
- the nesting of tuples. Web data models such as XML and RDF permit the nesting of elements within a single document, i.e. RDF data adds to RDF Graphs. Likewise tuples should be able to explicitly show how information units are interlinked.

This gives evidence that the coordination model of Linda needs to be rethought and extended in order to meet the requirements of Web-scale systems. A coordination model for the Semantic Web and Semantic Web services must be able

- to support autonomous activity of participants by decoupling interactions in time and reference - two agents are neither required to be available concurrently, nor must they know each others location or address,
- to process semantic information as the data being coordinated, and
- to provide access to large amounts of heterogeneous data that is dispersed over broad and dynamic networks.

In spite of these complex requirements it is clear that combining Linda models and semantic technologies introduces a new and powerful communication paradigm that provides the desired grounds for persistent, asynchronous and anonymous dissemination of machine-understandable information. This communication paradigm has been referred to as *“triplespace computing”* [11]. The target application scenarios for this novel middleware approach range from sharing information on the Semantic Web, distributed knowledge management and pervasive computing to a fully fledged communication and coordination platform for Semantic Web services or the Semantic Grid [32]. There, the potential application areas are just as diverse: Enterprise Application Integration (EAI), eHealth, digital multimedia systems and recommender systems, to only mention a few.

3.1 Required Extensions to Linda or Why Linda Is Not Enough

In order to comply the requirements of large scale Semantic Web applications Linda must be extended in several directions. We distinguish two categories of extensions to the original approach: (1) new types of tuplespaces, and (2) new types of tuples.

The former category aims at overcoming the technical problems of large scale distributed systems (e.g. heterogeneity, scalability, fault-tolerance, multiuser access) by proposing distribution strategies for multiple spaces or hierarchic spaces and augmented naming approaches. Such approaches were already considered in various non-semantic tuplespace implementations [6,9,27,30,34], as it was recognized that the traditional Linda approach does not suffice in the large. The second research direction looks at the necessary extension to tuples and tuple fields. The Semantic Web is mainly about the meaning of Web resources and their properties. The first problem with traditional Linda implementations is that tuples with the same number of field and the same field typing cannot be distinguished, which does not comply with the Semantic Web principle which foresees that all information is encoded in triples of URIs. Moreover, as we have seen in Section 2, the different semantic tuples in a space are not independent as in Linda, but highly correlated and depending on each other. Hence, the Semantic Web requires a reconsideration of the tuple model and the way tuples are matched in accordance to these concepts. The matching algorithms have to consider the meaning of the semantic tuples in order to provide the required degree of knowledge interpretation, and allow the retrieval of tuples taking into account the relationship to other tuples in the same space. This highly relates to the already discussed issues of resource identification – reference mechanism, namespace mechanism – which must be supported by the space in order to allow semantic data to be represented according to the specification of RDF.

Several research projects have already arisen that work on the specification and implementation of such extensions. We provide a short description of these projects in Section 5, but first we elaborate on the TripCom approach to semantics-aware tuplespaces.

4 Triplespace Computing in TripCom

The core vision of triplespace computing is to establish the Web paradigm of *“persistently publish and read”* for the Semantic Web and Semantic Web services. Currently these ideas are further conceptualized and implemented within the EU project TripCom. This section describes the basic concepts of this approach, as a precursor for the design and implementation of a TripCom triplespace system. First, it is necessary to revisit the definition of tuples and spaces, as it is indispensable to adapt these concepts to the norms of the Semantic Web. Thereafter we concentrate on the description of the required coordination primitives in order to make the Linda operations compatible to the requirements of the Semantic Web and Semantic Web services.

4.1 New Types of Tuples and Tuplespaces

Following the Linda paradigm a triplespace system should be able to represent *semantic information* through *tuples*. The expressivity of the information representation should be aligned to the expressivity of common Semantic Web languages, while respecting their semantics, so that tuples could be mapped to and from external Semantic Web resources. Regarding Semantic Web languages, we currently focus on RDF. RDF statements can be represented in a three fielded tuple (so-called *“triples”*) of the form $\langle$*subject, predicate, object*$\rangle$. Following the RDF abstract syntax each tuple field contains an URI (or, in the case of the object also a literal). Tuples can be addressed by means of URIs too, which are defined through the triplespace ontology (cf. Section 4.3). In this way tuples sharing the same subject, predicate and object can be addressed separately, which is consistent with the Linda model. If a tuple is removed from and reinserted into the space, it is allocated with a new URI, as the reinserted tuple is regarded as a new tuple despite its relatedness with the one previously deleted. When performing reasoning however, tuples sharing the same content are interpreted as duplicates despite different identifiers, as foreseen by the RDF semantics [18].

A second issue to be considered in this context is the way sets of related statements (i.e. RDF graphs) are represented at the level of tuples. Since both RDF statements and graphs are associated to URIs in the triplespace ontology (cf. Section 4.3), we can specify the membership relation through an additional triple using a dedicated meta-property: ⟨tupleURI p:partOf graphURI⟩. Moreover, the Triplespace API foresees a series of operations allowing the manipulation of RDF graphs (cf. Section 4.2).

The model of a semantic tuple can be further refined in terms of the knowledge representation language whose semantics is relevant for processing the tuple content in relation with matchings. As the prospected system foresees a semantic matching behavior in addition to the classical Linda procedures, it might be useful to differentiate between tuples embedding RDF, RDFS, OWL or WSML [12] data. While this distinction does not affect the basic tuple model, which remains a set of three fields, it triggers the usage of particular matching procedures, and thus needs to be stored in the space. Again, the triplespace ontology can be an instrument to capture this type of metadata about the tuples and their content. The coordination operations take into account this metadata in order to enforce the execution of a particular matching algorithm.

A triplespace is defined as a container for triples which encapsulate the RDF statements. A triplespace can be divided into virtual subspaces and physically partitioned across distributed kernels. Every space is addressed using a URI, which is installed by the creating user and captured in the triplespace ontology. Just as in the case of individual tuples, this URI is useful in terms of the REST communication model. A space may contain multiple (sub-)spaces, while it can only be contained in at most one parent space. The latter holds also for tuples, which are associated to a single space. In order to allow for overlapping between space the triplespace model resorts to the notion of *"scopes"* [28]. Scopes are temporary tuple containers. Unlike subspaces, which form the virtual structure of the triplespaces, they can be created individually by clients based on arbitrary filters. Scopes can be given different semantics, e.g. they could be seen as an alternative view on the structure of the tuplespace, or as a temporary copy of some tuples for retrieval by a client - in the latter case insertion and deletion operations would apply to the scope and not to the tuplespace as a whole. We note two final differences between spaces and their temporary counterparts: while spaces are defined to be non-overlapping, the notion of scopes does not impose this restriction. Consequently a tuple can be contained in a space - and implicitly in all the direct and indirect parent spaces of the original space - and in a multitude of possibly independent scopes (cf. [28]). The scopes are furthermore not stored in the triplespace ontology, which captures meta-aspects on *persistent* data containers such as graphs or spaces (cf. Section 4.3). Support for scopes is subject of ongoing work also at the level of the Triplespace API, which is introduced in the next section.

4.2 New Coordination Primitives

The Linda operations *out*, *in* and *rd* form the basis for any tuplespace implementation. The basic tuplespace primitives have however soon proven to be insufficient in various application contexts, and implementations of tuplespace platforms based on Linda have liberally extended the coordination language for their needs. We note that extensions to the coordination language can take different forms:

- the extension of the coordination primitives, i.e. the specification of new operations
- the extension of the coordination semantics, i.e. the redefinition of the meaning of the existing operations
- the extension of the coordination model, i.e. providing other instruments for expressing coordination such as programmability.

In this section we introduce the operations defined for the triplespace coordination model and thus our extensions to the Linda coordination primitives and their meaning in this context. We take a look at the semantics and the signatures of the operations and discuss the rationales for taking certain design decisions. An outline of the Triplespace API is given in Table 1.

The core operations are defined around the traditional Linda primitives. However, due to the characteristics of RDF, where triples are not independent data containers, but rather sets of interlinked ⟨*subject, predicate, object*⟩-tuples, it was necessary to extend the semantics of the operations. The *out* operation received a multi-write characteristics in order to allow the publication of whole RDF graphs within one call to the space. A similar operation termed *multiWrite* was for instance already foreseen in TSpaces [23].

With respect to tuple retrieval we differentiate three cases depending on the scope of the returned data. A retrieval call to the space does not return a single tuple/triple, but rather a whole RDF construct. The way such RDF constructs are created is seen to be an implementation detail (e.g. based on the Concise Bounded Descriptions approach [33]). This decision is motivated by the fact that a single RDF triple hardly has any meaning by itself. Reconsidering the examples in Section 2 it can be seen that already very simple statements consist of several triples that isolated would not communicate any useful information.

The primitive *rda* retrieves a single triple (read a triple) and the triples directly bound to it. This is hence the Semantic Web version of the Linda *rd* operation. The more frequently used operation is expected to be the plain *rd*, therefore also the simpler name. *rd* returns an undetermined number of triples and its neighbors. In consequence *rd* is an information retrieval operation. The amount of information (the number of triples) returned is limited by the indeterminism of the space platform and the possible incomplete set of visible nodes, i.e. *rd* does not guarantee that all matching information in a space is returned, although it aims for it. This conforms well to the original Linda definition and is clearly manifested by the open nature of the (Semantic) Web and the open world assumption of Semantic Web languages such as OWL. The third retrieval primitive called *rdg* is used to search for whole graphs (read graph). As explained previously, the triplespace *out* operation accepts a triple set (i.e. RDF graphs), which can optionally be associated with an identifier (a graph URI) to construct a so-called named graph [8] as an input. The *rdg* operation allows the retrieval of a single graph that contains at least one triple matching the template. This is particularly useful when coordinating Web service descriptions or purchase orders, where it is indispensable that all knowledge about one construct is retrieved at all times. To ensure that all data of a graph can be read at once, the system will store RDF graphs at the same physical location and hence they are either seen as a whole or not visible to a particular user at all. The three operations are also available in destructive mode. In this case they are referred to as *ina*, *in* and *ing*, respectively.

Table 1. The Triplespace API

out(Graph g, URI space, [URI graph, URI transaction]):void Inserts the triples included in the graph into the given space. By specifying a graph identifier a named graph is created. A transaction identifier can be provided to add the out to a given active transaction.
rda(Template t, [URI space, URI transaction, integer timeout]):Graph Returns *one* matching triple with any triples bound to it, e.g. following the Concise Bounded Descriptions approach. The request is executed against the given space, if provided, otherwise against the virtual global triplespace. The timeout is used as means to control the blocking characteristics of rda.
rd(Template t, [URI space, URI transaction, integer timeout]):Graph This operation generalizes rda: it returns an undetermined number of matching triples and their bound graphs; otherwise the functionality is the same.
rdg(Template t, [URI space, URI transaction, integer timeout]):Graph Returns the entire content of a named graph that contains a matching triple; used to coordinate whole objects.
ina(Template t, [URI space, URI transaction, integer timeout]):Graph This is the destructive version of rda.
in(Template t, [URI space, URI transaction, integer timeout]):Graph This is the destructive version of rd.
ing(Template t, [URI space, URI transaction, integer timeout]):Graph This is the destructive version of rdg.
subscribe(Template t, URI space, Listener l, [URI transaction]):URI Establishes a notification mechanism for triples matching the template. Subscriptions must be expressed against a given space. In case of a match the listener (e.g., in Java a class that is called in case of an event) is notified. The operation returns a handle to the successfully registered subscription in form of a URI.
unsubscribe(URI subscription, [URI transaction]):boolean This operation cancels a given subscription and returns true in case of successful execution.
create(URI space, [URI parent, URI transaction]):boolean This primitive creates a new space, as subspace of parent. In case no parent space is indicated the new space is installed as direct child of the virtual global space. It returns true after successful creation.
destroy(URI space, [URI transaction]):boolean This operation destroys the given space, its subspaces and all contained triples. Particular attention has therefore to be paid to rights management to avoid unauthorized removals.

The matching procedure for the six operations is based on templates. The precise syntax and semantics of a templates depends on the maturity of the space implementation and on the query languages and engines employed in the implementation; we generally refer to it as template in order to proceed with a stable interaction model. In the current release (March 2007) we use simple triple patterns that are very close to traditional Linda templates. A template is hence a tuple that contains both RDF resources and variables: ⟨*x:Coordination07Paper ?p ?o*⟩. In the future templates will consist of graph patterns (cf. Table 2) as common to most RDF query languages, e.g. SPARQL

[31]. Eventually, we expect the templates to evolve to fully fledged semantic queries or rules, which invoke reasoning engines and operate on asserted and inferred triples. In that way we can guarantee that the matching algorithms fully exploit the semantics of the information published. Even though it is clear that this enhanced matching algorithms will be more complex than pure Linda matching, we argue that it provides a good compromise between simplicity and matching at the level of meaning.

Table 2. Examples of Semantic Templates

TEMPLATE	DESCRIPTION

`?s a doap:Project;`
`   foaf:member ?o.`
	Matches all triples where the subject is of type doap:Project and where the same subject has triples indicating the members.

`?s ?p ?o.`
`?o a foaf:Person.`
	Matches all triples where the object is of type foaf:Person.

`?s foaf:name ?a;`
`   foaf:mbox ?b.`
	Matches the triples that contain subjects for which the name and a mailbox (foaf:mbox) are indicated.

Though retrieval by identifier does not correspond to the core Linda principle of associative addressing, it was previously considered in various tuplespace implementations such as for instance by the method *readTupleById* in TSpaces. As mentioned in Section 3 the Web uses unique identifiers (URI) to reference resources. The triplespaces are envisioned to provide a shared information space for the Semantic Web that inherits these basic notations regarding resource identification, and thus it was a natural choice to incorporate means to use URIs for retrieval. In TripCom this behavior is provided by use of metadata (cf. Section 4.3) that yields ontological descriptions of the triplespace contents. The name of graphs, the context of data and their interrelationships are modeled and stored in the triplespace ontology. The knowledge captured by this meta-model can be used to refine queries and thus also to query RDF graphs by their name if adequate: *match all triples belonging to the graph with identifier X*.

Furthermore, the retrieval operations will have a field of type integer for the specification of timeouts, and support for transactions. The timeout is used to control the blocking disposition of triplespace requests. A positive integer interrupts the search process after the provided number of seconds passed. If no timeout is specified the operation runs in the regular blocking mode. Note that this is particulary unadvised for *rd* calls due to its multi-search trait that does not come with a automatic retrieval interruption after a first discovery. Using *rd* without timeout results in data acquisition from the virtual global triplespace and this might eventually not terminate.

Transaction support was already added in tuplespace systems such as JavaSpaces [16] and TSpaces. Transactions play an important role in Web service communication and hence this extension was taken over to the coordination model of the triplespace. The first release does not yet integrate any support for transaction, while we plan to integrate transactions locally in a next step, i.e. a given transaction only wraps subsequent calls at a single access node. Eventually we want to install distributed transactions over multiple triplespace nodes too. This allows for transactional safe workflow executions involving multiple agents. Just as we do not guarantee that all data in a triplespace is visible at all times, due to physical or logical distribution, we do not guarantee that data encapsulated by transactions are visible to any other user.

The *subscribe* and *unsubscribe* methods are used to incorporate the notification service provided by triplespaces. Notification is an important tool for many interaction patterns and ensures flow-decoupling of concurrent processes. The use of notification is potentially a good compromise between easiness of implementation and expressive power in distributed systems, for which non-blocking operations do not provide appropriate solutions. A subscription is done by emitting a template to an interesting space. The subscription manager will then inform any subscriber about matching triples until a *unsubscribe* call is sent.

The last two operations depicted in Table 1 are management methods used to create and delete spaces. A space is created by giving it a new unique identifier and by possibly attaching it as a subspace to an already existing space. The semantics of *destroy* is more complex, as the removal of a space implies the deletion of all subspaces, and of the contained data. We therefore expect that removal is only allowed to the creator of the space, or at least that it depends on restrictive security measures. Security, privacy and trust measures are entirely neglected in this paper, as they are seen to be orthogonal to the presented concepts and are developed in parallel.

4.3 Triplespace Ontology

The coordination and data models for semantic tuplespaces provide the first building blocks towards the realization of the triplespace computing paradigm [11]. However, the crucial issues of distribution and scalability are only marginally addressed. The use of an ontology that describes the published data, the created spaces and their interrelationships and characteristics is expected to provide support in tackling the additional requirements of the open distributed scenarios targeted by TripCom. Ontology-driven middleware management is in fact seen to be one of the major assets of semantic tuplespaces compared to traditional space frameworks. The use of ontologies provides sophisticated means for data and infrastructure handling which directly influences and likely improves the necessary distribution and scalability measures.

In this section we introduce first ideas for a triplespace ontology and the expected benefits resulting thereof.

Metadata is an important tool to optimize management tasks, access procedures and, in case of distributed data sources, also to improve the distribution algorithms such as for replication or caching. Knowledge about the relationship of data items and their context allows for more precise and effective handling and faster processing of requests. Relational databases for instance use meta-information about tables, column types and access privileges. Object-oriented databases rely on information about the data structures, while data warehouses and reference information systems provide meta-information about the provenance and quality of information and their sources. If such information is normalized and formalized, i.e. in terms of an ontology, it is possible to reason about it, to infer and combine new facts, validate them or counteract possible ambiguity or incompleteness of information. This becomes particularly evident and useful when dealing with distributed data sources. In other words understanding the semantics of the meta-information is expected to improve and facilitate the management tasks of the system.

In TripCom we define a triplespace ontology for three particular tasks:

- the optimization of access patterns as mentioned in Section 4.2,
- the improvement of performance and scalability of the triplespace middleware by help of enhanced distribution mechanisms, and
- the management of the security and trust framework. The last task addresses issues like the modeling of user roles and access permissions. Further details are out of the scope of this paper.

The ontology modules so far developed are concentrating on the description of triples, graphs and the spaces they are published in. Moreover the ontology addresses the functionalities of the triplespace kernels: language and reasoning support, storage infrastructure, installed query engines. A triplespace kernel (TS Kernel) is the implementation of a triplespace access node. An informal excerpt of the triplespace ontology (classes and properties) is given in Table 3.[2]

In accordance with the definitions in Sections 4.1 and 4.2 we can see that *Data* items are either RDF *Triples* or RDF *Graphs*. As already pointed out in Section 4.1 it is also possible to serialize *formalisms* more expressive than RDF to triples. In order to ensure an adequate interpretation during matching or reasoning we link *Data* items to an identifier (URI) for the respective underlying language. Whenever a piece of data is accessed, if published for the first time, altered or simply read, it is possible to register an *AccessLogEntry*. The log entry contains information about the *Agent* that addressed the data item, as well as the time of access and the type of access. At all times the data is contained in a *Space*. The space itself can be contained in another triplespace within a hierarchy of spaces (built using the *isSubspaceOf* transitive property) and is shared by all nodes running a *Kernel* that manages the access to and the data of the space. Inversely a *Kernel* shares a given set of spaces, relationship which is captured by the property *sharesSpace*. Moreover kernels are linked to repositories in order to guarantee data persistency. The *Repositories* are accessed via some *QueryEngine* that resolves queries expressed in a particular *QueryLanguage*.

The previous paragraph depicted a short example of what is envisioned with the triplespace ontology. The captured metadata could also be used to refine templates, i.e. one could ask for triples matching a given pattern, but only if they were published by a particular agent.

The more important objective of the ontology is however the scalability and performance of the space installation and the discovery process (latency and quality). A concretization of the algorithms and the required ontology support is however only in its infancy at this stage of the TripCom project. First attempts allow to associate data with a particular kernel by use of the property *isManagedAtKernel*, or to point from one kernel to another (*seeAlsoKernel*) if they share particular characteristics: same type of data, same users, to name only a few possibilities.

[2] Further details on the triplespace ontology are available at http://www.tripcom.org/ontologies/

Table 3. Excerpt of the Triplespace Ontology

```
Triple :: Data
  partOf Graph

Graph :: Data
  hasPart Triple

Data
  formalism URI
  isContainedIn Space
  hasLogEntry AccessLogEntry
  isManagedAtKernel Kernel

Space
  isSubspaceOf Space
  isSharedAtKernel Kernel

AccessLogEntry
  publisher Agent
  date xsd:dateTime
  type AccessType

Kernel
  sharesSpace Space
  hasQueryEngine QueryEngine
  seeAlsoKernel Kernel

QueryEngine
  language QueryLanguage
  usesRepository Repository
```

In summary, the use of an integrated ontology-based meta information infrastructure brings along two major advantages for triplespace computing:

– the inference framework of the space middleware allows reasoning not only about the application data that is published and consumed by space users, but also about the administrative data (metadata).
– the use of Semantic Web languages, in particular the application of RDF, allows for an integrated platform without additional requirements on the space infrastructure, i.e. the administrative data is processed and stored by the same tools as the application data.

5 Related Work

This paper has described the conceptual model behind triplespaces, a middleware for coordinating knowledge processes on the Semantic Web. We consider this work to be the first comprehensive and well-grounded specification of a model for semantic Linda.

The initial idea of combining tuplespaces and Semantic Web information has been previously proposed in [11]. Subsequent proposals for a semantics-enabled coordination model such as [4,13,25] have however failed to address issues covered in this paper such as the particular representation of RDF syntax, the necessary revisions of the coordination primitives and the means for tuplespace partitioning and kernel distribution required to cope with the scalability and dynamism of the Semantic Web. The approach in [25] is deliberately targeted at a much wider range of application areas, which induces a lack of focus on Semantic Web-specific issues in terms of tuple and tuplespace models and the associated coordination operations. [4] proposes a minimal architecture for triplespace computing, but does not provide any details on the types of tuples and tuplespaces required in this context. The TSC project applies coordination principles to realize a communication middleware for Semantic Web services [13]. The approach is built upon an existing co-ordination system which led to many design decisions being simply carried over rather than re-assessed, as we have done, in a Semantic Web context. For example, the access at API level is to Java objects in the space encapsulating RDF graphs, preventing any lower granularity access at the triple level such as here. Further on, the system is targeted at Semantic Web services, while our triplespace approach covers the whole range of application settings on the Semantic Web. It is also unclear to what extent the aforementioned proposals continue to respect the basic principles of Linda, while the triplespace approach is clearly "backwards compatible".

Semantic Web Spaces [29] was conceived a generic lightweight coordination middleware for sharing and exchanging semantic data on the Web. While the types of tuples and tuplespaces foreseen in Semantic Web Spaces are similar to the approach introduced in this paper, their coordination model introduces a limited set of operations for handling RDF triples, while not considering extensions such as notifications or transactions, which are clearly required on the Semantic Web.

sTuples extended the JavaSpaces platform to support OWL data in tuple fields [21]. However, this approach has not further considered the implications of coordinating Semantic Web information, as we have done. Rather, OWL graphs are exchanged within tuples, and extracted and processed in other systems while in triplespaces we seek to integrate a Semantic Web framework within the system.

A further distinctive feature is the triplespace ontology. The usage of ontologies for middleware management purposes is acknowledged in [13,25,29]. However, only Semantic Web Spaces provides a brief outline how such a meta-model could look like. By contrast, our triplespace ontology is the result of a systematic ontology engineering process, carried on in collaboration with several potential users of such a triplespace platform in areas such as eHealth, Semantic Web services and Enterprize Application Integration.

6 Conclusion

In this paper we reconsidered triplespace computing and the envisioned added value of this novel paradigm for the Semantic Web and Semantic Web services with respect to heterogeneity, scalability and dynamism. First we discussed the requirements of a semantics-aware middleware and looked at the necessary extensions and adaptations of the original Linda coordination model with respect to the representation of formal knowledge and the interactions patterns on the Semantic Web. The paper presented moreover the concepts, models and interaction primitives for the triplespace platform of the TripCom project.

The triplespace model extends Linda mainly in what concerns the representation of semantic data in form of RDF triples and graphs. On the one hand it was necessary to revise the syntax and semantics of tuples and templates, on the other we adapted the interaction primitives, *out*, *in* and *rd* to reflect the fact that RDF triples are identifiable resources that presented nested and interlinked knowledge.

In order to make triplespaces scalable on Web-scale, in contrast to traditional approaches that rather focused on corporate and thus small-scale solutions, the tuplespace model had to be revised as well. Moreover, we expect additional support form ontology-driven space management. As matter of fact we expect the application of ontologies to be one of the major assets compared to conventional tuplespace installations.

While the heterogeneity problem is solved by the support for Semantic Web tools and dynamism is implicitly addressed by the inherited features of space-based computing, the scalability issue is still only marginally touched. We will therefore concentrate on mechanisms to tackle the significant challenges of distribution in large scale systems like the World Wide Web, Grid or pervasive computing environments. Semantic clustering of data, organization of spaces according to the internal structures of data and the joint usage of local and global spaces are possible starting points for future improvements to the existing semantic tuplespaces. Some of these ideas were already materialized by use of the triplespace ontology, but not yet implemented. We thus expect that upcoming work will take up these ideas, and that solutions for the distribution and scalability issues will be developed around them.

Moreover, the future of the Semantic Web is seen in the integration of rules languages with the currently available W3C recommendations RDF(S) and OWL. Such complex knowledge representation formalisms and the associated sophisticated reasoning services they enable are still missing in triplespace computing.

The triplespace computing concepts and models presented in this paper are the first steps in substantiating the ideas of [11]. We expect that further development will push the integration process of triplespace computing with the Semantic Web architecture, particularly for machine-to-machine communication:

> *The triplespace may become the Web for machines as the Web, based on HTML, became the Web for humans.* [11]

Acknowledgment

This work is funded by the European Commission under the project TripCom (IST-4-027324-STP). The authors would like to thank all members of the consortium for their advice and input, in particular the colleagues of the work packages WP2 and WP3.

References

1. Berners-Lee, T.: Semantic Web - XML2000. Talk: `http://www.w3.org/2000/Talks/1206-xml2k-tbl` (December 2000)
2. Berners-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American 284(5), 34–43 (May 2001)
3. Brickley, D., Guha, R.V.: RDF Vocabulary Description Language 1.0: RDF Schema. W3C Recommendation (February 2004)
4. Bussler, Ch.: A Minimal Triple Space Computing Architecture. In: 2nd WSMO Implementation Workshop (June 2005)
5. Cabral, L., Domingue, J., Galizia, S., Gugliotta, A., Norton, B., Tanasescu, V., Pedrinaci, C.: IRS-III: A Broker for Semantic Web Services based Applications. In: 5th Int'l Semantic Web Conf. pp. 201–214 (November 2006)
6. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: a Programmable Coordination Architecture for Mobile Agents. IEEE Internet Computing 4(4), 26–35 (July 2000)

7. Cardoso, J., Sheth, A.: Semantic Web Services, Processes and Applications. Springer, Heidelberg (2006)
8. Carroll, J.J., Bizer, Ch., Hayes, P., Stickler, P.: Named Graphs. Journal of Web. Semantics 3(4), 247–267 (October 2005)
9. Ciancarini, P., Knoche, A., Tolksdorf, R., Vitali, F.: PageSpace: An Architecture to Coordinate Distributed Applications on the Web. Computer Networks and ISDN Systems 28(7), 941–952 (May 1996)
10. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall PTR, Englewood Cliffs (August 2005)
11. Fensel, D.: Triple-Space Computing: Semantic Web Services Based on Persistent Publication of Information. In: IFIP Int'l Conf. on Intelligence in Communication Systems, pp. 43–53 (November 2004)
12. Fensel, D., Bussler, C.: The Web Service Modeling Framework WSMF. Electronic Commerce Research and Applications 1(2), 113–137 (Summer 2002)
13. Fensel, D., Krummenacher, R., Shafiq, O., Kuehn, E., Riemer, J., Ding, Y., Draxler, B.: TSC - Triple Space Computing. e&i Elektrotechnik und Informationstechnik, 124(1/2) (February 2007)
14. Fensel, D., Lausen, H., Polleres, A., de Bruijn, J., Stollberg, M., Roman, D., Domingue, J.: Enabling Semantic Web Services: The Web Service Modeling Ontology. Springer, Heidelberg (November 2006)
15. Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California Irvine (2000)
16. Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles, Patterns, and Practice. In: The Jini Technology Series, Addison-Wesley Longman Ltd (1999)
17. Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 80–112 (January 1985)
18. Hayes, P., McBride, B.: RDF Semantics. W3C Recommendation (February 2004)
19. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C Member Submission (May 2004)
20. Johanson, B., Fox, A.: Extending Tuplespaces for Coordination in Interactive Workspaces. Journal of Systems and Software 69(3), 243–266 (January 2004)
21. Khushraj, D., Lassila, O., Finin, T.W.: sTuples: Semantic Tuple Spaces. In: 1st Ann. Int'l Conf. on Mobile and Ubiquitous Systems: Networking and Services, pp. 268–277 (August 2004)
22. Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation (February 2004)
23. Lehman, T.J., McLaughry, St.W., Wyckoff, P.:T Spaces: The Next Wave. In: 32nd Hawaii Int'l Conf. on System Sciences (January 1999)
24. Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., Sycara, K.: OWL-S: Semantic Markup for Web Services. W3C Member Submission (November 2004)
25. Martín-Recuerda, F.: Towards CSpaces: A new perspective for the Semantic Web. In: 1st Int'l IFIP/WG12.5 Working Conf. on Industrial Applications of Semantic Web (August 2005)
26. McGuinness, D.L., van Harmelen, F.: OWL Web Ontology Language Overview. W3C Recommendation (February 2004)
27. Menezes, R., Tolksdorf, R.: Adaptiveness in Linda-based Coordination Models. In: Workshop on Engineering Self-Organising Applications, pp. 212–232 (July 2003)
28. Merrick, I., Wood, A.: Coordination with scopes. In: ACM Symposium on Applied Computing, pp. 210–217 (March 2000)

29. Nixon, L.J.B., Simperl, E.P.B., Antonenko, O., Tolksdorf, R.: Towards Semantic Tuplespace Computing: The Semantic Web Spaces System. In: 22nd Ann. ACM Symposium on Applied Computing (March 2007)
30. Omicini, A., Zambonelli, F.: Coordination for Internet Application Development. Autonomous Agents and Multi-Agent Systems 2(3), 251–269 (September 1999)
31. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF. W3C Working Draft (October 2006)
32. Shafiq, O., Toma, I., Krummenacher, R., Strang, Th., Fensel, D.: Using Triple Space Computing for communication and coordination in Semantic Grid. In: 3rd Semantic Grid Workshop (16th Global Grid Forum) (February 2006)
33. Stickler, P.: CBD - Concise Bounded Description. W3C Member Submission (September 2004)
34. Tolksdorf, R.: Laura — A service-based coordination language. Science of Computer Programming 31(2/3), 359–381 (July 1998)

Requirements for Routing in the Application Layer

Pamela Zave

AT&T Laboratories—Research, Florham Park, New Jersey USA
pamela@research.att.com

Abstract. In the application layer of networks, many application servers are middleboxes in the paths of messages from source to destination. Applications require, as a basic coordination mechanism, a way to route messages through the proper servers. This paper elaborates and justifies the requirements for such a coordination mechanism. It presents what is known about satisfying these requirements, and what questions still need to be answered.

1 Routing as a Coordination Mechanism

In any networked application, routing is a fundamental execution mechanism. When a node sends a message, routing determines which node will receive it.

The most familiar form of routing is routing for the network layer of the protocol stack, especially routing according to the "classic" Internet architecture. From this viewpoint, the sole purpose of routing is to get a message to its destination.

The network literature typically distinguishes between *routing*, meaning the process by which routes are advertised and local routing tables are maintained, and *forwarding*, meaning the step in which a router receives a message, looks its destination up in a table, and sends the message out again. Because this paper is concerned more with requirements than with mechanisms, there is no need to distinguish the two concepts. Both routing and forwarding are lumped together as "routing."

The main point of this paper is that applications require routing to serve a purpose in addition to getting a message to its destination. Application servers are often middleboxes that can only do their jobs if messages pass through them on their way from source to destination. Consequently, there should be an application-layer concept of routing whose purpose is to include appropriate application servers in the paths of messages, as well as to get them to their destinations. This form of routing *would serve as a coarse-grained coordination mechanism*, because it would govern both the inclusion and order of application servers in message paths.

Like most network concepts, routing can be hierarchical. Figure 1 shows how application routing would fit into the hierarchy. In the application layer, a message M_a passes through a middlebox on the way to its destination. The

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 19–36, 2007.

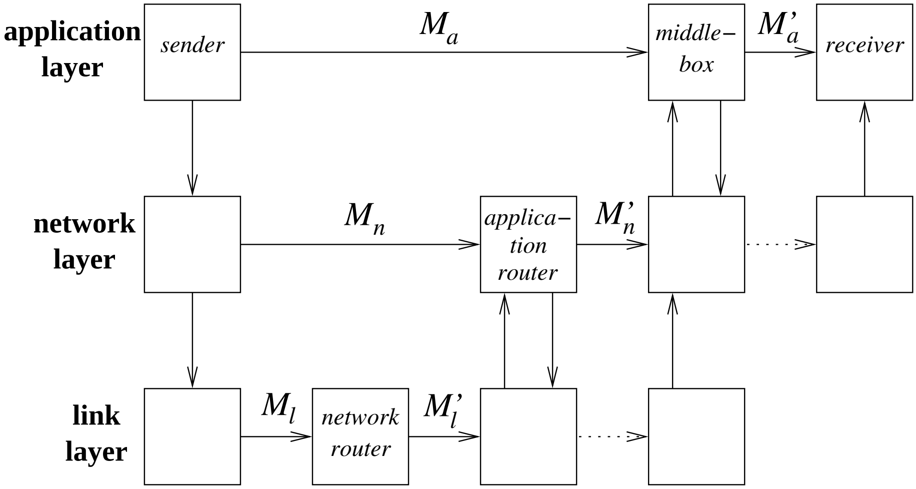


Fig. 1. Routing can be hierarchical, with application and network routing implemented in different layers. (The transport layer usually comes between the application and network layers in the protocol stack. It is omitted here because transport protocols interact little with routing.).

middlebox is a general application server, so it can modify the message to M'_a, absorb it, delay it, or replicate it.

In the network layer, M_a is encapsulated inside a message M_n that is routed to an application router by ordinary destination routing. The application router determines that M_a should be sent first to the middlebox. It changes M_n to M'_n before forwarding, so that ordinary destination routing will take it to the middlebox. A similar process goes on in the link layer, where network (IP) routers implement ordinary destination routing for the benefit of the network layer.

To set the stage for a discussion of exactly what application-layer routing should do, Section 2 gives four general reasons why an application server might be used as a middlebox rather than an endpoint. This establishes the importance of support for middleboxes.

Section 3 focuses on source/destination symmetry. This is the biggest difference between routing in the application and network layers, and hence the biggest unmet need in the application layer. Section 3 illustrates the bad effects of current deficiencies on service deployment, service maintenance, and security.

Section 4 introduces the routing capabilities of the Distributed Feature Composition (DFC) architecture [5]. DFC is a modular architecture for telecommunication services. It has been used successfully to build many voice-over-IP services, including corporate and consumer services in daily use [3,4]. DFC incorporates application routing in a way that is appropriate to telecommunications, and provides a good example of what application routing can offer to service developers. DFC routing has been accepted by the voice-over-IP industry, where it is part of a new standard [6] for programming application servers.

DFC routing may be a good start, but it is not sufficient to meet the needs of all Internet applications. Section 5 discusses additional requirements for routing in the application layer. It seems desirable to consider building middleware that can be shared among applications, but further research is necessary to design a sufficiently general capability.

Although this paper is focused on recommendations for the application layer of the Internet, it contains many examples from lower layers, particularly the network and link layers. This is because some of the application-layer concerns presented here are also relevant to lower layers, as described in [1] and [13]. Today's Internet has many deviations from the "classic" architecture, in which messages move transparently between endpoints. For example, a pragmatic definition of reachability in the Internet [14] combines the effects of routing, message filtering (primarily by firewalls), and Network Address Translators (NATs).

Despite the common themes found in all network layers, it seems best to focus on the application layer, for two reasons: (1) From a technical perspective, the arguments for enhanced routing are strongest and least controversial when applied to the application layer. (2) From a pragmatic perspective, the proposed application middleware would meet urgent needs of application builders, and incur relatively few obstacles to deployment. In lower layers, the need for change is not so obvious, and the obstacles to deployment are far greater.

2 Middleboxes in the Application Layer

In the application layer, as we would expect, some applications are provided by servers that act as the endpoints of message paths. Most Web servers function as endpoints.

The salient characteristic of the application layer, however, is that many applications are provided by servers that are not the endpoints of message paths. The view that servers can be middleboxes as well as endpoints is still somewhat controversial, because of the lasting influence of the classic Internet architecture. For this reason, it seems worthwhile to present in detail the motivations for servers as middleboxes. There are four such motivations, each discussed in one of the next subsections.

Arguments against middleboxes often invoke the end-to-end arguments [12], but these are really principles that distinguish between the network layer and the application layer. They say that the network layer should be minimal and highly efficient, because it is shared equally by all applications, and because it can perform few functions as well as application-aware software can. With this interpretation, there is no conflict whatsoever between the end-to-end arguments and application servers as middleboxes, because any application server is an "endpoint" with respect to the principles [13].

2.1 Being an Intermediary IS the Application

Some Internet applications have the purpose of acting as an intermediary between or among communicating endpoints. These applications can only be implemented by servers on the message paths between endpoints.

Intermediary applications perform many common functions, for example:

- They enhance security by blocking unwanted messages. This is the motivation for firewalls.
- They perform transcoding, protocol conversion, reformatting, or other functions that enable heterogeneous endpoints to communicate.
- They filter or transform content for particular audiences, for example, children or the disabled.
- They build multi-point connections out of point-to-point connections, and allow the endpoints to control them by switching and conferencing.
- They improve performance in application-dependent ways. For example, they cache Web pages.
- They improve reliability in application-dependent ways. For example, they implement automatic retrying or retargeting.

One of the most interesting categories of intermediary consists of servers representing third parties in the communication [2]. These servers might perform functions desired by the endpoints, such as acting as trusted brokers. Or they might act against the interests of the endpoints, for example by billing or wiretapping.

2.2 Servers Enhance Endpoints

Some network applications could conceivably be implemented in endpoints, but they are not implemented in endpoints for practical reasons.

For example, it obviously makes sense to put a voicemail capability in telephones, because most home answering machines work this way. When the endpoint device is a cellphone, however, there are important advantages to putting the voicemail capability in the network rather than in the device:

- Network voicemail provides an always-available network presence for an endpoint that is often unavailable.
- Network voicemail provides a large amount of persistent storage that is always accessible from any device.
- Network voicemail can employ speech recognition, speech search, and text-to-speech generation. Handheld wireless devices do not usually have access to the resources for such capabilities.
- Network software can be updated regularly, while small consumer devices do not usually have updatable software.

If a cellphone subscribes to voicemail in the network, then its calls must go through a middlebox that detects failure and redirects a failed call to a voicemail server.

Home-network applications can integrate two or more single-media devices into a single, virtual multimedia device. It would be very difficult to implement such an application within the devices themselves.

Finally, from a different perspective, service providers may wish to offer value-added communication services to consumers. They can only do this by implementing them in network servers and including these servers in message paths. If service providers can find a market for such services, it does not really matter whether the services could, in theory, be implemented in consumer endpoints.

2.3 Name Binding

In the application layer, there are many *names* serving many application-dependent purposes. An *address* is one of a node's many names, distinguished from others only by the fact that the network layer can route to it.

Applications create and use name spaces freely. Because of this, one of the most common and important functions performed by applications is to bind one name to another. The two names involved in an instance of name binding can differ in a large number of ways, for example:

- The first name can actually refer to a group of endpoints, while the second name refers to a member of the group. Similarly, the first name can be a service, while the second name is a server performing the service (as an endpoint).
- The first name can be published and long-lasting, while the second is the current network location of the endpoint. In other words, the endpoint is mobile.
- The first name can be public or anonymous or user-friendly, while the second name is private or secret or inscrutable.
- The first name is one of many roles or aliases employed by the owner of the second name.
- The first name can be global, while the second is local to a subnetwork. Similarly, the first name can be local to one subnetwork while the second is local to another subnetwork. This type of name binding is performed by NATs and by gateways between networks with different address spaces.

The kind of name binding that comes first to most peoples' minds is binding of global names by means of universally accessible lookup services such as DNS, called *lookup binding* here. Lookup binding supports location-independent names. Several additional lookup name spaces, serving different purposes within the Internet architecture, have been proposed [9,13]. Lookup binding is also part of popular peer-to-peer applications for file sharing and communication, each of which creates a global name space for its own users.

There is another way to bind names in the application layer, termed *path binding* to distinguish it from lookup binding. The simplest form of path binding is shown in the left half of Figure 2. The first name is the address of a server, here *a*, and the sender uses it as the destination field of a message. The addressed server itself binds this first name to a second name *a'*, changes the destination

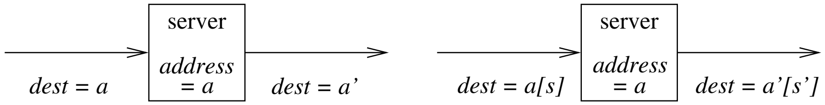


Fig. 2. *Path binding* is name binding performed by a server in the message path. The name to be bound is, or includes, the address of the server.

address of the message to *a'*, and forwards it. Path binding is different from lookup binding because the binding server is a middlebox in the message path between sender and receiver.

The right half of Figure 2 shows a simple variation on path binding in which a name consists of both an address *a* and a free-form string *s* encapsulated in the message. The address part gets the message to the binding server, while both parts contribute to the choice of second name. The properties of path binding are analyzed in [17].

Path binding is extremely common. The right half of Figure 2 depicts forwarding in a NAT, with *a* and *a'* being IP addresses, and *s* and *s'* being port numbers. If a link interface in an IP router is regarded as an implicit address, then all IP forwarding is path binding in the link layer.

Path binding is used in the application layer, in preference to lookup binding, in two situations. First, a path-binding server at address *a* need only bind names having *a* as their address part, rather than knowing how to bind all names in a namespace. Hence path binding is more local and easier to deploy than lookup binding. For example, Mobile IP [10] uses path binding to bind published mobile addresses to current network locations. This means that each "home agent" binds only its own address to the current location of its corresponding mobile endpoint, and need know nothing about other mobile endpoints.

The second situation in which applications use path binding is when they need to include an application server in a message path, and have no other mechanism with which to accomplish it. They introduce an artificial name binding for the purpose of including the server, rather than including a server for the purpose of performing a name binding. This situation is discussed further in Section 3.

2.4 Software Composition

Application servers are, among other things, modules of software. *Composition*—assembling complex software by composing simpler software modules—is how we make software development "scale up."

The final motivation for using servers as middleboxes is that this provides a valuable mechanism for software composition. When one or more servers sit on a message path between two endpoints, then the relevant software of each server is composed with the software of the other servers and the software of the endpoints in a pipes-and-filters configuration. Pipes-and-filters composition makes it relatively easy to augment or change an existing application by adding to or changing the servers in message paths.

Pipes-and-filters composition is evident in the deployment of proxies and reverse proxies between clients and Web servers. It is implicit in the use of firewalls, NATs, gateways, and other common network elements. The principle purpose of DFC (Section 4) is to support pipes-and-filters composition of telecommunication features.

Multiplayer games are a rapidly-growing application area that illustrates many of the themes in this section. For such games to be playable, they must satisfy stringent requirements for scalability, latency, and fairness. Achieving this on a global scale will probably require hierarchies of middleboxes. These hierarchies will be carefully engineered to optimize performance and arbitrate fairness.

3 Source/Destination Symmetry

In the network layer, the sole purpose of routing is to find the destination of a message. A router can be thought of as a server working on behalf of each destination, helping to get its messages delivered. There is no need to "find" the source of the message, and there are no servers acting on behalf of the source.

In the application layer, there are also servers that work on behalf of destinations, helping their messages to find the destinations. With this important exception, most reasons for including an application server in the path of a message are potentially symmetric. If there is a destination-related reason for including a server in a message path, probably there is a corresponding source-related reason.

This is why Web terminology includes "proxies," associated with clients, and "reverse proxies," associated with servers. In the world of Web technology, there are reverse proxies for security (firewalls) and performance (load balancing). There are proxies for security (anonymizing, blocking access to some sites) and performance (caching). There are also client-side (source-related) proxy functions that have no destination counterparts; these include reformatting for special devices and filtering out annoying forms of advertisement.

In telecommunications, which is a peer-to-peer service, symmetry is even more prominent. Once a telephone call has been established, its two parties are equal. Either party (or both parties) may wish to have features that perform switching, transfering, recording, or other mid-call functions.

Source/destination symmetry is the biggest difference between routing in the application and network layers. Because the network layer has little need for source-related servers, there is poor support for them in any layer.

A possible mechanism for including application servers on behalf of the source is *source-subscription routing*. In source-subscription routing, an address *a* can *source-subscribe* to another address *a'*. If there is such a subscription, then a message with source address *a* is first routed to the node at address *a'*. If this node is a server and chooses to forward the message, then the forwarded message is next routed to its destination address in the ordinary way. Obviously, an implementation of source-subscription routing requires a bit of history in the

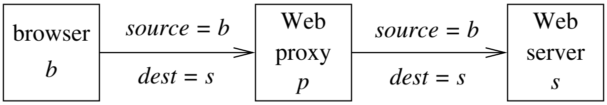


Fig. 3. Including a Web proxy in the path of an HTTP request, by means of source-subscription routing. Address b source-subscribes to address p.

message, to distinguish the hop originating at the sender from the hop originating at the server.

It is common for network administrators to want all browsers in their subnetworks to make requests through a particular Web proxy. Figure 3 shows the use of source-subscription routing to meet this goal. Address *b* of the browser *source-subscribes* to address *p* of the proxy, which means that every message with *source* = *b* is routed to *p* before it is routed to its destination. Note that no address translation is required to deliver the message to the Web server through the proxy.

In the absence of source-subscription routing, the common solution to the proxy problem is for the browser to send its original HTTP request with *destination* = *p*, and *s* encapsulated in the message. When the proxy forwards the message, it changes the destination to *s*. The proxy can be regarded as performing path binding, binding name *p[s]* to *s*. This is an example of using path binding to include a server in a message path rather than to perform "real" name binding, as mentioned in Section 2.3.

This common solution has two major deficiencies. The first is that every browser in the subnetwork must be configured to use *p*; ensuring this is well known to be a problem for administrators. The second is that the solution relies on the cooperation of the browser, which may not be forthcoming if the interests of the browser's user differ from the interests of the administrator. Source-subscription routing avoids both the configuration problem and the security problem by keeping both information and enforcement away from the browser.

As an alternative to the use of path binding to solve the proxy problem, one might think of putting a routing list or stack *[p, s]* in the message. Theoretically, this is provided for by IP source routing, although source routing is disabled in most subnetworks. In any case, a routing list has the same configuration and security deficiencies as the use of path binding.

For a second example of the use of source-subscription routing, consider the problem of making a connection to a mobile endpoint, as shown in Figure 4. Address *M* is the published mobile address of the endpoint, and address *m* is its current network location. The requestor of the connection knows the endpoint only by *M*. Ordinary destination routing and path binding enable the request message to pass through the location server for *M* and be delivered to *m*.

The need for source-subscription routing arises when the mobile endpoint replies to the request. The correct source address for the reply is *m*, yet the initiator must receive the message with *source* = *M*. This will enable the initiator

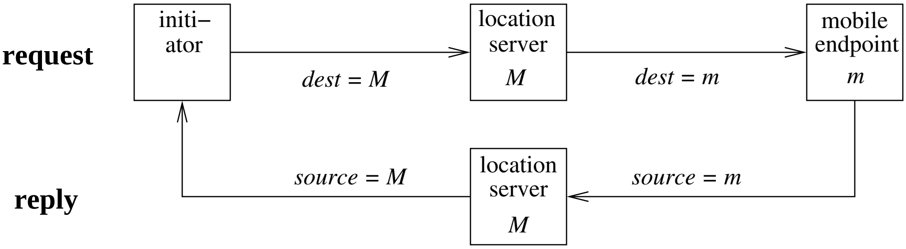


Fig. 4. Making a connection to a mobile endpoint, by means of source-subscription routing. Address m source-subscribes to address M.

to identify the message as a reply to its request. Furthermore, the initiator must continue to send all messages within the connection with *destination* = *M*. This is necessary so that each message from the initiator goes through the location server and is directed to the current location of *M*, which may be a new address *m'* rather than *m*.

With source-subscription routing, a current location *m* source-subscribes to *M* or some other suitable server address. The source server receives the reply and changes *m* to *M* in the source field before forwarding.

This design is very similar to Mobile IP, except that Mobile IP does not have the advantage of source-subscription routing. In its absence, the mobile endpoint sends its reply with *source* = *M*. In most cases, *M* does not belong to the address space of the subnetwork of *m*. Hence the reply is discarded by a local firewall that performs ingress filtering. This has been the major obstacle to the deployment of Mobile IP [10].

A mobile endpoint can request a connection as well as accept a request for a connection, then move during the lifetime of the connection. In Figure 4, if the mobile endpoint were the requestor of the connection, source-subscription routing would apply to the request message. It would be routed throught the endpoint's location server, and its source address *m* would be changed to *M*.

Figure 4 has some similarities to the deployment of NATs, where *M* would be the single public address of the subnetwork behind the NAT, and *m* would be a private address within the subnetwork. In the NAT case, port numbers would be used to distinguish among the many private addresses represented by *M*, as in the right half of Figure 2.

The NAT case is interesting here because the reply message from *m* is routed through *M* by yet another mechanism. It is clearly not IP source routing or source-subscription routing. It is not ordinary destination routing, because *M* is not found in the destination field of the message. Rather, the mechanism is an assumption that lower layers are configured so that a message *cannot* travel from *m* to the public Internet except by going through the NAT.

Arguably, NATs are active agents in all of the link, network, and transport layers, which accounts for the complexities that arise from them. This kind of layer spanning is neither possible nor desirable for typical applications, which may have no special administrative privileges, and which should be organized for easy maintenance. We return to the NAT example in Section 5.

4 Routing in the DFC Architecture

As mentioned in previous sections, the Distributed Feature Composition (DFC) architecture supports pipes-and-filters composition of software modules. These modules are intended to implement individual, user-controllable capabilities known as *features*. In practice, the unit of composition can be a whole application server or a software module ("virtual server") within an application server. DFC uses the term *box* to encompass application servers (whether real or virtual) and endpoints.

This section summarizes the capabilities of DFC routing. The emphasis is on routing behavior and the goals it satisfies, rather than on implementation; the implementation is straightforward once the behavior is understood.

4.1 Message Paths

The full path of a message is defined implicitly by the methods used in boxes. A box uses the *new* method to send a new message. A box uses the *continue* method to forward an existing message.

A full message path (see Figure 5) has a *source region* in which it is routed to boxes on behalf of the source, and a *target region* in which it is routed to boxes on behalf of the target.[1] When both source and target regions are exhausted, it is routed to its target in the ordinary way.[2]

Within the source region, boxes are included because of source-subscription routing. Within the target region, boxes are included because of target-subscription routing. Thus a message can be routed to a box because the target address subscribes to the box, then (when it is forwarded by the box) be routed to another box whose network address is the target.

The source or target subscription of an address is actually a list of boxes rather than a single box (the list can be empty). The boxes are routed to in the order listed. The boxes in a message path present because of the source (target) subscription of a single address are called the *source (target) zone* of the address.

When a box continues (forwards) a message, it can change the source or target address. If the box lies in the source region and changes the source address, then any remaining boxes in the source subscription of the original source address are omitted from the message path. Rather, routing immediately begins to traverse the source subscription of the new source address. For example, in Figure 5, the source-subscription of address *s1* is [A, B, C]. Box B changes the source address to *s2*, however, so rather than going to box C, the message is next routed to D. Similarly, if a box in the target region changes the target address, then the

[1] DFC terminology uses "target" instead of "destination", for brevity.

[2] It is also possible to define a *network region* between the source and target regions, for boxes that provide network functions such as billing.

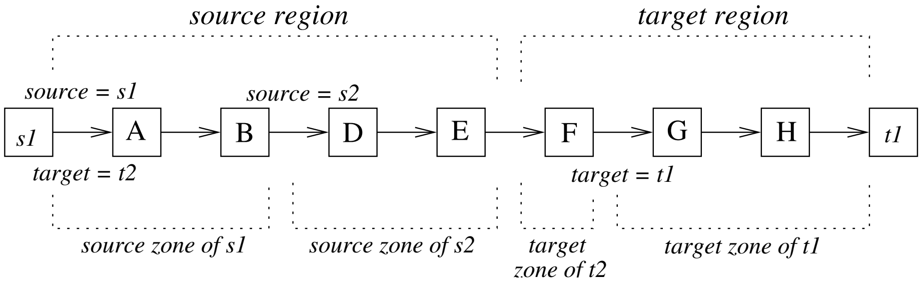


Fig. 5. A message path created by DFC routing

next box in the path is the first box of the target subscription of the new target address.

We are already familiar with target-region boxes that change the target address, because this is just another form of path binding. It is also useful, however, for source-region boxes to change the source address. For example, a box could change the original source address, which is the address of the calling device, to a personal address associated with the person sending the message. This has two benefits. First, the personal address is a better identifier of the sender for the benefit of the receiver. Second, the message can then be routed through boxes subscribed to by the personal address, thus utilizing their functions.

DFC routers must have access to the relevant subscription data. The necessary routing history of an individual message is carried along in the message, so that routers do not maintain state at that level. The routing history may be encrypted so that boxes cannot read it.

4.2 Composition, Modularity, and Additive Authority

The path mechanisms in the previous section meet the application-layer requirements of composition, modularity, and additive authority.

Composition dictates that there is no such thing as a unique server. If there is a reason why one server should be included in a message path, then the same reason might also apply to multiple servers. DFC routing supports composition in two ways. First, a subscription can be a list of any length, so that a zone can contain any number of boxes (servers). Second, a region can contain any number of zones.

Modularity dictates that servers should not know or need to know which other servers are present in the message path. When this requirement is satisfied, each server can be developed independently. Also, server configurations can be changed easily, because the servers themselves need not change. DFC routing was designed very specifically to satisfy this requirement.[3]

[3] This simplified view of modularity is appropriate for discussion of routing, which provides coarse-grained coordination. Finer-grained coordination may require analysis and management at the protocol level, as exemplified by [15].

The most common violation of the modularity requirement is the use of path binding to include servers in message paths (Sections 2.3 and 3). Often the purpose of these servers has nothing to do with name binding. If multiple servers are required, and if path binding is the only available means to include them, then every server must know the address of the next server in the path.

Composition and modularity can be illustrated with Web proxies. A user might wish to use all three of these proxies:

- A text-to-speech "edge service" proxy, which improves Web access for the visually impaired.
- Privoxy, which is a filtering proxy that can remove advertisements and other "Internet junk."
- A Tor client. The Tor client encrypts and chooses an anonymized, random route through participating Tor nodes to the requested server. The purpose of Tor is to protect the user's privacy, particularly from attacks by traffic analysis.

The proxies must be applied to each HTTP request in the order shown in Figure 6. Because the text-to-speech proxy gets each request before Privoxy, it gets each Web page after Privoxy has filtered it. Because Privoxy gets each request before the Tor client, it gets each Web page after Tor has decrypted it.

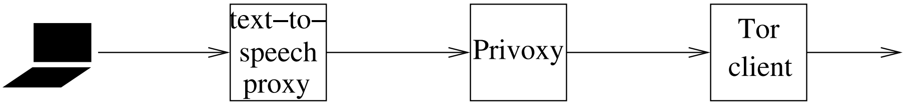


Fig. 6. Using multiple Web proxies

With DFC routing, it would be easy to deploy the proxies in this way. It would also be easy to make Tor optional for each such user, by allowing the user to toggle his subscription to the Tor client. This is valuable because fetches through Tor are inevitably slow.

With current technology, on the other hand, each proxy must be configured to send requests to the next one. It is impossible to remove Tor from the message path without altering the configuration of Privoxy. If Privoxy is running as a shared server in a local-area network, then it is impossible to apply Tor to the requests of some users and omit it from the simultaneous requests of other users.[4]

Finally, *additive authority* means that a server should be included in a message path if one of the endpoints desires it *or* if the relevant administrator desires it. As a general security requirement, it should be easy for an administrator to ensure that all messages of a particular type go through a particular security server. DFC routing supports additive authority because our implementations allow both parties to contribute to the subscription lists. In practice, almost every subscription has contributions from both parties.

[4] I am indebted to Trevor Jim for this example.

4.3 Usages

DFC is connection-oriented. For this reason, the only messages that are routed as described above are the messages that request connections. Within each inter-box hop of a request message, the reply message forms an independent box-to-box connection. All subsequent signaling takes place within these connections.

DFC boxes and signaling connections form dynamic graphs called *usages*. DFC usages have many interesting properties and behaviors, some of which are illustrated by the usage in Figure 7.

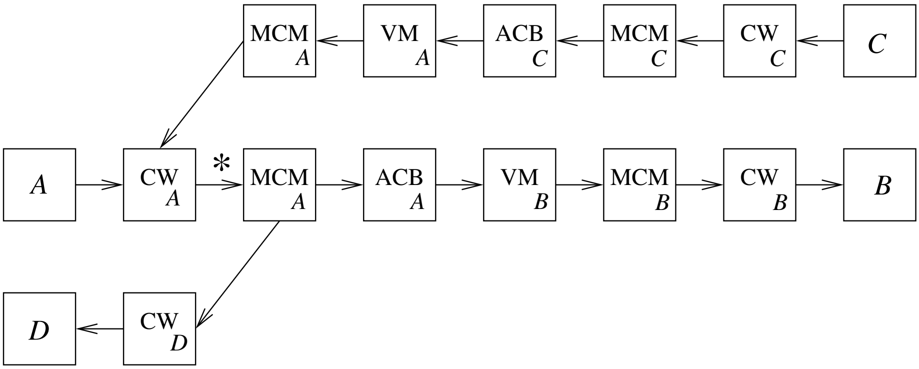


Fig. 7. DFC usages are dynamic graphs that evolve over time

There are four user endpoints: *A, B, C,* and *D*. DFC subscriptions actually name *types* of boxes rather than boxes themselves. Each of the four endpoints subscribes to the same box types, namely [CW, MCM, ACB] in the source region, and [VM, MCM, CW] in the target region. The acronyms stand for Call Waiting, Mid-Call Move, Automatic CallBack, and Voice Mail, respectively. Voice Mail is a target-side failure treatment: if the endpoint cannot be reached, redirect the call to a voicemail server. Automatic CallBack is a source-side failure treatment: if an attempt to place a call fails, remember it and try again later.

DFC maps between box types and boxes as follows. There are two categories of box type, *free* and *bound*. When a DFC router needs to route to a box of a free type, it simply creates a fresh new instance of the box type. In contrast, there is only one instance of a bound box type for each address that subscribes to the type. When a DFC router needs to route to a box of a bound type, it routes to the unique instance associated with the relevant address. In this example, Call Waiting is a bound type, and all others are free.

In Figure 7, *A* has placed a successful call to *B*. Each box instance is labeled with the address on behalf of which it is included in the usage. The arrows show the direction that the request message traveled to set up these connections.

In Figure 7, *C* has also placed a call to *A*. This call is also successful, because *A* has Call Waiting, which accepts the call. In this example, CW and MCM are *reversible* box types, which means that a subscriber subscribes to them in both regions, because they are useful to both callers and callees. Reversible box types are the ultimate example of source/destination symmetry. Although the request from A to B was routed to CW_A in the source region, the request from C to A is routed to CW_A in the target region. Now A has a signaling connection to both B and C, and can switch the voice channel so as to talk to either B or C at any moment.

A represents a user's home phone, and D represents the same user's cellphone. While A was connected to C and talking to B, the user noticed that it was time to go to work. He invoked the function of Mid-Call Move to move the B conversation to his cellphone, resulting in the third row of the figure. In the next instant the Mid-Call Move box will drop the connection marked with an asterisk, resulting in two separate usages. The user can hand telephone A to another family member to continue talking to C. Alternatively, hanging up telephone A will tear down the entire usage to C.

This is a specific example of a general pattern, which is that a long-lasting connection can be made up of connection segments that were not set up at the same time or in the same direction. The connection from A to C has segments set up from A to CW_A and from C to CW_A. The connection from B to D has segments set up from MCM_A to B and from MCM_A to D. Routing correctly in this situation requires a third method: in addition to forwarding with the *continue* method, a box can forward with the *reverse* method [16].

Similar structures might also arise in automotive infotronics, in which most communication consists of streams of sensor/actuator data. A sensor should be engaged in *some* end-to-end connection at all times, to ensure that critical data is not being lost, which means that some connection segments will be very long-lasting. At the same time, the exact configuration of data streams and functional components will vary with the vehicle's current mode of operation.

Despite the subtlety of usages, it is possible to prove useful properties about them [16]. For example, *any* connection between two endpoints X and Y with these same subscriptions contains the following subsequence of box instances: [CW_X, MCM_X, MCM_Y, CW_Y].[5] This invariant is true regardless of how the connection was initially formed or how it has evolved over time.

5 Unmet Requirements and Future Work

This section considers the possibility of building middleware that performs routing for the application layer. It would be deployed in the network as shown in Figure 1. It would make applications easier to build, deploy, maintain, and improve. It would also make them more secure.

This middleware should be shared among applications, amortizing its cost and, more importantly, facilitating convergence among applications. The value of current Internet applications is greatly reduced by the fact that each one

[5] After the instance of MCM_A moves the connection to D, we consider it an instance of MCM_D. It can be signaled from D to move the connection again.

tends to be an isolated fiefdom. Furthermore, this isolation limits imagination and innovation, as it prevents us from seeing the potential relationships among applications.

5.1 Deficiencies of DFC Routing

It seems that the DFC routing capability is a good start toward meeting the routing needs of Internet applications, but it is not sufficient. This section discusses its deficiencies for this purpose, and the research needed to remove the deficiencies.

DFC routing was designed for the single application of telecommunications. It has been amply illustrated that the principles of DFC routing are relevant for other applications, taken individually. Now we need to understand how routing could contribute to application convergence, which might mean routing messages through servers associated with different applications.

DFC routing was also designed for a single administrative domain. Multiple administrative domains are a fact of life in the Internet, and there are many questions about how DFC routing should be extended to include them. For example, consider the source and target regions in Figure 5. If the message path extended across multiple administrative domains, would it still look like this, or would there be a source and target region of the message path for each domain? Interestingly, there are good arguments for both answers to this question.

All the other deficiencies of DFC routing come down to efficiency in one way or another. As noted in Section 4, DFC routing is most often used to route messages to virtual servers within a physical application server. In this context, efficiency is not critical, and has received correspondingly little attention.

First, DFC routing often routes to boxes that will never be activated in this particular communication. These boxes simply behave transparently throughout their lifetime. The inefficiency of including transparent servers in message paths could be reduced by offering finer-grained selection criteria.

For an example, let us return to the NAT example at the end of Section 3. Say that a node with private address m is replying to a connection request from the open Internet. We want to use DFC routing to route the reply through a NAT, which will change the source address from private m to public M. Source-subscription routing is not very good for this purpose, because it will route *every* message from m to the NAT. If the message is destined for another node within the same subnetwork, then the NAT must behave transparently. An optimal subscription mechanism would route the message to the NAT only if the source is local to the subnetwork *and* the destination is not.

Second, a DFC request message goes through a DFC router each time it is forwarded. There should be a way to make the router visit optional, for situations in which it is not required. Note, however, that the mandatory router visit is an important security mechanism. Any optimization must be carefully designed to leave the security intact, allowing only trusted parties to circumvent it.

Finally, all DFC messages other than requests retrace the exact paths laid down by requests, with no servers omitted. Yet there are many situations in which subsequent messages could take a more direct route. The Session Initiation Protocol [11] allows the reply to a request to skip servers by distinguishing between *record-route* proxies and others. Even with DFC-style signaling to control IP media channels, the media packets themselves can travel directly from end to end [18].

5.2 Related Work

In the current Internet architecture, the only mechanisms available for influencing or altering ordinary destination routing are IP source routing, underlying network topology, path binding, and lookup binding. None of these as currently used is secure, robust, and general enough to meet the full spectrum of application requirements.

In [1], lookup binding is used to achieve the effective equivalent of destination subscription routing in DFC: a sender looks up the destination (a global, location-independent identifier), yielding a route to traverse to get to the destination. Section 2.3 showed that lookup binding and path binding are architecturally different ways to bind names. Section 1 showed application-level routing as implemented by path binding at a lower level. The use of lookup binding in [1] shows us that lookup binding is another implementation possibility for application-level routing.

There are several well-known middleware systems to support each of the publish/subscribe architecture and grid computing. There are also middleware systems for other coordination architectures such as distributed tuple spaces, e.g., [8]. In general these architectures are not concerned with middleboxes in message paths. As a result, their routing activities tend to be overlays on ordinary destination routing. For example, a grid architecture can be used to locate a desired resource for a potential client of that resource. The location is a destination address, which the client proceeds to use in the ordinary way.

Service-oriented architecture is such an active area that there are several service-oriented architectures. At the simpler end, as above, the architecture helps to locate a service for a client. Once located, the service is reached by ordinary destination routing. At the more complex end, "choreography" languages such as WS-CDL are used to create global descriptions of distributed Web-based services. As this technology matures, we will see whether choreography is compatible with DFC routing, or whether it requires tight logical coupling between servers. In the latter case, the compositional freedom supported by DFC routing would not be required or even allowed.

A new effort to formalize what routers (in the most general sense of the term) do promises to be relevant to the application layer [7]. In particular, it can help to define the space from which potential optimizations can be chosen.

5.3 Future Work

Before we can propose a specific middleware system for routing in the application layer, three broad open questions must be answered.

First, the most powerful routing functions are expensive in terms of visits to servers and routers. We must understand how to provide a large range of cost/function trade-offs, and how to guide users in making choices.

Second, the proposed new middleware must support convergence of different applications, and it must compose well with middleware for other purposes. These are areas in which there is little general knowledge, and much research to be done.

Third, we must understand how to extend the current routing scheme to multiple administrative domains. Both routing and subscription mechanisms must be examined from a security perspective.

6 Summary

This paper has explained why routing through middleboxes is an important coordination mechanism in the application layer, and justified the claim with examples from Web services, home networks, telecommunications, mobile IP, automotive infotronics, and multiplayer games.

The current mechanisms available for influencing routing in the Internet are not general enough to meet the needs of applications, nor do they enhance security or facilitate the deployment and maintenance of applications. The biggest gap is caused by the fact that applications have a great deal of source/destination symmetry, while routing at the network level is focused exclusively on the destination.

The application-specific routing capability of the DFC architecture is a better model, and meets many requirements that are shared by all applications. However, the fact that it was designed for a specific application means, inevitably, that some requirements of different applications are neglected. This paper identifies the deficiencies and explains the questions that must be answered before the routing requirements of all applications can be met.

References

1. Balakrishnan, H., Lakshminarayanan, K., Ratnasamy, S., Shenker, S., Stoica, I., Walfish, M.: A layered naming architecture for the Internet. In: Proceedings of SIGCOMM ‘04, ACM, New York (August 2004)
2. Blumenthal, M.S., Clark, D.D.: Rethinking the design of the internet: The end-to-end arguments vs. the brave new world. ACM Transactions on Internet Technology 1(1), 70–109 (August 2001)
3. Bond, G.W., Cheung, E., Goguen, H.H., Hanson, K.J., Henderson, D., Karam, G.M., Purdy, K.H., Smith, T.M., Zave, P.: Experience with component-based development of a telecommunication service. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) CBSE 2005. LNCS, vol. 3489, pp. 298–305. Springer, Heidelberg (May 2005)
4. Bond, G.W., Cheung, E., Purdy, K.H., Zave, P., Ramming, J.C.: An open architecture for next-generation telecommunication services. ACM Transactions on Internet Technology 4(1), 83–123 (February 2004)

5. Jackson, M., Zave, P.: Distributed Feature Composition: A virtual architecture for telecommunications services. IEEE Transactions on Software Engineering 24(10), 831–847 (October 1998)
6. JSR 289: SIP Servlet API Version 1.1. Java Community Process Early Draft Review, `http://www.jcp.org/en/jsr/detail?id=289` (2007)
7. Karsten, M., Keshav, S., Prasad, S.: An axiomatic basis for communication. In: Proceedings of the Fifth Workshop on Hot Topics in Networks. ACM SIGCOMM (2006)
8. Murphy, A.L., Picco, G.P., Roman, G.-C.: Lime: A coordination model and middleware supporting mobility of hosts and agents. ACM Transactions on Software Engineering and Methodology 15(3), 279–328 (July 2006)
9. O'Donnell, M.J.: Separate handles from names on the Internet. Communications of the ACM 48(12), 79–83 (December 2005)
10. Perkins, C.E.: Mobile IP. IEEE Communications (May 1997)
11. Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., Schooler, E.: SIP: Session Initiation Protocol. IETF Network Working Group Request for Comments, 3261 (2002)
12. Saltzer, J., Reed, D., Clark, D.D.: End-to-end arguments in system design. ACM Transactions on Computer Systems 2(4), 277–288 (November 1984)
13. Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., Shenker, S.: Middleboxes no longer considered harmful. In: Proceedings of the Sixth Usenix Symposium on Operating Systems Design and Implementation, ACM, New York (December 2004)
14. Xie, G., Zhan, J., Maltz, D.A., Zhang, H., Greenberg, A., Hjalmtysson, G., Rexford, J.: On static reachability analysis of IP networks. In: Proceedings of IEEE Infocom, IEEE, (March 2005)
15. Zave, P.: An experiment in feature engineering. In: McIver, A., Morgan, C. (eds.) Programming Methodology, pp. 353–377. Springer, Heidelberg (2003)
16. Zave, P.: Ideal connection paths in DFC. Technical report, AT&T Research (November 2003)
17. Zave, P.: Compositional binding in network domains. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 332–347. Springer, Heidelberg (2006)
18. Zave, P., Cheung, E.: Compositional control of IP media. In: Proceedings of the Second Conference on Future Networking Technologies. ACM SIGCOMM (2006)

Context-Aware Publish Subscribe in Mobile Ad Hoc Networks

Davide Frey and Gruia-Catalin Roman

Department of Computer Science and Engineering
Washington University in St. Louis
{frey,roman}@cse.wustl.edu

Abstract. The publish-subscribe communication paradigm is enjoying increasing popularity thanks to its ability to simplify the development of complex distributed applications. However, existing solutions in the publish-subscribe domain address only part of the challenges associated with the development of applications in dynamic scenarios such as mobile ad hoc networks. Mobile applications must be able to assist users in a variety of situations, responding not only to their inputs but also to the characteristics of the environment in which they operate. In this paper, we address these challenges by extending the publish-subscribe paradigm with the ability to manage and exploit context information when matching events against subscriptions. We present our extension in terms of a formal model of context-aware publish-subscribe. We propose a solution for its implementation in MANETs; and finally we validate our approach by means of extensive simulations.

1 Introduction

Publish-Subscribe has emerged as a communication paradigm able to facilitate the development of complex distributed applications in open network environments. The strong decoupling it introduces between communication parties enables applications to *publish* information without being aware of the identities of potential receivers or even of their existence. Similarly, it enables receivers to issue *subscriptions* that express their interests in messages with a given content regardless of the identity of their publishers. These characteristics make the paradigm well suited to scenarios where the set of communicating parties is subject to frequent changes as in Mobile Ad Hoc Networks (MANETs).

Consider the problem of disseminating traffic information in a network of vehicles. Messages need to be routed to vehicles regardless of their identity and based on the interests expressed by their drivers. A vehicle moving on a freeway might, for example, be interested in messages announcing traffic accidents or slowdowns between its current location and its intended exit. A vehicle approaching its destination might want to be notified about the availability of new parking spots in its vicinity. Similarly, a vehicle running out of fuel will request information about available gas stations. The publish-subscribe paradigm is a natural fit for these types of scenarios. Vehicles witnessing an accident, leaving their parking

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 37–55, 2007.

spots, or observing other relevant events can *publish* such information without having to know if there are any vehicles interested in receiving it.

The appropriateness of the publish-subscribe paradigm for mobile scenarios has motivated researchers to investigate routing techniques for the dissemination of events and subscriptions over dynamic network topologies. However, these techniques solve only part of the issues related to programming applications in mobile networks. Mobile applications are required to assist users in a variety of situations, responding not only to their inputs but also to the characteristics of the environment in which they operate. In the above example, a vehicle needs to be notified of available parking spots only when it approaches its destination and not while it is still on the freeway. Likewise, the interest in a specific parking spot increases or decreases depending on its location, or on the presence of other vehicles also looking for a parking place in its vicinity. Similarly, vehicles that enter a freeway after an accident has occurred should be notified about the accident even if the accident happened some time ago. Traditional publish-subscribe middleware offers only limited support for these aspects: for example events are not generally associated with any notion of persistence and propagate instantaneously through the network. This forces developers to deal with contextual aspects at the application level preventing the middleware from exploiting context information to optimize the dissemination of events.

In this paper, we address these limitations and develop a new kind of middleware that integrates the publish-subscribe paradigm with the requirements of context-aware mobile applications. Such an undertaking poses significant intellectual and technical challenges. Managing context at the middleware level requires a richer set of primitives than those available in current publish-subscribe implementations. Our middleware extends the publish-subscribe API and enriches events and subscriptions with notions of space, time, and, more generally, with the context information associated with publishers and subscribers. Publishers can thus constrain the diffusion of events by specifying that each event is relevant and/or visible only in a given context. In addition, they can exploit the time dimension and define persistent events that should remain available for a specified time after their publication. Similarly, subscribers can subscribe to events that are relevant in specified context domains and originate at publishers belonging to a particular context.

The paper is structured as follows. Section 2 presents our model of context-aware publish-subscribe. Section 3 proposes a routing and matching protocol for context-aware publish-subscribe in MANETs. Section 4 evaluates its performance by means of simulation. Section 5 places our work in the context of related efforts, and Section 6 concludes the paper.

2 Bringing Context into Publish-Subscribe

We introduce our extension to the publish-subscribe paradigm in the form of a basic formalization. This allows us to present our notion of context and discuss how it may be integrated into the publish-subscribe communication paradigm.

2.1 Basic Definitions

We assume a universe consisting of hosts that move freely through the physical space $\mathcal{S}$. Each host is characterized by a unique identifier $h \in H$, a vector of attribute-value pairs $a \in A$, and its location in space, $\lambda \in \mathcal{S}$. For the time being, we ignore the aspects related to communication between hosts and return to them in Section 3, in the description of our routing and matching protocol.

System Configuration. At every instant in time, we can describe the configuration of the system in terms of the distribution of hosts in the physical space and the values assigned to the attribute vectors associated with each host. We model this information by defining a *system configuration* as a pair consisting of two functions: the *spatial distribution* and the *system state*. The former expresses the geographical aspects of the system configuration by associating each host with its current geographical location. The latter captures its non-geographical aspects and associates each host with its current vector of attribute-value pairs.

In mathematical terms, we represent the spatial distribution and the system state, respectively, as two functions $f_s \in \mathcal{C}_s$ and $f_c \in \mathcal{C}_c$, where

$$\mathcal{C}_s = (H \rightarrow \mathcal{S}) \text{ and } \mathcal{C}_c = (H \rightarrow A).$$

Based on these definitions we model the system configuration as a pair $c = (f_s, f_c)$ where

$$c \in \mathcal{C} = \mathcal{C}_s \times \mathcal{C}_c.$$

Configuration Function. Both the geographical and non-geographical aspects of a system's configuration continuously vary through time due to the movement of hosts and due to changes in the hosts' attribute values. To model this dynamic aspect, we consider a linear notion of time with values from the set T. This allows us to capture the history of the system's evolution over time by means of a *configuration function* that associates each time instant $t \in T$ with the corresponding system configuration.

$$\mathcal{K} : T \rightarrow \mathcal{C}$$

2.2 Context Specifications

Informally, we can define context to include those aspects of the state of the environment that can affect a particular entity, henceforth called the *reference host*. In an environment like a mobile ad hoc network, it is natural to define context as *a set of hosts and their associated properties that are of interest to a given reference host, i.e., hosts that can affect its behavior, can communicate with it, or can carry out activities on its behalf*. This notion of context is reasonable in a MANET because, in the absence of any fixed infrastructure or dedicated servers, all information must be associated with one or more mobile hosts.

The hosts associated with a particular context can rarely be specified by explicit enumeration because it is often impossible to know in advance which hosts may be of interest to a given reference host, or even which hosts are in the system at a particular point in time. Thus, we introduce the notion of *context specification*. Context specifications allow a reference host to identify the hosts that are part of some context in terms of the properties they must have as individuals or as a group. *Individual properties* are defined on a host's location and attribute values. The properties of a group, henceforth called *relational properties*, relate the location and attributes of multiple hosts in the same system configuration. An individual property, for example, may express the fact that a host is "on the freeway and is traveling faster than 55mph". A relational property may instead identify the hosts that are "traveling faster than the hosts around them." The context defined by a given context specification is inherently dynamic. The set of hosts that are part of some context may vary as a result of changes in their locations, in their attribute values, and in those of other hosts in the system configuration. We model a context specification as a function that selects a group of hosts of interest from a given configuration. In this paper, we focus on context specifications with static properties that identify dynamic sets of hosts. However, in the most general case, the properties may also change over time; thus, the function, denoted by α, assumes as arguments a time instant and a system configuration, i.e.,

$$\alpha(t, \mathcal{K}(t)) \in \varLambda = (T \times \mathcal{C} \to \mathcal{P}(H))$$

where the symbol $\mathcal{P}(X)$ denotes the powerset of set X.

2.3 Bringing Context into Events and Subscriptions

Traditionally, publish-subscribe communication is achieved by means of event notification messages that propagate through the network to reach all matching subscribers. Publishers define the content of each event at publication time, while subscribers define subscription filters that operate on this content. The middleware determines which subscribers should receive a given event by means of a matching process that applies the filters associated with existing subscriptions to the content of every newly published event. Given a universe E of possible events, we model a specific event as a member of this set (i.e., $e \in E$) and a subscription as the set of events (i.e., $\tilde{e} \in \mathcal{P}(E)$) of interest to the subscriber. Formally, this reduces matching to a simple membership test, $e \in \tilde{e}$.

As we pointed out in Section 1, the publish-subscribe communication model exhibits several limitations that hamper its applicability in context-aware applications for mobile network scenarios. In the remainder of this section, we remove these limitations and add several new features that allow publish-subscribe middleware to respond not only to the interests of subscribers as they change over time, but also to the evolving context in which subscriptions and events exist.

- We unify the treatment of events and subscriptions by allowing both of them to remain available in the system until a specified *expiration* time.
- We allow events and subscriptions to exist within a limited scope, which we call *context of relevance* in the case of events and *context of interest* in the case of subscriptions.

- We enable publishers and subscribers to use context specifications to restrict the identity of their communication parties based on their current context by defining *publication* and *subscription domains.*

Persistent Events and Subscriptions. The first characteristic we introduce in our context-aware publish-subscribe model is the ability to define events that may remain available in the system for an arbitrary time after their publication. To achieve this, we have each publisher associate an expiration time $\tau_e \geq 0$ with each of its event notifications. This implies that, at each time instant τ, an event is active and may be matched by a subscription only if its expiration time τ_e has not elapsed.

The notion of expiration time allows us to distinguish between *instantaneous* and *persistent* events. Instantaneous events correspond to those available in traditional publish-subscribe middleware and are characterized by an expiration time that coincides with their publication time. Persistent events, on the other hand, are those for which expiration occurs later than the time of publication. Persistent events enrich the paradigm with the ability to maintain state in event notifications by extending their relevance over an arbitrarily long period of time. This allows the middleware to address situations like the freeway scenario mentioned above, in which the event notifying vehicles of the accident should remain available after the accident has occurred.

Following the same pattern as with event notifications, we also introduce an expiration time $\tau_s \in T$ in the case of subscriptions. According to intuition, a subscription is active and available for matching at a given time instant if its expiration time has not elapsed.

Contextual Relevance and Interest. The second characteristic we introduce in our model is the ability for publishers to associate their events with information about the context that may be affected by them. Consider the example in Figure 1a: an accident happens on the freeway at mile 32 as indicated in the figure. One of the cars involved in the accident publishes a persistent event to inform oncoming vehicles of the danger. Ideally, the event should propagate towards vehicles that are approaching the accident site and not to those that are already past the accident or traveling in the opposite direction. To make this possible, the publisher associates the event notification with a *context of relevance.* The context of relevance represents the context that, according to the publisher, will be affected by the event or in which the event will be relevant: in this case the vehicles that are traveling east between miles 20 and 32. The set of hosts that constitute the context of relevance is dynamic and thus cannot be computed once for all by the publisher and encoded in the content of the event. The context of relevance may, in fact, vary over time due the mobility of hosts or due to changes in the values of their attributes. We address these dynamic aspects by modeling the context of relevance as a context specification $r \in \Lambda$. The context of relevance expresses the opinion of the publisher regarding the context that may be affected by the event or in which the event should be considered relevant. This means that subscribers outside an event's context of relevance may still request that the event be delivered to them.

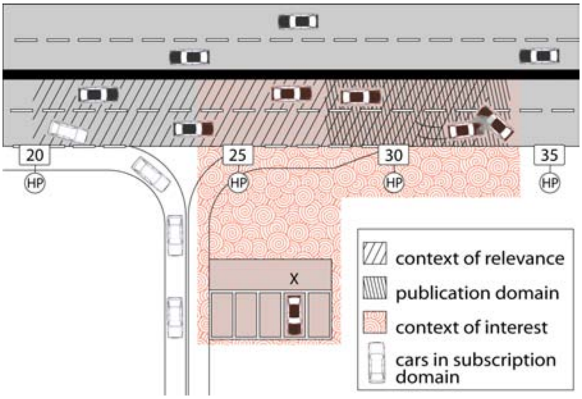
Consider again the above scenario. A short while after the accident, car X leaves its parking lot and intends to travel along the freeway where the accident happened. The middleware, should enable car X to issue a subscription that matches the events that may affect the road it intends to take. To make this possible, we allow subscribers to associate a *context of interest* with their subscriptions. Similar to the context of relevance, the context of interest is described by a context specification $\tilde{r} \in \Lambda$ and identifies a set of hosts that may change over time due. An event's context of relevance matches a subscription's context of interest if the sets of hosts they identify have a non-empty intersection. In Figure 1a, the context of interest of car X's subscription consists of the hosts in the area identified by the dashed contour. This allows subscriber X to receive the information about the accident from the hosts in the dashed area and decide on a different route.

Constraining the Sets of Publishers and Subscribers. The notions of relevance and of interest allow publishers and subscribers to associate a contextual scope to events and subscriptions. However, they do not allow them to restrict the identity of their communication parties based on their current contexts. Consider again the above scenario: the highway patrol has several stations along the freeway, each responsible for accidents that happen in a specific section. To detect accidents, each station subscribes to events regarding the freeway. However, it must also be able to specify that the publishers of these events should be located in the section it is responsible for. In this case the information about the accident should be received and handled by the station at mile 30 and not by the one at mile 25, regardless of the event's domain of relevance. To make this possible, each station associates its subscription with a *publication domain*: a context specification that, for each time instant and configuration, identifies the publishers whose events may be matched by a given subscription. In the above example, the station at mile 30 specifies that it wants to receive events published by cars involved in accidents between miles 28 and 33.

In a similar manner, publishers may want to restrict the identity of possibly matching subscribers based on their own context information. After intervening on the scene, the highway patrol attempts to speed up traffic by forcing vehicles with less than two occupants to exit at the last available exit before the accident. To inform vehicles of this decision it publishes a persistent event, stating that it should reach only the cars with only one occupant. To make this possible, the highway patrol associates a further context specification with its event: the *subscription domain*. This context specification states that the event may be received by subscribers that have a matching subscription filter, only if they are in the specified context (in this case, if they have only one occupant).

As with all other context specifications, we model the publication and subscription domains as two functions: respectively $\tilde{p} \in \Lambda$ and $\tilde{s} \in \Lambda$ that associate each time instant and configuration with a set of publishers or subscribers.

Events, Subscriptions, and Matching. The extensions we just defined allow us to enrich the publish-subscribe model with the notions of context-aware events,



(a) Accident Scenario

Event:

$p \in H$	publisher's identifier
$e \in E$	content of the event
$\tau_e \in T$	expiration of event
$r \in \Lambda$	context of relevance
$\tilde{s} \in \Lambda$	subscription domain

Subscription:

$s \in H$	subscriber's identifier
$\tilde{e} \in \mathcal{P}(E)$	filter on content of events
$\tau_s \in T$	expiration of subscription
$\tilde{r} \in \Lambda$	context of interest
$\tilde{p} \in \Lambda$	publication domain

(b) Event and Subscription Format

Fig. 1. Application Scenario and Event-Subscription Model

subscriptions, and matching. We define a context-aware *event* — henceforth called event — as a tuple $[p, e, \tau_e, r, \tilde{s}]$. As summarized in Figure 1b, the tuple includes the identity of the publisher, the content of the event, its expiration time, the context of relevance, and the subscription domain. In a similar manner, we define a context-aware *subscription* — henceforth called subscription — as a tuple $[s, \tilde{e}, \tau_s, \tilde{r}, \tilde{p}]$. As summarized in Figure 1b, this tuple consists of the identity of the subscriber, the subscription filter, the expiration time of the subscription, its context of interest, and its publication domain.

The definitions of context-aware events and subscriptions allow us to model the concept of *context-aware matching* as a direct extension of the matching operation for traditional publish-subscribe. As in the traditional case, the goal of matching is to determine whether a given event should be delivered to a given subscriber. However, our context-aware matching process must also take into account the current configuration of the system. Let us assume that each context specification in a given event-subscription pair is evaluated at time instant τ, and let $c = \mathcal{K}(\tau)$ denote the system configuration at that instant. Then we can state that a given event matches a given subscription if and only if the following five conditions are satisfied.

- (i) *standard matching*: the event content matches the subscription filter as in standard content-based publish-subscribe.

$$e \in \tilde{e}$$

– (ii) *lifetime validity*: the event and the subscription have not expired.

$$\tau \leq \tau_e \wedge \tau \leq \tau_s$$

– (iii) *interest-and-relevance overlap*: the context of relevance of the event and the context of interest of the subscription intersect each other.

$$r(\tau, c) \cap \tilde{r}(\tau, c) \neq \emptyset$$

– (iv) *publication domain matching*: the publisher is part of the context identified by the subscriber's publication domain.

$$p \in \tilde{p}(\tau, c)$$

– (v) *subscription domain matching*: the subscriber is part of the context identified by the event's subscription domain.

$$s \in \tilde{s}(\tau, c)$$

3 Routing and Matching in a Context-Aware Publish-Subscribe System

The definition of matching presented in Section 2.3 forms the basis for our routing and matching protocol supporting the context-aware publish-subscribe paradigm. The protocol is based on the idea that persistent events and subscriptions should be maintained by the hosts in the contexts of relevance and of interest until their expiration time. For a given event-subscription pair, the matching process is initiated by the hosts in the intersection of these two contexts, by evaluating conditions (i), (ii), and (iii): standard content-based matching, event and subscription lifetime, and the overlap between the contexts of relevance and of interest. If this evaluation is successful, further processing depends on whether the subscription includes a publication domain, i.e., whether it requires the evaluation of condition (iv). If this is the case, the host that is carrying out the matching operation must contact the publisher. Otherwise, it simply forwards the event to the subscriber for the evaluation of the subscription domain, i.e., condition (v), and for possible delivery to the application.

3.1 Assumptions and Requirements

To manage the complexity of context-aware matching, we built our protocol over a geocast routing service [17] and we assume that the contexts of relevance and of interest are always associated with some geographical component that identifies a region of the physical space. Moreover, we constrain the relational properties associated with the contexts of relevance and of interest to operate on groups of hosts within a one-hop neighborhood and on an application-defined subset of attributes, henceforth called *key attributes*. Hosts exchange periodic beacons that contain information about their current location, their location at the time their previous beacon was sent, and the current values of their *key attributes*. This allows relational properties to be evaluated locally by the hosts in the contexts of relevance and of interest, without issuing a query for each matching operation. For the publication and subscription domains, however, relational properties may exploit non-key attributes and may refer to groups of hosts beyond a one-hop range. To achieve this, we assume the use of a query dissemination service like the one in [22].

Finally, it is worth noting that all the context specifications in events and subscriptions refer to the system configuration when matching occurs. If publishers or subscribers want matching to operate on their state at publish or subscribe time, they can store this information as part of their events or subscriptions at the time they are issued.

3.2 Main Protocol Operation

The protocol exploits four data structures maintained by each network host (h) for the management of events and subscriptions.

- *Event table*: stores the unexpired events that have a geographical component of the context of relevance that includes host h.
- *Subscription table*: stores the unexpired subscriptions that have a geographical component of the context of interest that includes host h.
- *Local event table*: stores the unexpired events issued by host h.
- *Local subscription table*: stores the unexpired subscriptions issued by host h.

In addition, each host maintains a neighbor table that records the location and key attributes of neighboring hosts. This information, gathered with periodic beacons, is used in the evaluation of relational properties. Finally, the geocast protocol records the identifiers of recently received messages to prevent the transmission of duplicates.

Matching Subscription Interest Against Event Relevance. We begin our description by considering the publication of an event. The publisher first stores the event in its local event table. Then, it uses the geocast service to forward the event to the geographical region associated with its context of relevance. Each host in this region reacts to the receipt of the event with the following steps. First, if the event is persistent, the host stores it in its event table. This is done even if the non-geographical properties of its context of relevance are not currently satisfied. The dynamic nature of attribute values can in fact cause these properties to be satisfied at a later time. Second, the host carries out the first three steps of the context-aware matching process. First, it verifies whether the current values of its own attributes satisfy the individual properties associated with the context of relevance, and whether the key attributes of the hosts in its neighbor table satisfy the corresponding relational properties. If this is the case, it attempts to match the event against each unexpired subscription in its subscription table. This involves evaluating the subscription filter of each subscription against the content of the event and determining whether the attributes of the host and the key attributes of the hosts in its communication range satisfy the properties associated with the subscription's context of interest. If these first steps of the matching process are successful, the host takes action to deliver the event to the subscriber. In the accident scenario of Figure 1a, the hosts in the highlighted region will forward the event about the accident to the subscribers that expressed an interest in traffic information regarding the stretch of highway affected by the accident.

Delivering Events To Subscribers. For each event and subscription, the above matching process occurs at the intersection of the geographical regions associated with the contexts of relevance and of interest. Let us refer to this intersection as the *matching area.* A host that detects a positive match between an event and a subscription, prepares a *matched-event* message that encodes both the event and the information on how to reach the subscriber as shown in Figure 2 and discussed in greater detail in the following. Different matched-event messages generated by different hosts in the matching area for the same event-subscription pair are indistinguishable to the geocast protocol, which can therefore reduce the number of redundant messages by dropping multiple copies and by passively listening for other nodes' transmissions before forwarding a packet.

Matching Subscriptions against Persistent Events. Subscriptions are disseminated with a similar mechanism as events. A subscriber that issues a new subscription first stores it in its local table; then it forwards it to the hosts in the region associated with its context of interest. When a host in the region receives the subscription, it first stores it in its subscription table, and then it checks if its attributes and the key attributes of the hosts in its neighbor table satisfy the properties specified by the subscription's context of interest. If this is the case, it attempts to match the subscription against the unexpired persistent events contained in its event table. As in the above case, this involves evaluating whether, for each event, the conditions specified by the context of relevance are satisfied, and whether the subscription filter is satisfied by the content of the event. If these matching steps are successful, the event is forwarded to the subscriber as described above.

Managing the Publication and Subscription Domains. The protocol we just described exploits the contexts of relevance and of interest to direct events towards matching subscribers. In the following, we show how it can also evaluate the subscription and publication domains of events and subscriptions.

Subscription Domain. Let us consider an event with a subscription domain that is matched against a subscription without a publication domain. The subscription domain is evaluated by the subscriber upon receipt of the matched-event message. The subscriber uses its current location, its attribute values, and those in its neighbor table to evaluate individual properties and the relational properties referring to the key attributes of the hosts in its one-hop neighborhood. If this first evaluation is successful, it issues a query to retrieve information about

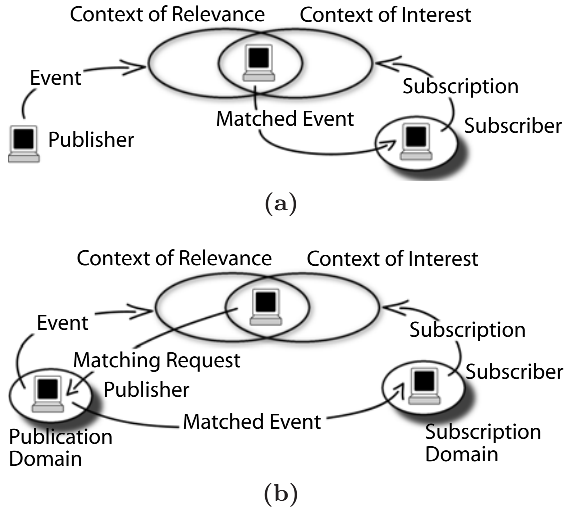


Fig. 2. Routing without (a) or with (b) a publication domain

the values of non-key attributes or of the attributes of hosts beyond the one-hop range. If the results of the query satisfy the relational properties of the subscription domain, the event is delivered to the application. Figure 2a shows the messages exchanged by the protocol in this situation.

Publication Domain. The protocol is slightly more complex when the publication domain is specified. A host that completes the first three steps of the matching process must in fact communicate with the publisher to retrieve information about the current configuration. To achieve this, it sends a *matching-request* message to the publisher. The message contains the identifier of the event being matched, the specification of the publication domain, and information on how to reach the subscriber. The publisher reacts to the matching-request by evaluating the publication domain and, in case of a positive match, by sending a matched-event message to the subscriber using the information contained in the matching request. A schematic view of this behavior is shown in Figure 2b.

Managing Host Mobility. So far, we have described our protocol by ignoring a key aspect of the system, host mobility. In the following we return to this issue and describe our mechanism to handle changes in host location. There are two aspects of our protocol that are affected by the ability of hosts to move: the ability to reach publishers and subscribers with matching-requests and matched-event messages during the matching process, and the maintenance of events and subscriptions in the regions of relevance and of interest.

Mobility of Publishers and Subscribers. To address the first issue, publishers and subscribers label their events and subscriptions with information about how they can be reached. Specifically, they associate each event or subscription with a *reachability region*, a geographical region computed on the basis of their current location and predicted movement. When routing a matching-request or matched-event message to a publisher or subscriber, a host uses the geocast service to deliver it to all the hosts in the reachability region. The description of the reachability region of the subscriber is also included in each matching-request sent to the publisher of a possibly matching event.

In principle, the reachability region should cover all the possible locations of a host throughout the lifetime of a persistent event or subscription. This, however, would result in exceedingly large regions for events and subscriptions with non-trivial lifetimes. Therefore, publishers and subscribers are allowed to update their events and subscriptions with new reachability regions. Updates are sent both periodically and whenever a publisher or subscriber gets within a specified *safe-distance* from the boundary of its reachability region.

Mobility in the Regions of Relevance and of Interest. To maintain events and subscriptions in dynamic regions of relevance and of interest, we combine the above updates to events and subscriptions with a reactive approach. Specifically, the hosts in the regions of relevance and of interest monitor their neighbors to determine if they have just entered a region associated with any of their events or subscriptions. To achieve this, hosts include, in each of their beacons, their current location and the one at the time they sent the previous beacon. If, using this information, a host determines that some neighbors have just entered any of the regions associated with the events and subscriptions in its tables, it broadcasts a *digest* message containing the necessary information to update their event and subscription tables. Finally, to improve reliability, publishers and subscribers may send each event and subscription to a region that is slightly larger than the actual region of relevance or of interest.

4 Simulation

We evaluated the performance of our protocol by means of a detailed simulation study based on OmNet++ [28] a popular open-source simulation system. We modeled the physical and MAC layers using Omnet++'s mobility framework. Specifically, we adopted an 802.11 MAC over which we placed a custom implementation of a receiver-based geocast protocol that provides the ability to route messages to a region of the physical space.

4.1 Simulation Setup

We consider a reference scenario consisting of 100 hosts in a $1000m \times 1000m$ area. The movement of hosts follows the random way-point mobility model, with a minimum speed of 1m/s, a pause time of 0s, and a maximum speed of 10m/s. For each measurement we executed 10 simulation runs of 450s each.

We model contextual attributes by associating each host with a randomly chosen integer value. According to the discussion in Section 3.1, the contexts of relevance and of interest are always associated with a geographical region, represented by a $100m \times 100m$ square. For the publication and subscription domains we consider the worst case scenario by using only non-geographical context specifications. Specifically, each is configured to match 50% of the available attribute values.

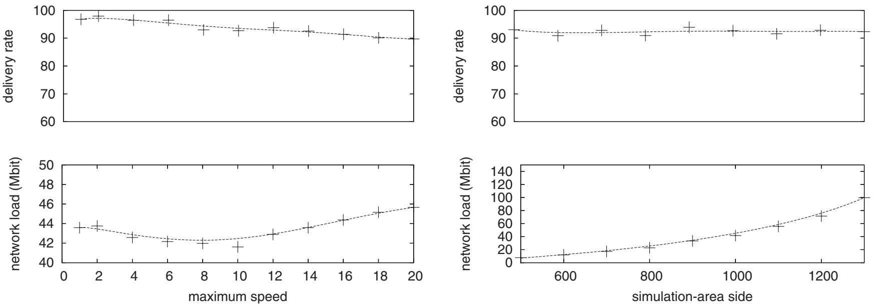
Publishers and subscribers each constitute 20% of the hosts in the network. Each subscriber issues a subscription every 50s: the subscription is either instantaneous or has a 50s lifetime. Similarly each publisher generates an event every 10s that is either instantaneous or has a lifetime of 10s. Moreover, 50% of the subscriptions are associated with a publication domain. The size of reachability regions of publishers and subscribers is set to $50m \times 50m$. Events and subscriptions are refreshed every 5s or whenever the publisher or subscribers gets within 10m of the region's boundary. In the absence of other traffic, hosts exchange beacon messages every 5s. Finally, to improve reliability, event and subscription updates are sent to regions with sides that are $60m$ larger that those of their actual destinations.

4.2 Protocol Performance

We consider two performance metrics in the evaluation of our protocol: delivery rate and communication cost. To evaluate the former, we compare our protocol against an ideal protocol that instantaneously matches events against subscriptions based on information about the entire system configuration. The delivery rate is the ratio between the number of events successfully delivered by our protocol and those that would be delivered by such an ideal system. The communication cost, on the other hand, is measured as the number of Mbits transmitted at the physical layer. To evaluate the impact of the application scenario on these two metrics, we individually vary four of the above parameters: the speed of hosts, the size of the network, its density, and the percentage of subscriptions that require the evaluation of the publication domain.

Impact of Host Speed. We first consider the performance of our protocol with increasing host speeds. The plots in Figure 3a show the results obtained with hosts that move with a minimum speed of 1m/s and a maximum speed ranging from 1m/s to 20m/s. According to expectations, delivery rate decreases with increasing speed from a value of over 96% with a maximum speed of 1m/s to a value of 90% with a maximum speed of 20m/s. This is due to the combination of two phenomena. First, the accuracy of the geocast protocol naturally decreases with speed. Second, the higher the speed, the more likely hosts are to move out of the regions of relevance and of interest. In general, a host that enters either region will receive the corresponding event or subscription in a digest message from one of the hosts already in the region. However, if there is no other host in range that can communicate this information, the new host will be unable to retrieve the event or subscription until the next refresh message.

The bottom plot in Figure 3a shows the effect of speed on communication cost for the protocol. The cost decreases when the maximum speed increases



(a) host speed

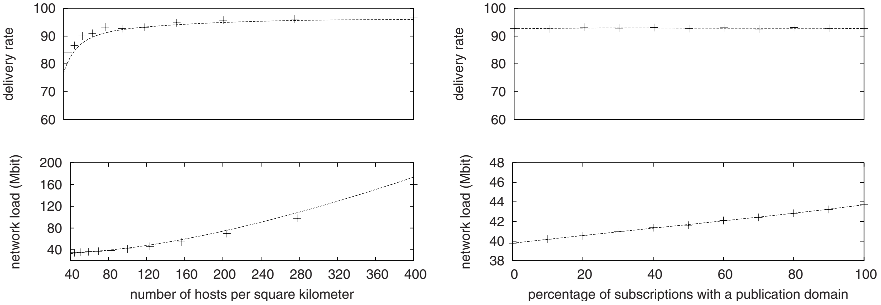
(b) system scale

Fig. 3. Impact of host speed and system scale over delivery rate and communication cost

from 1m/s to 10m/s as a result of the drop in the delivery rate associated with high mobility (93% with a maximum speed of 10m/s). When the speed increases even further, such a decrease is balanced by an increase in control traffic: digest messages, and refresh messages for events and subscriptions. It is worth noting that the cost remains limited (increasing from 42Mbit to 46Mbit) even at the maximum speed of 20m/s. In particular, we verified that flooding events in the same scenario would result in a cost ranging from 83Mbit to 94Mbit without the ability to manage the contextual aspects of events and subscriptions.

Impact of System Scale. Next, we evaluate the scalability of our protocol by varying the size of the network with a constant host density of 100 hosts per square kilometer. Results are depicted in Figure 3b. The top plot shows that the delivery rate of our protocol is largely independent of system scale and remains around a value of 92%. An increased scale at a constant host density simply results in a larger number of hops to reach the hosts in the matching area and to deliver matching events to subscribers. Our geocast-based protocol is able to address this increase without significantly decreasing its ability to deliver events. Moreover, the bottom plot in Figure 3b shows that network traffic increases as the cube of the side of the simulation area. This can be easily explained by observing that the number of events and subscriptions in the system grows linearly with the number of publishers and subscribers and thus quadratically with the simulation area. This, together with the fact that each event and subscription must travel a path that is approximately linear with the number of hosts, yields the cubic relationship shown in Figure 3b. In real-world applications, scalability should be further improved since the density of events and subscription is likely to be lower.

Impact of Host Density. Our next experiment analyzes the behavior of our protocol with different host densities. We vary density by changing the simulation area while employing a fixed number of hosts, one hundred. Specifically, we simulated



(a) host density **(b)** publication domain

Fig. 4. Impact of host density and of the evaluation of the publication domain on delivery rate and communication cost

areas yielding host densities ranging from 44 to 400 hosts per square kilometer. The results are shown in Figure 4a. The top plot highlights the good performance of our protocol in terms of delivery rate (over 90%) for host densities higher than 60 hosts per square kilometer. With lower densities, network connectivity may not be guaranteed at all times and the geocast protocol is more likely to experience routing errors due to local minima. Mechanisms to address this, such as perimeter routing or store-&-forward approaches, may improve performance at low densities. However, their evaluation is outside the scope of this paper.

The bottom plot in Figure 4a shows the behavior of the protocol in terms of network traffic. The increase in host density determines the presence of a larger number of hosts in the regions associated with the contexts of relevance and of interest. This causes an increase in the number of receivers for each event, which translates into the increase in network traffic evidenced by the figure.

Impact of the Publication Domain. The last scenario parameter we consider is the percentage of subscriptions that require the evaluation of the publication domain. As described in Section 3.2, the evaluation of this context specification requires publishers to be contacted after their events have been matched in the intersection of the contexts of relevance and of interest. The data depicted in the bottom plot of Figure 4b confirms the intuition that this process has an impact on the communication cost of the protocol. Matching-request messages account for almost 10% of the overall network traffic when the publication domain must be evaluated for every pair of event and subscription. The top plot in Figure 4b, on the other hand, shows that the delivery rate is almost unaffected by this parameter, despite the longer distance traveled by events before reaching matching subscribers.

5 Related Work

Recent advances in mobile computing infrastructures have led to increasing interest in context-aware applications that can respond not only to user inputs but also to the characteristics of the environment in which they operate. To support these applications, the research community has proposed several middleware solutions that aim to facilitate software development [11,20,16,25].

In particular, researchers have followed the success of publish-subscribe middleware for large-scale wired networks [4,2,23,24,5,14,10,21,13,15,26,27] and have investigated solutions for publish-subscribe communication in environments such as MANETs. The solutions in [30,18,29,1,8,7] take a first step in the development of protocols that implement the publish-subscribe model in a mobile environment. In this paper, we continue this effort and extend publish-subscribe with the ability to manage and react to context at the middleware level.

Such capabilities are available only in very basic forms in existing middleware. The work in [9] proposes a location-aware extension to the publish-subscribe model in which events and subscriptions can be associated with geographical scopes that resemble our publication and subscription domains. Nonetheless, the management of location information is only one of the needs of context-aware applications. Moreover, while the authors list MANETs as a possible application scenario, they only consider a general-purpose tree-based solution that is not backed up by a performance evaluation. Scalable Timed Events And Mobility (STEAM) [19] ties the locations of publishers with that of subscribers by proposing a proximity-based event model for MANETs. Events are distributed only in a limited geographical area centered around the publisher's location. This results in a limited model that cannot easily represent scenarios like those we consider in this paper. The same limitation affects the work in [12]. While its authors propose a notion of persistent events and subscriptions, their matching model requires the publisher and the subscriber of a matching event-subscription pair to be within a region centered around the other.

In [6], events are associated with a location of occurrence and subscriptions with a geographical predicate. Publishers are therefore unable to control the dissemination of their events. Moreover, the system exploits a centralized spatial matching engine that is clearly unsuitable for the MANET scenarios we target. Finally, Fulcrum [3] tackles the definition of context-aware publish-subscribe from a different angle and presents a system for large-scale context-aware publish-subscribe based on an open-implementation approach. Clients may specify user-defined strategies to implement complex filters that match combinations of events occurring at different locations in the network. Its current implementation, though, relies on a static backbone of brokers which may not always be available in scenarios such as MANETs.

6 Conclusions

Publish-subscribe has emerged as a communication paradigm able to facilitate the development of complex reactive applications in open network environments. Yet, current systems still offer only partial support for the management of context in mobile applications. In this paper, we proposed an extension to the publish-subscribe paradigm that enables the middleware to factor context information into events and subscriptions. This gives publishers and subscribers the ability to control the diffusion of messages and to restrict the identities of communication parties. We support this model with a protocol for the dissemination of context-aware events and subscriptions in MANETs. We based this initial protocol on geocast to aid the management of the geographical components of context, but we are also planning to investigate alternative routing approaches. Our simulation analysis shows that our protocol achieves high delivery rates in the presence of mobility while exhibiting good scalability properties. This suggests we can expect even better performance in many real-world scenarios.

Acknowledgements. This research was supported by the Office of Naval Research under MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily represent the views of the research sponsors.

References

1. Baldoni, R., Beraldi, R., Cugola, G., Migliavacca, M., Querzoni, L.: Structure-less content-based routing in mobile ad hoc networks. In: Proc. of ICPS, Santorini (Greece), IEEE Computer Society Press, Los Alamitos (July 2005)
2. Banavar, G., Chandra, T., Mukherjee, B., Nagarajarao, J., Strom, R., Sturman, D.: An efficient multicast protocol for content-based publish-subscribe systems. In: Proc. of the 19^{th} Int. Conf. on Distributed Computing Systems, Austin, TX, USA, IEEE Computer Society Press, Los Alamitos (1999)
3. Boyer, R.T., Griswold, W.G.: Fulcrum - an open-implementation approach to internet-scale context-aware publish/subscribe. In: Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 9, IEEE Computer Society Press, Los Alamitos (2005)
4. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM TOCS 19(3), 332–383 (August 2001)
5. Chand, R., Felber, P.A.: XNet: a reliable content based publish subscribe system. In: Proc. of the 23rd Symp. on Reliable Distributed Systems, Florianópolis, Brazil, IEEE Computer Society Press, Los Alamitos (October 2004)
6. Chen, X., Chen, Y., Rao, F.: An efficient spatial publish/subscribe system for intelligent location-based services. In: Proceedings of the 2nd international workshop on Distributed Event-Based systems, ACM Press, New York (2003)
7. Chen, Y., Schwan, K.: Opportunistic overlays: Efficient content delivery in mobile ad hoc networks. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 354–374. Springer, Heidelberg (November 2005)
8. Costa, P., Picco, G.P.: Semi-probabilistic Content-based Publish-subscribe. In: Proc. of the 25^{th} Int. Conf. on Distributed Computing Systems (ICDCS05), Columbus (OH, USA), pp. 575–585. IEEE Computer Society Press, Los Alamitos (June 2005)
9. Cugola, G., de Cote, J.E.M.: On introducing location awareness in publish-subscribe middleware. In: ICDCSW '05: Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS) (ICDCSW'05), pp. 377–382. IEEE Computer Society Press, Washington (2005)

10. Cugola, G., Di Nitto, E., Fuggetta, A.: The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. IEEE Trans. on Software Engineering 27(9), 827–850 (September 2001)
11. Dey, A., Abowd, G.: The context toolkit: Aiding the development of contextaware applications (1999)
12. Eugster, P.T., Garbinato, B., Holzer, A.: Location-based publish/subscribe. In: NCA '05: Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications, pp. 279–282. IEEE Computer Society Press, Washington (2005)
13. Fabret, F., Jacobsen, H.A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D.: Filtering algorithms and implementation for very fast publish/subscribe systems. ACM SIGMOD Record 30(2), 115–126 (2001)
14. Fiege, L., Mühl, G., Gärtner, F.C.: Modular event-based systems. Knowledge Engineering Review 17(4), 359–388 (2002)
15. Gruber, R.E., Krishnamurthy, B., Panagos, E.: The architecture of the READY event notification service. In: Dasgupta, P. (ed.) Proc. of the ICDCS Workshop on Electronic Commerce and Web-Based Applications, Austin, TX, USA, IEEE Computer Society Press, Los Alamitos (May 1999)
16. Julien, C., Roman, G.-C.: Egospaces: Facilitating rapid development of context-aware mobile applications. IEEE TSE 32(5), 281–298 (2006)
17. Maihüfer, C.: A survey on geocast routing protocols. IEEE Communications Surveys and Tutorials, 6(2), 2nd quarter issue (2004)
18. Meier, R., Cahill, V.: STEAM: Event-Based Middleware for Wireless Ad Hoc Network. In: Proc. of the 22^{nd} Int. Conf. on Distributed Computing Systems, Vienna, Austria, pp. 639–644. IEEE Computer Society Press, Los Alamitos (2002)
19. Meier, R., Cahill, V.: Steam: Event-based middleware for wireless ad hoc networks (2002)
20. Murphy, A.L., Picco, G.P., Roman, G.-C.: Lime: A coordination model and middleware supporting mobility of hosts and agents. ACM Trans. Softw. Eng. Methodol. 15(3), 279–328 (2006)
21. Pallickara, S., Fox, G.: NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In: Proc. of the 4th ACM/IFIP/USENIX Int. Middleware Conf, Rio de Janeiro, Brazil, pp. 41–61. ACM Press, New York (2003)
22. Payton, J.: A Query-Centric Approach to Supporting the Development of Context-Aware Applications for Mobile Ad Hoc Networks. PhD thesis, Washington University in St. Louis, Technical Report WUCSE-2006-49 (2006)
23. Pietzuch, P.R., Bacon, J.M.: Hermes: A distributed event-based middleware architecture. In: Proc. of the 1^{st} Int. Workshop on Distributed Event-Based Systems (DEBS), Vienna, Austria, IEEE Computer Society Press, Los Alamitos (July 2002)
24. Pietzuch, P.R., Bacon, J.M.: Peer-to-peer overlay broker networks in an event-based middleware. In: Proc. of the 2^{nd} Int. Workshop on Distributed Event-Based Systems (DEBS), San Diego, CA, USA, ACM Press, New York (June 2003)
25. Roman, M., Hess, C., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: A middleware infrastructure for active spaces. IEEE Pervasive Computing 1(4), 74–83 (2002)
26. Segall, B., Arnold, D., Boot, J., Henderson, M., Phelps, T.: Content Based Routing with Elvin4. In: Proc. of AUUG2K, Canberra, Australia (June 2000)

27. TIBCO Inc. TIBCO Rendezvous, `www.tibco.com`
28. Varga, A.: OMNeT++ Web page, `www.omnetpp.org` (2003)
29. Yoneki, E., Bacon, J.: An adaptive approach to content-based subscription in mobile ad hoc networks. In: Proc. of the 2nd IEEE Annual Conference on Pervasive Computing and Communications Workshops, Orlando, FL (March 2004)
30. Zhou, H., Singh, S.: Content-based multicast for mobile ad hoc networks. In: Proc. of the 1^{st} Annual Workshop on Mobile Ad Hoc Networking and Computing (Mobihoc 2000), Boston, MA, ACM Press, New York (August 2000)

A Prolog-Based Language for Workflow Programming

Steve Gregory and Martha Paschali

Department of Computer Science
University of Bristol, BS8 1UB, England
steve@cs.bris.ac.uk

Abstract. Workflow management systems control *activities* that are performed in a distributed manner by a number of human or automated *participants*. There is a wide variety of workflow systems in use, mostly commercial products, and no standard language has been defined in which to express workflow specifications. In this paper we propose Workflow Prolog, a new extension of Prolog. The language allows workflow systems to be implemented in a novel declarative style, while preserving the existing properties of Prolog, such as its familiarity and efficiency. We then demonstrate the expressiveness of the language by showing how it can express each of the *workflow patterns* that have previously been identified as the requirements of a workflow language.

1 Introduction

Workflow management systems (WFMSs) automate the flow of tasks, information, and data between people or other entities in an organization. A WFMS is controlled by a description of the business processes to be automated, where a *process* comprises a number of primitive *activities* performed by *participants*. Activities may be *manual* (performed by human participants) or *automated* (performed by software, etc.). Like processes in computer systems, a business process can have many *instances*, which run concurrently and might synchronize and communicate with each other. Differences from computational processes are that (a) activities are generally outside the control of the WFMS, and (b) business processes might "run" for a very long time.

In a WFMS, a business process is described by a *process definition* or *workflow specification*. In general, this may describe the control flow (the order of activities in a process), the data flow between participants, the resources needed, and possibly the detailed actions of activities. The process definition is written in a *workflow language*.

There are many different WFMSs, most of which are commercial products. A sample of 15 of them has been evaluated by van der Aalst *et al* in [1]. There is also a wide variety of workflow languages: most products use a different proprietary language, and independent workflow languages have also been proposed, for example [2, 24, 4]. These often provide a graphical syntax and sometimes have an underlying formal semantics. The languages differ fundamentally in major ways.

Although there is no standard workflow language, there have been some useful attempts to analyse the various existing languages and identify common characteristics.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 56–75, 2007.

One is the standardization effort by the Workflow Management Coalition [27, 28]. Another is the survey in [1], which identifies 20 *patterns* that abstract the control flow features found in the various languages and lists which languages support each pattern. The authors observed that none of the WFMSs surveyed supports all of the patterns. See also [29]. These workflow patterns are a good guide to the kind of features that a workflow language should support, and so we return to them in Section 3.

Many workflow languages are based on a formal framework, the most popular of which is probably Petri nets or some high-level variant; for example [2, 18]. These allow automatic verification of workflow properties, such as freedom from deadlock. There has also been some work on using various forms of logic for modelling and reasoning about workflow, which we briefly survey in Section 4.2.

In this paper, we are interested in a logic *programming* approach: i.e., using a logic-based language to *implement* WFMSs. In principle, this should offer a high-level declarative way to develop such systems. For this, we need a logic programming language that can conveniently express workflow problems.

Concurrent logic programming languages (e.g., [6, 23]) have long been used to implement concurrent applications, so they are obvious candidates for workflow programming in logic. However, they have several drawbacks with respect to workflow:

- A WFMS needs to interact with activities outside the system, which are performed by external participants: either humans or existing programs. This need is not addressed by concurrent logic programming languages.
- Communication by incrementally binding logical variables — a key feature of concurrent logic programming — is not a natural way to communicate with workflow participants. In general, it requires shared memory.
- Processes in a WFMS are persistent: they might run for days or even years, but a concurrent logic program runs only as long as its host machine is switched on.

Prolog systems with coroutining provide some of the expressive power of concurrent logic programming languages, and have additional advantages. However, they suffer from the same drawbacks as concurrent logic programming languages.

We propose a new logic programming language, Workflow Prolog, intended to enable the implementation of WFMSs in logic. Key features are:

- Workflow Prolog is an extension of Prolog, not a completely new language. A program deviates from Prolog only where necessary for workflow purposes. This means that users do not need to learn a new language, and can benefit from the availability of efficient and well-supported Prolog implementations.
- A *workflow variable*: an asynchronous bidirectional communication variable, allowing communication with workflow participants. This is used together with an asynchronous RPC (remote predicate call), which represents the effect of a workflow activity in the form of a goal: a relation on a workflow variable.
- A coroutining mechanism, using guards, to receive results from activities on workflow variables and suspend when necessary. Deep and nested guards are possible, for expressiveness.
- Time constraints, for a declarative treatment of time.
- A garbage collector that automatically cancels workflow activities.
- Persistence, to handle long-lived workflow processes.

In this paper, Section 2 describes the Workflow Prolog language, justifies its design, and illustrates its use with example programs. In Section 3 we show how each of the workflow patterns of van der Aalst *et al.* [1] can be expressed in the language. Conclusions, including related work, appear in Section 4.

2 Workflow Prolog

We view a WFMS as one component of a distributed system, which comprises the WFMS and a number of participants. A workflow programming language is needed to implement the WFMS but not the participants, which in general may be existing applications or even humans. Therefore, a workflow language need not be a complete distributed programming language; it just needs the ability to create and manage concurrent (business) processes and assign activities to participants. The language need not even feature true concurrency, provided it has some kind of coroutining ability. We attempt to demonstrate this below, particularly in Section 2.4.

Workflow Prolog is a new workflow programming language based on logic. One of our design aims has been to avoid designing a completely new language. Instead, we have added to an existing language (Prolog) only the features necessary for workflow programming. Workflow Prolog is essentially the same as Prolog but with a few extensions, notably a form of coroutining, which are described and justified below.

2.1 Workflow Variables

An *activity* is the smallest unit of work scheduled by a WFMS as part of a process [28]. After being invoked by the WFMS, the activity is performed independently by a participant; when completed, the participant returns the activity's results to the WFMS. We assume that there is no communication between the participant and the WFMS while performing the activity.

A workflow language therefore needs a way to (a) invoke an activity, identified by the name of the participant, (b) pass input data to the activity, (c) wait for the activity to complete, (d) receive output data when the activity is complete.

In general, this seems to require some kind of (bidirectional) message passing. In traditional concurrent logic programming languages, message passing is achieved by incrementally instantiating variables to lists [12], but we reject this method because:

- It is more general than needed. For example, messages are allowed to contain variables, and variables may be incrementally bound to structures other than lists, such as trees. Moreover, stream communication is not even needed for workflow, because each activity is defined to receive only one message and send one reply.
- In its general form, it requires shared memory, which is not available between a WFMS and participants.
- All variables are potentially communication channels, so the language would need special unification or matching features that allow goals to suspend on variables.

Another method which has been used is to add explicit "send" and "receive" message primitives or Linda-like blackboard primitives. But these are more general than we need for workflow purposes, and sometimes have no declarative meaning.

Instead, in Workflow Prolog we introduce the concept of a *workflow variable*. This acts as an asynchronous bidirectional communication channel between a Workflow Prolog program and a workflow participant. A workflow variable contains:

- *Destination*: the identity of the participant.
- *Message content*: data to be sent to the participant.
- *Message time*: absolute time when message was sent to the participant.
- *Reply content*: data returned by the participant.
- *Reply time*: absolute time when reply was sent by the participant.

As usual in logic programming languages, workflow variables are single-assignment: each component is set only once. The destination and message content are assigned by the Workflow Prolog program; the reply content is assigned by the participant; the message time and reply time are assigned by the implementation. This way, each workflow variable captures all available information about an activity.

Implementation. No special unification or matching is needed in a Workflow Prolog implementation, because workflow variables are distinct from regular variables and are used only by suspendable predicates. For example, a workflow variable cannot be unified with another workflow variable or any term except an unbound variable.

2.2 Remote Predicate Call

In imperative languages, such as Ada, bidirectional message passing is sometimes achieved using a procedure call syntax, in a remote procedure call (RPC). Input parameters of a procedure call are sent in a message to a server and output parameters are taken from the reply when it is received. The RPC is usually synchronous: it returns only after the reply is received.

In Workflow Prolog, also for bidirectional communication, we introduce an *asynchronous* form of RPC (remote predicate call):

```
rpc(Dest,MC,WV)
```

which creates a new workflow variable `WV`, sets the destination of `WV` to `Dest`, sets the message content of `WV` to `MC`, sends a message containing `MC` to `Dest`, and sets the message time of `WV` to the time at which the message was sent. The `rpc` goal succeeds immediately after sending the message. Later, when the participant replies, the reply content and reply time can be extracted from `WV`; see Sections 2.3 and 2.6.

The `rpc` goal is used in Workflow Prolog to invoke an activity and pass data to it. To wait for the activity to be complete, and retrieve results from it, the program needs to wait for the reply from the participant on the workflow variable; see Section 2.3.

An advantage of this method is that an `rpc` goal (representing an activity) has a declarative reading as a relation between the activity's input and output data, and possibly their times. E.g., if `M` is assessing proposal `P`, `rpc(M,P,WV)` might mean

$$(\text{“}\mathtt{M}\text{ approves }\mathtt{P}\text{”} \wedge \mathtt{WV.reply{=}accept}) \vee$$
$$(\text{“}\mathtt{M}\text{ rejects }\mathtt{P}\text{”} \wedge \mathtt{WV.reply{=}reject}).$$

Program 1 shows part of an example to receive and check project proposals submitted by students. The query creates a process to solicit a proposal from a student named zz1234 and have it checked by two “markers” named smith and jones. The rpc goal here represents an activity send_proposal (writing a proposal) performed by participant zz1234. When zz1234 completes the activity, the proposal will be available as the reply on workflow variable P, for checking by the markers.

```
:- proposal(zz1234,send_proposal,[smith,jones]).

proposal(Student,Request,Markers) :-
   rpc(Student,Request,P),
   check(P,Student,Request,Markers).
```

Program 1. Proposal/approval example

Our workflow variables and asynchronous RPC are quite similar to the “futures” of Multilisp [14]: both are used to invoke a concurrent process and access its result later. Differences are that our language is logic-based and that our RPC is intended solely for workflow purposes, not for general parallel computation.

Restrictions. In rpc(Dest,MC,WV), Dest and MC must be ground (variable-free) and WV an unbound variable. Because the effect of rpc is immediate, it is not allowed to appear in a context that might fail: it must appear at the Prolog top level (i.e., there must be no choicepoints) and it must not appear in a guard (see Section 2.3).

Implementation. The exact form of a message is not prescribed, except that it must contain a unique *activity identifier* to enable a reply to be matched with the corresponding workflow variable. Messages could be implemented by:

- email for messages and replies;
- email for messages with replies sent via a form on a web page;
- for an automated participant (a program): invoke the specified program, with its input taken from the message content and its output placed in the reply content.

2.3 Suspendable Goals and Guards

Our remote predicate call is asynchronous, to allow activities to be active concurrently. Therefore, a program needs the ability to wait until an activity completes, and to get the results from it. To do this we introduce a special coroutining mechanism.

In Workflow Prolog, goals only need to suspend on workflow variables, not regular variables. Moreover, instead of using features like Prolog’s freeze or when predicates to determine which variables to suspend on, we use guards [10], which are now a standard feature of concurrent programming languages: they were used in CSP [16] and introduced into logic programming by Clark and Gregory [5].

In Workflow Prolog, goals are executed sequentially from left to right with the exception of *suspendable goals*, which might suspend on workflow variables. A suspendable goal is a goal for a *suspendable predicate*, which is either:

- A built-in suspendable predicate:

```
replycontent(WV,RC)
```

This goal suspends on workflow variable WV if it has no reply yet, and otherwise succeeds, unifying RC with the reply content of WV. (There are also some other built-in suspendable predicates, described in Section 2.6.)

- A user-defined suspendable predicate. This is any predicate defined by clauses at least one of which has a non-empty guard.

In general, a Workflow Prolog clause takes the form

$$H \;\;\texttt{:-}\;\; G_1, \ldots, G_m \;\;\texttt{:}\;\; B_1, \ldots, B_n .$$

where H is the head, $G_1, \ldots, G_m$ is the *guard* and each G_i is a *guard goal*, $B_1, \ldots, B_n$ is the *body* and each B_j a *body goal*. Any guard goal or body goal may or may not be suspendable. If a guard is empty ($m = 0$), the clause is written

$$H \;\;\texttt{:-}\;\; B_1, \ldots, B_n .$$

If all clauses have empty guards, the predicate for H is not suspendable, and a goal matching H is executed in the same way as Prolog. Otherwise, for a suspendable predicate, Workflow Prolog *reduces* a goal matching H to the body of one of the clauses. It searches for a clause to use by executing each clause's guard in turn. Each guard may succeed, fail, or block (on a set of workflow variables). A guard executes from left to right, and *blocks* on a set of variables *Vs* if it reaches a suspendable goal that is blocked on *Vs*; this means that the guard cannot succeed or fail until at least one of the variables in *Vs* has a reply value. (Note that, if replycontent is the only suspendable goal used in guards, each guard will block on at most one variable.)

If a successful guard is found, the H goal *commits* by reducing to the corresponding body. If all guards fail, the goal fails. If no guard succeeds but one or more block, *H suspends* on the union of the sets of variables on which the guards are blocked.

If a goal H suspends, on a set of workflow variables *Vs*, it is delayed until there is a reply on any one of these variables from the corresponding participant, and the computation proceeds with the next goal to the right of H. The states of blocked guards are *not* saved: all work done in trying to reduce H will be repeated later when H is woken (see Section 2.4). This is one reason why side-effects are excluded from guards; another reason is that guards are expected to fail.

Program 2 defines the check predicate used in Program 1. This is a suspendable predicate because its clauses contain guards. If check is called before a reply arrives on workflow variable P, the goal will suspend on P because both guards block on P. Otherwise, the guards will extract the content of the reply, Proposal, and check whether it is valid. If so, the approve stage will be performed; otherwise, the proposal stage will be repeated, invoking a new send_proposal activity. ('\+' is Prolog's negation operator.)

```
check(P,Student,_,Markers) :-
  replycontent(P,Proposal), valid(Proposal) :
  approve(Proposal,Student,Markers).
check(P,Student,Request,Markers) :-
  replycontent(P,Proposal), \+ valid(Proposal) :
  proposal(Student,Request,Markers).
```

Program 2. Proposal/approval example (contd.)

Program 3 defines the approve predicate (not suspendable). Its rpc goals invoke two concurrent approve_proposal activities, one by each marker. The decide goal (suspendable) waits for the markers to reply. Initially, decide([R1,R2],…) suspends on both R1 and R2: its first two clauses block on R1 and its third clause blocks on R2. If *both* markers reply 'ok', the proposal is accepted (clause 1). If *one* marker replies with comments (clause 2 or 3), the comments and proposal are sent to the student to revise. In this case, the other marker's reply is ignored.

```
approve(Proposal,Student,Markers) :-
  rpcs(Markers,approve_proposal(Proposal),Rs),
  decide(Rs,Markers,Proposal,Student).

rpcs([],_,[]).
rpcs([Dest|Dests],Message,[WV|WVs]) :-
  rpc(Dest,Message,WV), rpcs(Dests,Message,WVs).

decide([R1,R2],_,Proposal,Student) :-
  replycontent(R1,ok), replycontent(R2,ok) :
  rpc(database,accepted(Student,Proposal),D),
  replycontent(D,ok).
decide([R1,_],[M1,M2],Proposal,Student) :-
  replycontent(R1,c(C)) :
  proposal(Student,revise(Proposal,C),[M1,M2]).
decide([_,R2],[M1,M2],Proposal,Student) :-
  replycontent(R2,c(C)) :
  proposal(Student,revise(Proposal,C),[M1,M2]).
```

Program 3. Proposal/approval example (contd.)

Restrictions. All suspendable goals must be ground unless they appear in a guard, to avoid problems with dependent goals on the right. Suspendable guard goals need not be ground because, if they block, guard goals to their right are not executed. Suspendable goals must appear at the Prolog top level (i.e., there must be no choicepoints). This prevents premature commitment, in the case of a cut following a suspended goal.

Implementation. A simple way to implement suspendable goals is by an interpreter that executes each guard until the goal commits or suspends. This approach is not too expensive because the interpreter is called only for suspendable goals and is very small: the guards themselves are not interpreted but are executed directly by Prolog, as are clause bodies. If no suspendable goals are used, the execution is almost the same as Prolog's, and there is no overhead relative to it.

One way to suspend a suspendable goal on a set of workflow variables would be to exploit the coroutining features provided in some Prolog systems, such as "attributed variables" [17, 15]. Each workflow variable would contain a pointer to the goals suspended on it, and the other variables that each goal is suspended on. Another method (used in the prototype implementation) is to store workflow variables and suspended goals using Prolog's record or assert. This is possible only because suspended goals are ground and workflow variables are used only by suspendable predicates (e.g., they cannot be unified). If suspendable goals contained variables, these would be lost (renamed) on copying the goal to the database. Using attributed variables may not be an advantage in Workflow Prolog because of the need for persistence (Section 2.4): between the times of suspending a goal and waking it up, the whole structure needs to be saved to and restored from a file.

2.4 Persistence and Waking Goals

As described in Section 2.3, each top-level goal in a Workflow Prolog query may succeed, fail, or suspend. If a goal fails, the query fails. If all goals succeed, the query succeeds. If some goals suspend, the query terminates with goals suspended. Each suspended goal may be suspended on one or more workflow variables; each workflow variable may have one or more goals suspended on it.

Goals are never woken up during a query, because it is impossible for a Workflow Prolog program to give a workflow variable a value. They are woken only when participants reply on workflow variables as the result of completing an activity. Activities, and especially processes (comprising many activities) may take days, weeks, or even years to complete, so a Workflow Prolog program may run for a long time, spending most of its time suspended, waiting for participants to reply. Therefore, Workflow Prolog is based on *persistent* execution: when a query terminates, normally with suspended goals, the query's state is saved in a file for later use. This includes:

- all suspended goals;
- the workflow variables on which each goal is suspended;
- the goals that are suspended on each workflow variable;
- the message time (for all workflow variables);
- the reply content and reply time (for workflow variables that have been replied to).

There are two important properties of the suspended goals that are saved in the state. First, because of the language restriction (Section 2.3), they contain no unbound variables, though they might (and normally do) contain workflow variables. Second, they are all suspendable goals from the top level of the query, not including suspendable guard goals or suspendable goals called from guards, because the state of blocked guards is never saved.

When a participant replies on a workflow variable `WV`:

- The state is read from the state file.
- The reply content and reply time of WV are stored in the state.
- The goals suspended on `WV` are removed from `WV` and all other workflow variables they are suspended on, and converted to a Prolog conjunction by appending them together in any order (because they are not dependent).
- The woken conjunction is executed in the same way as the initial query. We call this a *reply query*.
- The updated state is saved in the state file.

All of these steps are done mutually exclusively: if two participants reply about the same time, one will need to wait until the other has updated the state.

This way, a Workflow Prolog query is solved in *phases*: first the initial query and then a reply query phase whenever a participant responds. Eventually the query may succeed or fail, whereupon the state file may be deleted. Naturally, if many queries are active at any time, they are independent and each have their own state file.

Implementation. Workflow Prolog is a coroutining language, not a concurrent one: there is no need for a *run queue*, which concurrent systems normally use to store runnable threads that have been newly created, woken, or preempted. This is because:

- Each query, and reply query, is a Prolog conjunction, which is executed as a single thread. While executing, a query may spawn threads (each thread being a suspendable goal) but these are always suspended, not runnable; any suspendable goal that is runnable will be run in place by the query thread.
- Suspended goals are never woken up during a query.

Because there is no runnable queue, there can be no preemptive scheduling. Therefore, the programmer is responsible for avoiding nonterminating goals, but this is no more onerous than in Prolog. Assuming no nonterminating goals, fairness is assured. This is because whenever a goal suspends, it will not be woken up again until *all* goals in the query have suspended or succeeded.

2.5 Deep and Nested Guards

A guard can be an arbitrary Prolog conjunction, excluding `rpc` goals and other goals with side-effects. In this sense, guards can be *deep*, as in the earliest concurrent logic programming languages [25]. (Later languages used *flat* guards — restricted to certain built-in predicates — for implementation reasons.) However, although guards may be deep, they should be kept small, because they are executed again from the beginning whenever the parent goal is woken after being suspended.

A very useful kind of deep guard is a *nested* guard: one that calls a user-defined suspendable goal that is itself defined by clauses with guards. These can be used for what Conlon [7] called "peeling or-parallelism". Even without or-parallelism, nested guards are useful for expressing a disjunction between an arbitrary number of alternatives. The suspension criteria for a goal are determined dynamically, which cannot be done with flat guards or the `freeze` or `when` predicates provided in some versions of Prolog.

For example, the code in Program 1-3 works only for two markers, because of the definition of the `decide` predicate. To change the program to allow *N* markers, `decide` would need to be rewritten to use *N*+1 clauses. A better solution is to generalize it to allow an arbitrary number of markers by exploiting nested guards, as shown in Program 4. The first clause of `decide` handles an '`ok`' reply from *all* markers while the second clause handles comments from *any* marker.

```
decide(Rs,_,Proposal,Student) :- all_ok(Rs) :
  rpc(database,accepted(Student,Proposal),D),
  replycontent(D,ok).
decide(Rs,Markers,Proposal,Student) :-
  one_comment(Rs,C) :
  proposal(Student,revise(Proposal,C),Markers).

all_ok([]).
all_ok([R|Rs]) :- replycontent(R,ok), all_ok(Rs).

one_comment([R|_],C) :- replycontent(R,c(C)) : true.
one_comment([_|Rs],C) :- one_comment(Rs,C) : true.
```

Program 4. Nested guards

2.6 Time Constraints

Workflow Prolog handles time in a more declarative way than by simply assuming the existence of a "timer" activity. As mentioned above, each workflow variable has a message time and reply time. Each of these can be obtained, respectively, by:

`messagetime(WV,MT)`: This is not suspendable; it unifies `MT` with `WV`'s message time, assuming that `WV` is a workflow variable on which a message has been sent.

`replytime(WV,RT)`: This is suspendable: it suspends on workflow variable `WV` if it has no reply yet, and otherwise unifies `RT` with `WV`'s reply time.

The final type of time predicate is the *precedence constraint* , `T1<<T2`, which allows testing of a workflow variable's reply time even *before* a reply has been received. `T1<<T2` is a suspendable goal. `T1` and `T2` must each be an absolute time or of the form `replytime(WV)` where `WV` is a workflow variable. The four forms of this constraint, where $\mathtt{AT}_i$ is an absolute time and $\mathtt{WV}_i$ a workflow variable, are:

`AT1 << AT2`: Succeeds if `AT1` precedes `AT2`, fails otherwise.

`replytime(WV1) << replytime(WV2)`: If both `WV1` and `WV2` have been replied to, succeeds or fails, depending on their actual reply times. If neither has been replied to, suspends on both `WV1` and `WV2`. If `WV1` has been replied to but `WV2` has not, succeeds. If `WV2` has been replied to but `WV1` has not, fails.

`replytime(WV) << AT`: If `WV` has been replied to, succeeds if `WV`'s actual reply time is earlier than `AT`, otherwise fails. If not, and the current time is later than `AT`, fails. Otherwise, suspends on both `WV` and time `AT`.

`AT << replytime(WV)`: If `WV` has been replied to, succeeds if `WV`'s actual reply time is later than `AT`, otherwise fails. If not, and the current time is later than `AT`, succeeds. Otherwise, suspends on both `WV` and time `AT`.

In general, a suspended goal may now be suspended on workflow variables and/or time events. Goals suspended on a *time event* are woken up any time soon after the corresponding time is reached (by a special process or thread, or operating system scheduler), in the same way as goals suspended on workflow variables are woken up. We call this a *time query*.

The precedence constraint can be used to implement a timeout. In Program 4, suppose that each marker is required to respond within 24 hours; otherwise, the activity should be cancelled and the proposal sent to another marker for approval. Program 5 shows a variant of the `decide` predicate that does this. The first two clauses handle replies by markers, as before. The new third clause handles the case where one marker has not replied before the deadline: it invokes a new `approve_proposal` activity and replaces the old activity's workflow variable by the new one.

```
decide(Rs,_,Proposal,Student) :- all_ok(Rs) :
  rpc(database,accepted(Student,Proposal),D),
  replycontent(D,ok).
decide(Rs,Markers,Proposal,Student) :-
  one_comment(Rs,C) :
  proposal(Student,revise(Proposal,C),Markers).
decide(Rs,Markers,Proposal,Student) :-
  one_late(Rs,Rs1) :
  choose(Markers,Marker),
  rpc(Marker,approve_proposal(Proposal),R),
  decide([R|Rs1],Markers,Proposal,Student).

one_late([R|Rs],Rs) :- messagetime(R,T),
  Deadline is T+86400, Deadline << replytime(R) : true.
one_late([R|Rs],[R|Rs1]) :- one_late(Rs,Rs1) : true.
```

Program 5. Implementing a timeout using time constraints

Restrictions. A `replytime(WV,RT)` goal can only occur in a guard (unless `RT` is ground, which is unlikely), and must appear at the Prolog top level. `T1<<T2` must be ground and must appear at the Prolog top level.

Implementation. Time constraints can be implemented by, at regular intervals, restoring the state from the state file, finding time events that are earlier than the current time, executing goals that are suspended on them as a time query, and saving the updated state. This could be done by a process that runs continuously or by using the operating system's scheduler, such as the cron utility of Unix.

2.7 Data Flow

Although we have concentrated on control flow issues, the use of message passing via workflow variables also allows data to be transferred to and from activities (as the message and reply content). For real applications, this would not be a ground Prolog term, but might need to be a file: the WFMS would send to the participant the *contents* of the file, not just a filename. Similarly, for a file appearing in the reply content, data would be received from the participant and stored in a local temporary file at the time of reply. This implies that Workflow Prolog's persistent state (Section 2.4) may be extended to include a set of temporary files.

2.8 Cancellation and Garbage Collection

In WFMSs it is sometimes necessary to cancel activities after they have started. For example, in Program 1-3, an `approve_proposal` activity is invoked in two or more markers at the same time: if one marker replies with comments, the activity in each of the other markers should be cancelled.

It is easy to identify redundant activities: those whose workflow variable no longer appears in the suspended goals in the state. These are cancelled automatically by the garbage collector, which can also delete temporary files from the state (Section 2.7).

The garbage collector (which could be run at the end of each query phase or performed by a separate process that runs from time to time) identifies each workflow variable `WV` that appears in the state but is not referenced by the suspended goals, and:

- If `WV` has not been replied to, the corresponding activity is cancelled.
- If WV has been replied to, and the reply includes temporary files, these are deleted.

To cancel an activity, for an automated participant, the process executing the activity is killed. In the case of a human participant, a cancellation message is sent. It is sufficient to include in the cancellation message only the identifier of the activity to be cancelled, though in practice these messages might be customized to include an explanatory comment. E.g., a cancellation message sent to markers might inform them that they need not approve the proposal sent earlier because it is now obsolete.

It is conceivable that an activity should not be cancelled even if its result is not needed, e.g., if it has some permanent side-effect. In such cases, the programmer must make sure that the activity's workflow variable always appears in the suspended goals, if necessary by adding a special `replycontent` goal for this purpose.

2.9 Shared Workflow Variables

In the above examples, each concurrent workflow "thread" comprised only a single activity, but in general a thread may contain a sequence of activities or be even more complex. In these cases, each thread is represented by a suspendable goal which monitors its progress. Sometimes it is necessary for threads to synchronize or communicate with each other. This can be done by having some workflow variables shared between the corresponding suspendable goals. Only one goal can send a message (by `rpc`) on a workflow variable, but all goals can wait for its reply and access its value. To allow a workflow variable `WV` to be shared by several goals it must first be created, by a call to the `wvar` built-in predicate:

```
wvar(WV)
```

For example, consider a process in which candidates have to register for an exam before taking it, such that only one can take the exam at a time: the "exam" activities are mutually exclusive. Mutual exclusion can be implemented by inspecting the reply times of shared workflow variables. Program 6 is for the case of two candidates, `a` and `b`, but can easily be generalized. Each "candidate" thread accesses its own workflow variables and those of others; e.g., in `cand(a,AR,BR,AE,BE)`, `AR` and `AE` represent `a`'s registration and exam activities and `BR` and `BE` represent `b`'s. After a candidate successfully registers for the exam, it waits until all those that registered earlier have completed the exam before taking the exam itself. The `cand1` definition allows this candidate to take the exam if it has registered *and* the other candidate has completed the exam (clause 1) or if it registered earlier than the other did (clause 2).

```
:- wvar(AR), wvar(BR), wvar(AE), wvar(BE),
   cand(a,AR,BR,AE,BE), cand(b,BR,AR,BE,AE).

cand(Name,R,OR,E,OE) :-
  rpc(register,Name,R), cand1(Name,R,OR,E,OE).

cand1(Name,R,_,E,OE) :-
  replytime(R) << replytime(E),
  replytime(OE) << replytime(E) :
  rpc(exam,Name,E), replycontent(E,done).
cand1(Name,R,OR,E,_) :- replytime(R) << replytime(OR) :
  rpc(exam,Name,E), replycontent(E,done).
```

Program 6. Implementing mutual exclusion using shared workflow variables

Restrictions. Like `rpc`, `wvar` must appear at the Prolog top level (i.e., there must be no choicepoints) and it must not appear in a guard.

3 Workflow Patterns in Workflow Prolog

In this section we describe the workflow patterns described in [1] and show how each can be implemented in Workflow Prolog, in an attempt to demonstrate its expressive power as a workflow language. Most of the patterns refer to the proposal/approval example and programs of Section 2.

Pattern 1: Sequence. An activity is executed after completing another activity. Example: an "approve proposal" activity is executed after a "send proposal" activity.

To sequence two activities in Workflow Prolog, the first activity is invoked by an `rpc` goal, a suspendable goal waits for the reply on the first activity's workflow variable (by a clause's guard) and the same clause's body invokes the second activity by another `rpc` goal. See Programs 1-3.

Pattern 2: Parallel Split. A thread splits into two or more threads that are executed concurrently. Example: a "send proposal" activity enables the execution of the following activities "approve proposal" activity by all markers concurrently.

In Workflow Prolog, concurrent activities are each invoked by an `rpc` goal, like the `approve_proposal` activities in Program 3. In general, multiple concurrent *threads* can be invoked in a similar way: as well as an `rpc` goal to invoke each activity, there is a suspendable goal suspended on each activity's workflow variable, which invokes subsequent activities in the thread; an example of this appears in Program 6.

Pattern 3: Synchronization. A number of concurrent threads join together into a single thread. The following activity waits until all of the concurrent threads have completed. Example: an "accepted" activity may be enabled after the completion of "approve proposal" activities in all markers.

In Workflow Prolog, a suspendable goal suspends on all of the workflow variables that indicate the end of each activity or thread. In Program 3, the `decide` goal has this role: it waits for the two `approve_proposal` activities to complete.

Pattern 4: Exclusive Choice. One alternative branch is chosen to be executed. Example: after the execution of an "approve proposal" activity by one or more markers, either an "accepted" activity by the database or a "revise" activity by the student is invoked, depending on the result of the "approve proposal" activity.

In Workflow Prolog this is simply implemented using guards. If the choice between branches is to be made on the basis of the result of a workflow activity, the guard both waits for the activity to complete *and* tests its result, as in Program 3.

Pattern 5: Simple Merge. Two or more alternative branches join together without synchronization (because only one branch has been executed). Example: after one of the markers has rejected a proposal, the proposal must be revised by the student.

The simplest way to implement this is to include the continuation code at the end of every branch. In Program 3, a `proposal` goal appears in both the second and third clauses for `decide`. This duplication is not a problem because, of course, the `proposal` predicate is defined only once.

Pattern 6: Multi-choice. A number of alternative branches exist such that one or more of them can be selected for execution. For example, in a process to assess dissertations, an activity "assess dissertation" is executed by one marker or another, or both of them concurrently.

Because of Workflow Prolog's committed choice, this cannot be implemented with one clause for each alternative, like exclusive choice. It requires an extra clause for each combination of alternatives, or (more flexibly) it could be reformulated as a conjunction, where each conjunct either executes an activity or not, depending on its own condition. This is straightforward to implement; we omit the code for space reasons.

Pattern 7: Synchronizing Merge. Two or more branches join together into a single thread. If more than one branch was active concurrently, these branches are synchronized. Alternatively, if only one branch was chosen, no synchronization is needed. For example, after the execution of "assess dissertation" by one or both markers, an activity "store mark in database" is enabled.

In Workflow Prolog, a suspendable goal can suspend on a variable number of workflow variables, each being the reply to one of the concurrent activities or threads invoked in a multi-choice construct. This allows the implementation of a form of synchronizing merge, if not a fully general one. We omit the code for space reasons.

Pattern 8: Multi-merge. Two or more branches join together into a single one without synchronization, such that if more than one branch is executed (concurrently), the activity that follows the merge is executed for every branch. For example, after the execution of "assess dissertation" by *each* marker, the activity "store mark in database" is enabled for that marker's mark only.

Multi-merge can be implemented in Workflow Prolog as easily as synchronizing merge. Again, we omit the code, which is similar to that for the previous patterns.

Pattern 9: Discriminator. Two or more branches join together into one, such that when the first is completed the following activity is enabled. The program waits for the remaining branches (if any) to complete but then ignores them. Example: in

Section 2, a comment is required from one marker. Using a discriminator would still act on the first marker's comment but would wait for responses from the other markers.

A discriminator can be implemented in the same way as pattern 2 (parallel split) but with additional suspendable goals to wait for, and ignore, all but the first "branch" that completes. For example, the `decide` predicate of Program 4 is modified by adding a conjunction of `replycontent` goals to the body of the second clause.

Pattern 10: Arbitrary Cycles. One or more activities can be executed repeatedly. Example: in Section 2, the "revise" and "approve proposal" activities may need to be executed several times before the "accepted" activity is executed.

Cycles can be implemented by tail recursion. The result of an activity is tested in a clause guard; the recursive clause has a body that invokes a new activity and tests its result in a tail-recursive goal, like the `proposal` goal in `decide` in Program 3.

Pattern 11: Implicit Termination. If a subprocess ends (no activities are executed and none of them can be enabled) then the subprocess is terminated. In all the above examples, the processes should terminate when all activities are complete.

Workflow Prolog programs terminate implicitly because, as in other logic programming languages, there is no explicit "halt" instruction.

Pattern 12: Multiple Instances without Synchronization. Multiple instances of an activity are invoked that are independent, so there is no need to synchronize them.

In Workflow Prolog, each `rpc` goal invokes a new activity and workflow variable; if these appear in different suspendable goals, or none, they are independent.

Pattern 13: Multiple Instances with a priori Design Time Knowledge. Multiple instances of an activity are generated which need to synchronize, such that another activity starts when they all complete. The number of instances is known at design time.

This is the same as patterns 2 and 3 (parallel split and synchronization), except that all threads are of the same type. Since we know the number of instances we can include in the program *N* copies of an `rpc` goal, invoking *N* activities, and *N* calls to `replycontent` in a clause guard to wait for them to complete, as in Program 1-3.

Pattern 14: Multiple Instances with a priori Runtime Knowledge. Multiple instances of an activity are generated which need to synchronize, such that another activity may start when they all complete. In this pattern the number of instances is not known at design time but is known before the instances are created.

In Workflow Prolog this is implemented by a conjunction of `rpc` goals and `replycontent` goals, the numbers of which are determined at runtime. For example, in Program 4, the list of markers may be arbitrarily long.

Pattern 15: Multiple Instances without a priori Runtime Knowledge. Multiple instances of an activity are generated which need to synchronize, such that another activity may start when they all complete. In this pattern the number of instances is not known before the instances are created. New instances can be created even if some of the instances are still active or have already ended.

In Program 5, approve_proposal activities are invoked dynamically, depending on the outcome of the approve_proposal activities that are already active.

Pattern 16: Deferred Choice. One of several alternative branches is chosen to be executed. However, the choice is not made in advance, but depends on the environment. For example, in an exam process, a candidate has to register for an exam and then take the exam. Suppose there are two alternative exams he can take, and the choice is determined by which registration attempt succeeds first.

In Workflow Prolog this is implemented by first invoking all of the activities, waiting for one of them to complete, and then cancelling all others, using Workflow Prolog's special garbage collection, as in Program 7.

```
:- rpc(register1,Name,Reg1), rpc(register2,Name,Reg2),
   cand(Name,Reg1,Reg2).

cand(Name,Reg1,_) :- replycontent(Reg1,done) :
  rpc(exam1,Name,Exam), replycontent(Exam,done).
cand(Name,_,Reg2) :- replycontent(Reg2,done) :
  rpc(exam2,Name,Exam), replycontent(Exam,done).
```

Program 7. Pattern 16 (deferred choice)

Pattern 16 is described as a *state-based* pattern in [1] because it involves testing conditions that become true temporarily; e.g., the register1 activity is enabled only until register2 succeeds. However, as van der Aalst *et al.* point out, the same effect can also be achieved with message passing, using "messages to cancel previous messages"; this is what Workflow Prolog does, automatically by garbage collection.

Strictly speaking, redundant activities should be cancelled *atomically* when the required activity completes, but Workflow Prolog cannot do this, because of its asynchronous message passing; it only guarantees that unneeded activities are *eventually* cancelled. This sometimes gives undesirable results; e.g., in Program 7 the candidate might end up registering for both exams. However, in other applications, e.g., Program 1-3 and Program 8, our eventual form of cancellation is perfectly acceptable.

Pattern 17: Interleaved Parallel Routing. A number of activities are executed in an arbitrary order, decided at run time, like parallel split. The difference is that two activities cannot be executed at the same time; i.e., mutual exclusion.

This is described as a state-based pattern in [1] but, unlike pattern 16, it can be implemented in Workflow Prolog without cancellation messages. Mutual exclusion can be implemented by using reply times of workflow variables, as shown in Program 6.

Pattern 18: Milestone. An activity may be enabled (temporarily) only if some thread has reached a certain point in its execution. For example, in an exam process, there could be a milestone between "register" and "exam" activities. An "eat" activity can be executed zero or more times between these two activities.

Like deferred choice (pattern 16), this can be implemented in Workflow Prolog using cancellation messages (from garbage collection). A candidate first completes the register activity, and then tries to invoke both the exam and eat activities. If the eat activity succeeds, the exam activity is cancelled and retried later. If the exam activity succeeds, the process terminates. We omit the program for space reasons.

Pattern 19/20: Cancel Activity/Case. An enabled activity is disabled. For example, in the proposal/approval example of Section 2, one "approve proposal" activity may be cancelled when the other completes. Pattern 20 cancels all activities in a process.

In Workflow Prolog, activities are cancelled automatically by garbage collection. If a thread is waiting for the result of an activity, its suspended goal must be cancelled explicitly, by making it wake up and terminate; the activities will then be cancelled.

Program 8 shows another example of cancellation. A query order(c,b) is invoked when customer c orders book b. It forwards the order to a warehouse activity and starts a cancel_order activity to be performed by the customer. When either of these activities completes, the other is cancelled. This example highlights the need for customizable cancellation messages (Section 2.8). Starting the cancel_order activity involves sending a message like "Your order has been received; if you wish to cancel it, click here", while cancelling this activity would send a message such as "Your order has been despatched and cannot now be cancelled".

```
order(Customer,Book) :-
  rpc(warehouse,order(Customer,Book),Despatch),
  rpc(Customer,cancel_order(Book),Cancel),
  order1(Despatch,Cancel).

order1(Despatch,_) :-
  replycontent(Despatch,done) :: true.
order1(_,Cancel) :-
  replycontent(Cancel,done) :: true.
```

Program 8. Pattern 19/20 (cancellation)

4 Conclusions

4.1 Summary

We have presented a new extension to Prolog that we believe is sufficiently expressive for workflow applications but simple enough to implement efficiently and use.

The rpc predicate is simple but neatly captures a workflow participant's behaviour as a relation on a workflow variable. Garbage collection of workflow variables is fed back to the participant by cancelling activities. Coroutining using guards, especially nested guards, is more powerful than Prolog's freeze or when predicates. Persistence is also essential, and time constraints provide a powerful and declarative way to express time. Implementation is simplified by restricting suspended goals to be ground and because goals are never woken up during a query. A Workflow Prolog process cannot interact with itself or another Workflow Prolog process, because it has no way to act as a server (to receive unsolicited requests), but this is because the language is intended to coordinate workflow activities, not implement them.

In the introduction we argued that concurrent logic programming languages were not well-suited to workflow programming. However, it should be easy to extend them by adding features similar to those that we have proposed (excluding guards, which already exist). The main argument against such languages is that they are still relatively obscure, and probably more powerful than needed for workflow purposes.

We also claimed in the introduction that an (extended) logic programming language should offer a high-level declarative way to develop WFMSs. It is worth considering whether our approach offers any advantage, for workflow programming, over extensions to other well-known general purpose languages. As with other application areas, it is notoriously hard to prove this, since it depends on the languages with which the user is familiar. However, we would point to the fact that a Workflow Prolog program can be a very readable declarative description of the workflow behaviour, especially because of the time constraints.

A prototype implementation of Workflow Prolog has been developed in SWI-Prolog. It includes all features described in this paper, with messages to participants displayed on screen and replies read from the keyboard. Garbage collection is currently implemented only on success or failure of the query. The Workflow Prolog implementation is available from `http://www.cs.bris.ac.uk/~steve/wp/`.

4.2 Related Work

Bi and Zhao [3] use propositional logic to model workflow problems and present a method of verifying them. Concurrent (Constraint) Transaction Logic has been proposed [9, 22] for specifying, verifying, and scheduling workflow problems. Pokorny and Ramakrishnan [21] argue that Generalized Linear Temporal Logic is more suitable than transaction logic because of its ability to express temporal and fairness properties. Ma [19] also proposes linear temporal logic for modelling workflow, extending it in [11] to allow a specification to change over time. Wang and Fan [26] propose the use of Lamport's Temporal Logic of Actions (TLA) for modelling and analysing workflow, and use it to prove the absence of deadlock.

The idea of extending an existing language, instead of developing a new one, for workflow applications was also proposed by Forst *et al.* [13]. Their approach uses a "coordination toolkit" that can be added to computation languages, including Prolog.

The workflow patterns of van der Aalst *et al.* have received a lot of attention since they were published. As a recent example, they were used to evaluate the capabilities of the orchestration language Orc [8]; unlike Workflow Prolog, which is a practical programming language, Orc is a new process algebra. Coordination languages have also been applied to workflow recently by Omicini *et al.* [20].

Acknowledgements. We are grateful to the anonymous referees for some exceptionally detailed and perceptive comments, which have helped improve the presentation of this paper.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M., Kiepuszewski, B., Barros, A.P.: Workflow Patterns. Distributed and Parallel Databases 14(1), 5–51 (2003)
2. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (June 2005)

3. Bi, H.H., Zhao, J.L.: Applying propositional logic to workflow verification. In: Information Technology and Management 5(3-4), 293–318 (2004)
4. Chan, D., Leung, K.: Valmont: a language for workflow programming. In: Proc. 31st Annual Hawaii Int. Conf. on System Sciences, pp. 744–753. IEEE, NJ, New York (1998)
5. Clark, K.L., Gregory, S.: A relational language for parallel programming. In: Proc. Conf. on Functional Programming Languages and Computer Architecture, pp. 171–178. ACM Press, New York (1981)
6. Clark, K.L., Gregory, S.: Parallel programming in logic. In: ACM Trans. on Programming Languages and Systems, vol. 8(1), pp. 1–49. ACM Press, New York (January 1986)
7. Conlon, T.: Programming in Parlog. Addison-Wesley, London, UK (1989)
8. Cook, W., Patwardhan, S., Misra, J.: Workflow patterns in Orc. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 82–96. Springer, Heidelberg (2006)
9. Davulcu, H., Kifer, M., Ramakrishnan, C., Ramakrishnan, I.: Logic based modeling and analysis of workflows. In: Symp. on Principles of Database Systems, pp. 25–33 (1998)
10. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. In: Communications of the ACM, vol. 18(8), pp. 453–457. ACM Press, New York (August 1975)
11. Duan, Y., Ma, H.: Modeling flexible workflow based on temporal logic. In: Proc. 9th Int. Conf. on Computer Supported Cooperative Work in Design, vol. 1, pp. 508–513. IEEE Computer Society Press, Washington (2005)
12. van Emden, M.H., de Lucena Filho, G.J.: Predicate logic as a language for parallel programming. In: Clark, Tärnlund (eds.) Logic Programming, pp. 189–198. Academic Press, London (1982)
13. Forst, A., Kühn, E., Bukhres, O.: General purpose work flow languages. Distributed and Parallel Databases 3(2), 187–218 (April 1995)
14. Halstead, R.: MULTILISP: a language for concurrent symbolic computation. In: ACM Trans. on Programming Languages and Systems, vol. 7(4), pp. 501–538. ACM Press, New York (October 1985)
15. Hermenegildo, M., Cabeza, D., Carro, M.: Using attributed variables in the implementation of concurrent and parallel logic programming systems. In: Proc. 1995 Int. Conf. on Logic Programming, pp. 631–645. MIT Press, Cambridge (1995)
16. Hoare, C.A.R.: Communicating sequential processes. In: Communications of the ACM, vol. 21(8), pp. 666–677. ACM Press, New York (August 1978)
17. Holzbaur, C.: Metastructures vs. attributed variables in the context of extensible unification. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 260–268. Springer, Heidelberg (1992)
18. Liu, D., Wang, J., Chan, S., Sun, J., Zhang, L.: Modeling workflow processes with colored Petri nets. In: Computers in Industry, vol. 49(3), pp. 267–281. Elsevier, Amsterdam (December 2002)
19. Ma., H.: A workflow model based on temporal logic. In: Proc. 8th Int. Conf. on Computer Supported Cooperative Work in Design, vol. 2, pp. 327–332. IEEE, NJ, New York (2004)
20. Omicini, A., Ricci, A., Zaghini, N.: Distributed workflow upon linkable coordination artifacts. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 228–246. Springer, Heidelberg (2006)
21. Pokorny, L., Ramakrishnan, C.: Modeling and Verification of Distributed Autonomous Agents Using Logic Programming. In: Leite, J.A., Omicini, A., Torroni, P., Yolum, p. (eds.) DALT 2004. LNCS (LNAI), vol. 3476, pp. 148–165. Springer, Heidelberg (2005)

22. Senkul, P., Kifer, M., Toroslu, I.: A logic framework for scheduling workflows under resource allocation constraints. In: Proc. 28th VLDB Conf, pp. 694–702 (2002)
23. Shapiro, E.Y.: The family of concurrent logic programming languages. In: ACM Computing Surveys, vol. 21(3), pp. 413–510. ACM, New York (1989)
24. Stefansen, C.: SMAWL: a small workflow language based on CCS. In: Proc. CAISE05 Forum (Porto) (2005)
25. Takeuchi, A., Furukawa, K.: Parallel logic programming languages. In: Shapiro, E. (ed.) 3rd International Conference on Logic Programming. LNCS, vol. 225, pp. 242–254. Springer, Heidelberg (1986)
26. Wang, Y., Fan, Y.: Using temporal logics for modeling and analysis of workflows. In: IEEE Int. Conf. on E-Commerce Technology for Dynamic E-Business, pp. 169–174. IEEE, NJ, New York (2004)
27. Workflow Management Coalition. WFMC specification: the workflow reference model. Document no. WFMC-TC-1003 (1995)
28. Workflow Management Coalition. WFMC specification: terminology and glossary. Document no. WFMC-TC-1011 (1999)
29. Workflow Patterns Home Page. http://www.workflowpatterns.com/

Reactors: A Data-Oriented Synchronous/Asynchronous Programming Model for Distributed Applications

John Field[1], Maria-Cristina Marinescu[1], and Christian Stefansen[2]

[1] IBM T.J. Watson Research Center
{jfield,mariacm}@us.ibm.com
[2] Department of Computer Science, University of Copenhagen
cstef@diku.dk

Abstract. Our aim is to define the kernel of a simple and uniform programming model—the *reactor* model—suitable for building and evolving internet-scale programs. A reactor consists of two principal components: mutable state, in the form of a fixed collection of *relations*, and code, in the form of a fixed collection of *rules* in the style of datalog. A reactor's code is executed in response to an external *stimulus*, which takes the form of an attempted update to the reactor's state. As in classical process calculi, the reactor model accommodates collections of distributed, concurrently executing processes. However, unlike classical process calculi, our observable behaviors are sequences of states, rather than sequences of messages. Similarly, the interface to a reactor is simply its state, rather than a collection of message channels, ports, or methods. One novel feature of our model is the ability to compose behaviors both synchronously and asynchronously. Also, our use of datalog-style rules allows aspect-like composition of separately-specified functional concerns in a natural way .

1 Introduction

In modern web applications, the traditional boundaries between browser-side presentation logic, server-side "business" logic, and logic for persistent data access and query are rapidly blurring. This is particularly true for so-called web mash-ups, which bring a variety of data sources and presentation components together in a browser, often using asynchronous ("AJAX") logic. Such applications must currently be programmed using an agglomeration of data access languages, server-side programming models, and client-side scripting models; as a consequence, programs have to be entirely rewritten or significantly updated to be shifted between tiers. The large variety of languages involved also means that components do not compose well without painful amounts of scaffolding. Our ultimate goal is thus to design a uniform programming language for web applications, other human-driven distributed applications, and distributed business processes or web services which can express application logic, user interaction, and application logic using the same basic programming constructs. Ideally, such a language should also simplify composition, evolution, and maintenance of distributed applications. In this paper, we define a kernel programming model which is intended to address these issues and serve as a foundation for future work on programming languages for internet applications.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 76–95, 2007.

The reactor model is a synthesis and extension of key ideas from three linguistic foundations: synchronous languages [2, 4, 7], datalog [15], and the actor model [1]. From datalog, we get an expressive, declarative, and readily composable language for data query. From synchronous languages, we get a well-defined notion of "event" and atomic event handling. From actors, we get a simple model for dynamic creation of processes and asynchronous process interaction.

A reactor consists of two principal components: mutable state, in the form of a fixed collection of *relations*, and code, in the form of a fixed collection of *rules* in the style of datalog [15]. A reactor's code is executed in response to an external *stimulus*, which takes the form of an attempted update to the reactor's state. When a stimulus occurs, the reactor's rules are applied concurrently and atomically in a *reaction* to yield a *response* state, which becomes the initial state for the next reaction. In addition to determining the response state, evaluation of rules in a reaction may spawn new reactors, or generate new stimuli for the currently executing reactor or for other reactors. Importantly, newly-generated stimuli are processed *asynchronously*, in later reactions. However, we provide a simple mechanism to allow collections of reactors to react together as a unit when appropriate, thus providing a form of distributed atomic transaction.

As in classical process calculi, e.g., pi [12], the reactor model accommodates collections of distributed, concurrently executing processes. However, unlike classical process calculi, our observable behaviors are sequences of *states*, rather than sequences of *messages*. Similarly, the interface to a reactor is simply its state ("REST" style [6]), rather than a collection of message channels, ports, or methods. We accommodate information hiding by preventing certain relations in a reactor's state from being externally accessible, and by allowing the public relations to serve as abstractions of more detailed private state (as in database views). A significant advantage of using data as the interface to a component, and datalog as a basis for defining program logic, is that the combination is highly "declarative": it allows separately-specified state updates (written as rules) to be composed with minimal concern for control- and data-dependence or evaluation order.

Contributions. We believe that the reactor model is unique in combining the following attributes harmoniously in a single language: (1) data, rather than ports or channels as the interface to a component; (2) synchronous and asynchronous interaction in the same model, with the ability to generate processes dynamically; (3) expressive data query and transformation constructs; (4) the ability to specify constraints/assertions as a natural part of the core language; (5) distributed atomic transactions; and (6) declarative, compositional specification of functionality in an "aspect-like" manner.

2 Reactor Basics

A reactor consists of a collection of *relations* and *rules*, which together constitute a reactive, atomic, stateful unit of distribution. The full reactor syntax is given in Fig. 2.

Consider the declaration for `OrderEntryA` in Fig. 1. `OrderEntryA` defines a class of reactors that are intended to log orders—say, for an on-line catalog application. Reactor *instances* are created dynamically, using a mechanism we will describe shortly.

```
def OrderEntryA = {
  (* orders: id, itemid, qty *)
  public orders: (int, int, int).
  (* log: id, itemid, qty *)
  log: (int, int, int).

  log(id, itemid, qty) <-
    orders(id, itemid, qty).
}

def OrderEntryA′ = {
  (* ... same as OrderEntryA ... *)

  not log(id, itemid, qty) <-
    not orders(id, itemid, qty).
}
```

```
def OrderEntryB = {
  (* ... same as OrderEntryA ... *)

  (* orderIsNew is true if order has
     not previously been logged  *)
  ephemeral orderIsNew: ().
  orderIsNew() <-
    orders(id, itemid, qty),
    not -log(id, itemid, qty)
}

def OrderEntryC = {
  (* ... same as OrderEntryB ... *)

  (* delete any new order whose id is
     duplicate of a prev. logged id  *)
  not orders(id, itemid, qty) <-
    ^orders(id, itemid, qty),
    not -log(id, itemid, qty),
    -log(id, _, _).
}
```

Fig. 1. Order entry: variations

Relations. The state of a reactor is embodied in a fixed collection of *persistent relations*. Relations are sets of $(\tau_1, ..., \tau_n)$ tuples, where each τ_i is one of the types `int`, `string`, *enum-type-name*, or `ref` *reactor-type-name*. The primitive types have the usual meanings. Enumerations introduce a new type ranging over a finite set of constants. *Reactor references*, of the form `ref` *reactor-type-name*, are described in Section 4. Relations are empty when a reactor is instantiated. In addition to persistent relations, whose values persist between reactions, a reactor can declare *ephemeral* relations. These relations can be written and read in the same manner as persistent relations, but they are re-initialized with every reaction.

In the case of `OrderEntryA`, its state consists of two persistent relations, `orders` and `log`, each of which is a collection of 3-tuples of integer values. Relation `orders` has *access annotation* `public`, which means that the contents of `orders` may be read or updated by any client. By "update", we simply mean that tuples may be added to or deleted from `orders`; no other form of update is possible. Relation `log`, lacking any access annotation, is *private*, the default, and may thus only be read or updated by the reactor that contains `log`.

```
        REACTOR ::= def reactor-type-name = { {DECL .}* }
           ENUM ::= enum enum-type-name = { {atom-name ,}+ }
           DECL ::= ( RELATION-DECL | RULE-DECL ) .
  RELATION-DECL ::= [public | public write | publiac read][ephemeral] rel-name : ( {TYPE ,}* )
      RULE-DECL ::= HEAD-CLAUSE <- BODY
           BODY ::= {BODY-CLAUSE ,}+
    HEAD-CLAUSE ::= [not] [var-name .]rel-name[ˆ] ( {(_ | var-name | new reactor-name) ,}* )
    BODY-CLAUSE ::= [not] [var-name .][ˆ|-]rel-name ( {(_ | var-name) ,}* ) | BASIC-PREDICATE
BASIC-PREDICATE ::= EXP (< | > | <> | =) EXP
           TYPE ::= int | string | enum-type-name | ref reactor-type-name
            EXP ::= var-name | NUMERIC-LITERAL | STRING-LITERAL | EXP (+ | - | * | / | %) EXP | self
```

Fig. 2. Reactor syntax

The reaction process. A reaction begins when a reactor receives an *update bundle* from an external source. An update bundle is a total map from the set of relations of the recipient to pairs of sets (Δ^{+}, Δ^{-}), where Δ^{+} and Δ^{-} are sets of tuples to be added and deleted, respectively, from the target relation, and $\Delta^{+} \cap \Delta^{-} = \emptyset$. An update bundle should contain at least one non-empty set, i.e. completely empty update bundles are not well-formed. In the examples that follow, an update bundle will typically contain an update to a single relation, usually adding or deleting only a single tuple. However, an update bundle can in general update any of the public relations of a reactor, and add and delete arbitrary number of tuples at a time.

The state of a reactor before an update bundle is received is called its *pre-state*. A reaction begins when an update bundle is applied atomically to the pre-state of a reactor, yielding its *stimulus state*. The stimulus state of a reaction is (conceptually) a copy of each relation of the reactor with the corresponding updates from the update bundle applied. So, for example, in the case of `OrderEntryA`, if relation `orders` contained the single tuple $(0, 1234, 3)$ prior to a reaction, and a reaction is initiated by applying an update bundle with $\Delta^{+} = \{(1, 5667, 2)\}$ and $\Delta^{-} = \emptyset$, then the stimulus value of `orders` at the beginning of the reaction will be the relation $\{(0, 1234, 3), (1, 5667, 2)\}$. We will refer to the "value of relation r in the stimulus state" and "the stimulus value of r" interchangeably.

If a reactor contains no rules, the state of its relations at the end of a reaction—its *response state*—is the same as the stimulus state, and the reaction stores the stimulus values back to the corresponding persistent relations. Hence in its simplest form, a reaction is simply a state update. However, most interesting reactors have one or more rules which compute a response state distinct from the reactor's stimulus state (Section 3). Rule evaluation can also define sets of additions and deletions to/from the *future state* of either local relations or—via reactor references—relations of other reactors. These sets form the update bundles—one bundle per reactor instance referenced in a reaction—that initiate subsequent reactions in the same or other reactors. Update bundles thus play a role similar to messages in message-passing models of asynchronous computation.

It is important to note that from the point of view of an external observer, a reaction occurs *atomically*, that is, no intermediate states of the evaluation process are externally observable, and no additional update bundles may be applied to a reactor until the current reaction is complete.

Fig. 3 illustrates the life cycle of a typical collection of reactors, both from the point of view of an external observer (the top half of the figure) and internally (the bottom half of the figure schematically depicts reactor M during reaction i). The pre-state of reactor M during reaction i is labeled $\texttt{-}S_i$, its stimulus state is labeled $\texttt{\^{}}S_i$, its future state is labeled $S\texttt{\^{}}_i$, and its response state is labeled S_i. The terms pre-state, stimulus state, response state, and future state are meaningful only *relative* to a particular reaction, because one reaction's response state becomes the next reaction's pre-state, and references to a reactor's future state are used (along with the pre-state) to define the stimulus state for a subsequent reaction. The only "true" state, which persists between reactions, is the response state. An external observer therefore sees only a sequence of response states (more specifically, the response values of public relations). Each rule of a reactor can refer programmatically to relation values in all four states: it can read the pre-state of a relation (schematically depicted as `-r` in Fig. 3), the stimulus state (`^r`), and the response state (`r`); it can write the response state and the future state (`r^`).

3 Rules

Basic rule evaluation. Reactor rules (Fig. 2) are written in the style of datalog [15, 14]. The single rule of `OrderEntryA` can be read as "ensure that `log` contains whatever tuples are in `orders`". The right-hand side, or *body* of a reactor rule consists of one or more *body clauses*. In `OrderEntryA`, there is only one body clause, a *match predicate* of the form `orders(id, itemid, qty)`. A match predicate is a pattern which binds instances of elements of tuples in the relation named by the pattern (here, `orders`) to variables (here, `id`, `itemid`, and `qty`). As usual, we use '`_`' to represent a unique, anonymous variable. Evaluation of the rule causes the body clause to be matched to *each* tuple of `orders` and binds variables to corresponding tuple elements. Since the *head clause* on the left side of the rule contains the same variables as the body clause, it ensures that `log` will contain every tuple in `orders`.

In general, a RULE-DECL can be read "for every combination of tuples that satisfy BODY, ensure that the HEAD-CLAUSE is satisfied" (by adding or deleting tuples to the relation in HEAD-CLAUSE). The semantics of datalog rule evaluation ensures that no change is made to any relation unless necessary to satisfy a rule, and—for our chosen semantics—that rule evaluation yields a unique fixpoint result in which all rules are satisfied. Although our rule evaluation semantics is consistent with standard datalog semantics, we have made several significant extensions, including head negation, reference creation, and the ability to refer to remote reactor relations via reactor references.

Returning to reactor `OrderEntryA`, let us consider the case where the pre-state values of `orders` and `log` are, respectively, $\{(0, 1234, 3)\}$ and$\{(0, 1234, 3)\}$, and an update bundle has $\Delta^+ = \{(1, 5667, 2)\}$ and $\Delta^- = \emptyset$. Then the stimulus value of `orders` will be equal to $\{(0, 1234, 3), (1, 5667, 2)\}$. No rule affects the value of `orders`, so the response value of `orders` will be the same as the stimulus value. In the case of `log`, rule evaluation yields the response state $\{(0, 1234, 3), (1, 5667, 2)\}$, i.e., the least change to `log` consistent with the rule.

Now, starting with the result of the previous `OrderEntryA` reaction described above, consider the effect of applying another update bundle such that $\Delta^+ = \emptyset$ and $\Delta^- = \{(0, 1234, 3)\}$. This reaction will begin by deleting $(0, 1234, 3)$ from `orders`, yielding the stimulus state `orders` $= \{(1, 5667, 2)\}$, `log` $= \{(0, 1234, 3), (1, 5667, 2)\}$. Evaluating the rule after the deletion has no net effect on `log` (since the only remaining tuple in `orders` is already in `log`), hence we get the response state `orders` $= \{(0, 1234, 3)\}$ and `log` $= \{(0, 1234, 3), (1, 5667, 2)\}$. We thus see that the effect of this rule is to ensure that `log` contains every orderid ever seen in `orders`. If we wanted to ensure that `log` is maintained as an *exact* copy of the current value of `orders` (which would mean that it is no longer a log at all), we could add the additional *negative* rule depicted in definition `OrderEntryA'` in Fig. 1. The negative rule of `OrderEntryA'` has the effect of ensuring that if an orderid is not present in `orders`, it will also be absent from `log`; i.e., it encodes tuple *deletion*. While negation is commonly allowed in body clauses for most datalog dialects, negation on the head of a rule is much less common (though not unheard of, see, e.g., [16]).

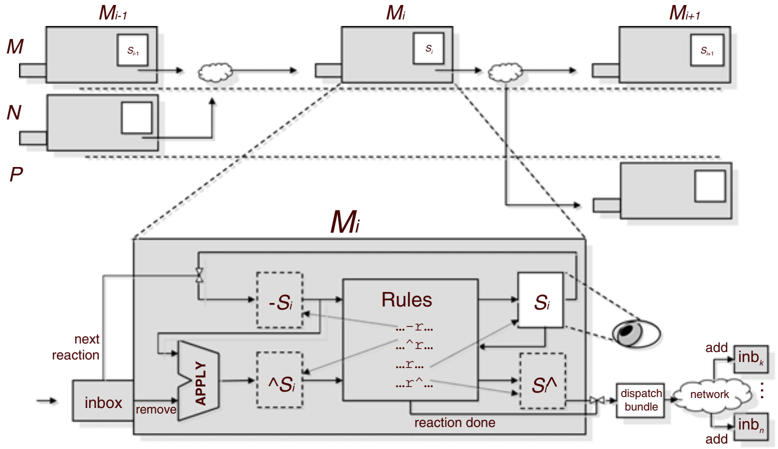


Fig. 3. Reaction schematic

Pre-state value and stimulus values of relations. Reactor definitions `OrderEntryB` and `OrderEntryC` of Fig. 1 add additional rules that further refine the behavior of `OrderEntryA`, using references to both pre-state and stimulus values of relations. `OrderEntryB` defines an *ephemeral* nullary relation `orderIsNew`, which functions as a boolean variable, initially false. The new rule in `OrderEntryB` sets `orderIsNew` to true (i.e., adds a nullary tuple) if `orders` contains a value not found in `log` prior to the reaction (i.e., in `log`'s pre-state value). The definition of `OrderEntryC` further refines `OrderEntryB` by causing any new order whose orderid is a duplicate of a previously logged orderid to be deleted. The new rule in `OrderEntryC` must distinguish the stimulus value of `orders`, i.e., `^orders` from its response value, i.e., `orders`, since the rule defines the response value to be something different from the stimulus value in the case where a duplicate order `id` is present.

It is important to note that the result of rule evaluation is oblivious to the order in which rules are declared. We believe this feature makes it much easier to update the functionality of a reactor by changing the rule set without concern for control- or data-dependencies. We see this feature demonstrated in the progression of examples depicted in Fig. 1, where rules can be mixed and matched liberally to yield updated functionality. Thus rules allow orthogonal functional "concerns" to be specified in an aspect-like fashion [10].

Initialization, constants, and reaction failure. Consider the pair of rules `r(x) <- s(x)` and `not r(x) <- s(x)`. These rules are inherently contradictory, since they require that `x` be both present and absent from relation `r`. In such cases, a *conflict* results. Because rules are conditionally evaluated, conflicts cannot in general be detected statically and must be detected during rule evaluation. If such a conflict occurs, the reaction *fails*: the reactor rolls back to its pre-state and no update bundles are dispatched.

Consider the reactor definition `Cell` depicted in Fig. 4. Each instance of a `Cell` is intended to hold exactly one value. Instances of `Cell` contain two relations: a public unary relation `val` containing the publicly-accessible value of the cell, and a private nullary (i.e., boolean) relation `live`. Recall that a reactor's relations are initially empty

```
def Cell = {
  public val:(int).
  live:().

  (* initializations *)
  (*1*) live() <- .
  (*2*) val(0) <- not -live().
```

```
  (* singleton constraint: if not
     satisfied, reaction fails;
     reactor rolls back *)
  (*3*) not live() <-
          val(x), val(y), x <> y.
}
```

Fig. 4. Cell

when the reactor is instantiated. Rules (1) and (2) together define an idiom which will allow us to initialize relations to non-empty values. First, consider rule (1). Rule (1) is an *unconditional rule*, and is an instance of the shorthand notation depicted in Fig. 5(b). Rule (1) defines `live` to be a a *constant*, since its response value evaluates to non-empty (i.e., "true") at the end of every reaction. Because of rule (1), `-live` in rule (2) is nonempty only during the first reaction in which the `Cell` is instantiated. Hence `val` will be initialized to 0 only once, in the reaction in which `Cell` is created. Thereafter, `-live` will be non-null, and the initialization will not recur, allowing `val` to be freely updated to arbitrary values.

Finally, consider rule (3) of `Cell`. The three clauses in its body collectively check to see whether `val` contains more than one value, i.e., whether it is a singleton. If not, the rule requires that its goal clause (left-hand side) be satisfied, i.e., that `live` be set to empty (false). However, any such attempt is inconsistent with the assertion in rule (1) that `live` is non-empty (true), hence any attempt to update `Cell` without maintaining the singleton invariant will result in a conflict and reaction failure. We thus see that the reactor model allows "assertions" and "integrity constraints" in the style of databases to be expressed in precisely the same form as rules that express state updates. When some assertion fails, the reaction rolls back. Fig. 5(d) depicts a notational convention that will allow us to use `FAIL` to define rules that represent assertions.

4 Asynchronous Reactor Composition

Up to this point, we have not explained how update bundles are generated, only how reactors react when an update bundle is applied. In this section, we show how updates are generated, and explain how this is intimately connected to asynchronous interaction.

Single-reactor asynchrony. Consider the reactor definition `Fibonacci` in Fig. 6, which computes successive values of a Fibonacci series. The relation `series` contains pairs whose first element is the ordinal position of the sequence value, and whose second element is the corresponding value of the sequence. The value of `series` is initialized using notational conventions (e) and (f) of Fig. 5. To compute the next element of the series, we need to first identify the last two elements of the series computed thus far. Universal quantification is required to determine the maximum element of a series; however, the body of a datalog rule can essentially encode only existential properties. To compute universal properties, we typically require auxiliary relations, thus, e.g., in `Fibonacci`, we use the ephemeral relation `notLargest` to contain all the indices of elements of `series` which are less than the maximum index.

	Notation	Translation	Comments

a	r1(exp_0) <-...ri(exp_i)....	r1(x0) <- ...ri(xi)...,
		x0 = exp_0, ...,
		xi = exp_i.
			Expressions exp$_i$ are instances of non-terminal EXP in Fig. 2; the xi are fresh variables.

b	*head* <- .	*head* <- 0 = 0.	
c	r(x1,...xn) := *body*.	r(x1,...xn) <- *body*. not r(x1′,...xn′) <- *body*, -r(x1′,...xn′), x1′ <> x1. ... not r(x1′,...xn′) <- *body*, -r(x1′,...xn′), xn′ <> xn.	
d	FAIL <-....	not live() <-....	Assumes the following definitions exist: live: (). live() <- not -live().

e	$body_0$:{
	$head_1$ <- $body_1$.
	...
	$head_n$ <- $body_n$.
	}
		$head_1$ <- $body_1$, $body_0$.
		...
		$head_n$ <- $body_n$, $body_0$.	

f	INIT	not -live()	Assumes same definitions as (d).
g	$head_1$, ···, $head_n$ <- *body*.	$head_1$ <- *body*. ... $head_n$ <- *body*.	

Fig. 5. Notational conventions

```
def Fibonacci = {
  (* complete series thus far: 1st elt.
     is index, 2nd elt. is value *)
  public read series: (int, int).
  (* must be true for reactor to run *)
  public write run: ().
  (* holds indices in the sequence
     less than the maximum *)
  ephemeral notLargest: (int).

  INIT: {
    series(1,0), series(2,1) <- .
    run() <- .
  }

  (* indices in series less than max *)
  (*1*) notLargest(n) <- series(n, _),
          series(n′, _), n′ > n.

  (* compute next series value *)
  (*2*) series^(n, x1+x2) <-
          not notLargest(n),
           series(n-1, x1),
          series(n, x2).

  (* halts if "run" set to false *)
  (*3*) FAIL <- not -run(), not ^run().
}
```

Fig. 6. Self-Reacting Fibonacci

Note that the relation in the head of rule (2) computing the next value of the Fibonacci sequence has the form series^. A relation name of this form refers to the *future state* of the relation. The future state defines the contents of an update bundle which is processed *after* the current reaction ends, in a subsequent reaction. One can thus think of the future value of a relation as defining an asynchronous update or dispatching a "message". As a result, successive values of the series are separately visible to external observers as they are added to the list. Rule (3) results in failure if both the pre-state and the stimulus values of run are false, thus preventing further updates to the series from being generated. Instances of Fibonacci can react to two distinct classes of update bundles: "internally" generated update bundles containing only new values of the series, and client-generated update bundles which only affect the value of run. A client cannot update series since series is not public. The Fibonacci reactor does not produce update bundles affecting the value of run, since it has no rules referring to the future value of run.

In general, distinct reactors operate *concurrently* and *independently*. Given this fact, it is possible for an update bundle to be generated by a client attempting to update the value of run while a previous reaction by the same instance is in progress. Since reactions take place atomically, we must enqueue pending client updates until the current reaction is complete. To this end, every reactor has an associated *inbox* queue containing a multiset of pending update bundles. When a reaction is complete, the reactor checks for a new update bundle in the queue. If it exists, the reactor dequeues it and uses it to initiate a reaction. If no update bundle is present, the reactor performs no further computation until a new update arrives. We make no assumptions about the order in which inbox items are processed, except that they must be processed fairly. Fig. 3 illustrates this process.

Reactor references and multi-reactor asynchrony. Until now, our examples have only considered a single reactor type. Consider now the definitions for reactors Sample, Sensor, and Nonce depicted in Fig. 7. Reactor types Sample and Sensor encode a "classical" asynchronous request/response interaction. To enable two reactor instances to communicate, we use *reactor references* such as those stored in relation rSensor. Rule (1) of Sample has the effect of dispatching an asynchronous request for the sensor value (maintained a Sensor reactor) whenever a client of Sample updates pulse. The expression s.req^(self) in Rule (1) contains an *indirect reference* to relation rSensor: after the reactor reference stored in relation rSensor is bound to variable s, we refer to relation req of the sensor instance indirectly using the expression s.req. Since we refer to the future value of s.req, an asynchronous update bundle is dispatched to the Sensor instance. The update bundle contains a self-reference to the requesting Sample instance, which is generated by the self construct.

```
def Sample = {
  (* rSensor: ref. to sensor; assumed
     to be initialized by client *)
  public rSensor: (ref Sensor).
  (* samples collected thus far; nonces
     distinguish sample instances *)
  public log: (ref Nonce, int).
  (* pulse: set to collect sample *)
  public write ephemeral pulse: ().
  (* response: holds sensor response *)
  public write ephemeral response:
    (int).

  (* request sample when pulse set *)
  (*1*) s.req^(self) <-
          pulse(), rSensor(s).

  (* process response:
     add sample to log *)
```

```
  (*2*) log(new Nonce, r) <-
          response(r).
}

def Sensor = {
  (* set when sample is to
     be collected; value is ref.
     to sample reactor *)
  public write ephemeral req:
    (ref Sample).
  (* val: current sensor value *)
  public val:(int).

  (* send resp. when client sets req *)
  r.response^(v) <- val(v), req(r).

  (* sensor value is a singleton *)
  FAIL <- val(x), val(y), x <> y.
}

def Nonce = {}
```

Fig. 7. Asynchronous query/response

A `Sensor` instance responds to a request (in the form of an update to relation `req`) by dispatching the current value of the sensor back to the corresponding `Sample` instance. It does so by by setting the `Sample`'s `response` relation via the reactor reference sent by the requester. The response is asynchronous, since `r.response^` refers to a future value. The requester processes the response from the `Sensor` instance by updating its `log` relation with the value of the response.

There are two ways of introducing references to reactors. The keyword `self` evaluates to a reference to the enclosing reactor. An expression of the form `new` *reactor-type-name* instantiates a new instance of the given reactor type. Instantiation expressions may only appear in the head of a rule. Rule (2) of `Sample` creates instances of the trivial reactor `Nonce`. A reactor reference is globally unique, hence trivial reactors such as `Nonce` can be used as sources of keys for relations. In particular, `Sample` uses instances of `Nonce` to distinguish multiple instances of the same sensor value. While `Nonce`s contain no rules, in general, when a reactor is instantiated in a reaction, its rules are evaluated along with the rules of the parent reactor, as we shall see in Section 5.

In order to instantiate and connect `Sample` and `Sensor` instances together, another reactor must contain rules of the form `s.rSensor(new Sensor) <- theSampler(s)` and `theSampler(new Sample) <-` . A request-response cycle between `Sample` and `Sensor` instances requires three distinct reactions: the reaction in which a `Sample` client sets `pulse` (which dispatches the request to the sensor), the reaction in which the sensor responds to the request, and the reaction in which the requester updates the value of `log`.

5 Synchronous Reactor Composition

In the example in Fig. 8, an instance of `MiniBank` receives asynchronous requests to transfer money between accounts. As with the example in Fig. 7, we use references to "plumb" the reactors together. However, unlike the previous example, the remote references in Fig. 8 refer to *response* values of relations, not future values. This means that if a reaction is initiated by an update bundle containing a new `req` tuple at an instance of `MiniBank`, the scope of the reaction will *extrude* to include both of the account reactors (referred to by variables `to` and `from`, respectively). This results in a composite, synchronous, atomic reaction involving *three* reactor instances. Scope extrusion is

```
def Acct = {
  (* account balance *)
  public balance: (int).

  (* balance is a singleton *)
  FAIL <-
    balance(x), balance(y), x <> y.

  (* negative balances not allowed *)
  FAIL <- balance(x), x < 0.
}
```

```
def Minibank = {
  (* transfer request:transfer amount,
     to account, from accout *)
  public write ephemeral transferReq:
    (int, ref Acct, ref Acct).

  to.balance(x+amt) :=
    ^transferReq(amt, to, _).

  from.balance(y-amt) :=
     ^transferReq(amt, _, from).
}
```

Fig. 8. Classic Transaction

an inherently dynamic process, similar to a distributed transaction—see Section 6 for details. MiniBank uses the notation of Fig. 5(c) to define "assignments" to singleton relations.

Note that the rules in Acct encode constraints on the allowable values of balance. In a composite reaction, all of the rules of all of the involved reactors must be satisfiable in order for the reaction to succeed. If any of the rules fails, the composite reaction fails, and all of the reactors revert to their pre-reaction states. A composite reaction is always initiated at a single reactor instance at which some asynchronously-generated update bundle is processed—in the case of the example in Fig. 8, reactor instance MiniBank.

The example in Fig. 9 shows how multiple user interface components can be instantiated *dynamically* based on the current contents of an associated database. This mimics the process of building dynamic, data-driven user interface components. The basic idea of DataDisplay is that a button and an output field are generated for each item in a database. ButtonWidget and OutputWidget are reusable user interface components representing the button and output field generated for each item in relation db. The buttons thus generated are "active": pushing them causes the associated data to be updated, which in turn results in updates to the UI. For example, rule (8) "wires" together corresponding button and database items such that when a button is pressed, the corresponding data item is decremented. Rule (7) sets the value of the output field to the value of the quantity currently maintained in the database. The rules for newly-created reactors are evaluated as part of the "parent" reactor that created them.

6 Synchronous Reactions: Scope Extrusion and Locking Details

Scope Extrusion. In response to an update bundle, a reactor evaluates all its rules and while doing so it may extrude the scope of the reaction to include other reactors. Extrusion can happen in two ways: First, when a new reactor is instantiated it is included in the scope of the reaction that caused it to be instantiated. Second, when the response state of any relation (local or remote) is written, the reactor that contains that relation and all reactors containing rules reading that relation are included in the scope of the ongoing reaction. We say a relation is written whenever a rule produces a response-state update for that relation regardless if this results in a state change or not (e.g. adding an tuple that already exists constitutes a write). One important exception is the *passive read*. A passive read is a read from the pre-state of a remote relation. It differs from other remote reads in that writes to the remote relation in itself will not cause the reaction to extrude to the reader.

A reaction is complete when all reactors included in the reaction have reached a state that satisfies their rules. If one or more involved reactors cannot reach a state that satisfies their rules, there has been a *conflict* and all involved reactors roll back to their pre-state, i.e. the state they were in before the update bundle occurred or before they were included in the reaction. If different reactors involved in the same composite reaction separately define future values for relations of the same target reactor instance, the updates are combined into a *single* update bundle, which will be dispatched at the end of the composite reaction to the target reactor. In case the reaction rolls back, no update bundles are produced. In this sense, from the point of view of an external observer, a

```
def ButtonWidget = {
  public label: (string).
  public write ephemeral pressed: ().
}

def OutputWidget = {
  public label: (string).
  public val: (string).
}

def DataDisplay = {
  (* database: itemid, quantity *)
  public db: (int, int).

  (* list of button / output widget
     pairs,indexed by itemid *)
  widgets: (int, ref ButtonWidget,
            ref OutputWidget).

  (* labels of widgets are constants *)
  (*1*) o.label("Inventory: ") <-
          widgets(_, _, o).
  (*2*) b.label("Click to decr") <-
          widgets(_, b, _).

  (* projection of relns. on itemids *)
  ephemeral oldDisplayItems: (int).
  ephemeral currDbItems: (int).
```

```
  (*3*) oldDisplayItems(i) <-
          -widgets(i, _, _).
  (*4*) currDbtems(i) <- db(i, _).

  (* create new child widgets when new
     item added to db *)
  (*5*) widgets(i, new ButtonWidget,
                new OutputWidget) <-
          db(i, _),
          not oldDisplayItems(i).

  (* delete widgets if corresp. items
     removed from db *)
  (*6*) not widgets(i, _, _) <-
          -widgets(i, _, _),
          not currDbItems(i).

  (* output val set to qty of
     corresp. item *)
  {*7*) o.val(toString(q)) :=
          widgets(i, _, o), db(i, q).

  (* button decrements qty. of
     corresp. item *)
  (*8*) db(i, q-1) :=
          widgets(i, b, _),
          b.pressed().
}
```

Fig. 9. Data-Driven UI

composite reaction has the same atomicity properties as a reaction involving a single reactor. If a reaction updates the future state of (local or remote relations in) reactors $C_1, \ldots, C_n$, it produces n different update bundles—one per target reactor—and each C_i will have a separate and independent reaction to its own update bundle. Even if the scope of C_i's reaction should happen to expand to include some other C_j, C_j's update bundle will not be processed (or even visible) in that reaction.

Locking. Conceptually a rule *reads* all relations that appear in the body as well as any relation that appears in negated form in the head. Thus a rule's need for read access can be determined statically. Whether a rule will *write* the head relation when evaluated can generally only be determined at runtime, because the read has to match a non-empty set of facts that satisfy the body for a write to occur[1]. For this reason we will use the term *read* to refer to the static property, regardless of any optimization that might avoid unnecessary reads. The term *write*, on the other hand, will be strictly reserved for the dynamic property, i.e. a head relation is considered to be written only if the body yields at least one match when evaluated.

When a reaction extends to include several reactors, the composite reaction should remain atomic. The following locking conventions ensure this property. A reactor locks when it agrees to react to an update bundle and remains locked for the duration of the reaction until either a quiescent response state is found and committed or a conflict causes the reactor to roll back to its pre-state. When a reactor is locked, it denies any

[1] When a tuple t is present in a relation `r`, we say that `r` t is a fact.

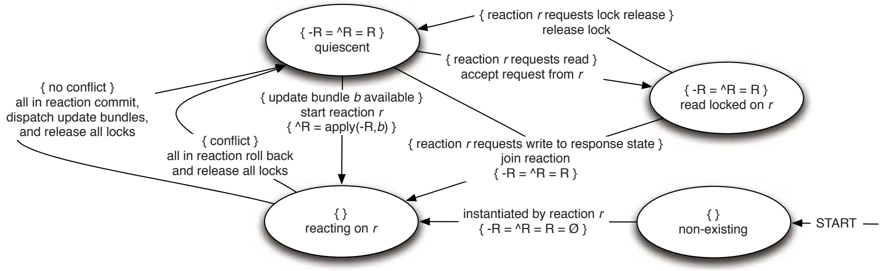


Fig. 10. Reactor state machine. Transitions are written $\{\textsf{precondition}\}\textsf{activity}\{\textsf{postcondition}\}$. States are written $\{\textsf{invariant}\}\textsf{state}$.

interaction (read-access, write-access, and beginning to react to other update bundles) with reactors that are not part of the same reaction. Refer to Fig. 10 for an overview.

Intuitively, *reaction* scope extends to include all reactors owning or (non-passively) reading relations being *written*, while *lock* dynamically extends to include all reactors owning relations being *read* (as well as all reactors in the reaction scope). When a remote reactor is read, an exclusive lock on the reactor is acquired and held for the entire duration of the reaction. Should the same remote reactor need to be written later in the same reaction, this defensive locking strategy guarantees that references to the remote reactor's pre-state and response state will in fact refer to two consecutive states of that reactor because it has not been free to serve any other reactions in the meantime. The problem of deadlock naturally arises in this setting; a reactor implementation would require that standard deadlock detection, avoidance, or recovery techniques (e.g., optimistic concurrency control) be used.

7 Advanced Semantic Issues

The rule evaluation model for reactors extends standard datalog with head clause negation (to express deletion) and reactor references. In this section, we review key aspects of standard datalog semantics and define our extensions. We evaluate the rules by forward derivation (also called bottom-up derivation). This strategy applies all rules on the facts to produce all possible consequences—the appropriate approach for our purpose of creating a completely defined new state.

7.1 Head Clause Negation: Semantics Via Translation

We handle negation in head clauses by transforming reactor rules to normal datalog rules. First, we treat each of the four states of a given relation as distinct relations from the point of standard datalog semantics. The goal of reactor rule evaluation is to determine a unique, minimal solution for the response and future values of local and remote relations. Let r_i represent the response or future value of a local relation we wish to compute (we will consider reactor references shortly). Let r_1^P, ... r_n^P denote the contents of the persistent relations immediately prior to a reaction; if the reactor has just been created the relations are empty by default. Let $\wedge\mathsf{r}_i^{\Delta^+}$, $\wedge\mathsf{r}_i^{\Delta^-}$ be the addition and the deletion sets of the update bundle applied to the reactor. The basic idea for determining the solution to r_i is as follows: (1) introduce a pair of auxiliary relations ($\underline{\mathtt{r}}_i^{\Delta^+}$,$\underline{\mathtt{r}}_i^{\Delta^-}$) which contains the sets of tuples that will be added to and deleted from r_i; (2) eliminate negation in head clauses by transforming the program to a normal datalog program containing references to $\underline{\mathtt{r}}_i^{\Delta^+}$ and $\underline{\mathtt{r}}_i^{\Delta^-}$; (3) evaluate the transformed program using standard datalog semantics; (4) for r_i a response state overwrite $\mathtt{r}_i^P$ to include $\underline{\mathtt{r}}_i^{\Delta^+}$ and exclude $\underline{\mathtt{r}}_i^{\Delta^-}$; for r_i a future state copy ($\underline{\mathtt{r}}_i^{\Delta^+}$,$\underline{\mathtt{r}}_i^{\Delta^-}$) into ($\mathtt{{}^\wedge r}_i^{\Delta^+}$,$\mathtt{{}^\wedge r}_i^{\Delta^-}$).

We now describe our program transformation in detail. Given a reactor C with persistent relations r_i and ephemeral relations t_i, let us redefine $\mathtt{{}^\wedge r}_i^{\Delta^+}$, $\mathtt{{}^\wedge r}_i^{\Delta^-}$, $\mathtt{{}^\wedge t}_i^{\Delta^+}$ and $\mathtt{{}^\wedge t}_i^{\Delta^-}$ to be the addition and the deletion sets of the update bundle applied to reactor C. We can assume that $\mathtt{{}^\wedge r}_i^{\Delta^+} \cap \mathtt{{}^\wedge r}_i^{\Delta^-} = \emptyset$ and $\mathtt{{}^\wedge t}_i^{\Delta^+} \cap \mathtt{{}^\wedge t}_i^{\Delta^-} = \emptyset$ because this property is checked by the originating reactor before creating new update bundles. Let $\mathtt{-r}_i$ denote the pre-, $\mathtt{{}^\wedge r}_i$ the stimulus, r_i the response, and $\mathtt{r{}^\wedge}_i$ the future state of the reaction.

Rewrite rules. Fig. 11 shows how our rewriting technique transforms a program with negation in the head clauses to a program without them. Let us redefine $\underline{\mathtt{r}}_i^{\Delta^+}$, $\underline{\mathtt{r}}_i^{\Delta^-}$, $\underline{\mathtt{t}}_i^{\Delta^+}$ and $\underline{\mathtt{t}}_i^{\Delta^-}$ the sets of additions/deletions to the response state of the persistent and ephemeral relations, correspondingly.

Rewrite:	r_i `<-` *body*	as:	$\underline{\mathtt{r}}_i^{\Delta^+}$ `<-` *body*	(I)
Rewrite:	`not` r_i `<-` *body*	as:	$\underline{\mathtt{r}}_i^{\Delta^-}$ `<-` $\underline{\mathtt{r}}_i$, *body*	(II)
Rewrite:	*head* `<-` r_i, *body*	as:	*head* `<-` $\underline{\mathtt{r}}_i$, `not` $\underline{\mathtt{r}}_i^{\Delta^-}$, *body*	(III)
Rewrite:	*head* `<-` `not` r_i, *body*	as:	*head* `<-` $\underline{\mathtt{r}}_i^{\Delta^-}$, *body* *head* `<-` `not` $\underline{\mathtt{r}}_i$, *body*	(IV)
Rewrite:	$\mathtt{r{}^\wedge}_i$ `<-` *body*	as:	$\underline{\mathtt{r{}^\wedge}}_i^{\Delta^+}$ `<-` *body*	(VII)
Rewrite:	$\mathtt{notr{}^\wedge}_i$ `<-` *body*	as:	$\underline{\mathtt{r{}^\wedge}}_i^{\Delta^-}$ `<-` *body*	(VIII)
Add:	r_i `<-` $\mathtt{{}^\wedge r}_i$			(V)
Add:	r_i `<-` $\underline{\mathtt{r}}_i^{\Delta^+}$			(VI)

Fig. 11. Rewrite rules defining the semantics of reactor rule evaluation in terms of normal datalog

Rewrite rule (I) computes the set of tuples to be added to r_i as the set of tuples that the body clauses resolve to. Rule (II) computes the deletion set very similarly; the only difference is adding a body clause which makes sure that a tuple gets deleted from a relation only if it was already there. The extra clause ensures that this rewriting rule does not introduce domain dependence—see further in this section for details. Rule (VI) adds the new addition sets to the response state as soon as they are computed; this ensures that the most current tuple additions are visible and propagating to the rest of the reactor rules. We would like to similarly account for the deletion sets but to reflect that in the response state we have to express it as negation in the rule head—exactly what the program transformation technique is trying to eliminate. Therefore the deletion sets must be accounted for and propagated via the reactor rules. As a result, rule (III) restricts the matching for the tuples in r_i to the ones that are not in the deletion set; conversely, rule (IV) allows matching on tuples in the deletion set. The rest of the rules are trivial. For t_i rules (I) to (VI) apply unchanged; rules (VII) and (VIII) do not apply because ephemeral relations do not have a future state.

State updates. At the beginning of a reaction the following assignments take place:

$$\mathtt{-r}_i ::= \mathtt{r}_i^P, \quad \mathtt{\char`\^r}_i ::= \mathtt{\char`\^r}_i^{\varDelta^+} \cup \mathtt{-r}_i \setminus \mathtt{\char`\^r}_i^{\varDelta^-}, \quad \mathtt{\char`\^t}_i ::= \mathtt{\char`\^t}_i^{\varDelta^+}$$

where '::=' should be read as relation overwriting. Intuitively the first assignment overwrites the pre-state of the current reaction's persistent relations with the contents of the relations prior to the reaction. The next assignments then apply the corresponding update bundles to the pre-state to obtain the stimulus state. Note that the deletion set of the update bundle $\mathtt{\char`\^t}_i^{\varDelta^-}$ for ephemeral relations has no effect.

All assignments are done outside datalog and have the effect of keeping a snapshot copy of the pre-state and the stimulus state in case the rules need to read them or the reaction rolls back. After the assignments take effect we can apply standard datalog techniques to evaluate the program rules up to a fixpoint. If at any point during the evaluation either $\underline{\mathtt{r}}_i^{\varDelta^+} \cap \underline{\mathtt{r}}_i^{\varDelta^-} \neq \emptyset$, $\underline{\mathtt{r\char`\^}}_i^{\varDelta^+} \cap \underline{\mathtt{r\char`\^}}_i^{\varDelta^-} \neq \emptyset$, $\underline{\mathtt{t}}_i^{\varDelta^+} \cap \underline{\mathtt{t}}_i^{\varDelta^-} \neq \emptyset$, or $\underline{\mathtt{t\char`\^}}_i^{\varDelta^+} \cap \underline{\mathtt{t\char`\^}}_i^{\varDelta^-} \neq \emptyset$ the evaluation stops and the reaction rolls back. If we reach the fixpoint (without either of the checks failing) we update $\mathtt{r}_i^P$ to take into account the deletion sets: $\mathtt{r}_i^P ::= r_i \setminus \underline{\mathtt{r}}_i^{\varDelta^-}$. Before quiescing, the reaction forms the update bundles ($\underline{\mathtt{r\char`\^}}_i^{\varDelta^+}$,$\underline{\mathtt{r\char`\^}}_i^{\varDelta^-}$) and ($\underline{\mathtt{t\char`\^}}_i^{\varDelta^+}$,$\underline{\mathtt{t\char`\^}}_i^{\varDelta^-}$) for other reactors and for itself, if applicable.

7.2 Semantics of Normal Datalog Programs: Stratification and Safety

As a result of applying our rewrite technique, we are left with a normal datalog program containing negation in body clauses only. Since it is possible to use negation to encode logical paradoxes, we wish to apply a syntactic constraint to rules that will ensure that a unique solution to collection of rules exists, without unduly affecting expressiveness. We adopt the *stratification* semantics for normal programs with negation [15]. The main idea behind stratification is to partition the program along negation such that for any relation we fully compute its content before applying the negation operator on it. For example, a program consisting of the rules `q(x) <- p(x,y), not q(y)` and `p(1,2) <-` is not stratified because it contains recursion through negation.

Another desirable property of a datalog program is *domain independence*: a solution should depend only on the known facts and not on the universal set of all facts. We would also like to ensure *finiteness*, or at least *weak finiteness*: that the evaluation of a rule from a finite set of facts yields a finite set of results, but infinite results caused by infinite recursion cannot be ruled out. To ensure domain independence and weak finiteness, we adopt the syntactic *safety* condition of Topor [14], which supports arithmetic expressions and negation. Briefly, a rule is safe if all of its variables are *limited*. A variable is limited if it occurs in a non-negated clause in the body, if is occurs in a negated clause in the body and is not used elsewhere, or if it occurs in an expression where a unique value of the variable can be computed given all the limited variables of the expression. A program is safe if all of its rules are safe.

7.3 Remote Reactor References

In a naive approach, every time a set of synchronously executing reactors C_i tries to extrude its scope to a new reactor C which is executing, C_i (1) waits for C to quiesce and accept the scope extrusion request, then (2) rolls back and restarts execution within the new scope expanded to include C.

A less naive approach will first statically compute the transitive closure of all the synchronously executing reactors based on the type information. The goal is to coordinate the order of rule evaluation with the order of locking—see Section 6. The algorithm will therefore compute the global stratification for the statically computed reactor closure; any time there is a choice of rules to evaluate next, the algorithm will choose the one that is consistent with the global stratification. This will make sure that positive information in relations is fully computed before evaluating its negation.

The program obtained by putting together the rules of all reactors C_k in a reaction will contain remote references to relations in C_k. To make all remote references local we define the following transformation. For every relation `r` in a reactor, the reactor defines a shadow copy `r~` and an implicit rule of the form `r~(self,x) <- r(x)`. Every remote reference `c.r(x)` to relation `r` can then be transformed into a local access to `r~(c,x)`. At this point the statically computed set of rules only contains local references and it is treated as a single program to which we can apply the transformation in Fig. 11.

7.4 Reactor Instantiation Details

The semantics of reactor instantiation must be defined with some care to ensure that the number of reactors instantiated in a reaction is unambiguous. Consider a rule whose head has the form `r`$(x_1, ..., x_i,$ `new M`$, x_{i+1}, ..., x_n)$, where `M` is a reactor type name and $x_k, k \in [1...n]$ are variables bound in the rule body. Then a tuple $\langle v_1, ..., v_i, \alpha, v_{i+1}, ..., v_n \rangle$ is a satisfying solution for the rule if and only if (1) α is a reference to a reactor of type `M`, (2) α is globally unique, (3) α did not exist prior to the current reaction, (4) $\langle u_1, ..., u_j, \alpha, u_{j+1}, ..., u_m \rangle$ does not satisfy any rule with a head of the form `s`$(y_1, ..., y_j,$ `new M`$, y_{j+1}, ..., y_m)$, and (5) $\langle u_1, ..., u_i, \alpha, u_{i+1}, ..., u_n \rangle$ does not satisfy any rule with a head of the form `r`$(x_1, ..., x_i,$ `new M`$, x_{i+1}, ..., x_n)$, unless $u_k = v_k$ for all $k \in [1...n]$. The basic idea behind this definition is that in each reaction, for each instance of `new` in a rule head, a new, globally unique reactor reference is generated *for each satisfying combination of values bound to variables in the rule*. The generalization of this semantics to rules containing multiple instances of `new` is straightforward.

Consider the rules `[1] r(new Foo, i) <- t(i)`, `[2] s(y) <- r(y, _)`, and `[3] s(new Foo) <- .` If relation `t` initially contains three tuples, then following the instantiation semantics above, rule `[1]` causes three new reactors to be instantiated. Similarly, rule `[3]` will generate one new reactor, distinct from those generated by rule `[1]`. If relation `s` is initially empty, it will contain four distinct reactor references at the end of the reaction: three from the reactors instantiated by rule `[1]`, and one from the reactor instantiated by rule `[3]`. If we were to duplicate any of rules `[1]-[3]`, the result of the reaction would remain the same, since the instantiation behavior of rules is dependent on their semantics, not their syntactic form.

Note that although the instantiation semantics distinguishes between `new` expressions and variables in rule heads, it does not distinguish reference-bound *variables* from other variables. Consider, e.g., the rules `[4] t(new Bar, i) <- r(i)` and `[5] s(new Bar, b) <- t(b, j)`. Rule `[5]` requires that there exist a unique, newly generated reference for each `b` satisfying `t(b, j)`. If `r` initially contains two tuples, then rules `[4]` and `[5]` each instantiate two new reactors.

It will be convenient to assume that reactor references generated by the same reactor are totally ordered, which allows them to be used as a source of ordered—but otherwise uninterpreted—keys. The `Nonce` reactors used in Fig. 7 serve this purpose.

8 Extended Example: Three-Tier Web Application

Fig. 12 depicts an extended example which follows the structure of a conventional three-tier web application for catalog ordering (note that this example makes extensive use of the notational abbreviations of Fig. 5). Unlike the example in Fig. 9, which combined "client" and "server" functionality in a single component, the example in Fig. 12 explicitly models a database (`DB`), web server (`WebServer`), and browser (`Browser`) as distinct components. The browser and web server communicate entirely asynchronously, while the web server and database communicate synchronously (i.e., transactionally). "Page content" in the browser is modeled by the `CompositeWidget` reactor type, which is instantiated with various primitive widget reactors (`OutputWidget` and `ButtonWidget` from Fig. 9 and a new `FormWidget`), depending on the type of page being displayed. Instances of `CompositeWidget` perform local (i.e., "browser-side") computation which performs basic form validation. Such functionality could easily be replaced with more elaborate browser-based widgets, e.g., with AJAX-style asynchronous behavior. One limitation of our current model is that all reactor references must be strongly typed, which makes it difficult to model a web browser reactor that can render arbitrary pages, also represented as reactors. In the future, we will consider more flexible type systems as well as weaker stratification requirements, which would allow a completely generic browser to be defined.

9 Related and Future Work

Related Work. Fundamentally, reactors are "reactive systems" [8], combining and extending features from several, largely unrelated areas of research: synchronous languages, datalog [15], and the actor model [1].

Esterel [2], Lustre [4], Signal [7], and Argos [11] are prominent synchronous languages. In synchronous languages, the term *causality* refers to dependencies, and all have restrictions on cyclic dependencies. Esterel only admits a program if all signals can be inferred to be either present or absent (as opposed to unknown); this is referred to as *constructiveness*. Esterel adopts a strict interleaving semantics, i.e. it assumes that reactions cannot overlap temporally. In Esterel signals are broadcast instantaneously so that all receptors of the signal will see it in the same instant and the signal will only exist in that reaction. The reactor model, on the other hand, supports both synchronous and asynchronous broadcasts (readers can react when a relation is changed) as well as

```
def DB = {
  (* trivial database: single value
     containing item inventory *)
  public inv: (int).
}

def WebServer = {
  (* reference to the database *)
  rDB: (ref DB)
  (* server accepts two request types
     from browsers: session initiation
     and form submission *)
  public ephemeral newSession:
    (ref Browser).
  public ephemeral formSubmit:
    (ref Browser, int).
  (* temp to hold new page *)
  ephemeral newPage:
    (ref CompositeWidget).

  (* generate page on every reaction *)
  newPage(new CompositeWidget) <- .
  (* each has link back to server *)
  c.rServer(self) <- newPage(c).

  (* newSession creates three primitive
     widgets for the new page *)
  newSession(_), newPage(c): {
    c.outWidget(new OutputWidget),
    c.formWidget(new FormWidget),
    c.buttonWidget(new ButtonWidget)
      <- .
    (* init primitive widget data,
       in particular, copy current
       inventory from db *)
    o.label("Available: "),
    o.val(toString(q)) <-
      c.outWidget(o), rDB(d), d.inv(q).
    f.label("Quantity to order: "),
    f.val("1") <- c.formWidget(f).
    b.label("Submit") <-
      c.buttonWidget(b).
  }

  (* formSubmit checks whether
     requested qty. is avail.; returns
     appropriate responses *)
  formSubmit(br, qr), newPage(c): {
    c.outWidget(newOutputWidget) <- .
    ephemeral reqOK() <-
      qr >= q, rDB(d), d.-inv(q).
    o.label("Success!") <-
      reqOK(), c.outWidget(o).
```

```
    d.inv(q’) := reqOK(), rDB(d),
      d.-inv(q), q’ = q - qr.
    o.label("Sorry!") <-
      not reqOK(), c.outWidget(o).
  }

  (* submit new page to browser *)
  br.showPage^(c) <- .
}

def Browser = {
  (* request to display new page *)
  public ephemeral showPage:
    (ref CompositeWidget).
  (* current page visible in browser *)
  thePage: (ref CompositeWidget).

  (* request to show new page updates
     current page; link to browser *)
  thePage(c) := showPage(c).
  c.rBrowser(self) <- showPage(c).
}

def CompositeWidget = {
  (* refs to server and browser *)
  rServer: (ref WebServer).
  rBrowser: (ref Browser).
  (* widgets on page; the form and
     button widgets are empty (not
     used) on the response page *)
  outWidget: (ref OutputWidget).
  formWidget: (ref FormWidget).
  buttonWidget: (ref ButtonWidget).

  (* perform local validation: ensure
     qty. requested less than inv. *)
  ephemeral validateOK: ()
  validateOK() <-
    buttonWidget(b), b.pressed(),
    formWidget(f), f.val(qty),
    outWidget(o), o.val(inv),
    toInt(qty) <= toInt(inv).
  (* validation OK: submit to server *)
  rServer.formSubmit^(br, toInt(qty))
    <- validateOK(), rBrowser(br),
       formWidget(f), f.val(qty).
  (* validation fails: just reset qty.
     to 1; do not submit *)
  f.val("1") <- not validateOK().
}

def FormWidget = {
  public label: (string).
  public val: (string).
}
```

Fig. 12. Mini three-tier web application

synchronous and asynchronous point-to-point communication (by writing directly into a public relation of the receiver).

Lustre and Signal also limit cyclic dependencies, but add *sampling* in the form of the construct `x = Exp when BExp` meaning that `Exp` should be evaluated only when `BExp` is true. This facility provides a sophisticated way of reading values from preceding reactions other than the immediately previous one. In the reactor model, such predicates can be expressed directly as `x(Exp) <- BExp` where `x` should be a singleton relation. Argos is based on State Charts and hierarchical automata and distinguishes itself from other synchronous languages by being graphical.

Generally speaking, the group of synchronous languages does not allow cycles in the data flow graph – only pre-state to response-state connections are permitted when referring to the same variable. In the reactor model, stratification provides a more refined classification that widely allows recursion while ruling out cases where the fixed point could be ambiguous (of course, programs may still loop infinitely). Reactors provide several features not found in synchronous languages, namely asynchrony, generativity, and distributed transactions. We are not familiar with any other language that combines these features.

Active databases [13] commonly express triggers of the form *Event–Condition–Action* (ECA), where the action is carried out if on receipt of a matching event the condition holds true. This can be expressed as `action <- event, condition` in the reactor model. The reactor model eliminates the distinction between conditions and events, and adds support for distribution, process generation, and synchronous composition.

Transaction Datalog [3] introduces transactions and database updates to datalog. In Transaction Datalog, inserts and deletes are special atoms in rule bodies, and backward derivation rather than forward derivation is used. To achieve concurrency in transactions a concurrent conjunction operator, `|`, is added. In the reactor model, all rules execute concurrently within the same reaction (subject to stratification) by default, and thus *sequentiality*, rather than concurrency, must be programmed explicitly when needed.

Future Work. While this paper has not focused on implementation, there are two broad areas that are amenable to optimization: query incrementalization, and efficient implementation of synchronous composite reactions through low-overhead concurrency control. The former has already been studied in the datalog community (e.g., [5]), and we intend to adapt those results appropriately to our setting. In the case of synchronous reactions, recent results on efficient implementation of software transactions (e.g., [9]) are likely to be relevant.

Other issues we plan to investigate include: (1) contract/interface type systems; (2) various abstraction facilities, such as reactor and rule parametricity and high-order rules, that read, write, and deploy other rules; (3) more sophisticated access control mechanisms; (4) function symbols (functors); (5) reactor garbage collection; (6) a truly distributed implementation; (7) support for long-running (rather than atomic) transactions.

Acknowledgments. The authors gratefully acknowledge the contributions of Rafah Hosn, Bruce Lucas, James Rumbaugh, Mark Wegman, and Charles Wiecha to the development of the ideas embodied in this work.

References

1. Agha, G.A., Mason, I.A., Smith, S.F., Talcott, C.L.: A foundation for actor computation. Journal of Functional Programming 7(1), 1–69 (January 1997)
2. Berry, G., Gonthier, G.: The ESTEREL synchronous programming language: design, semantics, implementation. Science of Computer Programming 19(2), 87–152 (1992)
3. Bonner, A.J.: Workflow, transactions and datalog. In: PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, New York, NY, USA, pp. 294–305. ACM Press, New York (1999)
4. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: a declarative language for real-time programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, pp. 178–188. ACM Press, New York (1987)
5. Dong, G., Topor, R.W.: Incremental evaluation of datalog queries. In: Hull, R., Biskup, J. (eds.) ICDT 1992. LNCS, vol. 646, pp. 282–296. Springer, Heidelberg (1992)
6. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
7. Gautier, T., Guernic, P.L.: SIGNAL: A declarative language for synchronous programming of real-time systems. In: Proceedings of the Functional Programming Languages and Computer Architecture, London, UK, pp. 257–277. Springer, Heidelberg (1987)
8. Harel, D., Pnueli, A.: On the development of reactive systems. In: Apt, K.R. (ed.) Logics and models of concurrent systems. NATO ASI Series F: Computer And Systems Sciences, vol. 13, pp. 477–498. Springer, Heidelberg (1989)
9. Harris, T., Marlow, S., Jones, S.P., Herlihy, M.: Composable memory transactions. In: ACM Conf. on Principles and Practice of Parallel Programming, Chicago, pp. 48–60 (June 2005)
10. Kiczales, G.: Aspect-oriented programming. ACM Computing Surveys 28, 4es, 154 (1996)
11. Maraninchi, F., Rémond, Y.: Argos: an automaton-based synchronous language. Computer Languages 27, 61–92 (2001)
12. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts I-II. Information and Computation 100(1), 1–77 (1992)
13. Paton, N.W., Díaz, O.: Active database systems. ACM Comput. Surv. 31(1), 63–103 (1999)
14. Topor, R.: Safe database queries with arithmetic relations. In: Proceedings of the 14th Australian Computer Science Conference (1991)
15. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. 1. Computer Science Press, ch. 3 (1988)
16. Wang, X.: Negation in Logic and Deductive Databases. PhD thesis, University of Leeds (1999)

A Theory for Strong Service Compliance*

Mario Bravetti and Gianluigi Zavattaro

Department of Computer Science, University of Bologna, Italy
{bravetti,zavattar}@cs.unibo.it

Abstract. We investigate, in a process algebraic setting, a new notion of compliance that we call *strong service compliance*: composed services are strong compliant if their composition is both deadlock and livelock free (this is the traditional notion of compliance) and whenever a message can be sent to invoke a service, this service is ensured to be ready to serve the invocation. We define also a new notion of refinement, called *strong subcontract pre-order*, suitable for strong compliance: given a composition of strong compliant services each one executing according to some specific contracts, we can replace the services with other services executing corresponding strong subcontracts preserving strong compliance. Finally, we present a characterization of the strong subcontract pre-order resorting to the theory of (should) testing pre-order.

1 Introduction

One of the main novelties emerged during the last years of research in the field of distributed computing is Service Oriented Computing (SOC). It is a novel paradigm based on services intended as autonomous and heterogeneous components that can be published and discovered via standard interface languages and publish/discovery protocols. One of the peculiarities of Service Oriented Computing, distinguishing it from other distributed computing paradigms (such as component based software engineering), is that it is centered around the so-called *message oriented architecture*. This means that, given a set of collaborating services, the current state of their interaction is stored inside the exchanged messages and not only within the services. From a practical viewpoint, this means that it is necessary to include, in the exchanged messages, the so-called correlation information that permits to a service to associate a received message to the correct session of interaction (in fact, the same service could be contemporarily involved in different sessions at the same time).

Web Services is the most prominent service oriented technology: Web Services publish their interface expressed in WSDL, they are discovered through the UDDI protocol, and they are invoked using SOAP.

Two main approaches for the composition of services are currently under investigation and development inside the SOC research community: service *orchestration* and service *choreography*. According to the first approach, the activities

* Research partially funded by EU Integrated Project Sensoria, contract n. 016004.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 96–112, 2007.

of the composed services are coordinated by a specific component, called the orchestrator, that is responsible for invoking the composed services and collect their responses. Several languages have been already proposed for programming orchestrators such as XLANG [Tha01], WSFL [Ley01] and WS-BPEL [OAS].

Choreography languages are attracting a lot of attention within W3C, where the most credited choreography language WS-CDL [W3C] is currently under development. Choreographies represent a "more democratic" alternative approach for service composition with respect to orchestrations. Indeed, orchestrations require the implementation of central points of coordination; on the contrary, choreography languages support a high level description of peer-to-peer interactions among services that directly communicate without the mediation of any orchestrator. More precisely, the aim of choreography languages is to support the high level description of systems that should be actually implemented as combination of autonomous, loosely coupled and heterogenous services.

In this paper we continue a formal investigation of service composition initiated in [BZ07]. In particular, in [BZ07] we have presented a process algebraic modeling of the notion of contract intended as a "behavioural interface" of services describing the possible flows of invocations that are received and/or emitted by the service. Based on this contract language, we have formalized the notion of correct composition of services, and we have defined a notion of subcontract preorder characterizing the possibility to replace services with subservices without breaking the correctness of the composition. The notion of correctness considered in [BZ07] requires that the composed services are both deadlock and livelock free.

In this paper we consider a stronger notion, called *strong correctness*, that requires also the following: if a service invocation is ready to be executed, the corresponding invoked service should be ready to serve the request. This intuitive correctness assumption was not taken under consideration is previous work on service compliance such as our previous paper [BZ07] and the paper by Carpineti et al. [CCL$^+$06]. For instance, in these papers, the service $S_1 = \overline{a}_{S_2}|\overline{b}_{S_2}$, invoking in parallel the two operations a and b provided by the service S_2, is considered compliant with a service $S_2 = a; b$ that serves first a and then b. In practice, it could happen that the request on the operation b could reach S_2 before the request on operation a. In this case, usual service invocation protocols (such as, e.g., SOAP) returns an exception indicating the unavailability of the requested operation. *Strong compliance* considers also this kind of exceptions as errors that breaks the correctness of the composition. For instance, according to strong compliance, S_1 is not compliant with the above S_2 but it is compliant with another service $S_2 = a|b$ ready to serve both the requests on a and b.

In order to formalize strong compliance, we need to slightly modify the calculus introduced in [BZ07]. The first difference is that we add locations to services and to output actions. This because the kind of exceptions described above is meaningful assuming that messages are directed to a specific target service, and the exception is raised when the target service is not ready to serve the request contained in the message. The second difference is that the calculus that we consider in the present paper has a standard output prefix instead of the output $\tau;\overline{a}$ always prefixed by an internal τ action considered in [BZ07]. This assumption, permitted us to prove in [BZ07] several interesting results. The most important one is that a suitable subcontract pre-order can be defined abstracting away from the input alphabet of the services in the context. Even if we remove this assumption considering a standard output prefix, the new notion of strong compliance allows us to achieve an even stronger result: a suitable strong subcontract pre-order can be defined abstracting away from both the input and the output alphabet of the other services in the context. This is a rather important property that permits to define a general notion of strong subcontract which is independent of the context in which the corresponding service substitutions are done.

We foresee at least two main applications for our notion of strong subcontract. On the one hand, it can be exploited in the *service discovery* phase. Consider, for instance, a service system defined in terms of the contracts that should be exposed by each of the service components. The actual services to be combined could be retrieved independently one from the other (e.g. querying contemporarily different service registries) collecting those services that either exposes the expected contract, or one of its strong subcontract. On the other hand, the notion of strong subcontract could be useful in *service updates* in order to ensure backward compatibility. Consider, e.g., a service that should be updated in order to provide new functionalities; if the new version exposes a strong subcontract of the previous service, our theory ensures that the new service is a correct substitute for the previous one.

The last technical contribution of the paper is a characterization of our notion of subcontract achieved resorting to the theory of testing [DH84], in particular to the should testing pre-order investigated in [RV05]. This characterization permits to have an effective procedure to prove whether a contract is a strong subcontract of another one. In fact, the definition of strong subcontract is not directly applicable because it contains a universal quantification on all possible contexts.

Structure of the Paper. Section 2 reports the syntax and the operational semantics of the calculs that we use in our theory for strong service compliance. Section 3 reports the formalization of strong service compliance while Section 4 reports about our investigation of the new notion of strong subcontract pre-order. Finally, Section 5 contains conclusive remarks and a comparison with the related literature.

2 Contracts and Service Compositions

In this section we introduce the syntax and the operational semantics of the calculus that we use in the following section to investigate formally the notion of strong compliance.

2.1 Syntax

We assume a denumerable set of action names $\mathcal{N}$, ranged over by $a, b, c, \ldots$. The set $\mathcal{N}_{con} = \{a_* \mid a \in \mathcal{N}\}$ is the set of contract action names. Moreover, we consider a denumerable set Loc of location names, ranged over by $l, l', l_1, \cdots$. The set $\mathcal{N}_{loc} = \{a_l \mid a \in \mathcal{N}, l \in Loc\}$ is the set of located action names. The set $\mathcal{A}_{con} = \mathcal{N}_{con} \cup \{\overline{a}_* \mid a_* \in \mathcal{N}_{con}\}$ is the set of input and output contract actions. The set $\mathcal{A}_{loc} = \mathcal{N}_{loc} \cup \{\overline{a}_l \mid a_l \in \mathcal{N}_{loc}\}$ is the set of input and output located actions. We use $\tau \notin \mathcal{N}$ to denote an internal (unsynchronizable) computation. Given a set of located action names $I \subset \mathcal{N}_{loc}$, we denote: with $\overline{I} = \{\overline{a}_l \mid a_l \in I\}$ the set of output actions performable on those names and with $I_l = \{a \mid a_l \in I\}$ the set of action names with associated location l.

Definition 1. (Contracts and Systems) *The syntax of contracts is defined by the following grammar*

$$\begin{array}{rcl} C & ::= & \mathbf{0} \quad | \quad \mathbf{1} \quad | \quad \tau \quad | \quad a_* \quad | \quad \overline{a}_* \quad | \quad a \quad | \quad \overline{a}_l \quad | \\ & & C;C \quad | \quad C{+}C \quad | \quad C|C \quad | \quad C\backslash M \quad | \quad C^* \end{array}$$

where $M \subseteq \mathcal{N}_{con}$. The set of all contracts C is denoted by $\mathcal{P}_{con}$. In the following we will omit trailing "$\mathbf{1}$" when writing contracts.
The syntax of systems (contract compositions) is defined by the following grammar

$$P \quad ::= \quad [C]_l \quad | \quad P \| P \quad | \quad P \backslash\!\backslash L$$

where $L \subseteq \mathcal{A}_{loc}$. A system P is well-formed if: (i) every contract subterm $[C]_l$ occurs in P at a different location l and (ii) no output action with destination l is syntactically included inside a contract subterm occurring in P at the same location l, i.e. actions $\overline{a}_l$ cannot occur inside a subterm $[C]_l$ of P. The set of all well-formed systems P is denoted by $\mathcal{P}$. In the following we will just consider well-formed systems and, for simplicity, we will call them just systems.

The contracts are specified using a typical process algebra. We use **0** and **1** to denote the two possible final states of a service execution: unsuccessful or successful, respectively. The possible basic actions are the typical internal τ action and the input/output actions. We distinguish between input/output actions executed internally (synchronization between two threads of the same service) and input/output actions involving different services. The first kind of interaction occurs on the contract action names decorated with the subscript star: input actions are denoted with a_*, while output actions with $\overline{a}_*$. The second kind of interaction occurs on the standard actions names: input actions are denoted with a, while output actions with $\overline{a}_l$, where l is the name of the target location of the output action. The composition operators are the standard sequence $\_;\_$, choice $\_+\_$, parallel $\_|\_$, restriction (only on local contract action names) $\_\backslash\_$, and repetition $\_^*$.

The syntax of compositions permits to represent a service located at location l, and executing according to the contract C, simply as $[C]_l$. Services are composed using parallel composition $\_\|\_$ and restriction $\_\backslash\!\backslash\_$. The restriction operator for compositions distinguishes between input and output actions, e.g., we write $[C]_l\backslash\backslash\{a_l, \overline{b}_{l'}\}$ to state that the service $[C]_l$ cannot perform inputs on a (e.g., because a is an output port of the service running at l) and cannot perform outputs on the port b of the service running at location l' (e.g., because b is an output port of that service).

2.2 Operational Semantics

The operational semantics is defined in terms of a labeled transition system defined in two steps; we first define the semantics of contracts and, based on the corresponding transition systems, we define the semantics for service compositions.

In the following, we take α to range over the set of syntactical actions $SAct = \mathcal{A}_{con} \cup \mathcal{N} \cup \{\overline{a}_l \mid a_l \in \mathcal{N}_{loc}\} \cup \{\tau\}$.

The operational semantics of contracts is defined by the rules in Table 1 (plus symmetric rules) while the operational semantics of systems is defined by the rules in Table 2 (plus symmetric rules). We take β to range over the set of actions executable by contracts and systems, $Act = \mathcal{A}_{con} \cup \mathcal{N} \cup \mathcal{A}_{loc} \cup \{\tau\}$. We take λ to range over the set of transition labels $\mathcal{L} = Act \cup \{\surd\}$, where $\surd$ denotes successful termination.

Table 1. Semantic rules for contracts (symmetric rules omitted)

	$\mathbf{1} \xrightarrow{\surd} \mathbf{0}$	$\alpha \xrightarrow{\alpha} \mathbf{1}$
$\dfrac{C \xrightarrow{\lambda} C'}{C{+}D \xrightarrow{\lambda} C'}$	$\dfrac{C \xrightarrow{\lambda} C' \quad \lambda \neq \surd}{C;D \xrightarrow{\lambda} C';D}$	$\dfrac{C \xrightarrow{\surd} C' \quad D \xrightarrow{\lambda} D'}{C;D \xrightarrow{\lambda} D'}$
$\dfrac{C \xrightarrow{a_*} C' \quad D \xrightarrow{\overline{a}_*} D'}{C \mid D \xrightarrow{\tau} C' \mid D'}$	$\dfrac{C \xrightarrow{\surd} C' \quad D \xrightarrow{\surd} D'}{C \mid D \xrightarrow{\surd} C' \mid D'}$	$\dfrac{C \xrightarrow{\lambda} C' \quad \lambda \neq \surd}{C \mid D \xrightarrow{\lambda} C' \mid D}$
$\dfrac{C \xrightarrow{\lambda} C' \quad \lambda \notin M \cup \overline{M}}{C \backslash M \xrightarrow{\lambda} C' \backslash M}$	$C^* \xrightarrow{\surd} \mathbf{0}$	$\dfrac{C \xrightarrow{\lambda} C' \quad \lambda \neq \surd}{C^* \xrightarrow{\lambda} C';C^*}$

The operational semantics for contracts is standard for process algebra with sequential composition and repetition. In particular, the label $\surd$ is used to denote the successful completion of a contract. Given $C; D$, the contract D can be activated if C has an outgoing transition labeled with $\surd$; while given the repetition C^*, we have that it can either complete (it has an outgoing transition labeled with $\surd$) or execute one instance of the contract C before becoming C^* again. The unique specific comment is concerned with the rule for synchronization that is admitted only for input/output actions executed on local contract action names a_*. In fact, the synchronization on global names will be considered in the semantics of systems.

Table 2. Semantic rules for contract compositions (symmetric rules omitted)

$$\frac{C \xrightarrow{a} C'}{[C]_l \xrightarrow{a_l} [C']_l} \qquad \frac{C \xrightarrow{\overline{a}_{l'}} C'}{[C]_l \xrightarrow{\overline{a}_{l'}} [C']_l} \qquad \frac{P \xrightarrow{\lambda} P' \quad \lambda \neq \surd}{P\|Q \xrightarrow{\lambda} P'\|Q}$$

$$\frac{P \xrightarrow{a_l} P' \quad Q \xrightarrow{\overline{a}_l} Q'}{P\|Q \xrightarrow{\tau} P'\|Q'} \qquad \frac{P \xrightarrow{\surd} P' \quad Q \xrightarrow{\surd} Q'}{P\|Q \xrightarrow{\surd} P'\|Q'} \qquad \frac{P \xrightarrow{\lambda} P' \quad \lambda \notin L}{P\backslash\!\backslash L \xrightarrow{\lambda} P'\backslash\!\backslash L}$$

Also the operational semantics for systems is defined in a standard way: the unique nonstandard operator is restriction that, as discussed above, distinguishes between input and output actions executed on the same name. Observe that the synchronization rule considers only nonlocal standard names.

In the remainder of the paper we use the following notations: $P \xrightarrow{\lambda}$ to mean that there exists P' such that $P \xrightarrow{\lambda} P'$ and, given a sequence of labels $w = \lambda_1\lambda_2 \cdots \lambda_{n-1}\lambda_n$ (possibly empty, i.e., $w = \varepsilon$), we use $P \xrightarrow{w} P'$ to denote the sequence of transitions $P \xrightarrow{\lambda_1} P_1 \xrightarrow{\lambda_2} \cdots \xrightarrow{\lambda_{n-1}} P_{n-1} \xrightarrow{\lambda_n} P'$ (in case of $w = \varepsilon$ we have $P' = P$, i.e., $P \xrightarrow{\varepsilon} P$).

3 Strong Compliance

We now define the notion of strong correct composition of contracts. Intuitively, a composition of services is strongly correct if it is guaranteed that all services eventually reach successful completion (it is both deadlock and livelock free) and everytime a process may invoke an operation on a service, the target service should be ready to serve the request. This second assumption is new with respect to [BZ07] (where we only assumed guaranteed completion) and characterize the new notion of *strong* compliance.

As discussed in the introduction, the rational behind strong compliance is that standard protocols for service invocation usually raise exceptions in the case the target of a service invocation is not ready to serve it. In order to formalize situations in which exception cannot be raised, we define the auxiliary operator $\mathtt{nso}(P)$ that evaluates non-synchronizable outputs immediately executable by P, i.e. outputs that do not have a corresponding input, and the predicate $\mathtt{exceptionFree}(P)$ that indicates whether none of the above undesired exceptions can be raised in the system P.

Definition 2. (Exception freedomness) *We first define* $\mathtt{nso}(P)$ *inductively on the structure of* P*:*

$$\mathtt{nso}([C]_l) = \{\overline{a}_l \mid C \xrightarrow{\overline{a}_l} C'\}$$
$$\mathtt{nso}(P_1\|P_2) = (\mathtt{nso}(P_1) - \{\overline{a}_l \mid P_2 \xrightarrow{a_l} P_2'\}) \cup (\mathtt{nso}(P_2) - \{\overline{a}_l \mid P_1 \xrightarrow{a_l} P_1'\})$$
$$\mathtt{nso}(P\backslash\!\backslash L) = \begin{cases} \{\mathtt{exception}\} & \textit{if}\ \mathtt{nso}(P) \cap L \neq \emptyset \\ \mathtt{nso}(P) & \textit{otherwise} \end{cases}$$

where `exception` *is an auxiliary name denoting the existence of an output without a corresponding input. Finally, we define:*

$$\texttt{exceptionFree}(P) \textit{ if and only if } \texttt{nso}(P) = \emptyset$$

Definition 3. (Strongly correct composition) *A system P is a strongly correct composition, denoted $P \Downarrow$, if for every P' such that $P \xrightarrow{\tau}^* P'$ both the following hold:*

- `exceptionFree`(P') *and*
- *there exists P'' such that $P' \xrightarrow{\tau}^* P'' \xrightarrow{\surd}$.*

Informally, a system is strongly correct if no exceptions can be raised in any of the reachable states, and the successful termination of all composed services (denoted with the transition labeled with $\surd$) is eventually reached. In [BZ07] we have defined the notion of correct composition, denoted with $P \downarrow$, that corresponds to the above definition without the first item about the exception freedomness of all reachable states.

4 Contract Refinement

In this section we investigate a suitable notion of refinement for contracts compatible with strong correctness; intuitively, a contract C' is a *strong subcontract* of C if it is a "good" substitute of C, i.e. given a system P containing the service $[C]_l$, we can replace it with $[C']_l$ preserving the strong correctness of P.

4.1 Input-Output Strong Subcontract Relation

In general, see for instance [CCL$^+$06], the subcontract relation depends on the alphabet (i.e. the possible actions) of the services present in P. For instance, we can consider $a + (c; P)$ subcontract of a assuming that the action $\overline{c}_l$ is not in the alphabet of the other services (indeed, this implies that the new branch $c; P$ of the subcontract cannot interfere with the other services in the system).

We start defining a notion of subcontract parameterized on the input and output alphabets of the services in the potential contexts. Then we prove that, thanks to the new exception freedomness assumption, we can abstract away from these alphabets.

We first formally define the input and output alphabets of systems.

Definition 4. (Input and Output sets) *Given the contract $C \in \mathcal{P}_{con}$, we define $I(C)$ as the subset of $\mathcal{N}$ of the potential input actions of C:*

$$I(\mathbf{0}) = I(\mathbf{1}) = I(\tau) = I(a_*) = I(\overline{a}_*) = I(\overline{a}_l) = \emptyset \qquad I(a) = \{a\}$$
$$I(C;C') = I(C+C') = I(C|C') = I(C) \cup I(C') \qquad I(C\backslash M) = I(C^*) = I(C)$$

We define $O(C)$ as the subset of $\mathcal{N}_{loc}$ of the potential output actions of C:

$$O(\mathbf{0}) = O(\mathbf{1}) = O(\tau) = O(a) = O(a_*) = O(\overline{a}_*) = \emptyset \qquad O(\overline{a}_l) = \{\overline{a}_l\}$$
$$O(C;C') = O(C+C') = O(C|C') = O(C) \cup O(C') \qquad O(C\backslash M) = O(C^*) = O(C)$$

Note that the set M in $C \setminus M$ does not influence $I(C \setminus M)$ and $O(C \setminus M)$because it contains only contract names outside $\mathcal{N}$. Given the system P, we define $I(P)$ as the subset of $\mathcal{N}_{loc}$ of the potential input actions of P:

$$I([C]_l) = \{a_l \mid a \in I(C)\} \qquad I(P \| P') = I(P) \cup I(P') \qquad I(P \backslash\!\backslash L) = I(P) - L$$

We define $O(P)$ as the subset of $\mathcal{N}_{loc}$ of the potential output actions of P:

$$O([C]_l) = O(C) \qquad O(P \| P') = O(P) \cup O(P') \qquad O(P \backslash\!\backslash L) = O(P) - L$$

In the following we make the nonrestrictive assumption that the other services composed in parallel with the service that we want to substitute are of the form $([C_1]_{l_1} \| \cdots \| [C_n]_{l_n}) \backslash\!\backslash L$; this is not restrictive because it is always possible to use standard renaming techniques to avoid capture of names while extending the scope of restrictions. We denote with $\mathcal{P}_{conpres}$ the subset of systems of the form $([C_1]_{l_1} \| \cdots \| [C_n]_{l_n}) \backslash\!\backslash L$.

Note that, given $P = ([C_1]_{l_1} \| \ldots \| [C_n]_{l_n}) \backslash\!\backslash I \cup \overline{O} \in \mathcal{P}_{conpres}$, we have $I(P) = (\bigcup_{1 \le i \le n} I([C_i]_{l_i})) - I$ and $O(P) = (\bigcup_{1 \le i \le n} O([C_i]_{l_i})) - O$. In the following we let $\mathcal{P}_{conpres,I,O}$, with $I, O \subseteq \mathcal{N}_{loc}$, denote the subset of systems of $\mathcal{P}_{conpres}$ such that $I(P) \subseteq I$ and $O(P) \subseteq O$.

In the next definition we use the following notation: given a contract $C \in \mathcal{P}_{con}$, we use $oloc(C)$ to denote the subset of Loc of the locations target of the output actions occurring inside C.

Definition 5. (Input-Output strong subcontract relation) *A contract C' is a subcontract of a contract C with respect to a set of input located names $I \subseteq \mathcal{N}_{loc}$ and output located names $O \subseteq \mathcal{N}_{loc}$, denoted $C' \preceq\!\!\!\preccurlyeq_{I,O} C$, if and only if for all $l \in Loc$ such that $l \notin oloc(C) \cup oloc(C')$ and $P \in \mathcal{P}_{conpres,I,O}$ such that $l \notin loc(P)$ we have*

$$([C]_l \| P) \Downarrow \quad \Rightarrow \quad ([C']_l \| P) \Downarrow$$

The following Proposition states an intuitive contravariant property: given $\preceq\!\!\!\preccurlyeq_{I',O'}$, and the greater sets I and O (i.e. $I' \subseteq I$ and $O' \subseteq O$) we obtain a smaller relation $\preceq\!\!\!\preccurlyeq_{I,O}$ (i.e. $\preceq\!\!\!\preccurlyeq_{I,O} \subseteq \preceq\!\!\!\preccurlyeq_{I',O'}$). This follows from the fact that extending the sets of input and output actions means considering a greater set of discriminating contexts.

Proposition 1. *Let $C, C' \in \mathcal{P}_{con}$ be two contracts, $I, I' \subseteq \mathcal{N}_{loc}$ be two sets of input channel names such that $I' \subseteq I$ and $O, O' \subseteq \mathcal{N}_{loc}$ be two sets of output channel names such that $O' \subseteq O$. We have:*

$$C' \preceq\!\!\!\preccurlyeq_{I,O} C \quad \Rightarrow \quad C' \preceq\!\!\!\preccurlyeq_{I',O'} C$$

Proof. Let us suppose $C' \preceq\!\!\!\preccurlyeq_{I,O} C$. Consider now $l \in Loc$, $l \notin oloc(C) \cup oloc(C')$, and $P \in \mathcal{P}_{conpres,I',O'}$, $l \notin loc(P)$, such that $([C]_l \| P) \Downarrow$. As $I' \subseteq I$ and $O' \subseteq O$, then also $P \in \mathcal{P}_{conpres,I,O}$. Thus, as we suppose $C' \preceq\!\!\!\preccurlyeq_{I,O} C$, $([C']_l \| P) \Downarrow$. This implies that also $C' \preceq\!\!\!\preccurlyeq_{I',O'} C$.

The following Proposition states an intermediary result useful in subsequent proofs.

Proposition 2. *Let $C, C' \in \mathcal{P}_{con}$ be contracts and $I, O \subseteq \mathcal{N}_{loc}$ be sets of located names and let $C' \preccurlyeq_{I,O} C$. For every $l \in Loc$, $l \notin oloc(C) \cup oloc(C')$, and $P \in \mathcal{P}_{conpres,I,O}$, $l \notin loc(P)$, such that $([C]_l \| P)\Downarrow$, we have*

$$\big([C']_l \backslash\!\!\backslash (I([C']_l) - I([C]_l)) \| P\big)\Downarrow \qquad \textit{and} \qquad \big([C']_l \backslash\!\!\backslash \overline{(O(C') - O(C))} \| P\big)\Downarrow$$

Proof. *We discuss the result concerned with restriction of outputs (the proof for the restriction of inputs is symmetric). Let $C' \preccurlyeq_{I,O} C$. Given any $P \in \mathcal{P}_{conpres,I,O}$ such that $([C]_l \| P)\Downarrow$, we will show that $([C']_l \backslash\!\!\backslash \overline{(O(C') - O(C))} \parallel P)\Downarrow$. We first observe that $([C]_l \parallel P \backslash\!\!\backslash (O(C') - O(C)))\Downarrow$. Since $C' \preccurlyeq_{I,O} C$, we derive $([C']_l \parallel P \backslash\!\!\backslash (O(C') - O(C)))\Downarrow$.*
As a consequence $([C']_l \backslash\!\!\backslash \overline{(O(C') - O(C))} \parallel P \backslash\!\!\backslash (O(C') - O(C)))\Downarrow$. We can conclude $([C']_l \backslash\!\!\backslash \overline{(O(C') - O(C))} \parallel P)\Downarrow$.

The following proposition states an important property which is a direct consequence of the assumption of exception freedomness of correct compositions.

Proposition 3. *Let $C, C' \in \mathcal{P}_{con}$ be contracts and $I, O \subseteq \mathcal{N}_{loc}$ be sets of located names and let $C' \preccurlyeq_{I,O} C$. For every $l \in Loc$, $l \notin oloc(C) \cup oloc(C')$, and $P \in \mathcal{P}_{conpres,I,O}$, $l \notin loc(P)$, such that $([C]_l \| P)\Downarrow$,*

$$([C']_l \| P) \xrightarrow{\tau}{}^{*} ([C'_{der}]_l \| P_{der}) \quad \Rightarrow \quad \begin{cases} \forall\, a_{l'} \in O(C') - O(C).\; C'_{der} \overset{\overline{a}_{l'}}{\not\longrightarrow} \\ \forall\, a \in I(C') - I(C).\; P_{der} \overset{\overline{a}_{l}}{\not\longrightarrow} \end{cases}$$

Proof. *We proceed by contradiction for both statements.*

Concerning the first statement. Suppose that there exist C'_{der}, P_{der} such that $([C']_l \| P) \xrightarrow{\tau}{}^{*} ([C'_{der}]_l \| P_{der})$ and $C'_{der} \xrightarrow{\overline{a}_{l'}}$ for some $a_{l'} \in O(C') - O(C)$. We further suppose (without loss of generality) that such a path is minimal, i.e. no intermediate state $(C'_{der2} \| P_{der2})$ is traversed, such that $C'_{der2} \xrightarrow{\overline{a}_{l'}}$ for some $a_{l'} \in O(C') - O(C)$. This implies that the same path must be performable by $([C']_l \backslash\!\!\backslash \overline{(O(C') - O(C))} \parallel P)$, thus reaching the state $([C'_{der}]_l \backslash\!\!\backslash \overline{(O(C') - O(C))} \parallel P_{der})$. However, since in the state C'_{der} of contract C' we have $C'_{der} \xrightarrow{\overline{a}_{l'}}$ for some $a_{l'} \in O(C') - O(C)$ and the execution of $\overline{a}_{l'}$ is disallowed by restriction, we will have $\mathtt{out}([C'_{der}]_l \backslash\!\!\backslash \overline{(O(C') - O(C))}) = \{\mathtt{exception}\}$, thus $([C']_l \backslash\!\!\backslash \overline{(O(C') - O(C))} \parallel P)\not\Downarrow$ contradicting Proposition 2.

Concerning the second statement. Suppose that there exist C'_{der}, P_{der} such that $([C']_l \| P) \xrightarrow{\tau}{}^{*} ([C'_{der}]_l \| P_{der})$ and $P_{der} \xrightarrow{\overline{a}_{l}}$ for some $a \in I(C') - I(C)$. We further suppose (without loss of generality) that such a path is minimal, i.e. no intermediate state $(C'_{der2} \| P_{der2})$ is traversed, such that $P_{der2} \xrightarrow{\overline{a}_{l}}$ for some $a \in I(C') - I(C)$. This implies that the same path must be performable by $([C']_l \parallel P \backslash\!\!\backslash (I([C']_l) - I([C]_l)))$, thus reaching the state $([C'_{der}]_l \parallel P_{der} \backslash\!\!\backslash (I([C']_l) - I([C]_l)))$. However, since in the state P_{der} of system P

we have $P_{der} \xrightarrow{\overline{a}_l}$ for some $a \in I(C') - I(C)$ and the execution of $\overline{a}_l$ is disallowed by restriction, we will have $\texttt{out}(P_{der} \setminus\!\!\setminus \overline{(I([C']_l) - I([C]_l))}) = \{\texttt{exception}\}$*, thus* $([C']_l \parallel P \setminus\!\!\setminus \overline{(I([C']_l) - I([C]_l))}) \not\Downarrow$*. This implies* $([C']_l \setminus\!\!\setminus (I([C']_l) - I([C]_l)) \| P) \not\Downarrow$ *contradicting Proposition 2.*

We are finally ready to prove the main results of this subsection. We separate these results in two independent Propositions; the first one states that the set of potential inputs of the other contracts in the system is an information that does not influence the strong subcontract relation, the second one states the same about outputs.

Proposition 4. *Let $C \in \mathcal{P}_{con}$ be a contract, $O \subseteq \mathcal{N}_{loc}$ be a set of located output names and $I, I' \subseteq \mathcal{N}_{loc}$ be two sets of located input names such that $O(C) \subseteq I, I'$. We have that for every contract $C' \in \mathcal{P}_{con}$,*

$$C' \preccurlyeq_{I,O} C \quad \Longleftrightarrow \quad C' \preccurlyeq_{I',O} C$$

Proof. *Let us suppose $C' \preccurlyeq_{I',O} C$ (the other direction is symmetric). Given any $l \in Loc$, $l \notin oloc(C) \cup oloc(C')$, and $P \in \mathcal{P}_{conpres,I,O}$, $l \notin loc(P)$, such that $([C]_l \| P) \Downarrow$, we will show that $([C']_l \| P) \Downarrow$. We first observe that $([C]_l \parallel P \setminus\!\!\setminus (I - O(C))) \Downarrow$. Since $C' \preccurlyeq_{I',O} C$ and $O(C) \subseteq I'$, we derive $([C']_l \parallel P \setminus\!\!\setminus (I - O(C))) \Downarrow$. Due to Proposition 3 we have that $([C']_l \| P \setminus\!\!\setminus (I - O(C)))$ can never reach by τ transitions a state where outputs in $O(C') - O(C)$ are executable by some derivative of C', so we conclude $([C']_l \| P) \Downarrow$.*

Proposition 5. *Let $C \in \mathcal{P}_{con}$ be a contract, $O, O' \subseteq \mathcal{N}_{loc}$ be two sets of located output names such that for every $l \in Loc$ we have $I(C) \subseteq O_l, O'_l$, and $I \subseteq \mathcal{N}_{loc}$ be a set of located input names. We have that for every contract $C' \in \mathcal{P}_{con}$,*

$$C' \preccurlyeq_{I,O} C \quad \Longleftrightarrow \quad C' \preccurlyeq_{I,O'} C$$

Proof. *Let us suppose $C' \preccurlyeq_{I,O'} C$ (the other direction is symmetric). Given any $l \in Loc$, $l \notin oloc(C) \cup oloc(C')$, and $P \in \mathcal{P}_{conpres,I,O}$, $l \notin loc(P)$, such that $([C]_l \| P) \Downarrow$, we show that $([C']_l \| P) \Downarrow$. We observe that $([C]_l \parallel P \setminus\!\!\setminus \overline{(O - I([C]_l))}) \Downarrow$. Since $C' \preccurlyeq_{I,O'} C$ and $I([C]_l) \subseteq O'$, we derive $([C']_l \parallel P \setminus\!\!\setminus \overline{(O - I([C]_l))}) \Downarrow$. As a consequence $([C']_l \setminus\!\!\setminus I(C') - I(C) \parallel P \setminus\!\!\setminus \overline{(O - I([C]_l))}) \Downarrow$ and $([C']_l \setminus\!\!\setminus I(C') - I(C) \parallel P) \Downarrow$. Due to Proposition 3 we have that $([C']_l \setminus\!\!\setminus I(C') - I(C) \parallel P)$ can never reach by τ transitions a state where outputs in $\overline{I([C']_l) - I([C]_l)}$ are executable by some derivative of P, so we conclude $([C']_l \| P) \Downarrow$.*

These last two Proposition permits us to forget about the restrictions on the input/output alphabets of the services in the context in which we apply the substitution of one contract with one of its subcontracts, considering always $\preccurlyeq_{\mathcal{N}_{loc},\mathcal{N}_{loc}}$. We denote this relation simply with $\preccurlyeq$ and we call it the *strong subcontract pre-order*.[1] We define $\preccurlyeq$ assuming a limited set of possible contexts and then we prove that this limitation is not relevant. The new set of

[1] It is easy to see that $\preccurlyeq_{\mathcal{N}_{loc},\mathcal{N}_{loc}}$ is reflexive and transitive.

contexts does not contain restrictions, i.e., we consider $[C_1]_{l_1} \| \cdots \| [C_n]_{l_n}$ instead of $([C_1]_{l_1} \| \cdots \| [C_n]_{l_n}) \backslash\backslash L$. We call $P \in \mathcal{P}_{conpar}$ the subset of systems of the form $[C_1]_{l_1} \| \cdots \| [C_n]_{l_n}$.

Definition 6. (Strong subcontract pre-order) *A contract C' is a strong subcontract of a contract C denoted $C' \preceq C$, if and only if for all $l \in Loc$ such that $l \notin oloc(C) \cup oloc(C')$ and $P \in \mathcal{P}_{conpar}$ such that $l \notin loc(P)$ we have*

$$([C]_l \| P)\Downarrow \quad \Rightarrow \quad ([C']_l \| P)\Downarrow$$

Proposition 6. *Let $C, C' \in \mathcal{P}_{con}$ be two contracts:*

$$C \preceq C' \qquad \textit{if and only if} \qquad C' \preceq_{\mathcal{N}_{loc}, \mathcal{N}_{loc}} C$$

Proof. The if *part is simple as $\preceq$ is defined as $\preceq_{\mathcal{N}_{loc}, \mathcal{N}_{loc}}$ assuming a subset of possible contexts P.*

We now prove the only-if *part. Supposed $P = ([C_1]_{l_1} \| \ldots \| [C_n]_{l_n}) \backslash\backslash L$, let $I, O \subset \mathcal{N}_{loc}$ be such that $I = \{a_l \mid a_{l_i} \in L \wedge 1 \leq i \leq n\}$ and $O = \{a_l \mid \overline{a}_l \in L\}$ (in O only outputs on the location l in the hypothesis of the proposition are cosidered). We have that $([C]_l \| (P \backslash\backslash L))\Downarrow \iff ([C]_l \| (P \backslash\backslash I \cup \overline{O}))\Downarrow \iff ([C]_l \| (P\{\tau; \mathbf{0}/\alpha | \alpha \in \overline{O}\} \backslash\backslash I))\Downarrow \iff ([C]_l \| P')\Downarrow$, where P' is obtained from $P'' \equiv P\{\tau; \mathbf{0}/\alpha | \alpha \in \overline{O}\}$ as follows. We call $M \in \mathcal{N}$ the (finite) set of action names occurring in C and C'. We consider an arbitrary injective function $rel : M \rightarrow (\mathcal{N} - M)$ that maps each action name a in M into a fresh name $rel(a)$. For each $a_{l'} \in I$, we do the following: (i) we replace each syntactical occurrence of a inside the unique subterm $[C'']_{l'}$ of P'' with $rel(a)$, and (ii) we replace each syntactical occurrence of $\overline{a}_{l'}$ inside P'' with $\overline{rel(a)}_{l'}$. Since the same chain of "$\iff$" holds for C' (using the same relabeling function "rel"), we have that the result is a direct consequence of the definition of strong subcontract pre-order applied to P'.*

4.2 Independent Refinement

In this subsection we prove that the strong subcontract pre-order $\preceq$, which has been defined assuming that the other services in the context are kept unchanged while applying the refinement, is suitable also for a more general refinement that is applied independently on all services contemporarily. This is an important property for a refinement notion suitable for service oriented computing; indeed, services are loosely coupled in the sense that they can be updated/modified independently one from the other ones. Independent refinements can be defined as follows.

Definition 7. (Independent subcontract pre-order) *A pre-order $\leq$ over $\mathcal{P}_{con}$ is an independent subcontract pre-order if, for any $n \geq 1$, contracts $C_1, \ldots, C_n \in \mathcal{P}_{con}$ and $C'_1, \ldots, C'_n \in \mathcal{P}_{con}$ such that $\forall i. C'_i \leq C_i$, and distinguished location names $l_1, \ldots, l_n \in Loc$ such that $\forall i. l_i \notin oloc(C_i) \cup oloc(C'_i)$, we have*

$$([C_1]_{l_1} \| \ldots \| [C_n]_{l_n})\Downarrow \quad \Rightarrow \quad ([C'_1]_{l_1} \| \ldots \| [C'_n]_{l_n})\Downarrow$$

We will show that the maximal individual subcontract pre-order corresponds to the pre-order $\preceq\!\!\!\!<$ defined in the previous subsection. This is achieved defining a more general class of pre-orders called *singular subcontract pre-orders*, showing that all independent subcontract pre-orders are also *singular subcontract pre-orders*, and finally observing that $\preceq\!\!\!\!<$ (which is the maximum of all singular subcontract pre-orders) is also a singular subcontract pre-order.

Intuitively a pre-order $\leq$ over $\mathcal{P}_{con}$ is a singular subcontract pre-order whenever the strong correctness of systems is preserved by refining just one of the contracts. More precisely, for any $n \geq 1$, contracts $C_1, \ldots, C_n \in \mathcal{P}_{con}$, $1 \leq i \leq n$,$C'_i \in \mathcal{P}_{con}$ such that $C'_i \leq C_i$, and distinguished location names $l_1, \ldots, l_n \in Loc$ such that $\forall k \neq i.\, l_k \notin oloc(C_k)$ and $l_i \notin oloc(C_i) \cup oloc(C'_i)$, we require

$$([C_1]_{l_1} \parallel \ldots \parallel [C_i]_{l_i} \parallel \ldots \parallel [C_n]_{l_n}) \Downarrow \quad \Rightarrow \quad ([C_1]_{l_1} \parallel \ldots \parallel [C'_i]_{l_i} \parallel \ldots \parallel [C_n]_{l_n}) \Downarrow$$

By exploiting commutativity and associativity of parallel composition we can group the contracts which are not being refined and get the following cleaner definition. We recall that $\mathcal{P}_{conpar}$ denotes the subset of systems of the form $[C_1]_{l_1} \| \ldots \| [C_n]_{l_n}$.

Definition 8. (Singular subcontract pre-order) *A pre-order $\leq$ over $\mathcal{P}_{con}$ is a singular pre-order if, for any $C, C' \in \mathcal{P}_{con}$ such that $C' \leq C$, $l \in Loc$ such that $l \notin oloc(C) \cup oloc(C')$, $P \in \mathcal{P}_{conpar}$ such that $l \notin loc(P)$ we have*

$$([C]_l \| P) \Downarrow \quad \Rightarrow \quad ([C']_l \| P) \Downarrow$$

It is easy to see that the strong subcontract pre-order $\preceq\!\!\!\!<$ is the maximal singular subcontract pre-order as it relates all pairs of contracts that satisfy the property stated in the Definition 8.

In order to prove the existence of the maximal independent subcontract pre-order, we will prove that every pre-order that is an independent subcontract is also a singular subcontract (Theorem 1), and vice-versa (Theorem 2).

Theorem 1. *If a pre-order $\leq$ is an independent subcontract pre-order then it is also a singular subcontract pre-order.*

Proof. *Suppose that $\leq$ is an independent subcontract pre-order. Consider $n \geq 1$, $C, C' \in \mathcal{P}_{con}$, $l \in Loc$ and $P \in \mathcal{P}_{conpar}$ such that $l \notin loc(P)$. From $([C]_l \| P) \Downarrow$ and $C' \leq C$, we can derive $([C']_l \| P) \Downarrow$ by just taking in the definition of independent subcontract pre-order, $C_1 = C$, $C'_1 = C'$, $C_2 \ldots C_n$ to be such that $P = (C_2 \| \ldots \| C_n)$ and finally C'_i to be C_i for every $i \geq 2$ (since $\leq$ is a pre-order we have $C \leq C$ for every C).*

Theorem 2. *If a pre-order $\leq$ is a singular subcontract pre-order then it is also an independent subcontract pre-order*

Proof. *Consider $n \geq 1$, contracts $C_1, \ldots, C_n \in \mathcal{P}_{con}$ and $C'_1, \ldots, C'_n \in \mathcal{P}_{con}$ such that $\forall i.\, C'_i \leq C_i$, and distinguished location names $l_1, \ldots, l_n \in Loc$ such that $\forall i.\, l_i \notin oloc(C_i) \cup oloc(C'_i)$. For any i we let $P_i = [C_i]_{l_i}$ and $P'_i = [C'_i]_{l_i}$. If $(P_1 \| \ldots \| P_n) \Downarrow$ we can derive $(P'_1 \| \ldots \| P'_n) \Downarrow$ in n steps: at the i-th step we replace P_i with P'_i without altering the correctness of the system.*

We can, therefore, conclude that there exists a maximal independent subcontract pre-order and it corresponds to "$\preccurlyeq$ ".

4.3 Resorting to Should Testing

The remainder of this section is devoted to the definition of an actual procedure for determining that two contracts are in strong subcontract relation. This is achieved resorting to the theory of *should-testing* [RV05].

In the following we use the following abuse of notation: "$C \backslash\!\backslash M$" stands for "$C\{\mathbf{0}/\alpha | \alpha \in M\}$". This allows us, e.g., to achieve a transition system isomorphic to that of $[C]_l \backslash\!\backslash I$, with $I = \{a_l \mid a \in M\}$ for some $M \subseteq \mathcal{N}$, simply by considering $[C \backslash\!\backslash M]_l$.

First, we need a preliminary result that is a direct consequence of the fact that $C' \preccurlyeq_{\mathcal{N}_{loc}, \bigcup_{l \in Loc} I([C]_l)} C$ if and only if $C' \preccurlyeq C$, due to Proposition 5.

Lemma 1. *Let $C, C' \in \mathcal{P}_{con}$ be contracts. We have*

$$C' \backslash\!\backslash (I(C') - I(C)) \preccurlyeq C \quad \Rightarrow \quad C' \preccurlyeq C$$

Proof. We will show that the hypothesis yields $C' \preccurlyeq_{\mathcal{N}_{loc}, \bigcup_{l \in Loc} I([C]_l)} C$. From this we can derive the result by using Proposition 5. Given any $l \in Loc$, $l \notin oloc(C) \cup oloc(C')$, and $P \in \mathcal{P}_{conpres, \mathcal{N}_{loc}, \bigcup_{l \in Loc} I([C]_l)}$, $l \notin loc(P)$, such that $([C]_l \| P) \Downarrow$, we will show that $([C']_l \| P) \Downarrow$. We have $([C' \backslash\!\backslash I(C') - I(C)]_l \parallel P) \Downarrow \iff ([C' \backslash\!\backslash I(C') - I(C)]_l \parallel P \backslash\!\backslash \overline{I([C']_l) - I([C]_l)}) \Downarrow \iff ([C']_l \parallel P \backslash\!\backslash \overline{I([C']_l) - I([C]_l)}) \Downarrow \iff ([C']_l \| P) \Downarrow$.

Note that the opposite implication trivially holds (by taking $O = \mathcal{N}_{loc}$ and $I = \mathcal{N}_{loc}$ in Proposition 2).

In the following we denote with $\preceq_{test}$ the *should-testing* pre-order defined in [RV05] where we consider the set of actions used by terms as being $\mathcal{L}$ (i.e. we consider located input and output actions, unlocated input actions, and $\surd$ is included in the set of actions of terms under testing as any other action). We denote here with $\surd'$ the special action for the success of the test (denoted by $\surd$ in [RV05]).

In order to resort to the theory defined in [RV05], we first define a transformation on the finite labeled transition system (LTS) of a system P. The trasformation is performed in two steps:

1. First, for every state s of the LTS we do the following: called $I(s)$ the set of labels of outgoing input transitions from the state s, for every label in $I(P) - I(s)$ we add to the state s an outgoing input transition with that label that leads to the $\mathbf{0}$ state.
2. Then, for every output transition $\overline{a}_l$, we replace the output transition with a pair of connected transitions (the first one has the same source as the previous output transition and the second one has the same destination as the previous output transition): a τ transition followed by an $\overline{a}_l$ output transition.

The same transformation is defined also on contracts C. Here, we describe the effect of the two above transformations applied to the contracts checked in the testing scenario that we are going to formalize. The first transformation on the labeled transition system is used to capture those output actions performed by the tester for which there is no corresponding input action in the tested process; the new added input can synchronize with these actions and lead to the state $\mathbf{0}$. This disallows the possibility for the tester to complete its execution, thus to succeed. The second transformation, on the other hand, is necessary to check whether all output actions performable by the tested contract can be received by the tester: in fact, the *tau* transition added before the output action permits to enter in a state where only the output action is executable. If the tester does not consume this output the contract stucks and the test cannot succeed.

Now we derive a normal form for systems and contracts of our calculus that corresponds to terms of the language in [RV05]. The normal form of the system P (denoted with $\mathcal{NF}(P)$) is defined as follows, by using the operator $rec_X\theta$ (defined in [RV05]) that represents the value of X in the solution of the minimum fixpoint of the finite set of equations θ,

$$\mathcal{NF}(P) = rec_{X_1}\theta \qquad \text{where } \theta \text{ is the set of } i\text{-indexed equations}$$

$$X_i = \textstyle\sum_j \lambda_{i,j}; X_{der(i,j)}$$

where, assuming to enumerate the states in the transformed (according to the two-steps procedure above) labeled transition system of P starting from X_1, each variable X_i corresponds to the i-th state of the transformed labeled transition system of P, $\lambda_{i,j}$ is the label of the j-th outgoing transition from X_i, and $der(i,j)$ is the index of the state reached with the j-th outgoing transition from X_i. We assume empty sums to be equal to $\mathbf{0}$, i.e. if there are no outgoing transitions from X_i, we have $X_i = \mathbf{0}$. The normal form of a contract C (denoted with $\mathcal{NF}(C)$) is defined in the same way.

Theorem 3. *Let $C, C' \in \mathcal{P}_{con}$ be two contracts. We have*

$$\mathcal{NF}(C' \backslash\!\!\backslash I(C') - I(C)) \preceq_{test} \mathcal{NF}(C) \quad \Rightarrow \quad C' \underline{\prec\!\!\prec}\, C$$

Proof. According to the definition of should-testing of [RV05], since $\mathcal{NF}(C' \backslash\!\!\backslash (I(C') - I(C))) \preceq_{test} \mathcal{NF}(C)$ we have that, for every test t, if $\mathcal{NF}(C)$ **shd** *t, then also $\mathcal{NF}(C' \backslash\!\!\backslash (I(C') - I(C)))$* **shd** *$t$, where Q* **shd** *t iff*

$$\forall w \in \mathcal{L}^*, Q'. \qquad Q \|_{\mathcal{L}} t \xrightarrow{w} Q' \qquad \Rightarrow \qquad \exists v \in \mathcal{L}^*, Q'' : Q' \xrightarrow{v} Q'' \xrightarrow{\surd'}$$

where $\|_{\mathcal{L}}$ is the CSP parallel operator: in $R\|_{\mathcal{L}}R'$ transitions of R and R' with the same label λ (with $\lambda \neq \tau, \surd'$) are required to synchronize and yield a transition with label λ.

Let us now consider $l \in Loc$, $l \notin oloc(C) \cup oloc(C')$, and $P \in \mathcal{P}_{conpar}$, $l \notin loc(P)$, such that $([C]_l \| P)\Downarrow$. We consider $t = \mathcal{NF}(P)\{\surd; \surd'/\surd\}\{\overline{a}/\overline{a}_l | a \in \mathcal{N}\}$, i.e., the normal form of P where: we replace each occurrence of $\surd$ with the *sequence* $\surd;\surd'$ *and we turn every output action* $\overline{a}_l$ *directed to* $[C]_l$ *into* $\overline{a}$*. We denote with* $\overline{t}$ *the term obtained by turning each* $a_{l'}$ *occurring in* t *into* $\overline{a}_{l'}$*, and each* $\overline{a}_{l'}$ $(\overline{a})$ *into* $a_{l'}$ (a)*. From the definition of* **shd** *it follows that* $\mathcal{NF}(C)$ $\mathbf{shd}$ $\overline{t}$*. Since* $\mathcal{NF}(C' \backslash\!\backslash (I(C') - I(C))) \preceq_{test} \mathcal{NF}(C)$*, we have that also* $\mathcal{NF}(C' \backslash\!\backslash (I(C') - I(C)))$ $\mathbf{shd}$ $\overline{t}$*. From the definition of* **shd** *and from the fact that output transitions are always preceded by* τ *transitions (which guarantees that no output can be enabled that does not have a synchronizing input transition) we can conclude that* $(C' \backslash\!\backslash I(C') - I(C) \| P) \Downarrow$*. The thesis directly follows from Lemma 1.*

5 Related Work and Conclusion

We have considered a new notion of correctness for service compositions, modeled using process calculi, in which we assume that whenever a message is sent to a service in order to invoke a particular operation, the service should be ready to serve it. We call this new notion *strong compliance*, and we develop around it an entire theory. It comprises a suitable refinement for services based on a *strong subcontract pre-order*, the proof that this refinement can be applied on each of the service inside a composition independently, and an effective procedure that can be used to prove whether a contract is a subcontract of another one.

The theory of contracts reported in this paper is different from the theory reported in our previous paper [BZ07] in several aspects. The calculus considered in [BZ07] imposes a limitation to output actions that are always preceeded by τ internal actions; on the contrary, in this paper we consider standard input and output prefixes. In this paper we add locations to services and output operations in order to indicate the target of a message; this reflects more faithfully the Web Services technology in which invocations include both the address of the service and the operation to be invoked. Moreover, in this paper we consider a stronger notion of compliance that requires to completely revisit the corresponding notion of (strong) subcontract. The most interesting result is that, even if we consider a more general language than [BZ07], we can achieve even stronger results thanks to the notion of strong compliance. In particular, the strong subcontract pre-order can now be defined abstracting away from both the input and the output alphabets of the services in the context. Moreover, the characterization of the strong subcontract pre-order (achieved also in this paper resorting to the theory of testing) requires new nontrivial technicalities.

It is worth noting that there are some important differences between our form of testing and the traditional one proposed by De Nicola-Hennessy [DH84]. The main difference is that, besides requiring the success of the test, we impose also that the tested process should successfully complete its execution. Moreover, all output actions of both the tester and the tested process should be immediately receivable. Another difference is in the treatment of divergence: we do not follow the traditional catastrophic approach, but the fair approach introduced by the theory of should-testing by Rensink-Vogler [RV05]. In fact, we do not impose that all computations must succeed, but that all computations can always be extended in order to reach success.

We conclude our analysis of related work considering the theory of contracts proposed by Fournet et al. [FHR$^+$04] and by Carpineti et al. [CCL$^+$06].

In [FHR$^+$04] contracts are CCS-like processes; a generic process P is defined as compliant to a contract C if, for every tuple of names $\tilde{a}$ and process Q, whenever $(\nu\tilde{a})(C|Q)$ is stuck-free then also $(\nu\tilde{a})(P|Q)$ is. Our notion of contract refinement differs from stuck-free conformance mainly because we consider a different notion of stuckness. In [FHR$^+$04] a process state is stuck (on a tuple of channel names $\tilde{a}$) if it has no internal moves (but it can execute at least one action on one of the channels in $\tilde{a}$). In our approach, an end-states different from successful termination is stuck (independently of any tuple $\tilde{a}$). Thus, we distinguish between internal deadlock and successful completion while this is not the case in [FHR$^+$04]. Another difference follows from the exploitation of the restriction $(\nu\tilde{a})$; this is used in [FHR$^+$04] to explicitly indicate the local channels of communication used between the contract C and the process Q. In our context we can make a stronger *closed-world* assumption (corresponding to a restriction on all channel names) because service contracts do not describe the entire behaviour of a service, but the flow of execution of its operations inside one session of communication.

The closed-world assumption is considered also in [CCL$^+$06] where, as in our case, a service oriented scenario is considered. In particular, in [CCL$^+$06] a theory of contracts is defined for investigating the compatibility between one client and one service. Our paper consider multi-party composition where several services are composed in a peer-to-peer manner. Moreover, we impose service substitutability as a mandatory property for our notion of refinement; this does not hold in [CCL$^+$06] where it is not in general possible to substitute a service exposing one contract with another one exposing a subcontract.

As future work, we plan to investigate strong compliance also in the context of choreography languages, considering a process algebraic modeling of choreographies that follows the proposals by Busi et al. [BGG$^+$05, BGG$^+$06] and by Carbone et al. [CHY07]. In particular, we intend to define a procedure for extracting from a choreography the set of contracts (and strong subcontracts) of services that could correctly play the roles specified in the choreography.

References

[BZ07] Bravetti, M., Zavattaro, G.: Contract based Multi-party Service Composition. In: FSEN'07, volume to appear of LNCS (2007)

[BGG$^+$05] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration: A synergic approach for system design. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 228–240. Springer, Heidelberg (2005)

[BGG$^+$06] Busi, N., Gorrieri, R., Guidi, C., Lucchi, R., Zavattaro, G.: Choreography and orchestration conformance for system design. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 63–81. Springer, Heidelberg (2006)

[CHY07] Carbone, M., Honda, K., Yoshida, N.: Structured Communication-Centred Programming for Web Services. In: ESOP'07, volume to appear of LNCS (2007)

[CCL+06] Carpineti, S., Castagna, G., Laneve, C., Padovani, L.: A Formal Account of Contracts for Web Services. In: Bravetti, M., Núñez, M., Zavattaro, G. (eds.) WS-FM 2006. LNCS, vol. 4184, pp. 148–162. Springer, Heidelberg (2006)

[DH84] De Nicola, R., Hennessy, M.: Testing Equivalences for Processes. Theoretical Computer Science 34, 83–133 (1984)

[FHR+04] Fournet, C., Hoare, C.A.R., Rajamani, S.K., Rehof, J.: Stuck-Free Conformance. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 242–254. Springer, Heidelberg (2004)

[Ley01] Leymann, F.: Web Services Flow Language (wsfl 1.0). Technical report, IBM Software Group (2001)

[RV05] Rensink, A., Vogler, W.: Fair testing. CTIT Technical Report TR-CTIT-05-64, Dep. of Computer Science, University of Twente (2005)

[OAS] OASIS. Web Services Business Process Execution Language Version 2.0. http://www.oasis-open.org/committees/download.php/10347/wsbpel-specification-draft-120204.htm

[Tha01] Thatte, S.: XLANG: Web services for business process design. Microsoft Corporation (2001)

[W3C] W3C. Web Services Choreography Description Language. http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/

Towards a Theory of Refinement in Timed Coordination Languages

Jean-Marie Jacquet and Isabelle Linden

Institute of Informatics, FUNDP – University of Namur
{jmj,ili}@info.fundp.ac.be
http://www.info.fundp.ac.be/~{jmj,ili}

Abstract. Building upon previous work on timed coordination languages, this paper presents a novel notion of refinement for these languages which satisfies the substitutability property: if the implementation I refines the specification S and if $C[S]$ is deadlock free, for some context C, then $C[I]$ is also deadlock free.

1 Introduction

As motivated by the constant expansion of computer networks and illustrated by the development of distributed applications, the design of modern software systems centers on re-using and integrating software components. This induces a paradigm shift from stand-alone applications to interacting distributed systems, which, in turn, naturally calls for well-defined methodologies and tools aiming at integrating heterogeneous software components.

In this context, a clear separation between the *interactional* and the *computational* aspects of software components has been advocated by Gelernter and Carriero in [14]. Their claim has been supported by the design of a model, Linda ([8]), originally presented as a set of inter-agent communication primitives which may be added to almost any programming language. Besides process creation, this set includes primitives for adding, deleting, and testing the presence/absence of data in a shared dataspace.

A number of other models, now referred to as coordination models, have been proposed afterwards (see [24,25] for a comprehensive survey of many of them). One the extensions, of interest for this paper, concerns the introduction of time. It is motivated both by industrial proposals such as JavaSpaces ([13]) and TSpaces ([29]) as well as by the coding of applications which evidence the fact that data and requests rarely have an eternal life. For instance, a request for information on the web has to be satisfied in a reasonable amount of time. Even more crucial is the request for an ambulance which, not only has to be answered eventually but within a critical period of time. The list could also be continued with software in the areas of air-traffic control, manufacturing plants and telecommunication switches, which are inherently reactive and, for which, interaction must occur in "real-time".

In our recent work ([17,19,20,21]), we have proposed different ways of introducing time in coordination languages. For that purpose, we have used the

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 113–131, 2007.

classical *two-phase functioning* approach to real-time systems and have proved that this approach was effective for modeling coordination in reactive systems.

However, although the need for techniques and tools to reason about concurrent programs is widely recognized in the concurrency community, using process algebras such as CCS ([22]), CSP ([16]), π-calculus ([23]) and is met by a large body of work (see for instance [1,3,4,12]), little attention has been paid to programming methodologies in the coordination community. This lack is even more crucial in the context of real-time systems for which strict delays have to be guaranteed.

This paper aims at contributing to this effort. Work in other settings like the B-method ([2]), the FDR tool ([26]), the concurrency workbench ([10]) have evidenced that refinement is a fundamental notion for reasoning. After having shown that the traditional definitions of refinement do not transpose directly in a satisfactory way to Linda-like languages and, consequently, to our timed coordination framework, we shall propose a new notion of refinement which satisfies the substitutability property: if the implementation I refines the specification S and if $C[S]$ is deadlock free, for some context C, then $C[I]$ is also deadlock free. This property is particularly crucial since it enables a compositional way of reasoning and thereby helps model checking to scale. Moreover, as Linda-like languages can be embedded in our time setting, our notion of refinement also applies to Linda-like languages and thus to a wide range of languages.

The rest of this paper is organized as follows. Section 2 introduces our timed coordination models. Section 3 explains why the classical notions of trace-refinement and failure-refinement are not suited for the coordination context. In the aim of laying down the foundations to explain our refinement in section 5, section 4 studies an event based semantics. Finally, section 6 draws our conclusions and compares our work with related work.

2 Timed Coordination Languages

Our approach to the introduction of time in coordination languages follows the classical two-phase functioning approach to real-time systems illustrated by languages such as Lustre ([9]), Esterel ([5]) and Statecharts ([15]). This approach may be described as follows. In a first phase, elementary actions of statements are executed. They are assumed to be atomic in the sense that they take no time. Similarly, composition operators are assumed to be executed at no cost. In a second phase, when no actions can be reduced or when all the components encounter a special timed action, time progresses by one unit.

In that context, four families of timed coordination languages have been introduced in [17]. They are obtained

1. by introducing delays stating that a communication primitive should only be processed after some units of time;
2. by stating that tuples on the tuple space are only valid for some units of time and that, similarly, requests for tuples are to be made during a period of time;

3. by introducing delays for some specific points in time;
4. by specifying absolute intervals of time in which actions should be processed and, dually, by associating such intervals with tuples.

The first two families are said to incorporate a relative notion of time since the delays and the validity of tuples and of operations are defined at execution time with respect to their moments of consideration. In contrast, the last two families are said to incorporate an absolute notion of time because they refer statically to specific instants of a clock.

The expressiveness of these families has been studied in [17,19,20,21]. Two interesting conclusions may be extracted from these papers. On the one hand, from a programming point of view, the language embodying relative delays and relative timed primitives (namely embodying the features of points 1 and 2 above) is the most interesting one. On the other hand, from an implementation point of view, the language incorporating absolute delays and absolute timed primitives (namely the features of points 3 and 4 above) is the most fundamental one. We shall thus use the former language is this paper. Formally it is defined as follows.

Definition 1. *Let Stoken be a denumerable set, the elements of which are subsequently called* tokens *and are typically represented by the letters t and u. Let Stime be the set of time units or durations defined as the set composed of the positive integers and of ∞ denoting infinity. Elements of Stime are typically represented by the letter d. Let Sprocvar be a denumerable set disjoint with Stoken and Stime, the elements of which are typically denoted by X and are called* procedure variables*. Define the language $\mathcal{L}$ as the set of agents A generated by the following grammar*

$$\begin{aligned} C &::= tell_d(t) \mid ask_d(t) \mid get_d(t) \mid nask_d(t) \mid delay(d) \\ A &::= C \mid A \,;\, A \mid A \,||\, A \mid A \,+\, A \mid X \end{aligned}$$

where the durations d in the subscripts are not null.

As easily noticed by the careful reader, the communication primitives of the language $\mathcal{L}$ are thus basically the Linda primitives equipped with time. Indeed, the Linda primitives `out`, `in` and `rd` for, respectively, putting an object t in a shared dataspace, getting it and checking for its presence are renamed as `tell`, `get` and `ask` for compatibility with the syntax used in our previous publications. Moreover, a primitive `nask(t)` has been added to test the absence of t on the shared dataspace.

These primitives are enriched with durations (syntactically denoted by subscripts) with the following intuition:

- the execution of $tell_d(t)$ adds t to the dataspace but for d units of time only,
- if the execution of the $ask_d(t)$, $nask_d(t)$, and $get_d(t)$ primitives need to suspend[1], this may only occur during d units of time, after which the primitives fail.

[1] Because of the non availability of t for the `ask` and `get` primitives or because of the availability of t for the `nask` primitive.

To these primitives is added the primitive `delay(d)` whose purpose is to force time to pass by d units of time.

The composition operators are the traditional ones in concurrency theory: $;$, $||$ and $+$ are used to respectively denote sequential composition, parallel composition and external choice. Finally, following [11], the letter X is used to denote an abstraction of procedure call and to allow recursion. These procedure calls are defined as guarded agents by means of declarations, as follows.

Definition 2. *Define the set $\mathcal{G}$ of guarded agents as the set of the agents G given by the following grammar:*

$$\begin{aligned} C &::= tell_d(t) \mid ask_d(t) \mid get_d(t) \mid nask_d(t) \mid delay(d) \\ A &::= C \mid A \; ; \; A \mid A \, || \, A \mid A \, + \, A \mid X \\ G &::= C \mid G \; ; \; A \mid G \, || \, G \mid G \, + \, G \end{aligned}$$

where the durations in the subscripts are not null.

Definition 3. *A declaration D is a list of associations $<X, G>$ between procedure variables and guarded agents. Such a list may be infinite, so that D is formally regarded as a mapping from procedure variables to guarded agents. For the ease of reading, we shall also rewrite $<X, G>$ as $X = G$.*

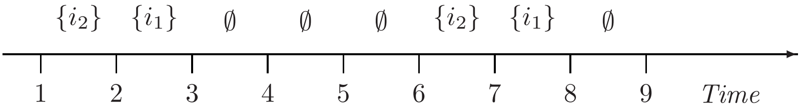
To simplify the notations, we shall subsequently assume a declaration D to be given and will omit it when no confusion is introduced.

As easily observed from the above definitions, the main property of guarded agents is that any call to a procedure variable X is always preceded by at least one communication primitive C. This (classical) property ensures that equations of the form $X = G$ are well-defined. Note that we do not require that procedure calls are preceded by a tick clock. For instance, $X = tell_1(t) \; ; \; X$ is allowed. It corresponds to a process that infinitively produces occurrences of the token t in the same unit of time. Such behaviors are named Zeno-behaviors (see eg [18]). They will be treated in a companion paper.

Example 1. An example may help to understand the above concepts. Let us code a producer process which recursively produces items outdated after two units of time and which takes a rest of five units of time after each item is produced. Such a process may be coded as follows, where *Prod* represents the producer and i the item being produced:

$$Prod = tell_2(i) \; ; \; delay(5) \; ; \; Prod$$

It is worth observing the contents of the shared dataspace during the execution of *Prod*. Started at time 1 with an empty dataspace, *Prod* first executes the primitives $tell_2(i)$ which puts i for two units of time, namely until time 3. The execution of the primitive itself takes no time so that *Prod* then executes the $delay(5)$ operation which forces it to stay idle until time 6. As a result, assuming *Prod* is the only process being executed, the dataspace is again empty between times 3 and 6. This is summarized in the following picture where the contents of the dataspace is drawn in each interval of time after the execution of the $tell(i)$ primitive and where the subscript associated with i denotes the current life of the item.



Similarly, a process which recursively requests the item i for one unit of time and takes two units of time to actually consume it may be coded as follows:

$$Cons = get_1(i)\ ;\ delay(2)\ ;\ Cons$$

The computation of $Prod \mid\mid Cons$ is then worth observing. As its first action is to get i, the agent $Cons$ has to wait that $Prod$ has produced i. It can then get it and then calls itself recursively after having wait two units of time, namely at time 3. At that moment, $Cons$ tries to get i again but for one unit of time only. However, this is too short since $Prod$ will only put a new occurrence of i at time 6. As a result, the agent $Cons$ is then be blocked for the rest of the computation.

As just illustrated by the above example, tokens and primitives get older as time evolves. This is formally captured by the following definitions.

Definition 4

1. *Given an agent $A \in \mathcal{L}$, we denote by A^- the agent defined inductively as follows:*[2]

$$\begin{aligned} delay(d)^- &= delay(max\{0, d-1\} \\ tell_d(t)^- &= tell_d(t) \\ ask_d(t)^- &= ask_{max\{0,d-1\}}(t) \\ nask_d(t)^- &= nask_{max\{0,d-1\}}(t) \\ get_d(t)^- &= get_{max\{0,d-1\}}(t) \end{aligned} \qquad \begin{aligned} X^- &= X \\ (B\ ;\ C)^- &= B^-\ ;\ C \\ (B \mid\mid C)^- &= B^- \mid\mid C^- \\ (B\ +\ C)^- &= B^-\ +\ C^- \end{aligned}$$

2. *Define the set of timed stores*[3] *Ststore as the set of multisets of elements of the form t_d where t is a token and d is a duration. Given a timed store σ, we denote by σ^- the new store obtained by decreasing the duration associated with the tokens by one unit and by removing those associated in σ with 1 unit of time: precisely, if all the notations are understood to relate to multi-sets,*

$$\sigma^- = \{t_{d-1} : t_d \in \sigma, d > 1\}$$

[2] We extend classical arithmetic on natural numbers by asserting that $\infty - 1 = \infty$.

[3] We use store as a synonym for shared date space.

According to the two-phase functioning approach, a temporal step will be done when no communication primitives can be executed. However, given that the execution of tell primitives can always proceed, a temporal step only makes sense if the agent under consideration offers the hope of an execution step in the future, namely if it contains in an executable position an ask, nask or get primitive or delay primitive associated with a non null duration. This is formally expressed by the two following definitions.

Definition 5. *Let* Scom *denote the set of communication primitives* $tell_d(t)$, $ask_d(t)$, $get_d(t)$, $nask_d(t)$ *and* $delay(d)$ *for any* $d \in Stime$ *and* $t \in Stoken$. *Define* $\mathcal{F} : \mathcal{L} \rightarrow \mathcal{P}(Scom)$ *as the following function: for any communication primitive* c*, procedure variable* X *defined in* D *by the declaration* $<X, G>$*, and agents* A *and* B*,*

$$\begin{array}{cc} \mathcal{F}(c) = \{c\} & \mathcal{F}(A\ ;\ B) = \mathcal{F}(A) \\ \mathcal{F}(X) = \mathcal{F}(G) & \mathcal{F}(A\ +\ B) = \mathcal{F}(A) \cup \mathcal{F}(B) \\ & \mathcal{F}(A \mid\mid B) = \mathcal{F}(A) \cup \mathcal{F}(B) \end{array}$$

Definition 6. *For any agent* A*, the predicate* $A \gg$ *holds iff the set* $\mathcal{F}(A)$ *contains at least one primitive ask, nask, get or delay associated with a non null duration.*

Example 2. Consider the agents $Prod$ and $Cons$ of example 1 at the end of the first unit of time. $Prod$ has become $delay(5)\ ;\ Prod$ and $Cons$ has become $delay(2)\ ;\ Cons$. Let us name these agents $Prod'$ and $Cons'$, respectively. Intuitively, both agents are worth being continued. This is met by the above formalization. Indeed, one has $\mathcal{F}(Prod' \mid\mid Cons') = \{delay(5), delay(2)\}$ and, consequently, $(Prod' \mid\mid Cons') \gg$. In contrast, consider $Cons$ alone after one unit of time. As no item i is produced, $Cons$ has become $get_0(i)\ ;\ delay(2)\ ;\ Cons$. Let us denote by $Cons''$ this agent. One has $\mathcal{F}(Cons'') = \{get_0(i)\}$ and, accordingly, $Cons'' \not\gg$. This translates the fact that it is not interesting to continue the computation of $Cons$ (which will remain blocked forever).

The computations in $\mathcal{L}$ may be modeled by a transition system written in Plotkin's style. Following the explanation given above, the configurations to be considered consist of an agent together with a multi-set of timed tokens, denoting the tokens currently available for the computation together with their durations.

To easily express successful termination, we shall introduce particular configurations composed of a special terminating symbol E together with a multi-set of tokens. For uniformity purposes, we shall abuse language and qualify E as an agent. However, to meet the intuition, we shall always rewrite agents of the form $(E\ ;\ A)$, $(E \mid\mid A)$, and $(A \mid\mid E)$ as A. This is technically achieved by defining the extended set of agents as follows, and by justifying the simplifications by imposing a bimonoid structure. Finally, in contrast to the agents of definition 1, we shall allow communication primitives associated with null durations.

Communication primitives

$$(\mathbf{T}) \quad \frac{d > 0}{\langle tell_d(t) \mid \sigma \rangle_u \rightarrow \langle E \mid \sigma \cup \{t_d\} \rangle_u}$$

$$(\mathbf{A}) \quad \frac{d, k > 0}{\langle ask_d(t) \mid \sigma \cup \{t_k\} \rangle_u \rightarrow \langle E \mid \sigma \cup \{t_k\} \rangle_u}$$

$$(\mathbf{N}) \quad \frac{d > 0,\ \nexists k > 0 : t_k \in \sigma}{\langle nask_d(t) \mid \sigma \rangle_u \rightarrow \langle E \mid \sigma \rangle_u}$$

$$(\mathbf{G}) \quad \frac{d, k > 0}{\langle get_d(t) \mid \sigma \cup \{t_k\} \rangle_u \rightarrow \langle E \mid \sigma \rangle_u}$$

$$(\mathbf{D}) \quad \langle delay(0) \mid \sigma \rangle_u \rightarrow \langle E \mid \sigma \rangle_u$$

General operators

$$(\mathbf{S}) \quad \frac{\langle A \mid \sigma \rangle_u \rightarrow \langle A' \mid \sigma' \rangle_u}{\langle A \ ; \ B \mid \sigma \rangle_u \rightarrow \langle A' \ ; \ B \mid \sigma' \rangle_u}$$

$$(\mathbf{P}) \quad \frac{\langle A \mid \sigma \rangle_u \rightarrow \langle A' \mid \sigma' \rangle_u}{\begin{array}{c} \langle A \mid\mid B \mid \sigma \rangle_u \rightarrow \langle A' \mid\mid B \mid \sigma' \rangle_u \\ \langle B \mid\mid A \mid \sigma \rangle_u \rightarrow \langle B \mid\mid A' \mid \sigma' \rangle_u \end{array}}$$

$$(\mathbf{C}) \quad \frac{\langle A \mid \sigma \rangle_u \rightarrow \langle A' \mid \sigma' \rangle_u}{\begin{array}{c} \langle A \ + \ B \mid \sigma \rangle_u \rightarrow \langle A' \mid \sigma' \rangle_u \\ \langle B \ + \ A \mid \sigma \rangle_u \rightarrow \langle A' \mid \sigma' \rangle_u \end{array}}$$

$$(\mathbf{R}) \quad \frac{< X, G > \in D,\ \langle G \mid \sigma \rangle_u \rightarrow \langle A' \mid \sigma' \rangle_u}{\langle X \mid \sigma \rangle_u \rightarrow \langle A' \mid \sigma' \rangle_u}$$

Time passage

$$(\mathbf{Ti}) \quad \frac{A \neq E, A \gg, \langle A \mid \sigma \rangle_u \not\rightarrow}{\langle A \mid \sigma \rangle_u \rightsquigarrow \langle A^- \mid \sigma^- \rangle_{u+1}}$$

Fig. 1. Transition rules of $\mathcal{L}$

Definition 7. *Define the set of extended agents $\mathcal{L}^e$ as the set of the agents Ae defined by the following grammar*

$$\begin{aligned} C &::= tell_d(t) \mid ask_d(t) \mid get_d(t) \mid nask_d(t) \mid delay(d) \\ A &::= C \mid A \ ; \ A \mid A \mid\mid A \mid A \ + \ A \mid X \\ Ae &::= E \mid A \end{aligned}$$

Moreover, we shall subsequently assert that the structure $(Seagent, E, \ ; \ , \ || \ )$ is a bimonoid and simplify elements of $Seagent$ accordingly. Finally, we extend the A^- notation by stating that $E^- = E$.

The transition rules are the ones given in figure 1. An operational semantics may be defined directly from them by reporting the traces of all the computation steps made during the executions. Formally, it is specified in definition 9 where δ^+ and δ^- are respectively used as ending marks respectively denoting successful and failing computations. Moreover, given a set S, we respectively denote by S^ω and S^* the set of infinite sequences and finite sequences composed from elements of S.

Definition 8

1. *Define the set of configurations $Stconf$ as $\mathcal{L}^e \times Ststore \times Stime$. Configurations are denoted as $\langle A \mid \sigma \rangle_u$, where A is an agent, σ is a timed store of $Ststore$ and u is a time.*
2. *Define the set $Sthist$ as the set $Ststore^\omega \cup Ststore^*.\{\delta^+, \delta^-\}$*

Definition 9. *Define the operational semantics $\mathcal{O}_h : \mathcal{L} \to \mathcal{P}(Sthist)$ as the following function. For any agent A,*

$$\begin{aligned}\mathcal{O}_h(A) = {} & \{{\sigma^0}_{u_0}.\cdots.{\sigma^n}_{u_n}.\delta^+ : \langle A_0 \mid \sigma^0\rangle_{u_0} \mapsto \cdots \mapsto \langle A_n \mid \sigma^n\rangle_{u_n}, \\ & \qquad A_0 = A, \sigma^0 = \emptyset, u_0 = 1, A_n = E, n \geq 0\} \\ & \cup \\ & \{{\sigma^0}_{u_0}.\cdots.{\sigma^n}_{u_n}.\delta^- : \langle A_0 \mid \sigma^0\rangle_{u_0} \mapsto \cdots \mapsto \langle A_n \mid \sigma^n\rangle_{u_n} \not\mapsto, \\ & \qquad A_0 = A, \sigma^0 = \emptyset, u_0 = 1, A_n \neq E, n \geq 0\} \\ & \cup \\ & \{{\sigma^0}_{u_0}.\cdots.{\sigma^n}_{u_n}.\cdots : \langle A_0 \mid \sigma^0\rangle_{u_0} \mapsto \cdots \mapsto \langle A_n \mid \sigma^n\rangle_{u_n} \mapsto \cdots, \\ & \qquad A_0 = A, \sigma^0 = \emptyset, u_0 = 1, \forall n \geq 0 : A_n \neq E\}\end{aligned}$$

where the $\mapsto$ arrow denotes either a $\longrightarrow$ transition or a $\leadsto$ transition.

Note that the behavior of (untimed) Linda primitives is obtained as a particular case of the transitions where all the communication primitives are associated with a 1 time. Our language $\mathcal{L}$ thus subsumes Linda and, consequently, the refinement theory we shall develop applies to Linda-like languages as well.

3 Trace and Failure Sets Refinements

In the traditional lines, exemplified by [2] and [26], one may define a first notion of refinement by stating that an implementation I refines a specification S (both I and S being agents of the $\mathcal{L}$ language) if any trace of execution made by I can also be made by S, namely if $\mathcal{O}_h(I) \subseteq \mathcal{O}_h(S)$.

Unfortunately, trace refinement takes only into account the actual traces computed by the agents regardless of their possible interactions with an environment. As a result, trace refinement does not enjoy the required property of substitutability.

Consider for instance $A = get_1(a)$ and $B = get_1(b)$. One has $\mathcal{O}_h(A) = \{\emptyset_1.\delta^-\} = \mathcal{O}_h(B)$ and, consequently, A trace-refines B. Consider now $C = tell_1(b)$. The parallel composition $B \mid\mid C$ has just one successful computation $\emptyset_1.\{b\}_1.\emptyset_1.\delta^+$ whereas the alternative parallel composition $A \mid\mid C$ has just one failing computation $\emptyset_1.\{b\}_1.\{b\}_2.\delta^-$. However, if trace refinement would enjoy the substitutability property, given that $B \mid\mid C$ is deadlock free, then $A \mid\mid C$ should also be deadlock free.

This phenomenon is quite well known in concurrency theory and the solution is to use a failure semantics instead of a trace semantics. Intuitively, the idea is to complete the description of a trace ending in a deadlock by a description of the actions that the process can refuse. In other terms, if the environment offers the actions of the refusal set then the environment in parallel with the considered agent is still in a deadlock state. In the above example, this would allow to distinguish the reason of the deadlock of A and B: the former deadlocks because of the absence of the token a on the shared space whereas the latter deadlocks because of the absence of the token b on the shared space. As the deadlocks have different explanations, the failure semantics of A and B would be different and, consequently, one would not have that A refines B.

However, although intuitive, failures have at least three drawbacks which prevent us from a direct transposition in our time coordination setting and, more generally, for Linda-like languages.

First, traditional algebras such as CCS and CSP are based on the synchronous communication of events. In this context, it is reasonable to build refusal sets from actions which cannot be made by the agent under consideration even if they are offered by the environment. However, in the $\mathcal{L}$ context and more generally for Linda-like languages, what matters is not so much which actions are made but more importantly the current contents of the shared dataspace. However, in contrast to actions of which many can be offered by the environment, only one contents of the shared space is of interest, namely the minimal one which allows computations to be continued. This has lead us in previous work to consider reactive sequences to build fully abstract semantics of Linda-like languages (see [6,7]) and, basically amounts to abandoning the idea of failure sets.

Second, the nature of the choice operator in the synchronous setting of CCS and CSP defines the failure sets of $A + B$ as the union of the failure sets of A and B. Rephrased in our setting, this would lead to consider that $ask_1(a)$ failure-refines $ask_1(a) + tell_1(b)$. However, $ask_1(a)$ waits for one unit of time before deadlocking whereas $ask_1(a) + tell_1(b)$ has no deadlock.

Third, as failures are traditionally defined in an untimed setting, there is no provision for time. One could think of using timed failure sets defined in [27] but, in view of the above two points, we prefer to define directly our notion of refinement. To that end, we first rephrase the transition system of figure 1 in terms of events and characterize the deadlocks of $\mathcal{L}$ agents.

4 Event Based Semantics

A first set of events corresponds to the consultations and modifications of the store. Accordingly, the addition of a token t with duration d to the store is denoted by the event t^+_d, a check of the presence of the token t is denoted by the event t°, and check of its absence, by $t^\bullet$. Moreover, the removal of an occurrence of t out of the store corresponds to the event t^-.

The second kind of event corresponds to an internal step of the agent, without interaction with the environment. We will denote such an event τ. Finally, the last event corresponds to the tick of the clock. It is denoted by ν.

Definition 10. *Define Sevent as the set* $\{t^+_d, t^\circ, t^\bullet, t^- \;:\; t \in \mathit{Stoken},\ d \in \mathit{Stime}\} \cup \{\tau, \nu\}$*. Moreover, let* $T \subseteq \mathit{Stoken}$ *be a set of tokens. Define* $\mathit{events}(T)$ *as the set* $\{t^+_d, t^\circ, t^\bullet, t^- \;:\; t \in T,\ d \in \mathit{Stime}\}$

We associate with an agent the set of the events it can be responsible for.

Definition 11. *Given an agent A, we define*

$$A^+ = \{t \;:\; \mathit{tell}_d(t) \in \mathcal{F}(A), d > 0\} \qquad A^- = \{t \;:\; \mathit{get}_d(t) \in \mathcal{F}(A), d > 0\}$$
$$A^\circ = \{t \;:\; \mathit{ask}_d(t) \in \mathcal{F}(A), d > 0\} \qquad A^\bullet = \{t \;:\; \mathit{nask}_d(t) \in \mathcal{F}(A), d > 0\}$$

These sets are useful to characterize which agent may fail on some stores. An agent fails if it is not able to do any internal step, nor to compute a primitive, nor to do a temporal step. This is the case of an agent A for which $\mathcal{F}(A)$ contains only obsolete primitives (with 0 as duration) or *ask*, *get*, *nask* primitives on a store that does not allow any of them to fire. Note that if an *ask* and a *nask* primitive in $\mathcal{F}(A)$ have the same token as argument (i.e. $A^\circ \cap A^\bullet$ is not empty) the agent A is computable whatever the store is. Similarly if $A^- \cap A^\bullet$ is not empty, the agent is always able to compute one primitive whatever the store is.

Proposition 1. *Let A be an agent. There exists a store on which it cannot fire any computational transition iff the following three conditions hold: (i) $A^+ = \emptyset$, (ii) $delay(0) \notin \mathcal{F}(A)$ and (iii) $(A^\circ \cup A^-) \cap A^\bullet = \emptyset$.*

Proof. An induction on the syntactic structure of A establishes that for any agent A, store σ and time u, there exist an agent B and a store ρ satisfying $\langle A \mid \sigma\rangle_u \to \langle B \mid \rho\rangle_u$ if and only if one of the five following cases occurs.

1. there exist $t \in Stoken$, $d > 0$ such that $tell_d(t) \in \mathcal{F}(A)$
2. there exist $t \in Stoken$, $d > 0$ such that $ask_d(t) \in \mathcal{F}(A)$ and $t \in \sigma^*$
3. there exist $t \in Stoken$, $d > 0$ such that $get_d(t) \in \mathcal{F}(A)$ and $t \in \sigma^*$
4. there exist $t \in Stoken$, $d > 0$ such that $nask_d(t) \in \mathcal{F}(A)$ and $t \notin \sigma^*$
5. $delay(0) \in \mathcal{F}(A)$

where σ^* denotes the multisets of the tokens occurring in σ without their subscript duration. Conversely, A is unable to fire a computational step on the store σ at time u if and only if all these conditions are falsified. Therefore, for a given agent A, there exist a store σ and a time u satisfying $\langle A \mid \sigma\rangle_u \not\to$ if and only if it is possible to provide a store σ and a time u satisfying the five following conditions: (1) $A^+ = \emptyset$, (2) $A^\circ \cap \sigma^* = \emptyset$, (3) $A^- \cap \sigma^* = \emptyset$, (4) $A^\bullet \subseteq \sigma^*$, (5) $delay(0) \notin \mathcal{F}(A)$. Conditions (1) and (5) are directly satisfied by an agent A failing on a store. As $\mathcal{F}(A)$ is finite, conditions (2) to (4) express a finite set of conditions which is always satisfiable but in case a token has both to be and not to be in the store, i.e. if $(A^\circ \cup A^-) \cap A^\bullet \neq \emptyset$.

Definition 12. *Let A be an agent. We denote by $A \downarrow$ the existence of a store on which A fails.*

Proposition 2. *Let A be an agent. It cannot fire any transition on any store iff $A \downarrow$ and $A \not\gg$*

Proof. On the one hand, the condition $A \downarrow$ occurs if and only if $\mathcal{F}(A)$ only contains *ask*, *get* and *nask* primitives. On the other hand, the condition $A \not\gg$ occurs if and only if none of the *ask*, *get*, *nask* or *delay* primitives in $\mathcal{F}(A)$ have a positive duration. The conjunction of the two conditions occurs then if and only if $\mathcal{F}(A)$ involves only ask_0, get_0 and $nask_0$ primitives, and therefore, if and only if it is unable to fire any transition on any store.

The transition system of figure 1 can be rephrased as a transition system where computation steps are reformulated as their corresponding events. For instance,

RULES FOR PRIMITIVES

$$(\mathbf{T}) \quad \frac{d > 0}{tell_d(t) \xrightarrow{t_d^+} E}$$

$$(\mathbf{A}) \quad \frac{d > 0}{ask_d(t) \xrightarrow{t^\circ} E}$$

$$(\mathbf{N}) \quad \frac{d > 0}{nask_d(t) \xrightarrow{t^\bullet} E}$$

$$(\mathbf{G}) \quad \frac{d > 0}{get_d(t) \xrightarrow{t^-} E}$$

$$(\mathbf{D}) \quad delay(0) \xrightarrow{\tau} E$$

COMPOSITIONAL RULES

$$(\mathbf{S}) \quad \frac{A \xrightarrow{a} A', a \neq \nu}{A \; ; \; B \xrightarrow{a} A' \; ; \; B}$$

$$(\mathbf{P}) \quad \frac{A \xrightarrow{a} A', a \neq \nu}{\begin{array}{c} A \,||\, B \xrightarrow{a} A' \,||\, B \\ B \,||\, A \xrightarrow{a} B \,||\, A' \end{array}}$$

$$(\mathbf{C}) \quad \frac{A \xrightarrow{a} A', a \neq \nu}{\begin{array}{c} A \, + \, B \xrightarrow{a} A' \\ B \, + \, A \xrightarrow{a} A' \end{array}}$$

$$(\mathbf{R}) \quad \frac{g_X \xrightarrow{a} A', a \neq \nu}{X \xrightarrow{a} A'}$$

TEMPORAL RULE

$$(\mathbf{Ti}) \quad \frac{A \neq E, A \gg, A \downarrow}{A \xrightarrow{\nu} A^-}$$

Fig. 2. Tagged transition system for $\mathcal{L}$

the label t_d^+ is used to indicate the addition of the token t with duration d by the computation of a $tell_d(t)$ primitive. The resulting tagged transition system is described in figure 2.

A first relation between the two transition systems is easy to establish.

Proposition 3. *Let A be an agent of $\mathcal{L}$, $A \xrightarrow{\nu}$ if and only if there are a store σ and a time u such that $\langle A \mid \sigma \rangle_u \not\rightsquigarrow$.*

The computation of an agent may be defined as a sequence of events. As for the semantics $\mathcal{O}_h$, a computation may be infinite or finite and, in that latter case, terminated by the symbol δ^+ to denote a successful termination or by the symbol δ^- to denote a deadlock computation.

The ability to actually perform a sequence of events depends on the contents of the store. For instance, a t° event may only occur on stores containing the token t. Conversely a $t^\bullet$ event may only occur on stores that contain no occurrence of t. Similarly, event ν can only occur on stores on which the agent is blocked. However, as described until now, the temporal event ν does not contain enough information to decide whether it can be fired on a given store. From now on, we associate with a temporal step ν two sets F and G in order to indicate respectively which tokens have to be present and absent from the store in order to allow a transition to take place. The careful reader will directly notice the extension made with respect to the failure set semantics discussed in the previous section.

Definition 13. *Define the set of the tagged temporal events Stevent as follows*

$$Stevent = \{t^+_d, t^\circ, t^\bullet, t^- \ : \ t \in Stoken\} \cup \{\tau\} \cup \{\nu^F_G : F, G \subseteq Stoken\}$$

A functional interpretation of the events allows to relate them with respect to the corresponding effects on stores.

Definition 14. *For any event e of Sevent we define a partial function f_e : $Ststore \times Stime \nrightarrow Ststore \times Stime$ as follows: for any store σ, any time u*

$$\begin{aligned}
f_{t^+_d}(\sigma_u) &= (\sigma \cup \{t_d\})_u && \\
f_{t^\circ}((\sigma \cup \{t_d\})_u) &= (\sigma \cup \{t_d\})_u && \\
f_{t^\bullet}(\sigma_u) &= \sigma_u && \textit{if there is no } d \textit{ such that } t_d \in \sigma \\
f_{t^-}(\sigma \cup \{t_d\})_u &= \sigma_u && \\
f_\tau(\sigma_u) &= \sigma_u && \\
f_{\nu^F_G}(\sigma_u) &= \sigma^-_{u+1} && \textit{if } G \subseteq \sigma^* \textit{ and } F \cap \sigma^* = \emptyset
\end{aligned}$$

where σ^* denotes the set of the tokens occurring in σ without their subscript duration.

The following lemma already establishes a link between the two transition systems.

Lemma 1. *For any agents A and B, any stores σ and ρ and any time u, one has*

1. *$\langle A \mid \sigma\rangle_u \rightarrow \langle B \mid \rho\rangle_u$ iff there is $e \in Sevent \setminus \{\nu\}$ such that $A \xrightarrow{e} B$ and $\rho_u = f_e(\sigma_u)$*
2. *$\langle A \mid \sigma\rangle_u \not\twoheadrightarrow$ iff for all $e \in Sevent \setminus \{\nu\}$ the fact that $A \xrightarrow{e}$ implies that $f_e(\sigma_u)$ is undefined*
3. *$\langle A \mid \sigma\rangle_u \rightsquigarrow$ iff for all $e \in Sevent \setminus \{\nu\}$ the fact that $A \xrightarrow{e}$ implies that $f_e(\sigma_u)$ is undefined and that $A \xrightarrow{\nu}$*
4. *$\langle A \mid \sigma\rangle_u \not\mapsto$ iff there is no $e \in Sevent$ such that $A \xrightarrow{e}$*

Proof. The proof of the first property is obtained by induction on the syntax of agents, first for guarded ones and then on arbitrary ones. The second and third properties result directly from the first property. Finally, the fourth property occurs if and only if $\mathcal{F}(A)$ contains only ask_0, get_0 and $nask_0$ primitives, i.e. iff $A \neq E$, $A \downarrow$ and $A \not\gg$.

5 Refinement

5.1 Auxiliary Notions

We are now in a position to define our notion of refinement. To make our theory more general, we introduce a hiding operator Δ_T whose purpose is to hide some tokens as local details, simply observed as τ steps. The rules to be added to the tagged transition system of figure 2 are given in figure 3. Some auxiliary notations will also be needed.

RULES

$$(\mathbf{\Delta_1}) \quad \frac{A \xrightarrow{a} A', a \in events(T) \cup \{\nu\}}{\Delta_T A \xrightarrow{a} \Delta_T A'}$$

$$(\mathbf{\Delta_2}) \quad \frac{A \xrightarrow{a} A', a \notin events(T)}{\Delta_T A \xrightarrow{\tau} \Delta_T A'}$$

Fig. 3. Tagged transition system for $\Delta_T A$

Definition 15. *For any agent A and A' and any event a we denote*

1. $A \Rightarrow A'$ *whenever there are agents A_i for $i = 1, \ldots, n$ such that* $A \xrightarrow{\tau} A_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} A_n \xrightarrow{\tau} A'$
2. $A \Rightarrow^a A'$ *whenever there are agents A_i for $i = 1, \ldots, n$ such that* $A \xrightarrow{\tau} A_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} A_n \xrightarrow{a} A'$
3. $A \Rightarrow^\nu A'$ *whenever there are agents A_i for $i = 1, \ldots, n$ such that* $A \xrightarrow{\tau} A_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} A_n \xrightarrow{\nu} A'$
4. $A \Rightarrow^\omega$ *whenever there is an infinite sequence of agents A_i for $i \in \mathbb{N}$ such that* $A \xrightarrow{\tau} A_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} A_i \xrightarrow{\tau} \ldots$

Similarly, for any agents A and A', any set of token T and any event a we denote

1. $\Delta_T A \Rightarrow \Delta_T A'$ *whenever there are agents A_i for $i = 1, \ldots, n$ such that* $\Delta_T A \xrightarrow{\tau} \Delta_T A_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} \Delta_T A_n \xrightarrow{\tau} \Delta_T A'$
2. $\Delta_T A \Rightarrow^a \Delta_T A'$ *whenever there are agents A_i for $i = 1, \ldots, n$ such that* $\Delta_T A \xrightarrow{\tau} \Delta_T A_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} \Delta_T A_n \xrightarrow{a} \Delta_T A'$
3. $\Delta_T A \Rightarrow^\nu \Delta_T A'$ *whenever there are agents A_i for $i = 1, \ldots, n$ such that* $\Delta_T A \xrightarrow{\tau} \Delta_T A_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} \Delta_T A_n \xrightarrow{\nu} \Delta_T A'$
4. $\Delta_T A \Rightarrow^\omega$ *whenever there is an infinite sequence of agents A_i for $i \in \mathbb{N}$ such that* $\Delta_T A \xrightarrow{\tau} \Delta_T A_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} \Delta_T A_i \xrightarrow{\tau} \ldots$

5.2 Refinement

A refinement relation is defined as follows.

Definition 16. *A relation R on $\mathcal{L}^e \times \mathcal{P}(Stoken) \times \mathcal{L}^e$ is a* refinement relation *iff any triple (I, T, S) in R satisfies the following properties*

1. $tokens(S) \subseteq T$
2. *for any event a and agent I' such that $\Delta_T I \Rightarrow^a \Delta_T I'$ there is some S' in $\mathcal{L}^e$ such that $S \Rightarrow^a S'$ with $(I', T, S') \in R$.*
3. *for any agent I' such that $\Delta_T I \Rightarrow^\nu \Delta_T I'$ there is some S' such that $S \Rightarrow^\nu S'$ with $(I', T, S') \in R$.*
4. $\Delta_T I \Rightarrow^\omega$ *implies* $S \Rightarrow^\omega$.

5. $\Delta_T I \Rightarrow \Delta_T I' \not\rightarrow$ implies $S \Rightarrow S' \not\rightarrow$, for some agents I' and S'
6. $\Delta_T I \Rightarrow \Delta_T E$ implies $S \Rightarrow E$

where the transition rules for $\Delta_T A$ are provided by figure 3.

In this definition, it is worth observing that an interaction of the implementation I with the store has to be a possible interaction of the specification S with the store. Similarly, temporal events of the implementation have to be temporal events made by the specification. But these are not the only required properties. The ability of actions by the specification has to be preserved. This is expressed in conditions 6, 5 and 4 associated with condition 2, by the fact that termination, failure or zeno behavior of the implementation are acceptable only if they respectively correspond to termination, failure and zeno behavior of the specification.

Example 3. To illustrate the above definition, let us reconsider the producer *Prod* of example 1:

$$Prod = tell_2(i)\ ;\ delay(5)\ ;\ Prod$$

Assume that the production of item i is actually more complex and consists in producing two parts, say ii and iii, from which i is obtained by a special composition. Focussing on the interactions only, we may code this behavior by replacing the action $tell_2(i)$ by an agent Aux defined as follows:

$$\begin{aligned} Aux &= (Prod\_ii \mid\mid Prod\_iii)\ ;\ Composer \\ Prod\_ii &= tell_1(ii) \\ Prod\_iii &= tell_1(iii) \\ Composer &= (get_1(ii) \mid\mid get_1(iii))\ ;\ tell_2(i) \end{aligned}$$

Call $Prod\_aux$ the agent $Prod$ obtained by replacing $tell_2(i)$ by Aux, namely

$$Prod\_aux = Aux\ ;\ delay(5)\ ;\ Prod\_aux$$

Obviously, $Prod$ and $Prod\_aux$ are not equivalent since, for instance, the computation of $Prod\_aux$ makes the item ii appear and not that of $Prod$. However, if we regard the production and consumption of the items ii and iii as details, we may say that $Prod\_aux$ is a more complex version of $Prod$. Technically, $Prod\_aux$ is said to refine $Prod$. It is also worth observing from the example that a strict respect of the timing has to be preserved. For instance, defining $Composer$ as

$$Composer = (get_1(ii) \mid\mid get_1(iii))\ ;\ delay(1)\ ;\ tell_2(i)$$

would not be correct since then $Cons$ in parallel with $Prod\_aux$ would miss the item i and would not be able to consume it at all (in contrast to what happened in example 1).

As in traditional concurrency theory, many refinement relations may be defined so that, as usual, we shall focus now on the maximal one defined as the union of all the refinement relations.

Definition 17. *The refinement relation $\trianglelefteq$ is the union of all the refinement relations on $\mathcal{L}^e \times \mathcal{P}(Stoken) \times \mathcal{L}^e$. Moreover, for the ease of reading, we write $A \trianglelefteq_T S$ whenever the triple (A, T, S) is in $\trianglelefteq$.*

5.3 Properties

Proposition 4. *For any agents A,B,C of $\mathcal{L}^e$ and any T set of tokens,*

1. *if $tokens(A) \subseteq T$, then $A \trianglelefteq_T A$*
2. *if $tokens(A) \subseteq T$, then $A \trianglelefteq_T A\ +\ A$*
3. *if $tokens(A) \subseteq T$, then $A\ +\ A \trianglelefteq_T A$*
4. *if $tokens(B) \cup tokens(C) \subseteq T$, if $A \trianglelefteq_T B$ and if $B \trianglelefteq_T C$ then $A \trianglelefteq_T C$*
5. *if $A \Rightarrow B$ and if $A \trianglelefteq_T S$ then $B \trianglelefteq_T S$.*

Proof. The first property is obtained by observing that the relation R on $\mathcal{L}^e \times \mathcal{P}(Stoken) \times \mathcal{L}^e$ defined as follows $R = \{(A, T, A) : tokens(A) \subseteq T\}$ is a refinement relation. It is then included in $\trianglelefteq$.

Properties 2 to 5 are direct consequences of property 1, by observing that inclusion of the set of tokens and the five required properties are satisfied.

Definition 18. *An agent A of $\mathcal{L}$ is said to be deadlock free iff its operational semantics $\mathcal{O}_h$ does not contain any failing computations.*

An important property is that an agent refining a deadlock-free specification is also deadlock-free.

Theorem 1 (deadlock-freeness preservation). *For any agents A and S of $\mathcal{L}$ such that $A \trianglelefteq_T S$, if S is deadlock-free, then A is also deadlock-free.*

Proof. The proof is established by contradiction. Let s be a deadlocking history of $\mathcal{O}_h(A)$. This history can be written $s = \sigma^1_{u_1}.\cdots.\sigma^n_{u_n}.\delta^-$ for some stores $\sigma^i (i = 1, \cdots, n)$ and times $u_i (i = 1, \cdots, n)$ with $\sigma^1_{u_1} = \emptyset_1$. One then has, for a sequence of agents $A_i (i = 1, \cdots, n)$ of $\mathcal{L}^e$ with $A_1 = A$,

$$\langle A_1 \mid \sigma_1 \rangle_{u_1} \mapsto \cdots \mapsto \langle A_n \mid \sigma_n \rangle_{u_n} \not\mapsto .$$

Proposition 3 and lemma 1 provide then a sequence of events $e_i (i = 1, \cdots, n-1)$ of *Sevent* such that

$$A_i \xrightarrow{e_i} A_{i+1} \text{ and } \sigma^{i+1}_{u_{i+1}} = f_{e_i}(\sigma^i_{u_i}) \text{ for } i = 1, \cdots, n-1.$$

Moreover, thanks to lemma 1, as $\langle A_n \mid \sigma_n \rangle_{u_n} \not\mapsto$, the agent A_n is not able to fire any tagged transition. According to the transition system of figure 3, one has then for any $i = 1, \cdots, n-1$, $\Delta_T A_i \xrightarrow{e_i} \Delta_T A_{i+1}$ or $\Delta_T A_i \xrightarrow{\nu} \Delta_T A_{i+1}$.

Let us now define the subsequence of the events preserved by the filter, i.e. those events in $T \cup \{\nu\}$. Formally, it is provided by the subsequence $e_{j_k}(k = 1, \cdots, m)$ such that $j_0 = 1$ and $\Delta_T A_{j_k} \Rightarrow^{e_{j_{k+1}}} \Delta_T A_{j_{k+1}}$.

The definition of the refinement provides a sequence of agents $S_k(k=1, \cdots, m)$ with $S_1 = S$, such that $S_k \Rightarrow^{e_{j_{k+1}}} S_{k+1}$. Moreover, as $\Delta_T A_{j_m} \Rightarrow \Delta_T A_n \not\rightarrow$, there is some S_{m+1} such that $S_m \Rightarrow S_{m+1} \not\rightarrow$. As $f_\tau(\rho) = \rho$ for any store ρ, one has, thanks to proposition 3 and lemma 1, for some agents $S_k^1, \ldots, S_k^{l(k)}$,

$$\langle S_k \mid \rho_k \rangle_{u_k} \rightarrow \langle S_k^1 \mid \rho_k \rangle_{u_k} \rightarrow \cdots \rightarrow \langle S_k^{l(k)} \mid \rho_k \rangle_{u_k} \mapsto \langle S_{k+1} \mid \rho_{k+1} \rangle_{u_{k+1}}$$

with $\rho_k = \sigma_{j_k} \cap T$ for $k = 1, \cdots, m$ and

$$\langle S_m \mid \rho_m \rangle_{u_m} \rightarrow \langle S_m^1 \mid \rho_m \rangle_{u_m} \rightarrow \cdots \rightarrow \langle S_{m+1} \mid \rho_m \rangle_{u_m} \not\mapsto$$

which concludes the proof.

The following lemma will help to establish the substitutability property.

Lemma 2. *For any agents A, B and S of $\mathcal{L}$ and for any set of tokens T such that $A \trianglelefteq_T S$ and $tokens(A) \cap tokens(B) \subseteq T$, one has*

1. $A \ ; \ B \trianglelefteq_{T \cup tokens(B)} S \ ; \ B$ *and* $B \ ; \ A \trianglelefteq_{T \cup tokens(B)} B \ ; \ S$
2. $A \ + \ B \trianglelefteq_{T \cup tokens(B)} S \ + \ B$ *and* $B \ + \ A \trianglelefteq_{T \cup tokens(B)} B \ + \ S$
3. $A \mid\mid B \trianglelefteq_{T \cup tokens(B)} S \mid\mid B$ *and* $B \mid\mid A \trianglelefteq_{T \cup tokens(B)} B \mid\mid S$

Proof. The proofs being similar we only establish here that for $A \ ; \ B$. Consider the relation R_{aux} defined as

$$\{((A \ ; \ B), T', (S \ ; \ B)) : T' = T \cup tokens(B), tokens(A) \cap tokens(B) \subseteq T, A \trianglelefteq_T S\}$$

and let us establish that $R = \trianglelefteq \cup R_{aux}$ is a refinement relation, in which case, as it includes $\trianglelefteq$, the relation R is the $\trianglelefteq$ relation itself.

Let $(A \ ; \ B, T', S \ ; \ B)$ be an element of R_{aux}, and T be a set of token such that $T \cup tokens(B) = T'$, $tokens(A) \cap tokens(B) \subseteq T$ and $A \trianglelefteq_T S$. Let us examine each of the six properties required in definition 16

1. The inclusion of the sets of tokens is direct from the definition of R_{aux}.
2. Assume $\Delta_{T'}(A \ ; \ B) \Rightarrow^a \Delta_{T'}(C)$. According to the transition system of figure 3, three cases are possible.
   case 1. $C = A' \ ; \ B$ with $\Delta_{T'}(A) \Rightarrow^a \Delta_{T'}(A')$ and $A' \neq E$. In this case, as $tokens(A) \cap T' \subseteq T$, one also has $\Delta_T(A) \Rightarrow^a \Delta_T(A')$. As $A \trianglelefteq_T S$, there is then some S' such that $S \Rightarrow^a S'$ and $A' \trianglelefteq_T S'$. Therefore, according to the definition of R the triple $(A' \ ; \ B, T', S' \ ; \ B)$ is also in R. The conclusion then follows from the fact that $S \ ; \ B \Rightarrow^a S' \ ; \ B$.
   case 2. $C = B$ with $\Delta_{T'}(A) \Rightarrow^a \Delta_{T'}(E)$. In this case, one similarly has $\Delta_T(A) \Rightarrow^a \Delta_T(E)$. As $A \trianglelefteq_T S$, one has then $S \Rightarrow^a S'$ and $S' \Rightarrow E$. As $B \trianglelefteq_{T'} B$, it is then direct that $B \trianglelefteq_{T'} S' \ ; \ B$ and $(B, T', S' \ ; \ B)$ is in R.
   case 3. $C = B'$ with $\Delta_{T'}(A) \Rightarrow \Delta_{T'}(E)$ and $\Delta_{T'}(B) \Rightarrow^a \Delta_{T'}(B')$. In this case, one also has $\Delta_T(A) \Rightarrow \Delta_T(E)$. As $A \trianglelefteq_T S$, one has $S \Rightarrow E$. Moreover, as $tokens(B) \subseteq T'$ one has $B \Rightarrow^a B'$. Therefore one has $S \ ; \ B \Rightarrow^a B'$ with (B', T', B') in R which suffices.

3. The case $\Delta_{T'}(A \ ; \ B) \Rightarrow^{\nu} \Delta_{T'}(C)$ is treated exactly as $\Delta_{T'}(A \ ; \ B) \Rightarrow^{a} \Delta_{T'}(C)$.
4. Assume $\Delta_{T'}(A \ ; \ B) \Rightarrow^{\omega}$. Two cases are to be distinguished.
case 1. $\Delta_{T'}(A) \Rightarrow^{\omega}$. In this case, one also has $\Delta_T(A) \Rightarrow^{\omega}$ and then $S \Rightarrow^{\omega}$. Therefore one has $S \ ; \ B \Rightarrow^{\omega}$ which suffices.
case 2. $\Delta_{T'}(A) \Rightarrow \Delta_{T'}(E)$ and $\Delta_{T'}(B) \Rightarrow^{\omega}$. In this case, one also has $\Delta_T(A) \Rightarrow \Delta_T E$ and then $S \Rightarrow E$. Moreover, as $tokens(B) \subseteq T'$, one has $B \Rightarrow^{\omega}$. Therefore one has $S \ ; \ B \Rightarrow^{\omega}$ which suffices.
5. Assume $\Delta_{T'}(A \ ; \ B) \Rightarrow \Delta_{T'}(C) \not\rightarrow$. Two cases are to be distinguished.
case 1. $C = A' \ ; \ B$ and $\Delta_{T'}(A) \Rightarrow \Delta_{T'}(A') \not\rightarrow$. In this case, one also has $\Delta_T(A) \Rightarrow \Delta_T(A') \not\rightarrow$ and then $S \Rightarrow S' \not\rightarrow$. Therefore one has $S \ ; \ B \Rightarrow S' \ ; \ B \not\rightarrow$ which suffices.
case 2. $\Delta_{T'}(A) \Rightarrow \Delta_{T'}(E)$ and $\Delta_{T'}(B) \Rightarrow \Delta_{T'}(C) \not\rightarrow$. In this case, one also has $\Delta_T(A) \Rightarrow \Delta_T(E)$ and then $S \Rightarrow E$. Moreover, as $tokens(B) \subseteq T'$, one has $B \Rightarrow C \not\rightarrow$. Therefore one has $S \ ; \ B \Rightarrow C \not\rightarrow$ which suffices.
6. Finally, assume $\Delta_{T'}(A \ ; \ B) \Rightarrow \Delta_{T'}(E)$. This situation occurs only if $\Delta_{T'}(A) \Rightarrow \Delta_{T'}(E)$ and $\Delta_{T'}(B) \Rightarrow \Delta_{T'}(E)$. In this case, on the one hand, as $tokens(A) \cap T \subseteq T$, one also has $\Delta_T(A) \Rightarrow \Delta_T(E)$ and $S \Rightarrow E$. On the other hand, as $tokens(B) \subseteq T'$, one has $B \Rightarrow E$. Therefore $S \ ; \ B \Rightarrow E$ which suffices.

Definition 19. *Let $\square$ be a fresh symbol. Define the set of* contexts *$Scontext$ by the following rule where A represents an agent.*

$$C ::= \square \mid A \mid C \ ; \ A \mid A \ ; \ C \mid C \parallel A \mid A \parallel C \mid C + A \mid A + C$$

The application of a context C to an agent B is defined as the new agent obtained by replacing the place holder $\square$ in C, if any, by B. This is subsequently denoted as $C[B]$.

Proposition 5. *Let S be a specification and A an agent refining S with respect to T. For any context $C[.]$ such that $tokens(C) \cap tokens(A) \subseteq T$, then $C[A]$ refines $C[S]$ with respect to $T \cup tokens(C)$.*

Proof. The proposition is established by using lemma 2 and a simple induction on the structure of the context.

Theorem 2. *Let S be a specification and A an agent refining S with respect to T. For any context $C[.]$ such that $tokens(C) \cap tokens(A) \subseteq T$, if $C[S]$ is deadlock free then $C[A]$ is also deadlock free.*

Proof. For any context $C[.]$ such that $tokens(C) \cap tokens(A) \subseteq T$, proposition 5 ensures that $C[A] \trianglelefteq_{T \cup tokens(C)} C[S]$. Proposition 1 then ensures that if $C[S]$ is deadlock free, then $C[A]$ is also deadlock free.

6 Conclusion

Building upon previous work on timed coordination languages, this paper has presented a novel notion of refinement for these languages which satisfies the substitutability property: if the implementation I refines the specification S and if $C[S]$ is deadlock free, for some context C, then $C[I]$ is also deadlock free.

To our best knowledge, the article [28] is the only piece of work which has developed a refinement theory in the context of coordination languages. However, this work takes the complementary perspective of using a first order temporal logic to write specifications and of employing an axiomatic semantics to derive properties. Moreover, it uses a Prolog-like rule format for manipulating tuples. As appreciated by the reader, our work is based on an algebraic perspective. Accordingly, specifications and implementations are agents of the same language, and are related thanks to an abstraction operator and a refinement relation. Moreover, another family of coordination models, featuring Linda-like primitives, traditional concurrent operators and time, are tackled.

Refinements have been studied for classical concurrent languages. We have shown that trace refinement, underlying among others the B method ([2]), is not suited for our purposes. Moreover, refinement based on failure sets, classically used for process algebras such as CCS ([23]) and CSP ([16]), is also not adapted to our coordination context. We have thus refined the notion of refusal sets by replacing actions by tokens to be present or absent from the shared dataspace and have imposed restrictions on temporal transitions. The resulting refinement relation has then been shown to be adequate to obtain the substitutability property.

Because this property has a compositional flavor, it is expected that it will help to scale model checking. Our future research will aim to contribute to this area by building a tool similar to FDR ([26]) dedicated to our timed coordination languages.

Acknowledgments

We would like to thank the University of Namur for its support as well as the anonymous reviewers for their helpful comments.

References

1. Abadi, M., de Alfaro, L. (eds.): CONCUR 2005. LNCS, vol. 3653. Springer, Heidelberg (2005)
2. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
3. Baier, C., Hermanns, H. (eds.): CONCUR 2006. LNCS, vol. 4137. Springer, Heidelberg (2006)
4. Ball, T., Jones, R.B. (eds.): CAV 2006. LNCS, vol. 4144. Springer, Heidelberg (2006)
5. Berry, G., Gonthier, G.: The Esterel Synchronous Programming Language: Design, Semantics, Implementation. Science of Computer Programming, vol. 19 (1992)
6. Brogi, A., Jacquet, J.-M.: Modeling Coordination via Asynchronous Communication. In: Garlan, D., Le Métayer, D. (eds.) COORDINATION 1997. LNCS, vol. 1282, pp. 238–255. Springer, Heidelberg (1997)

7. Brogi, A., Jacquet, J.-M., Linden, I.: Fully Abstract Semantics for a Coordination Model with Asynchronous Communication and Enhanced Matching. Fundamenta Informaticae 73(4), 431–478 (2006)
8. Carriero, N., Gelernter, D.: Linda in Context. Communications of the ACM 32(4), 444–458 (1989)
9. Caspi, P., Halbwachs, N., Pilaud, P., Plaice, J.: Lustre: a Declarative Language for Programming Synchronous Systems. In: Proc. POPL'87, ACM Press, New York (1987)
10. Cleaveland, R., Parrow, J., Steffen, B.: The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. ACM Transactions on Programming Languages and Systems 15(1), 36–72 (1993)
11. de Bakker, J., de Vink, E.: Control Flow Semantics. MIT Press, Cambridge (1996)
12. Etessami, K., Rajamani, S.K. (eds.): CAV 2005. LNCS, vol. 3576. Springer, Heidelberg (2005)
13. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces: Principles, Patterns, and Practice. Addison-Wesley, London (1999)
14. Gelernter, D., Carriero, N.: Coordination Languages and Their Significance. Communications of the ACM 35(2), 97–107 (1992)
15. Harel, D.: Statecharts: a Visual Formalism for Complex Systems. Science of Computer Programming, vol. 8 (1987)
16. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
17. Jacquet, J.-M., De Bosschere, K., Brogi, A.: On Timed Coordination Languages. In: Porto, A., Roman, G.-C. (eds.) COORDINATION 2000. LNCS, vol. 1906, pp. 81–98. Springer, Heidelberg (2000)
18. Johansson, K.H., Egerstedt, M., Lygeros, J., Sastry, S.: On the Regularisation of Zeno Hybrid Automata. System and Control Letters 38, 141–150 (1999)
19. Linden, I., Jacquet, J.-M.: On the Expressiveness of Absolute-Time Coordination Languages. In: De Nicola, R., Ferrari, G.L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 232–247. Springer, Heidelberg (2004)
20. Linden, I., Jacquet, J.-M., De Bosschere, K., Brogi, A.: On the Expressiveness of Relative-Timed Coordination Models. Electronical Notes in Theoretical Computer Science 97, 125–153 (2004)
21. Linden, I., Jacquet, J.-M., De Bosschere, K., Brogi, A.: On the Expressiveness of Timed Coordination Models. Science of Computer Programming 61(2), 152–187 (2006)
22. Milner, R.: A Calculus of Communication Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
23. Milner, R.: Communicating and Mobile Systems: the Pi-Calculus. Cambridge University Press, Cambridge (1999)
24. Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R. (eds.): Coordination of Internet Agents: Models, Technologies, and Applications. Springer, Heidelberg (2001)
25. Papadopolous, G.A., Arbab, F.: Coordination Models and Languages. Advances in Computers, vol. 48 (1998)
26. Roscoe, A.W.: Theory and Practice of Concurrency. Prentice-Hall, Englewood Cliffs (1997)
27. Schneider, S.: Concurrent and Real-time Systems: the CSP Approach. John Wiley & Sons, Chichester (2000)
28. Semini, L., Montangero, C.: A Refinement Calculus for Tuple Spaces. Science of Computer Programming 34(2), 79–140 (1999)
29. Wyckoff, P.: T Spaces. IBM Systems Journal, vol. 37(3) (1998)

A Calculus for Mobile Ad Hoc Networks
(Extended Abstract)

Jens Chr. Godskesen*

IT University of Copenhagen
Rued Langgaards Vej 7
DK-2300 Copenhagen S, Denmark
jcg@itu.dk

Abstract. We suggest a *Calculus for Mobile Ad Hoc Networks*, CMAN. A node in a network is a process equipped with a location, it may communicate with other nodes using *synchronous spatially oriented broadcast* where only the current neighbors receive the message. Nodes may autonomously change their neighbor relationship and thereby change the network topology. We define a natural reduction semantics and a reduction congruence as well as a labeled transition semantics and prove a weak contextual bisimulation to be a *sound* and *complete* co-inductive characterization of the reduction congruence. Finally, we apply CMAN on a small example of a cryptographic routing protocol.

1 Introduction

The use of wireless networks is becoming more and more important due to the increasing and widespread use of communicating mobile devices. The application area for wireless networks is broad, spanning from ambient intelligence, wireless local area networks, sensor networks, and cellular networks for mobile telephony.

Our work is devoted to a particular kind of wireless networks, the so called *Mobile Ad Hoc Networks* (MANETS). MANETS are self organizing wireless networks without centralized access points or any other central control components. Hence they do not contain a pre-deployed infrastructure for routing messages. An ad hoc network may be formed when a collection of mobile nodes join together and agree on how to route messages for each other over possibly multiple hops.

The communication primitive for wireless devices is message broadcast. However in contrast to the conventional technology in wired local area networks, say the Ethernet, where broadcasted messages reach every node in the network, then for wireless networks broadcast is *spatially oriented* meaning that messages will only reach those nodes within the communication range (the cell) of the emitting node. Another difference between wired and wireless network technology is that interference is a much harder and severe problem in wireless systems. Also, in wireless networks communication links between entities cannot always be considered bidirectional.

* Supported by grant no. 272-05-0258 from the Danish Research Agency.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 132–150, 2007.

Calculi for broadcast systems were first studied by Prasad in the work on the CBS calculus [16] and later in a mobile setting by Ene and Muntean in the $b\pi$ calculus [5], and by Ostrovsky, Prasad, and Taha in HOBS [15]. Recently wireless broadcast systems have been studied by Nanz and Hankin in CBS# [13] and by Merro in CMN [8]. In the former calculi broadcast scope is *transitive* in that if two nodes P and Q both can communicate with a third node then P and Q can also communicate with each other whereas this is not necessarily the case for CBS# and CMN. The calculus CWS [9] by Mezzetti and Sangiorgi also studies wireless broadcast but at a much lower level of abstraction, in particular they take the phenomenon of interference into account.

Another characteristic of MANETS is that nodes may be mobile, not only do they enter and leave the network, but also they autonomously change localities and thereby change their connections and hence the topology of the network. Mobility of processes has been addressed by many calculi, like π [11], Mobile Ambients [3], Seal [4], and Homer [7], and some even take the notion of spatially oriented communication into account like Mobile Ambients and Hennessy and Riely's $D\pi$ [17]. However only very limited work has so far been devoted to calculi for broadcast and mobility, like $b\pi$ and HOBS, and to our knowledge the only reported work on calculi for spatially oriented broadcast and mobility is CBS# and CMN.

The goal of our work is to define a *Calculus for Mobile Ad Hoc Networks* (CMAN) that facilitates mobility and spatially oriented broadcast. As in CMN we adopt that communication between nodes in a network is carried out on bidirectional links, and further we assume that nodes in a network may move arbitrarily as in both CBS# and CMN. We shall refrain from dealing with interference in this paper.

The neighborhood relation in CBS# is dealt with at the semantic level, the semantics is parameterized and quantified over a set of configurations (graphs). In CMN and CWS the neighborhood relation is taken care of by a metric function that tells if two physical locations are close enough to communicate. Here instead we choose *logical locations* and follow to some extent the ideas by De Nicola et al. [14] letting the topology be explicitly part of the network syntax and letting the topology change as a consequence of computational steps. We choose as a key design principle of our calculus that the specification of a node's control behaviour must be independent of and not intermixed with its neighborhood coordination as this would render models in the calculus unnecessarily complex.

We follow the approach from CBS# and CMN (and CWS) letting broadcast be spatially oriented, but in contrast to CBS#, where broadcast messages may be received after a change of network topology, we let broadcast as in CMN be *atomic* in the sense that all neighbors at the time of the broadcast, and only those, can listen to and receive the broadcasted message. Another similarity with CMN is that we allow broadcasted messages to be lost for some potential recipients. However, opposite to CMN where broadcast is carried out on channels that may be restricted, we let broadcasted messages be transmitted on an unrestricted medium.

One important factor of motivation is that we want to be able to model cryptographic routing protocols for MANETS, like ARAN [18]. For that reason we choose to adopt a data (term) language as the one known from the Applied π-Calculus [2].

A *node*, $\lfloor p \rfloor_l^\sigma$, in our calculus is modeled as a (sequential) *process* p located at some (logical) *location* l and connected to other nodes at locations σ. A location is an abstract name that cannot be referred by the node's process. Nodes put together in parallel constitute a *network*, say

$$P = \lfloor p \rfloor_l^m \parallel \lfloor q \rfloor_m^l \parallel \lfloor r \rfloor_n \ ,$$

where the current topology is that the node at location l, $\lfloor p \rfloor_l^m$, is connected to the node at location m, $\lfloor q \rfloor_m^l$, (and vice versa). The node at location n is disconnected from any other node. Mobility is obtained by a simple reduction, say that the node at location n autonomously moves and becomes (bidirectionally) *connected* to the node at location l,

$$\lfloor p \rfloor_l^m \parallel \lfloor q \rfloor_m^l \parallel \lfloor r \rfloor_n \searrow \lfloor p \rfloor_l^{mn} \parallel \lfloor q \rfloor_m^l \parallel \lfloor r \rfloor_n^l \ . \tag{1}$$

Similarly, nodes may arbitrarily *disconnect*, say

$$\lfloor p \rfloor_l^{mn} \parallel \lfloor q \rfloor_m^l \parallel \lfloor r \rfloor_n^l \searrow \lfloor p \rfloor_l^n \parallel \lfloor q \rfloor_m \parallel \lfloor r \rfloor_n^l \ . \tag{2}$$

A node containing a process $\langle t \rangle.p$ may broadcast t and a node with $(x).q$ can receive a broadcasted message. *Synchronous spatially oriented broadcast* is realized by a *broadcast reduction* labelled by the location of the emitting node, say

$$\lfloor \langle t \rangle.p \rfloor_l^{nm} \parallel \lfloor (x).q \rfloor_m^l \parallel \lfloor (x).r \rfloor_n^l \searrow_l \lfloor p \rfloor_l^{nm} \parallel \lfloor q\{t/x\} \rfloor_m^l \parallel \lfloor r\{t/x\} \rfloor_n^l \ , \tag{3}$$

where the node at location l broadcasts to all nodes to which it is connected in the current topology, or similarly

$$\lfloor \langle t \rangle.p \rfloor_l^{nm} \parallel \lfloor (x).q \rfloor_m^l \parallel \lfloor (x).r \rfloor_n^l \searrow_l \lfloor p \rfloor_l^{nm} \parallel \lfloor (x).q \rfloor_m^l \parallel \lfloor r\{t/x\} \rfloor_n^l \ , \tag{4}$$

where the broadcasted message to one of l's neighbors, in this case the node at location m, is lost. As a special case, a disconnected node in a network may broadcast without anyone listening

$$\lfloor \langle t \rangle.p \rfloor_l \parallel \lfloor (x).q \rfloor_m \parallel \lfloor (x).r \rfloor_n \searrow_l \lfloor p \rfloor_l \parallel \lfloor (x).q \rfloor_m \parallel \lfloor (x).r \rfloor_n \ . \tag{5}$$

A novel contribution of our work is that we choose to work with a family of broadcast reductions, one for each locality in the network. This allows an external observer to observe the locality (node) in charge of the synchronous broadcast.

However, since it may be unrealistic for an observer to cover the whole network we introduce the notion of a *hidden node*, i.e. a node with the location name restricted. A hidden node, say $\nu k.\lfloor \langle t \rangle.r \rfloor_k$, may connect to other nodes extruding its location name,

$$\lfloor (x).p \rfloor_l^m \parallel \lfloor q \rfloor_m^l \parallel \nu k.\lfloor \langle t \rangle.r \rfloor_k \searrow \nu k.(\lfloor (x).p \rfloor_l^{mk} \parallel \lfloor q \rfloor_m^l \parallel \lfloor \langle t \rangle.r \rfloor_k^l) \ , \tag{6}$$

and subsequently send (receive) messages to (from) its neighbors, e.g.

$$\nu k.(\lfloor (x).p \rfloor_l^{mk} \parallel \lfloor q \rfloor_m^l \parallel \lfloor \langle t \rangle .r \rfloor_k^l) \searrow \nu k.(\lfloor p\{t/x\} \rfloor_l^{mk} \parallel \lfloor q \rfloor_m^l \parallel \lfloor r \rfloor_k^l) \ , \tag{7}$$

but the emission from a hidden node cannot be observed by an external observer, hence the reduction (7) is not a broadcast reduction.

As in the seminal work on barbed bisimulation [12,10] we strive to have an as simple as possible reduction semantics and to allow an external global observer to have minimal observability, in our case: reductions $\searrow_l$ for broadcast, and reductions $\searrow$ for connections, disconnections, and broadcast from hidden nodes. Similar to the semantics of CMN and CBS# we choose to abstract from observability of node mobility. Indistinguishability under these observations gives rise to a natural equivalence which in turn induces a natural congruence over networks, i.e. the equivalence in all contexts closed under structural congruence. In the present paper we show how to obtain a labeled transition semantics such that (early contextual) weak bisimulation is a *sound* and *complete* co-inductive characterizations of the reduction congruence. Due to lack of space we focus on weak congruence and bisimulation, but the soundness and completeness results carry naturally over to strong congruence and bisimulation.

The paper is organized as follows: The language of CMAN is presented in Section 2. The reduction semantics and the natural reduction congruence follows in Section 3. In Section 4 we provide the labeled transition system semantics and give the co-inductive characterization of the reduction congruence. Then, for a sub-calculus of CMAN, in Section 5 we demonstrate a considerably simpler characterization of the reduction congruence. We end the paper with a simple example of a cryptographic routing protocol and a conclusion.

2 Syntax

As already touched upon above a network in CMAN consists of nodes composed in parallel, some nodes may be hidden, and each node is a sequential process at some abstract location connected to other locations.

Our process definition is similar to the one in [1], a variant of the *Applied π-Calculus* ($A\pi$) [2]. $A\pi$ is a simple extension of the π-Calculus [11] with value passing, primitive functions, and term equations.

Terms. Terms are defined relative to an infinite set of *names* $\mathcal{N}$ ranged over by n, an infinite set of *variables* $\mathcal{X}$ ranged over by x, and two disjoint finite sets, $\mathcal{F}$ and $\mathcal{G}$, of *constructor* and *destructor* symbols ranged over by f and g respectively. Formally, destructors are defined to be partial functions, i.e. the application of a destructor to a tuple of terms is only defined in case the tuple matches one of the destructors defining equations (we refer the reader to [1]).

Then the set of terms is defined as follows:

$$s, t ::= n \mid x \mid f(t_1, \ldots, t_k) \mid (t_1, \ldots, t_i) \ ,$$

where f is a constructor symbol with arity k. We let $\mathcal{T}$ denote the set of all terms with no variables.

Processes. As mentioned above, processes in CMAN are based on the process constructs from $A\pi$. We choose although to omit the notion of a *channel*, letting everyone able to listen be a potential receiver of the broadcasted message.[1] We assume a set of process variables $\mathcal{Z}$ ranged over by z. The set of processes is defined by the grammar:

$$p, q ::= 0 \mid \langle t \rangle.p \mid (x).p \mid \mathit{if}\ (t = s)\ \mathit{then}\ p\ \mathit{else}\ q \mid \mathit{let}\, x = t\ \mathit{in}\ p \mid$$
$$\mathit{let}\, x = g(t_1, \ldots, t_i)\ \mathit{in}\ p\ \mathit{else}\ q \mid \nu n.p \mid z \mid \mathit{rec}\ z.p\ .$$

The process 0 is the inactive process. $\langle t \rangle.p$ may output t and in so doing become p. The process $(x).p$ binds x in p and may input a term t and replace all free occurrences of x in p by t. The process $\mathit{if}\ \ t = s\ \ \mathit{then}\ p\ \mathit{else}\ q$ is a standard conditional. The local definition $\mathit{let}\ x = t\ \mathit{in}\ p$ binds the variable x in p and executes p with all free occurrences of x replaced by t. The process $\mathit{let}\ x = g(t_1, \ldots, t_k)\ \mathit{in}\ p\ \mathit{else}\ q$ also binds x in p, if the destructor application $g(t_1, \ldots, t_k)$ evaluates to a term t then x is bound to t in p, otherwise the process becomes q. The process $\nu n.p$ binds the name n in p and restricts n to p. Finally, $\mathit{rec}\ z.p$ is a recursively defined process where $\mathit{rec}\ z$ binds z in p. [2]

We let $p\{t/x\}$ denote the process p where any free occurrence of x is substituted by t (taking care that names in t are not bound in p by the use of α-conversion if needed). Likewise, $p\{q/z\}$ denotes the process p where z is substituted by q. The set of *free names* in a process p is denoted by $\mathit{fn}(p)$, and its *free variables* are denoted by $\mathit{fv}(p)$. A process p is (variable) *closed* if $\mathit{fv}(p) = \emptyset$. $\mathbf{P}$ denotes the set of all closed processes and as usual we identify processes up to α-equivalence.

Networks. Assume a finite set of *location* names $\mathcal{L}$ ranged over by l and k. We assume $\mathcal{N} \cap \mathcal{L} = \emptyset$ and let m range over $\mathcal{N} \cup \mathcal{L}$. We let σ range over sets of location names and let ϵ denote the empty set. The set of networks is defined by the grammar:

$$P, Q, R ::= 0 \mid \lfloor p \rfloor_l^\sigma \mid \nu m.P \mid P \parallel Q\ .$$

The network 0 denotes the empty network. $\lfloor p \rfloor_l^\sigma$ is a singleton network with the node at location l containing the process p and connected to nodes in σ. $\nu m.P$ is the network P with the (location or term) name m hidden, and finally $P \parallel Q$ is the parallel composition of the two networks P and Q. [3] As a shorthand we allow to write $\Pi_{i \in I} P_i$ for the parallel composition of all networks P_i, $i \in I$.

We let the hiding operator have higher precedence than parallel composition. We write $\lfloor p \rfloor_l$ instead of $\lfloor p \rfloor_l^\epsilon$. When $\tilde{m} = \{m_1, \ldots, m_i\}$ we write $\tilde{m}m$ for $\tilde{m} \cup \{m\}$ and we write $\nu\tilde{m}$ instead of $\nu m_1 \ldots \nu m_i$. We write σl instead of $\sigma \cup \{l\}$ and let $\sigma\sigma'$ denote the union of disjoint sets σ and σ'.

[1] Another approach would be to broadcast on a given channel as in CMN and $b\pi$.

[2] Notice, that in the present version of CMAN we have left out parallel composition and replication of processes.

[3] As in [14] we have no operator for having an unbounded number of network nodes.

The set of *free names* in a network P, denoted by $fn(P)$, is defined as expected and so is the set of *free variables* $fv(P)$. We let $P\{t/x\}$ denote the network P where all free occurrences of x in P is substituted by t (taking care that names in t are not bound in P using α-conversion if needed). The set of *free locations* in a network P, denoted by $fl(P)$, is inductively defined by: $fl(\lfloor p \rfloor_l^\sigma) = \{l\}$, $fl(\nu m.P) = fl(P) \backslash \{m\}$, and $fl(P \parallel Q) = fl(P) \cup fl(Q)$. The set of *free connections* in a network P, denoted by $fc(P)$, is inductively defined by: $fc(\lfloor p \rfloor_l^\sigma) = \sigma$, $fc(\nu m.P) = fc(P) \setminus \{m\}$, and $fc(P \parallel Q) = fc(P) \cup fc(Q)$. Finally, the set of free locations and connections in a network P is denoted by $flc(P) = fl(P) \cup fc(P)$.

As a syntactical convention we allow to write $P_{l \oplus k}$ meaning that the node in P (if any) with location name l is connected to a node with location name k, and symmetrically node k in P (if any) is connected to l. Formally we define $P_{l \oplus k}$ inductively by: $0_{l \oplus k} = 0$, and $(\lfloor p \rfloor_l^\sigma)_{l \oplus k} = \lfloor p \rfloor_l^{\sigma k}$, $(\lfloor p \rfloor_k^\sigma)_{l \oplus k} = \lfloor p \rfloor_l^{\sigma l}$, and $(\lfloor p \rfloor_m^\sigma)_{l \oplus k} = \lfloor p \rfloor_m^\sigma$ if $m \notin \{l, k\}$, $(P \parallel P')_{l \oplus k} = P_{l \oplus k} \parallel P'_{l \oplus k}$, $(\nu m.P)_{l \oplus k} = \nu m.(P_{l \oplus k})$ if $m \notin \{l, k\}$. Similarly, we let $P_{l \ominus k}$ denote the network where k is not connected to node l, and vice versa. We let $l \oplus k$ and $l \ominus k$ have higher precedence than the hiding operator.

Well-formedness. We say that a network P is *well-formed* if each node in P is not connected to itself and if each free location in P is unique. Formally, well-formedness is inductively defined by:

- $\lfloor p \rfloor_l^\sigma$ is well-formed if $l \notin \sigma$.
- $P \parallel Q$ is well-formed if P and Q are well-formed and if $fl(P) \cap fl(Q) = \emptyset$.
- $\nu m.P$ is well-formed if P is well-formed.

In the sequel we consider only the set of well-formed networks and we identify networks up to *alpha*-equivalence. The set of well-formed and variable closed networks is denoted by $\mathbf{N}$.

3 Reduction Semantics

We provide our calculus with a reduction semantics defined through the use of evaluation contexts, structural congruence, and reduction rules.

As usual we say that a binary relation $\mathcal{R}$ on $\mathbf{P}$ is a *congruence* if $p \mathrel{\mathcal{R}} q$ implies $C(p) \mathrel{\mathcal{R}} C(q)$ for any process context C. Structural congruence on $\mathbf{P}$, $\equiv_{\mathbf{P}}$, is the least congruence and equivalence relation that is closed under α-conversion and the rules in Table 1. Likewise, we say that a binary relation $\mathcal{R}$ on $\mathbf{N}$ is a *congruence* if $P \mathrel{\mathcal{R}} P'$ implies $\nu m.P \mathrel{\mathcal{R}} \nu m.P'$ for all m, and $P \parallel Q \mathrel{\mathcal{R}} P' \parallel Q$ for all Q with $fl(Q) \cap (fl(P) \cup fl(P')) = \emptyset$. Structural congruence on $\mathbf{N}$, $\equiv$, is the least congruence and equivalence relation that is closed under α-conversion and the rules in Table 2.

Reduction Rules. We define a reduction $\searrow_l \subseteq \mathbf{N} \times \mathbf{N}$ for each $l \in \mathcal{L}$ as the least relation closed under structural congruence, parallel composition, and satisfying the rules in Table 3. Also, we define $\searrow \subseteq \mathbf{N} \times \mathbf{N}$ as the least relation

Table 1. Structural congruence, processes

$$
\begin{array}{c}
\mathit{let}\ x = t\ \mathit{in}\ p \ \equiv_{\mathbf{P}}\ p\{t/x\} \qquad \mathit{if}\ (t = t)\ \mathit{then}\ p\ \mathit{else}\ q \equiv_{\mathbf{P}} p \\
\mathit{if}\ (t = s)\ \mathit{then}\ p\ \mathit{else}\ q \ \equiv_{\mathbf{P}}\ q\ , \quad \text{if } t \neq s \\
\mathit{let}\, x = g(t_1, \ldots, t_i)\ \mathit{in}\ p\ \mathit{else}\ q \ \equiv_{\mathbf{P}}\ p\{t/x\}\ , \quad \text{if } g(t_1, \ldots, t_i) = t \\
\mathit{let}\, x = g(t_1, \ldots, t_i)\ \mathit{in}\ p\ \mathit{else}\ q \ \equiv_{\mathbf{P}}\ q\ , \quad \text{if } g(t_1, \ldots, t_i) \text{ not defined}
\end{array}
$$

Table 2. Structural congruence, networks

$$
\begin{array}{c}
P \parallel 0 \equiv P \quad P \parallel Q \equiv Q \parallel P \quad (P \parallel P') \parallel P'' \equiv P \parallel (P' \parallel P'') \\
\lfloor p \rfloor_l^\sigma \equiv \lfloor q \rfloor_l^\sigma\ , \text{ if } p \equiv_{\mathbf{P}} q \qquad \lfloor \nu n.p \rfloor_l^\sigma \equiv \nu n.\lfloor p \rfloor_l^\sigma \\
\nu m.\nu m'.P \equiv \nu m'.\nu m.P \qquad \nu m.P \parallel Q \equiv \nu m.(P \parallel Q)\ , \text{ if } m \notin \mathit{fn}(Q) \cup \mathit{flc}(Q)
\end{array}
$$

closed under structural congruence, parallel composition, and restriction, and satisfying the rules in Table 3. We let $\searrow^*$ denote the reflexive and transitive closure of $\searrow$.

A reduction due to rule (*con*) in Table 3 signifies that a bidirectional connection within the network has taken place, and likewise a reduction due to (*dis*) means that a disconnection has happened.

A reduction due to rule (*brd*) means that the node at location l synchronously broadcasts a message to neighbors to which it is currently connected and which are capable of listening. Notice that the rule (*brd*) captures that broadcast is an atomic step, hence no node outside the range of the emitting node at the time of transmission can ever receive the broadcasted message. Also note that broadcasted messages may be lost, i.e. not only will neighbors to which l is connected but which are not listening for sure lose the message, but also connected neighbors that are listening are not guaranteed to receive the emitted message as demonstrated by reduction (4) in the Introduction.

Rule (*res*) allows broadcasting from non-hidden localities to be observable, and dually rule (*hide*) makes emission from hidden nodes unobservable. For reduction examples we refer the reader to (1) – (7) in the Introduction.

Reduction Congruence. Based on the reductions above we introduce a natural weak congruence for CMAN. We say that a binary relation $\mathcal{R}$ on $\mathbf{N}$ is *weak reduction closed* if whenever $P \mathrel{\mathcal{R}} Q$ then $P \searrow P'$ implies the existence of some Q' such that $Q \searrow^* Q'$ and $P' \mathrel{\mathcal{R}} Q'$, and $P \searrow_l P'$ implies the existence of some Q' such $Q \searrow^* \searrow_l \searrow^* Q'$ and $P' \mathrel{\mathcal{R}} Q'$.

Definition 1. *A symmetric relation* $\mathcal{R}$ *on* $\mathbf{N}$ *is a* weak reduction bisimulation *if it is weak reduction closed and if* $P \mathrel{\mathcal{R}} Q$ *implies* $\mathit{fl}(P) = \mathit{fl}(Q)$*.*

Our weak reduction bisimulation is an equivalence relation. It abstracts from internal computation and connectivity, however we do not use barbs as for

Table 3. Reduction rules

$$(con)\ \frac{}{\lfloor p \rfloor_l^{\sigma} \parallel \lfloor q \rfloor_k^{\sigma'} \searrow \lfloor p \rfloor_l^{\sigma k} \parallel \lfloor q \rfloor_k^{\sigma' l}} \qquad (dis)\ \frac{}{\lfloor p \rfloor_l^{\sigma k} \parallel \lfloor q \rfloor_k^{\sigma' l} \searrow \lfloor p \rfloor_l^{\sigma} \parallel \lfloor q \rfloor_k^{\sigma'}}$$

$$(brd)\ \frac{}{\lfloor \langle t \rangle.p \rfloor_l^{\sigma\sigma'} \parallel \mathit{\Pi}_{m \in \sigma} \lfloor (x).p_m \rfloor_m^{\sigma_m l} \searrow_l \lfloor p \rfloor_l^{\sigma\sigma'} \parallel \mathit{\Pi}_{m \in \sigma} \lfloor p_m\{t/x\} \rfloor_m^{\sigma_m l}}$$

$$(res)\frac{P \searrow_l P'}{\nu m.P \searrow_l \nu m.P'} \quad m \neq l \qquad (hide)\frac{P \searrow_l P'}{\nu l.P \searrow \nu l.P'}$$

instance in [12], but instead make broadcasting from nodes be observable through reductions of type $\searrow_l$.

Definition 2. *A relation $\mathcal{R}$ on* $\mathbf{N}$ *is a* weak reduction congruence *if it is a weak reduction bisimulation and a congruence.*

We let $\cong$ denote the largest weak reduction congruence.

4 Labeled Transition System Semantics

In order to give an alternative co-inductive characterization of the weak reduction congruence, $\cong$, we provide a labeled transition system semantics of our calculus. We begin with the semantics for plain processes and proceed with the semantics for networks.

Process Semantics. Let the set of *process actions*, $\mathcal{A}_{\mathbf{P}}$, ranged over by λ be defined by:

$$\lambda ::= (t) \mid \nu\tilde{n}\langle t \rangle$$

where $t \in \mathcal{T}$. The action (t) describes that the term t is received by a process and the action $\nu\tilde{n}\langle t \rangle$ denotes the emission of the term t with names in $\tilde{n}$ bound. If $\tilde{n} = \emptyset$ we write $\langle t \rangle$ instead of $\nu\emptyset\langle t \rangle$. When $\lambda = \nu\tilde{n}\langle t \rangle$ we write $\nu n\lambda$ for $\nu\tilde{n}n\langle t \rangle$. We let $fn(\lambda)$ $(bn(\lambda))$ denote the bound (free) names in λ.

The operational semantics for processes is defined as a labeled transition system $(\mathbf{P}, \mathcal{A}_{\mathbf{P}}, \rightarrow)$ where $\rightarrow \subseteq \mathbf{P} \times \mathcal{A}_{\mathbf{P}} \times \mathbf{P}$ is the least set defined by the rules in Table 4 and closed by $\equiv_{\mathbf{P}}$. The rule (*out*) states that the process $\langle t \rangle.p$ can broadcast the term t. (*in*) states that $(x).p$ can receive any term t and let it be substituted for any free occurrence of x in p. The rule (*rec*) is the usual rule for recursion, and (*res*) is the usual rule for restriction. The rule (*open*) takes care of extrusion of restricted names.

Networks Semantics. The set of *network actions* $\mathcal{A}$ ranged over by α is defined by:

$$\alpha ::= \beta \mid \gamma \qquad \beta ::= \bar{l} \mid \bar{l}\sigma\nu\tilde{n}\langle t \rangle \mid l\overline{\sigma}(t) \mid \tau \qquad \gamma ::= l\triangleright \mid \nu l.l\triangleright \mid l \triangleleft k \mid \tau$$

Table 4. Transition Rules, Processes

$$(out)\frac{}{\langle t\rangle.p \xrightarrow{\langle t\rangle} p} \qquad (in)\frac{}{(x).p \xrightarrow{(t)} p\{t/x\}} \qquad (rec)\frac{p\{rec\ z.p/z\} \xrightarrow{\lambda} p'}{rec\ z.p \xrightarrow{\lambda} p'}$$

$$(res)\frac{p \xrightarrow{\lambda} p'}{\nu n.p \xrightarrow{\lambda} \nu n.p'}\quad n \notin fn(\lambda) \cup bn(\lambda) \qquad (open)\frac{p \xrightarrow{\lambda} p'}{\nu n.p \xrightarrow{\nu n\lambda} p'}\quad n \in fn(\lambda)$$

where $t \in \mathcal{T}$. Actions are grouped into broadcast and mobility actions ranged over by β and γ respectively. The action $\overline{l}$ denotes that the node at location l has completed a broadcast computation. The action $\overline{l}\sigma\nu\tilde{n}\langle t\rangle$ is an output action, it means that the node at location l may broadcast the message t with names in $\tilde{n}$ bound to the nodes with locations in σ. The action $l\overline{\sigma}(t)$ is an input action, meaning that t may be received from the node at location l by the nodes with locations in σ. The action $l\triangleright$ ($\nu l.l\triangleright$) means that the (hidden) node at location l may move. Finally, the action $l\triangleleft k$ indicates that the two nodes at locations l and k respectively are disconnecting. As usual τ denotes an internal computation.

For convenience we write $\nu\tilde{m}.l\triangleright$ for $l\triangleright$ if $\tilde{m} = \emptyset$, likewise if $\tilde{m} = \{l\}$ we write $\nu\tilde{m}.l\triangleright$ for $\nu l.l\triangleright$. We let $bn(\alpha)$ ($fn(\alpha)$) denote the bound (free) names in α, and we let $bl(\alpha)$ ($fl(\alpha)$) denote the bound (free) locations in α.

The operational semantics for networks is defined by a labeled transition system $(\mathbf{N}, \mathcal{A}, \rightarrow)$ where $\rightarrow \subseteq \mathbf{N} \times \mathcal{A} \times \mathbf{N}$ is the least relation satisfying the rules in Table 5 and 6, omitting the symmetric counterparts of the three rules (*synch*), (par_1), and (par_2).

The rule (*brd*) in Table 5 states that a node at location l may broadcast its message to any node with location in σ. Rule (*lose*) represents that broadcast messages may be arbitrarily lost, nodes with locations in σ' will not receive the message broadcasted by l. Hence we may have:

$$P_1 = \lfloor\langle n\rangle.p\rfloor_k^{lm} \xrightarrow{\overline{k}lm\langle n\rangle} \lfloor p\rfloor_k^{lm} \quad \text{and} \quad P_1 \xrightarrow{\overline{k}l\langle n\rangle} \lfloor p\rfloor_k^{lm} \ .$$

The two rules (rec_1) and (rec_2) show how broadcasted terms may be received by nodes, e.g. we may have:

$$Q_1 = \lfloor(x).q\rfloor_l^k \parallel \lfloor(x).r\rfloor_m^k \xrightarrow{k\overline{lm}(n)} \lfloor q\{n/x\}\rfloor_l^k \parallel \lfloor r\{n/x\}\rfloor_m^k = Q_2 \ .$$

The actual synchronization between broadcast and reception of messages is shown in (*synch*), for instance:

$$P_1 \parallel Q_1 \xrightarrow{\overline{k}\epsilon\langle n\rangle} \lfloor p\rfloor_k^{lm} \parallel Q_2 \quad \text{and} \quad P_1 \parallel \lfloor(x).r\rfloor_m^k \xrightarrow{\overline{k}l\langle n\rangle} \lfloor p\rfloor_k^{lm} \parallel \lfloor r\{n/x\}\rfloor_m^k \ .$$

The rules ($open_1$) and (*close*) make sure that extrusion of bound names is treated properly, where (*close*) signals the completion of a broadcasting session. As an

Table 5. Transition Rules, Network Broadcast

$$(brd)\frac{p \xrightarrow{\nu\tilde{n}\langle t\rangle} p'}{\lfloor p\rfloor_l^{\sigma} \xrightarrow{\overline{l}\sigma\nu\tilde{n}\langle t\rangle} \lfloor p'\rfloor_l^{\sigma}} \qquad (lose)\frac{P \xrightarrow{\overline{l}\sigma\sigma'\nu\tilde{n}\langle t\rangle} P'}{P \xrightarrow{\overline{l}\sigma\nu\tilde{n}\langle t\rangle} P'} \qquad (rec_1)\frac{p \xrightarrow{(t)} p'}{\lfloor p\rfloor_l^{\sigma k} \xrightarrow{k\overline{l}(t)} \lfloor p'\rfloor_l^{\sigma}}$$

$$(rec_2)\frac{P \xrightarrow{l\overline{\sigma}(t)} P' \quad Q \xrightarrow{l\overline{\sigma'}(t)} Q'}{P \parallel Q \xrightarrow{l\overline{\sigma\sigma'}(t)} P' \parallel Q'} \qquad (close)\frac{P \xrightarrow{\overline{l}\epsilon\nu\tilde{n}\langle t\rangle} P'}{P \xrightarrow{\overline{l}} \nu\tilde{n}.P'}$$

$$(hide)\frac{P \xrightarrow{\overline{l}} P'}{\nu l.P \xrightarrow{\tau} \nu l.P'} \qquad (open_1)\frac{P \xrightarrow{\overline{l}\sigma\nu\tilde{n}\langle t\rangle} P'}{\nu n.P \xrightarrow{\overline{l}\sigma\nu\tilde{n}n\langle t\rangle} P'} \quad n \in fn(t) \setminus \tilde{n}$$

$$(synch)\frac{P \xrightarrow{\overline{l}\sigma\sigma'\nu\tilde{n}\langle t\rangle} P' \qquad Q \xrightarrow{l\overline{\sigma'}(t)} Q'}{P \parallel Q \xrightarrow{\overline{l}\sigma\nu\tilde{n}\langle t\rangle} P' \parallel Q'} \quad \tilde{n} \cap fn(Q) = \sigma \cap fl(Q) = \emptyset$$

$$(par_1)\frac{P \xrightarrow{\beta} P'}{P \parallel Q \xrightarrow{\beta} P' \parallel Q} \quad fl(\beta) \cap fl(Q) = bn(\beta) \cap fn(Q) = \emptyset$$

$$(res_1)\frac{P \xrightarrow{\beta} P'}{\nu m.P \xrightarrow{\beta} \nu m.P'} \quad m \notin fl(\beta) \cup fn(\beta) \cup bn(\beta)$$

example of the result of an application of rule ($open_1$) we may take:

$$P_1{}' = \nu n.P_1 \xrightarrow{\overline{k}lm\nu n\langle n\rangle} \lfloor p\rfloor_k^{lm} \ ,$$

and assuming $n \notin fn(Q_1)$, taking care to avoid name clashing in the (*synch*) rule, we may apply the rule (*close*) to obtain:

$$P_1{}' \parallel Q_1 \xrightarrow{\overline{k}} \nu n.(\lfloor p\rfloor_k^{lm} \parallel Q_2) \ .$$

The rule (par_1) is a standard rule for concurrency, say:

$$Q_1 \xrightarrow{k\overline{l}(n)} \lfloor q\{n/x\}\rfloor_l^k \parallel \lfloor (x).r\rfloor_m^k \ ,$$

and beyond taking care to avoid name clash it implies for instance:

$$P_1 \parallel \lfloor (x).r\rfloor_m^k \not\xrightarrow{\overline{k}lm\langle n\rangle} \ ,$$

because m is a free location in both $\overline{k}lm\langle n\rangle$ and $\lfloor (x).r\rfloor_m^k$, hence the side condition in (par_1) enforces networks not to externally broadcast messages to nodes it already contains. Likewise, (*synch*) enforces:

$$P_1 \parallel Q_1 \not\xrightarrow{\overline{k}m\langle n\rangle} \ ,$$

because $m \in fl(Q_1)$.

Table 6. Transition Rules, Network Mobility

$$(\mathit{con}_1)\frac{}{\lfloor p\rfloor_l^{\sigma} \xrightarrow{l\rhd} \lfloor p\rfloor_l^{\sigma}} \qquad (\mathit{dis}_1)\frac{}{\lfloor p\rfloor_l^{\sigma k} \xrightarrow{l\lhd k} \lfloor p\rfloor_l^{\sigma}}$$

$$(\mathit{con}_2)\frac{P \xrightarrow{\nu\tilde{m}.l\rhd} P' \qquad Q \xrightarrow{\nu\tilde{m}'.k\rhd} Q'}{P \parallel Q \xrightarrow{\tau} \nu\tilde{m}'\tilde{m}.(P' \parallel Q')_{l\oplus k}} \quad \tilde{m}\cap\tilde{m}' = \tilde{m}\cap \mathit{flc}(Q) = \tilde{m}'\cap \mathit{flc}(P) = \emptyset$$

$$(\mathit{dis}_2)\frac{P \xrightarrow{l\lhd k} P' \qquad Q \xrightarrow{k\lhd l} Q'}{P \parallel Q \xrightarrow{\tau} P' \parallel Q'}$$

$$(\mathit{open}_2)\frac{P \xrightarrow{l\rhd} P'}{\nu l.P \xrightarrow{\nu l.l\rhd} P'} \qquad (\mathit{res}_2)\frac{P \xrightarrow{\gamma} P'}{\nu m.P \xrightarrow{\gamma} \nu m.P'} \quad m \notin \mathit{fl}(\gamma)\cup \mathit{bl}(\gamma)$$

$$(\mathit{par}_2)\frac{P \xrightarrow{\gamma} P'}{P \parallel Q \xrightarrow{\gamma} P' \parallel Q} \quad \mathit{bl}(\gamma)\cap \mathit{flc}(Q) = \emptyset$$

The rule (res_1) is defined as usual, but (hide) is a new special rule added for the same reason as the rule with the same name in Table 3, i.e. to hide broadcast from hidden nodes. Hence for instance:

$$\nu k.({P_1}' \parallel Q_1) \xrightarrow{\tau} \nu k.\nu n.(\lfloor p\rfloor_k^{lm} \parallel Q_2) \ ,$$

is the result of letting the hidden node at location k in ${P_1}'$ complete a broadcast communication.

Mobility of nodes is obtained through the rules (con_1) and (dis_1) and their respective synchronization rules (con_2) and (dis_2) in Table 6. The rule (con_1) states that a node at a free location l may connect to any other node as demonstrated by the rule (con_2). As an example:

$$\lfloor p\rfloor_l \xrightarrow{l\rhd} \lfloor p\rfloor_l \ , \quad \lfloor q\rfloor_k \xrightarrow{k\rhd} \lfloor q\rfloor_k \ , \text{ and } \ \lfloor p\rfloor_l \parallel \lfloor q\rfloor_k \xrightarrow{\tau} \lfloor p\rfloor_l^k \parallel \lfloor q\rfloor_k^l \ .$$

Dually, (dis_1) states that a node at location l with a neighbor at location k may disconnect from k and in so doing remove k from the set of connections of the node. The mutual disconnection of bidirectionally connected nodes is taken care of by the rule (dis_2). For instance,

$$\lfloor p\rfloor_l^k \xrightarrow{l\lhd k} \lfloor p\rfloor_l \ , \quad \lfloor q\rfloor_k^l \xrightarrow{k\lhd l} \lfloor q\rfloor_k \ , \text{ and } \ \lfloor p\rfloor_l^k \parallel \lfloor q\rfloor_k^l \xrightarrow{\tau} \lfloor p\rfloor_l \parallel \lfloor q\rfloor_k \ .$$

Special care must be given to hidden nodes. The rules (open_2), (par_2), and (con_2) allow the location names for hidden nodes to be properly extruded, in particular taking care to avoid clashes between bound location names and free locations and connections. As an example, assuming $l \neq k$,

$$\nu l.\lfloor p\rfloor_l \xrightarrow{\nu l.l\rhd} \lfloor p\rfloor_l \ \text{ and } \ \nu l.\lfloor p\rfloor_l \parallel \lfloor q\rfloor_k \xrightarrow{\tau} \nu l.(\lfloor p\rfloor_l^k \parallel \lfloor q\rfloor_k^l) \ .$$

Illustrating the use of (par_2) we may have:

$$\nu l.\lfloor p\rfloor_l \parallel \lfloor q\rfloor_k \xrightarrow{\nu l.l\triangleright} \lfloor p\rfloor_l \parallel \lfloor q\rfloor_k \ ,$$

and from (con_2) we may then get if $m \notin \{l, k\}$,

$$\nu l.\lfloor p\rfloor_l \parallel \lfloor q\rfloor_k \parallel \nu m.\lfloor r\rfloor_m \xrightarrow{\tau} \nu m.\nu l.(\lfloor p\rfloor_l^m \parallel \lfloor q\rfloor_k \parallel \lfloor r\rfloor_m^l) \ .$$

The rule (res_2) is defined as usual.

The close correspondence between the reduction semantics and the labeled transition system semantics is demonstrated by the lemmas below.

Lemma 1. *$P \xrightarrow{\tau}\equiv P'$ iff $P \searrow P'$.*

Lemma 2. *$P \xrightarrow{\overline{l}}\equiv P'$ iff $P \searrow_l P'$.*

Bisimulation Semantics. Below we give a co-induction characterization, a weak bisimulation, of the weak reduction congruence $\cong$. Our characterization follows the contextual style as found in e.g. [4,14].

Let $\overset{\tau}{\Longrightarrow}$ be the reflexive and transitive closure of $\xrightarrow{\tau}$ and define $\overset{\overline{l}}{\Longrightarrow}$ by $\overset{\tau}{\Longrightarrow}\xrightarrow{\overline{l}}\overset{\tau}{\Longrightarrow}$.

Definition 3. *A symmetric relation $\mathcal{R}$ on $\mathbf{N}$ is a* weak bisimulation *if $P \mathcal{R} Q$ implies $fl(P) = fl(Q)$ and for all $p \in \mathbf{P}$,*

1. *if $P \xrightarrow{\tau} P'$ then $\exists Q'.\ Q \overset{\tau}{\Longrightarrow} Q'$ and $P' \mathcal{R} Q'$*
2. *if $P \xrightarrow{\overline{l}} P'$ then $\exists Q'.\ Q \overset{\overline{l}}{\Longrightarrow} Q'$ and $P' \mathcal{R} Q'$*
3. *if $P \xrightarrow{l\overline{\sigma}(t)} P'$ then $\forall \sigma'.\ l \notin \sigma',\ \exists Q'.$*
   $Q \parallel \lfloor\langle t\rangle.p\rfloor_l^{\sigma\sigma'} \overset{\overline{l}}{\Longrightarrow} Q'$ and $P' \parallel \lfloor p\rfloor_l^{\sigma\sigma'} \mathcal{R} Q'$
4. *if $P \xrightarrow{\nu\tilde{m}.l\triangleright} P'$ then $\forall\ k.\ k \notin fl(P) \cup \tilde{m}.\ \forall\sigma.\ \sigma \cap \tilde{m}k = \emptyset.\ \exists Q'.$*
   $Q \parallel \lfloor p\rfloor_k^\sigma \overset{\tau}{\Longrightarrow} Q'$ and $\nu\tilde{m}.(P' \parallel \lfloor p\rfloor_k^\sigma)_{l\oplus k} \mathcal{R} Q'$
5. *if $P \xrightarrow{l\triangleleft k} P'$ then $\forall\sigma.\ k \notin \sigma.\ \exists Q'.$*
   $Q \parallel \lfloor p\rfloor_k^{\sigma l} \overset{\tau}{\Longrightarrow} Q'$ and $P' \parallel \lfloor p\rfloor_k^\sigma \mathcal{R} Q'$

The notion of weak bisimulation in a broadcasting framework as defined by Definition 3 is a key contribution of this paper and deserves some comments. Requirement 1 in the definition is standard. Requirement 2 demands that the completion of a broadcast computation by a node at some visible location l in P must be matched by a completed broadcast communication by a node at the same location l in Q, but probably preceded and followed by internal computations. Requirement 3 states that if nodes at locations σ in P may receive t from a broadcasting node at location l in the environment, then Q composed with any such node may let the node emit t and complete a (weak) broadcast communication with Q, and in so doing Q and the node together become a network that can match the reception of t by the nodes at σ in P. Requirement 4 states that if a (possibly hidden) node in P (bidirectionally) connects to an external node at some fresh location k then Q an the new external node can make a number of internal computations and then match P being connected to the node at location k. Finally, requirement 5 demands that if the node at location l in P is about to disconnect from location k in its environment, then Q in an environment with a single node at location k that is connected to l can make internal computations, and match P and the node at location k together in parallel when the two are disconnected.

Notice that the three latter requirements in Definition 3 are contextual because they are demands on the network execution environment to provide external input of data terms, to connect with new fresh localities, and to disconnect from environmental locations respectively.

Let $\approx$ be the largest weak bisimulation. In the full version of this paper we show that:

Theorem 1. $\approx$ *is a congruence.*

Because $\approx$ is a congruence it's sufficient to establish that $\approx$ is weak reduction closed in order to show $\approx\, \subseteq\, \cong$, this follows from Lemma 1 and 2. Then in order to show $\approx\, =\, \cong$ it just remains showing $\cong$ to be a weak bisimulation.

Theorem 2. $\approx\, =\, \cong$.

Because the establishment of bidirectional connections and deconnections are unobservable in a weak bisimulation semantics the following lemma holds:

Lemma 3. *If* $l, k \in \mathit{fl}(P)$ *then* $P_{l \oplus k} \approx P_{l \ominus k}$.

It is not difficult to show that $\equiv$ is a weak bisimulation, and as an example we may show that the inactive network is weak bisimilar to a hidden node with an inactive process, i.e. $0 \approx \nu k.\lfloor 0 \rfloor_k$, because $\mathcal{R} \cup \mathcal{R}^{-1}$ is a weak bisimulation up to $\equiv$ where

$$\mathcal{R} = \{(\nu\tilde{m}.(0 \parallel P), \nu\tilde{m}k.(\lfloor 0 \rfloor_k \parallel P)_{\sigma \oplus k}) \mid \tilde{m} \cup \sigma \subseteq \mathit{fl}(P),\ k \notin \mathit{fl}(P)\}$$

letting $P_{\sigma \oplus k}$ be defined by $(\ldots(P_{l_1 \oplus k})\ldots)_{l_i \oplus k}$ whenever $\sigma = \{l_1, \ldots, l_i\}$.

5 Connection Closed Networks

The definition of weak bisimulation is contextual and therefore it is hard to prove bisimulation equivalence between networks. For the class of *connection closed* networks however it turns out that our framework becomes significantly simpler.

We say that a network P is connection closed if each node in P is connected only to other nodes within P, i.e. if $\mathit{fc}(P) \subseteq \mathit{fl}(P)$. For instance, all networks in the examples (1) – (7) in the Introduction are connection closed, but $\lfloor 0 \rfloor_l^m$ is not. We let $\mathbf{N}_c$ denote the subset of $\mathbf{N}$ of connection closed networks.

Let a binary relation on $\mathbf{N}_c$ be a *c-congruence* if it is closed by hiding and by (well-formed) parallel composition of networks in $\mathbf{N}_c$, and let structural c-congruence be the least c-congruence and equivalence relation on $\mathbf{N}_c$ closed under α-conversion and the rules in Table 2. As in Definition 2 we define a weak equivalence abstracting from internal computation, but now only over connection closed networks.

Definition 4. *A symmetric relation $\mathcal{R}$ on $\mathbf{N}_c$ is a* weak reduction c-congruence *if it is weak reduction closed, a c-congruence, and if $P \mathrel{\mathcal{R}} Q$ implies $\mathit{fl}(P) = \mathit{fl}(Q)$.*

We let $\cong_c$ be the largest weak reduction c-congruence.

Like weak reduction congruence was characterized by weak bisimulation we may also characterize weak reduction c-congruence by a co-inductively defined bisimulation. Let R_k range over networks in $\mathbf{N}_c$ where $k \in \mathit{fl}(R_k)$.

Definition 5. *A symmetric relation $\mathcal{R}$ on $\mathbf{N}_c$ is a* weak c-bisimulation *if $P \mathrel{\mathcal{R}} Q$ implies $\mathit{fl}(P) = \mathit{fl}(Q)$ and*

$$\textit{if } P \xrightarrow{\tau} P' \textit{ then } \exists Q'.\ Q \overset{\tau}{\Longrightarrow} Q' \textit{ and } P' \mathrel{\mathcal{R}} Q'$$

$$\textit{if } P \xrightarrow{\bar{l}} P' \textit{ then } \exists Q'.\ Q \overset{\bar{l}}{\Longrightarrow} Q' \textit{ and } P' \mathrel{\mathcal{R}} Q'$$

$$\begin{array}{l}\textit{if } P \xrightarrow{\nu\tilde{m}.l\triangleright} P' \textit{ then } \forall R_k \in \mathbf{N}_c.\mathit{fl}(R_k) \cap (\mathit{fl}(P) \cup \tilde{m}) = \emptyset.\ \exists Q'.\\ \qquad Q \parallel R_k \overset{\tau}{\Longrightarrow} Q' \textit{ and } \nu\tilde{m}.(P' \parallel R_k)_{l\oplus k} \mathrel{\mathcal{R}} Q'\end{array}$$

Let $\approx_c$ be the largest weak c-bisimulation. One may show, applying proofs similar to the proofs of Theorem 1 and 2, that $\approx_c$ is a c-congruence and that

Theorem 3. $\cong_c = \approx_c$

As an example, we may then (writing $\langle t \rangle$ for $\langle t \rangle.0$) show $\nu k.\lfloor \langle n \rangle.\langle n \rangle \rfloor_k \approx_c \nu k.\lfloor \langle n \rangle \rfloor_k \parallel \nu l.\lfloor \langle n \rangle \rfloor_l$ because

$$\begin{array}{l}\{\ (\nu\tilde{m}k.(\lfloor \langle n \rangle.\langle n \rangle \rfloor_k \parallel Q)_{\sigma\oplus k}, \nu\tilde{m}kl.(\lfloor \langle n \rangle \rfloor_k \parallel (\lfloor \langle n \rangle \rfloor_l \parallel Q)_{\sigma_1\oplus l})_{\sigma_2\oplus k}),\\ \quad (\nu\tilde{m}k.(\lfloor \langle n \rangle \rfloor_k \parallel Q)_{\sigma\oplus k}, \nu\tilde{m}kl.(\lfloor \langle n \rangle \rfloor_k \parallel (\lfloor 0 \rfloor_l \parallel Q)_{\sigma_1\oplus l})_{\sigma_2\oplus k}),\\ \quad (\nu\tilde{m}k.(\lfloor 0 \rfloor_k \parallel Q)_{\sigma\oplus k}, \nu\tilde{m}kl.(\lfloor 0 \rfloor_k \parallel (\lfloor 0 \rfloor_l \parallel Q)_{\sigma_1\oplus l})_{\sigma_2\oplus k})\\ \quad \mid \sigma \cup \sigma_1 \cup \sigma_2 \subseteq \mathit{fl}(Q)\ \}\end{array}$$

is a weak c-bisimulation up to structural c-congruence.

6 An Example: ARAN

As mentioned in the Introduction a key motivation for our work is to establish a framework that allows to reason about security properties for MANETS. In [6] an attack on the cryptographic routing protocol ARAN [18] was identified and below we recapture the principles of this attack.

The goal of ARAN is to ensure secure requests for routing in ad hoc networks by making requests and replies be signed and checked in every hop, hence messages cannot be altered and therefore the protocol is claimed to be safe in that no false routing information can be imposed by malicious nodes. The basic idea of the protocol is that a receiver of a message is obliged to check its signature and if the message is correctly signed the signature is removed and signed by the node itself before the new message is forwarded. It is assumed that all valid nodes in the network a priori have a private public key pair and a certificate and also that the public key of the certificate authority is known to every node.

In order to illustrate the attack it is sufficient to consider only a network consisting of three nodes: the initiator of a route request, the destination of the request, and an attacker. The attacker is not a valid node and hence it has not been authorized by the certificate authority.

The simplified ARAN protocol we consider goes as follows: The initiator broadcasts a signed request *rdp* to its neighbors and awaits a signed reply *rep* in return, if the reply is successfully returned the initiator broadcasts *success*. Hence the destination must be an immediate neighbor in order for a route to exist. The destination of the route request on the other hand waits for a signed route request, checks that it is properly signed and if so returns a signed reply to the initiator. Upon reception of the reply the initiator validates the signed message.

To model the cryptographic primitives, let $\{ok, pk, sk, sign\}$ be a set of constructor symbols and let $\{check, get\}$ be a set of destructor symbols where ok has arity 0, where get, pk, and sk have arity 1, where $sign$ has arity 2, and where $check$ has arity 3. We let $pk(n)$ be the constructor for a public key based on some seed n, and we let $sk(m)$ be a private (secret) key based on the seed m. The application of the constructor *sign*

$$sign(pk(n), sk(m)) \quad ,$$

then denotes the signing of the public key $pk(n)$ with the secret key $sk(m)$. We let the destructors *check* and *get* be defined by:

$$check(t, sign(t, sk(s)), pk(s)) = ok \quad , \quad get(sign(t, sk(s))) = t \quad .$$

That is, checking the signature of a message t with the public key matching the private key by which the message was signed yields the result ok. The destructor *get* simply returns the contents of a signed message. By convention we introduce two auxiliary destructors, *fst* and *snd*, that returns the first and second element of a pair respectively.

As shorthands for the process expressions, whenever q is 0, we abbreviate $\mathit{if}\ t = s\ \mathit{then}\ p\ \mathit{else}\ q$ by $\mathit{if}\ t = s\ \mathit{then}\ p$, we write $\mathit{let}\ x = g(t_1, \ldots, t_k)\ \mathit{in}\ p$ instead of $\mathit{let}\ x = g(t_1, \ldots, t_k)\ \mathit{in}\ p\ \mathit{else}\ q$, and also, as before we write $\langle t \rangle$ for $\langle t \rangle.q$.

Table 7. ARAN processes

$$
\begin{aligned}
p_0 &\stackrel{\text{def}}{=} \nu n_1.\mathit{let}\ x_{cert} = \mathit{sign}(\mathit{pk}(n_1), \mathit{sk}(n_0))\ \mathit{in}\ p_1\\
p_1 &\stackrel{\text{def}}{=} \mathit{let}\ x_{sreq} = \mathit{sign}(\mathit{rdp}, \mathit{sk}(n_1))\ \mathit{in}\ \langle (x_{sreq}, x_{cert})\rangle.(x).p_2\\
p_2 &\stackrel{\text{def}}{=} \mathit{let}\ x_1 = \mathit{fst}(x)\ \mathit{in}\ \ \mathit{let}\ x_2 = \mathit{snd}(x)\ \mathit{in}\ \ p_3\\
p_3 &\stackrel{\text{def}}{=} \mathit{if}\ \mathit{rep} = \mathit{get}(x_1)\ \mathit{then}\ \mathit{let}\ x_{key} = \mathit{get}(x_2)\ \mathit{in}\ p_4\\
p_4 &\stackrel{\text{def}}{=} \mathit{if}\ \mathit{check}(x_{key}, x_2, \mathit{pk}(n_0)) = \mathit{ok}\ \ \mathit{then}\ p_5\\
p_5 &\stackrel{\text{def}}{=} \mathit{if}\ \mathit{check}(\mathit{rep}, x_1, x_{key}) = \mathit{ok}\ \ \mathit{then}\ \langle \mathit{success}\rangle\\
\\
q_0 &\stackrel{\text{def}}{=} \nu n_3.\mathit{let}\ x_{cert} = \mathit{sign}(\mathit{pk}(n_3), \mathit{sk}(n_0))\ \mathit{in}\ (x).q_1\\
q_1 &\stackrel{\text{def}}{=} \mathit{let}\ x_1 = \mathit{fst}(x)\ \mathit{in}\ \ \mathit{let}\ x_2 = \mathit{snd}(x)\ \mathit{in}\ \ q_2\\
q_2 &\stackrel{\text{def}}{=} \mathit{if}\ \mathit{rdp} = \mathit{get}(x_1)\ \mathit{then}\ \mathit{let}\ x_{key} = \mathit{get}(x_2)\ \mathit{in}\ q_3\\
q_3 &\stackrel{\text{def}}{=} \mathit{if}\ \mathit{check}(x_{key}, x_2, \mathit{pk}(n_0)) = \mathit{ok}\ \mathit{then}\ \langle (\mathit{sign}(\mathit{rep}, \mathit{sk}(n_3)), x_{cert})\rangle
\end{aligned}
$$

The simplified one shot version of the ARAN protocol is defined by:

$$A = \nu n_0.(\lfloor p_0 \rfloor_l \parallel \lfloor q_0 \rfloor_k)\ \ ,$$

where p_0 and q_0 are defined in Table 7. The process p_0 defines the behaviour of the initiator of the protocol, and q_0 defines the behaviour of the destination.

The intruder, which in this example can only relay messages, is defined as a hidden node by:

$$I = \nu m.\lfloor \mathit{rec}\ z.(x)\langle x\rangle.z \rfloor_m\ \ . \tag{8}$$

Observe, that since the intruder is a hidden node broadcasting of messages from I cannot be observed.

A correctness criterion for the ARAN protocol is as stated above that the routing messages must be validated in each and every hop, in that each hop should always be between certified nodes only. For instance, it must not be possible for a non-certified node (an intruder) to be part of a valid route in ARAN. This criterion may be checked by verifying as to whether the protocol is unaffected by running together with an intruder doing relays as defined by (8).

The composition of A and the intruder can do the following computation:

$$A \parallel I \stackrel{\overline{l}}{\Longrightarrow} \nu n_0.\nu n_1.\nu m.(\lfloor (x).p_2 \rfloor_l^m \parallel \lfloor q_0 \rfloor_k \parallel \lfloor \langle t\rangle.\mathit{rec}\ z.(x)\langle x\rangle.z \rfloor_m^l) = P\ \ , \tag{9}$$

where

$$t = (\mathit{sign}(\mathit{rdp}, \mathit{sk}(n_1)), \mathit{sign}(\mathit{pk}(n_1), \mathit{sk}(n_0)))\ \ .$$

We argue $A \not\approx A \parallel I$, and hence demonstrate that the simple version of the ARAN protocol is not robust and therefore subject of attack from an intruder doing relays. A can match the weak output transition (9) above by the four moves:

$$A \xRightarrow{\bar{l}} \nu n_0.\nu n_1.(\lfloor (x).p_2 \rfloor_l \parallel \lfloor q_0 \rfloor_k) = Q \ ,$$

$$A \xRightarrow{\bar{l}} Q_{l\oplus k} \ ,$$

$$A \xRightarrow{\bar{l}} \nu n_0.\nu n_1.(\lfloor (x).p_2 \rfloor_l \parallel \lfloor \nu n_3.\langle t' \rangle \rfloor_k) = Q' \ ,$$

$$A \xRightarrow{\bar{l}} {Q'}_{l\oplus k} \ ,$$

where $t' = (\mathit{sign}(\mathit{rdp}, \mathit{sk}(n_3)), \mathit{sign}(\mathit{pk}(n_3), \mathit{sk}(n_0)))$.

Clearly $P \not\approx Q$ because $P \xRightarrow{\bar{k}}$ which cannot be matched by Q. Notice that the in-equivalence follows due to computations by P where the intruder is part of a route from the initiator to the destination whereas Q is a state where the request has been lost. Because $P \not\approx Q$ also $P \not\approx Q_{l\oplus k}$ due to Lemma 2 since $Q = Q_{l\ominus k}$.

The final part of the proof is due to the fact that the state P where the intruder got the request can be followed by a computation in which the message is lost when the intruder performs a (hidden) broadcast, [4] i.e.

$$P \xrightarrow{\tau} \nu n_0.\nu n_1.(\nu m.(\lfloor (x).p_2 \rfloor_l^m \parallel \lfloor q_0 \rfloor_k \parallel \lfloor \mathit{rec}\ z.(x)\langle x \rangle.z \rfloor_m^l)) = P' \ .$$

Then, since in Q' the destination cannot escape being able to broadcast, because for all $R \in \{R \mid Q' \xRightarrow{\tau} R\} = \{Q', {Q'}_{l\oplus k}\}$ it holds that $R \xrightarrow{\bar{k}}$, and since $P' \not\xRightarrow{\bar{k}}$ it turns out that $P \not\approx Q'$. It then follows from Lemma 2 that also $P \not\approx {Q'}_{l\oplus k}$ because $Q' = {Q'}_{l\ominus k}$.

7 Conclusion

We have defined a broadcasting calculus, CMAN, for MANETS that supports synchronous spatially oriented broadcast and dynamic changes of the network topology. CMAN is equipped with a natural reduction semantics and congruence, and a co-inductive sound and complete bisimulation characterization. The characterization is shown to be particularly simple for connection closed networks. CMAN has been applied on a small example of a cryptographic routing protocol. A major advantage of CMAN is that it permits direct description of features of MANETS that would be hard do describe in classical calculi.

In the future the process language of CMAN should be extended with concurrency, and we consider also extending the network language with a replication like construct that allows to reason about infinitely many (copies of) instances of nodes. Also, it would be of interest to understand how the semantics should be altered to cater for unidirectional communication links.

[4] Alternatively the intruder could disconnect from the initiator and then make the broadcast to an empty set of receivers.

As of now nodes are allowed to move around arbitrarily connecting to any other node, however that freedom may seem to be too liberal for many applications, and hence the mobility capabilities may be restricted in our future work by imposing more structure on the networks.

Finally, a challenging topic would be to continue the work of how to formalize and reason about security properties for MANETS, and in particular to investigate to what extent the current behavioural equivalences are sufficient to cater for more extensive security analysis.

Acknowledgments. Thanks to the anonymous reviewers for valuable comments on earlier versions of this paper.

References

1. Abadi, M., Blanchet, B.: Analyzing Security Protocols with Secrecy Types and Logic Programs. Journal of the ACM 52(1), 102–146 (January 2005)
2. Abadi, M., Fournet, C.: Mobile vales, new names, and secure communication. In: Nielson, H.R. (ed.) 28th ACM Symposium on Principles of Programming Languages, London, UK, pp. 104–115. ACM Press, New York (2001)
3. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Nivat, M. (ed.) ETAPS 1998 and FOSSACS 1998. LNCS, vol. 1378, Springer, Heidelberg (1998)
4. Castagna, G., Vitek, J., Nardelli, F.Z.: The seal calculus. Information and Computation 201(1), 1–51 (2005)
5. Ene, C., Muntean, T.: A broadcast-based calculus for communicating systems. In: 6th International Workshop on formal Methods for Parallel Programming: Theory and Applications, San Francisco (2001)
6. Godskesen, J.Chr.: Formal verification of the ARAN protocol using the applied π-calculus. In: Proceedings of Sixth International IFIP WG 1.7 Workshop on Issuses in the Theory of Security, (WITS), pp. 99–113, Vienna, Austria (March 2006)
7. Godskesen, J.Chr., Hildebrandt, T.: Extending Howe's method to early bisimulations for typed mobile embedded resources with local names. In: Ramanujam, R., Sen, S. (eds.) FSTTCS 2005. LNCS, vol. 3821, pp. 140–151. Springer, Heidelberg (December 2005)
8. Merro, M.: An observational theory for mobile ad hoc networks. Electron. Notes Theor. Comput. Sci. 173, 275–293 (2007)
9. Mezzetti, N., Sangiorgi, D.: Towards a calculus for wireless systems. Electr. Notes Theor. Comput. Sci. 158, 331–353 (2006)
10. Milner, R.: Functions as processes. In: Proceedings of the seventeenth international colloquium on Automata, languages and programming, New York, NY, USA, pp. 167–180. Springer, Heidelberg (1990)
11. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. Journal of Information and Computation 100, 1–77 (September 1992)
12. Milner, R., Sangiorgi, D.: Barbed bisimulation. In: Proceedings ICALP '92, vol. 623, pp. 685–695. Springer-Verlag, Heidelberg (1992)
13. Nanz, S., Hankin, C.: A framework for security analysis of mobile wireless networks. Theor. Comput. Sci. 367(1), 203–227 (2006)
14. De Nicola, R., Gorla, D., Pugliese, R.: Basic observables for a calculus for global computing. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1226–1238. Springer, Heidelberg (2005)

15. Ostrovsky, K., Prasad, K.V.S., Taha, W.: Towards a primitive higher order calculus of broadcasting systems. In: PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming, pp. 2–13. ACM Press, New York (2002)
16. Prasad, K.V.S.: A calculus of broadcasting systems. Sci. Comput. Program 25(2-3), 285–327 (1995)
17. Riely, J., Hennessy, M.: A typed language for distributed mobile processes (extended abstract). In: Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, pp. 378–390 (1998)
18. Sanzgiri, K., LaFlamme, D., Dahill, B., Levine, B.N., Shields, C., Belding-Royer, E.M.: Authenticated routing for ad hoc networks. IEEE Journal on Selected Areas in Communication, special issue on Wireless Ad. hoc Networks 23(3), 598–610 (March 2005)

A Theory of Nested Speculative Execution

Cristian Ţăpuş and Jason Hickey

California Institute of Technology

Abstract. Implementing distributed applications is a challenging task. Developers are confronted with issues like fault-tolerance, efficient synchronization mechanisms, and the correctness of the distributed code. Transactions are a simple and powerful mechanism for establishing fault-tolerance. To allow multiple processes to cooperate in a transaction we relax the isolation property. We call the new abstraction a speculation. This paper introduces a new programming model based on speculative execution. Speculations provide distributed coordinated rollback and enable optimistic execution of synchronization points. We present an operational semantics for nested speculative execution that specifies distributed speculations precisely. We also discuss two approaches to implementing support for speculations.

1 Introduction

Writing distributed applications can be a challenging task. Some of the main problems that one is confronted with when writing distributed code are fault-tolerance, the efficiency of the distributed synchronization mechanisms, and the correctness of the programs.

Transactions are one of the earliest and simplest abstractions for reliable concurrent programming [1]. They provide fault-isolation by guaranteeing the atomicity, the consistency and the durability of the actions performed as part of the transaction. Traditional transactions also provide isol ation, which prevents the independent actions inside of a transaction to be visible to the rest of the world until the transaction either aborts or commits.

In this paper we consider the case where multiple processes may cooperate in a transaction. This requires relaxing the transactional isolation property. We call these transactions with relaxed isolation *speculations*. We introduce them as programming language primitives.

A *speculation* defines a computation that is based on an assumption whose verification occurs in parallel with the computation. For example, upon entering a critical section of a program that is guarded by a shared mutex one process can speculatively assume that no other process executes inside the critical section. It starts executing the critical section code, while it verifies in parallel that the assumption was indeed correct. If the assumption is invalidated the speculation is *aborted* and the process rolls back its state to what it was before starting the speculation. However, if the assumption was correct the speculation is *committed* and the process continues its execution.

We introduce the speculative programming model through three calls that correspond to starting a speculation, aborting and committing it. These calls have the following types associated with them.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 151–170, 2007.

$$\begin{array}{ll}\text{speculate} & : \text{void} \rightarrow \text{int}\\ \text{abort} & : \text{int} \rightarrow \bot\\ \text{commit} & : \text{int} \rightarrow \text{void}\end{array}$$

The speculate function returns an integer, where the integer represents the local identifier of the speculation when the speculate function is initially called, and -1 if the speculation is later rolled-back. The local identifier of the speculation is used by the *abort* and the *commit* calls to identify the speculation (computation) that needs to be rolled back or committed. We call the *commit branch* the execution that follows after an initial *speculate* call until either an *abort* or a *commit* call are encountered. If the speculation is aborted the process is rolled back to the state it had when the speculate call was executed, and the value -1 is returned. In this case, the program may take an alternate execution path, as if it never executed on the *commit branch*. We call the code executed in this case the *abort branch*. Conceptually, the *abort* and the *commit* calls do not return a value. Their main purpose is to guide the flow of the speculative program. Thus, speculations have dynamic scoping, based on when the *abort* or *commit* functions are called.

Also, the speculative execution model presented in this paper has a component that enables distributed rollback actions to occur. A process may disseminate speculative data to other processes through shared objects, making their computation speculative as well. If the initiator of the speculation rolls back it forces the other processes that used its speculative data to roll back as well.

Speculations provide programs written in imperative languages (like C) with an exception mechanism similar to that found in pure functional languages (like Haskell). This semantics is also extended to distributed applications.

The contributions of this paper include:

- the introduction of a new programming model based on speculations,
- the definition of new speculative programming language constructs for distributed applications,
- an overview of the formal specification of the semantics of speculative execution,
- a brief discussion of two implementations, and
- a discussion of possible uses of the speculative execution model in designing tools for verifying and enforcing the correctness of distributed applications.

The rest of the paper is organized as follows. We introduce speculations through examples in Section 2. Next, we provide a formal model for nested speculations in Section 3. We present related work and compare speculations with similar research initiatives in Section 5. We conclude by discussing future directions of this research.

2 Examples

2.1 Improving Performance with Speculations

Consider a total order protocol implemented over a reliable network that might reorder packets. Each message sent in the network is tagged with an identifier containing a sequence number. For the purpose of this example we consider sequence numbers to be

```
read message $M$ from network;   $t_0 \leftarrow time$
while $time < t_0 + T \wedge \exists\, m' \,.\, last(m') < M$ do
    process messages from network
end while
if $\exists\, m' \,.\, last(m') < M$ then
    specid(M) = speculate();   deliver $M$
    if receive message $M'$ s.t. $M' < M$ then
        abort(specid(M))
    if $\forall\, m' \,.\, last(m') > M$ then
        commit(specid(M))
else
    deliver $M$
end if
```

$time$ is current system time
$last(m')$ is last message seen from machine m'
T is the waiting window before speculating

Fig. 1. Algorithm for total-order communication using speculations

real numbers, where gaps in the sequence of messages are inherent. When a message M is received, the message is held in a receive queue until all messages M' with smaller IDs have been delivered to the application, and only then is the message processed. We will further use the terms *received* and *delivered* as follows. A message is received when the message is passed from the network driver to our protocol. A message is delivered to the destination when the protocol makes it available to the application layer that uses our protocol. The decision of whether to deliver message M or not cannot always be taken at the time when M is received. We assume that there is a bound on the time between when the message is sent until it is received. This, together with the assumption that communication is reliable, provides an upper bound on how long a message would be stored in the receive queue.

To optimize performance, our algorithm uses speculations and a sliding window mechanism, similar to the TCP window. Figure 1 gives the pseudo-code for this algorithm. The recipient of a message uses speculations in the following way: if the first message, M, in the receive queue has not been delivered within time T, the recipient enters a new speculation that assumes M may be delivered at this time. Once the recipient enters the speculation, it delivers the message. The speculation can be committed once the recipient has seen messages from every other machine with IDs larger than that of M. If the recipient, however, receives a message, M', with ID smaller than M, then the recipient must abort the speculation and return to the state where M was at the head of the queue and waiting to be delivered. The new message, M' will be put at the head of the queue and the procedure is repeated.

Two possible outcomes of executing this protocol are illustrated in Figure 2. On the left hand side of the image the outcome of a successful speculative message delivery is shown. In this case the protocol waits for a given time T after the receipt of the message before it starts speculating. It speculates that M is deliverable and begins computing based on the value provided by M. This way useful computation is accrued.

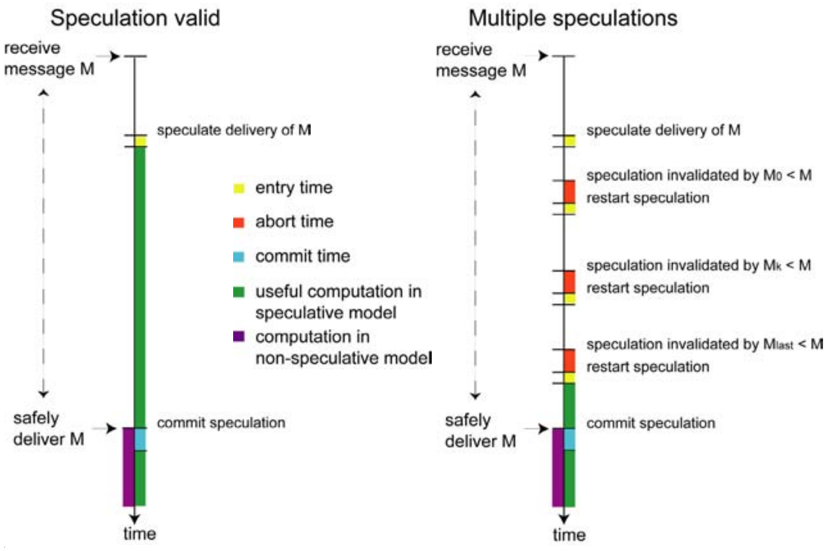


Fig. 2. Two possible speculative executions of the speculative total order protocol

The confirmation that M is deliverable is received later on. In the speculative version the confirmation triggers the commit of the speculation and the earlier started computation continues. In the nonspeculative model, shown on the same graph, the useful computation based on message M is only started upon receiving the confirmation of M's deliverability.

The right hand side of the figure shows the case when the speculation that is based on the deliverability of the last received message is invalidated several times. However, as shown, at some point the message that should be delivered to the application arrives. In this case we fall back to the case shown on the left hand side of the figure, where speculating on the deliverability of that message is successful.

A brief discussion of the benefits of using speculations for the total order protocol example. In this paper we are only going to present an overview of a more complex probabilistic model that captures the benefits of speculative execution. Consider the set of parameters described below that characterize our model.

U the time from the reception of the message until the moment it can be safely delivered for the single speculation case

V the time from the reception of the message until the moment it can be safely delivered for the multiple speculation case

W the time from the reception of the message M until the reception of the last message that should be delivered before M.

T the waiting window size

T_e the time spent entering a speculation

T_a the time spent aborting a speculation

T_c	the time spent committing a speculation
p	the probability that a message could be delivered upon reception
q	the probability that $U > T$
r	the probability that $V > T$

The condition for which the latency of our system is less than that of the classical model is given by the following inequality:

$$pqE[U]_T^\infty + (1-p)r(E[V]_T^\infty - E[W]_T^\infty) \geq pq(T+T_e+T_c) + (1-p)r(T_a+T_e+T_c)$$

Each term on each side of the inequality corresponds to one of the two cases presented in Figure 2, as follows. For the first case, the first element on each side of the inequality presents us with the following intuition: if the safe delivery time for the message is greater than the waiting window size plus the cost of entering a speculation and the cost of commiting it then we gain useful computation if we speculate. For the second case, we consider the second element on each side of the inequality. The intuition is that if the time from when we delivered the last message with an ID less than message M until the safe delivery time for M is greater than the time to abort a speculation (which is the one we invalidated due to that last message with a lower ID than M) plus the time to enter a new speculation and commit it then we gain useful computation.

A detailed mathematical analysis of of this example is presented in [2].

Other applications of speculations to improve performance. The problems that can be optimized using speculations do not reduce to the class of total-order protocols. If a computation is based on a condition that is usually expensive to compute, but that most of the time returns an easy to estimate result, we can use idle computing units to concurrently execute the verification and the computation. The computation would be executed speculatively, assuming a certain return by the verification procedure. This mechanism can increase the overall performance of the system. Smart devices, like intelligent network cards and graphic cards, are becoming a commodity with high computation abilities that is idle most of the time. We envision that verification tasks could be shipped to such devices, while the computation speculatively executes on the main CPU.

2.2 A Distributed Speculative Web Reservation System

Another interesting example considers the interaction between different active entities (processes) in the system while they perform speculative operations. We use a reservation system to illustrate this. A client needs to reserve a plane ticket and a hotel room. She contacts a flight agent for the airfare, and a hotel for lodging. The programs, in pseudo-code, using speculative constructs for the client and the two agents are shown in Figure 3.

Figure 4 shows the speculative dependencies for a successful reservation. The client speculates that the prices will be acceptable and that she will succeed with the reservation (speculation $s1$) and requests quotes from both the flight agent and the hotel. Upon receipt of the requests both the flight agent and the hotel become part of the client's speculation. The hotel successfully processes the request and sends a quote to the client. The flight agent computes a quote based on her local information, which

```
Client
if ($s_c$=speculate() $>$ 0)
    request-hotel
    request-flight
    get-quotes
    if !available then
        $abort(s_c)$
    else if !expensive
        pay for services
        $commit(s_c)$
    else $abort(s_c)$
else
    try different agents
```

```
Flight Agent
receive-request
check-reservations
if ($s_f$=speculate() $>$ 0)
    send price-quote
    check with airline
    if unavailable then
        $abort(s_f)$
    else
        send-confirmation
        get payment
        $commit(s_f)$
else
    do nothing
```

```
Hotel
receive-request
check-availability
if room then
    send price-quote
    get payment
else
    send NO-ROOM
```

Fig. 3. Speculative programs for reservation system

might be inaccurate, since the airline is the ultimate authority for availability. It speculates that the reservation will be confirmed by the airline (speculation $s2$), and sends the quote to the client, pending a final confirmation from the airline. When the client receives the speculative quote from the flight agent her actions become speculative based on the information provided by the agent. Therefore, her computation depends now on both $s1$ and $s2$. Furthermore, when she sends payment to the hotel it absorbs the hotel in the speculation as well. When the airline confirms the reservation the information is forwarded to the client and speculation $s2$ is committed on the flight agent's end. The commit eventually gets propagated to the client and to the hotel. In the meanwhile, the client decides that the operation was successful and commits its speculation as well, which eventually propagates to both agents. The hotel may receive the commit messages in any order, without it having an effect on its correct behavior.

Figure 5 illustrates the behavior of the system in case of an aborted speculation. The first set of events is identical to the previous case. The flight agent runs the commit call because the reservation is successful on the airline's end. After receiving the quotes from the flight agent and the hotel, the client decides the prices are too high and aborts her speculation. This triggers the rollback of all the entities: the client, the hotel and the flight agent. The client continues executing the abort branch which involves contacting another pair of agents (as described in Figure 3).

This example shows the ability of speculations to increase the parallelism and to restore the state of a distributed system to a consistent state upon an optimistic assumption being invalidated. The same type of problem has seen great exposure and has prompted the development of new transactional models in the database community [3].

3 Operational Semantics

Nested speculations, where processes are allowed to start speculations while they are executing inside speculations, are a refinement of the single speculation model [4]. They allow for finer grained speculative execution and provide, in certain cases, rollback to a more recent state than the single speculation model.

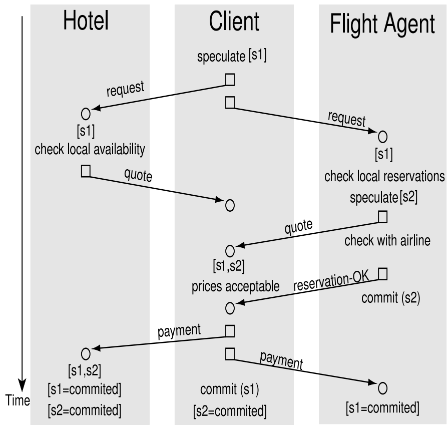


Fig. 4. The reservation succeeds

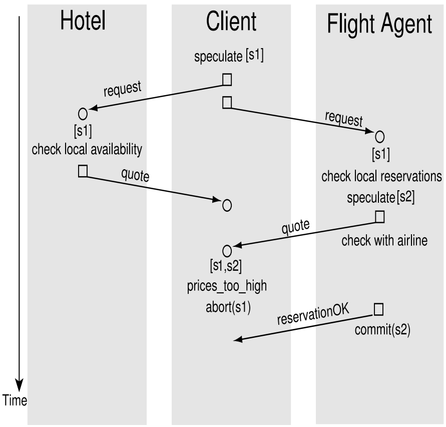


Fig. 5. The reservation fails

Note: In the figures above the send operations are shown as squares, while receives are represented by circles.

We present a speculative distributed objects system model consisting of processes and shared objects. Shared objects store values that can be accessed by any process in the system. They will be the vehicles to propagate speculations in the system.

Processes execute programs and can start speculations by executing the *speculate* call. After a speculation is started, we say that the speculation is *active* until a commit or an abort call is executed. We say that a process is *executing inside* a speculation if the process's program is executed as part of a speculative computation. An object becomes part of a speculation, or is involved in a speculation if a process that is inside a speculation accesses the object. A process is *absorbed* in a speculation if it reads data from an object that belongs to that speculation.

We say a process or an object belongs to a speculation if it started that speculation or if it was absorbed in the speculation.

This section presents the semantics of our speculative primitives in the context of a distributed environment. The semantics is presented as a set of operational rules that capture the transition of the distributed system state when speculative actions occur in the system. We present a subset of the operational semantics rules that define our model.

We believe that presenting the detailed operational semantics of distributed speculative execution is critical in providing other researchers with a clear specification of this useful concept (speculations) in order to facilitate the development of alternate approaches, and aid in their comparison.

3.1 Syntax of the Primitives

The terms of the language that we consider are defined in Figure 6. The base language ($\mathcal{L}$) can be any language that does not posses operations for reading from and writing to shared objects.

Construct	Description
$e ::= \mathcal{L}$	The base language
$speculate(e_1 \oplus e_2)$	Speculate call
$commit(s)$	Commit call
$abort(s)$	Abort call
$let\ v = read(o_j)\ in\ e[v]$	Read the value of object o_j
$write(o_j, x)$	Write value x to object o_j
$e\ ;\ e$	Sequencing

Fig. 6. Syntactically valid terms

The speculative construct $speculate(e_1 \oplus e_2)$ defines a speculation. In the speculative mode, the program executes e_1. If it executes an $abort(s)$, the speculation s is aborted and the process rolls back and executes e_2. If a $commit(s)$ is encountered in e_1, the process will never roll back its state to take the e_2 branch. We refer to e_1 as the "commit" branch, and to e_2 as the "abort" branch.

The $let\ v = read(o_j)\ in\ e[v]$ construct assigns the value of shared object o_j to variable v, which is bound in e. The write operation is represented by $write(o_j, x)$. It stores the value x in shared object o_j.

Syntactically valid terms can be sequenced using the "$;$" separator.

3.2 Terminology and Notation

The speculative operational semantics presented in this section uses the notation shown in Table 1. The operational semantics defines a reduction system that operates on states. If a distributed system in state Δ_i reduces in one step to state Δ_i' we write $\Delta_i \Rightarrow \Delta_i'$. The meaning of $\Delta_i \Rightarrow \Delta_i'$ is that the state of one, and only one, process changes as part of the reduction step. Formally, this is written as follows.

In this model the state of the distributed system (Δ) is composed of two entities: the states of the objects in the system Θ, and the states of the processes running in the system Π.

The state of a shared object (O_j) is characterized by the value it stores (V) and by an optional checkpoint list ($\mathcal{K}$). A checkpoint of the object is taken each time the object enters a new speculation. The checkpoint list is ordered, with the most recent checkpoint being listed first. Each checkpoint in the optional checkpoint list stores a dependency list $(\mathcal{D}_{\mathcal{S}})$ and the value (V') the object had before entering the speculation. The dependency list $(\mathcal{D}_{\mathcal{S}})$ stores the identifiers of the speculations the checkpoint depends on. The use of the dependency list will become clear when we present the operational semantics rules. The checkpoint list is empty if the object is not part of any speculation.

The state of a speculative process (p_i) is defined by three components:

- The list of speculations that the process started $(\mathcal{L}_{\mathcal{S}})$.
- An optional checkpoint list, where the most recent checkpoint is listed first (c).
- An environment (Γ), and the instructions of the program it executes (e). The environment, (Γ), contains definitions of the local variables of the process.

Table 1. Notation for speculative processes

$\Delta ::= \Theta \ \ \ddagger \ \ \Pi$	State of a speculative distributed system
$\Theta ::= o_1 : O_1 \ ; \ \ldots \ ; \ o_m : O_m$	Speculative state of objects $o_1, \ldots, o_m$
$\Pi ::= p_1 : P_1 \ldots p_n : P_n$	Speculative state of processes $p_1, \ldots, p_n$
$O_j ::= \Big[\ \mathcal{K} \ \Big\vert V \Big]$	Speculative state of an object
$\mathcal{K} ::= \ \kappa_q \ ; \ \ldots \ ; \ \kappa_1$	Ordered list of checkpoints; most recent first
$\kappa ::= \langle \mathcal{D}_{\mathcal{S}}, V' \rangle$	Checkpoint of a speculative object
$\mathcal{D}_{\mathcal{S}} ::= s \ ; \ \ldots \ ; \ s'$	List of speculation ids
$P_i ::= : \Big[\ \mathcal{L}_{\mathcal{S}} \ \Big\vert \mathcal{C} \ \Big\vert \Gamma \vdash e \Big]$	Speculative state of process p_i
$\mathcal{C} ::= c_r \ ; \ \ldots \ ; \ c_1$	Ordered list of checkpoints; most recent first
$c ::= \langle \mathcal{D}_{\mathcal{S}} \ , \ \Gamma' \ , \ e' \rangle$	Checkpoint of a speculative process
$\mathcal{L}_{\mathcal{S}} ::= s \ ; \ \ldots \ ; \ s'$	List of speculation ids

The checkpoint of a process has itself three components:

- The dependency list ($\mathcal{D}_{\mathcal{S}}$) represents the list of speculations on which the checkpoint depends. The order of speculation identifiers in the list is relevant; the list is ordered with the most recent checkpoint first.
- The local environment of the process (Γ'), at the time the process became part of the speculation.
- The expression to be executed in case of rollback (e'), the "abort" branch of the speculation.

The dependency list saved in a checkpoint ($\mathcal{D}_{\mathcal{S}}$) is defined as an ordered list of speculation identifiers, represented as follows: $\{s_0, s_1, ...s_q, ...s_k\}$. The order of the speculation identifiers in the dependency list gives the order in which speculations were entered by the current process or by other processes or objects whose speculative data was read at some point by the current process. We define the concatenation, or merger, operator ($\overset{\leftrightarrow}{\cup}$) on dependency lists, as follows:

If $\mathcal{D}_{\mathcal{S}} = \{s_{i_0}, s_{i_1}, ...s_{i_p}\}$ and $\mathcal{D}_{\mathcal{S}}{}' = \{s_{j_0}, s_{j_1}, ...s_{j_q}\}$, then

$\mathcal{D}_{\mathcal{S}} \overset{\leftrightarrow}{\cup} \mathcal{D}_{\mathcal{S}}{}' = \{s_{i_0}, s_{i_1}, ...s_{i_p}, s_{j_0}, s_{j_1}, ...s_{j_q}\}$.

Next we present the most significant rules of the operational semantics.

3.3 Speculate

A process outside any speculation successfully starts a new speculation by using the $speculate(\oplus)$ construct. A new speculation is created and added to the list of speculations created by the process. A checkpoint of the process is also taken. The checkpoint becomes part of that process's state and is added to the top of the checkpoints stack. The process advances with the execution to the next instruction in its program.

We only show the case when a speculative process starts a new speculation. The case when a nonspeculative process initiates its first speculation is similar and trivial. The speculation identifier is added to the list of speculations started by the process. A new checkpoint is saved in the checkpoint stack. The speculation dependency list associated with the checkpoint is computed by adding the speculation identifier to the front of the dependency list of the previous checkpoint. By adding the speculation s to the front of the dependency list $\mathcal{D}_{\mathcal{S}}$, we force the speculation to depend on all the previous speculations the process is involved in. The rule describing these actions is shown next.

Spec

$$\Theta \ \ddagger\ p_i : \left[\mathcal{L}_{\mathcal{S}} \,\middle|\, \langle \mathcal{D}_{\mathcal{S}}\ ,\ \Gamma'\ ,\ e' \rangle\ ;\ \mathcal{C}\ \middle|\ \Gamma \vdash \mathit{speculate}(e_1 \oplus e_2); e_3 \right]\ ;\ \Pi$$
$$\Longrightarrow$$
$$\Theta \ \ddagger\ p_i : \left[s;\mathcal{L}_{\mathcal{S}} \,\middle|\, \langle \{s\} \overset{\leftrightarrow}{\cup} \mathcal{D}_{\mathcal{S}}\ ,\ \Gamma\ ,\ e_2;e_3\rangle\ ;\ \langle \mathcal{D}_{\mathcal{S}}\ ,\ \Gamma'\ ,\ e'\rangle\ ;\mathcal{C}\ \middle|\ \Gamma \vdash e_1; e_3\right]\ ;\ \Pi$$

3.4 Reading from a Shared Object

The process is outside any speculation and the object is inside a speculation. See rule READ-SPEC-OBJ**.** A nonspeculative process executing a read from a speculative object becomes itself speculative. The process's state depends on speculative information, so the process is absorbed in the object's speculation. A checkpoint of the process is created to allow rollback if the speculation is later aborted.

Read-Spec-Obj

$$o_j : \left[\ \langle \mathcal{D}_{\mathcal{S}}, V' \rangle\ ;\ \mathcal{K}\ \middle|\ V \right]\ ;\ \Theta\ \ddagger\ p_i : \left[\mathcal{L}_{\mathcal{S}}\ \middle|\ \langle\,\rangle\ \middle|\ \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v] \right]\ ;\ \Pi$$
$$\Longrightarrow$$
$$o_j : \left[\ \langle \mathcal{D}_{\mathcal{S}}, V' \rangle\ ;\ \mathcal{K}\ \middle|\ V \right]\ ;\ \Theta\ \ddagger$$
$$p_i : \left[\mathcal{L}_{\mathcal{S}}\ \middle|\ \langle \mathcal{D}_{\mathcal{S}}\ ,\ \Gamma\ ,\ \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v] \rangle\ \middle|\ \Gamma, v : V \vdash e[v] \right]\ ;\ \Pi$$

The process and the object are inside different speculations. See rule READ-SPEC-NESTED**.** The most interesting case for reading the value of a shared object is when the process and the object belong to different speculations. After the read operation is performed the state of the process depends on the speculative data stored in the shared object at the time of the access. The process saves a checkpoint that depends both on the speculation list its current state depends on and the speculation list the object's state depends on. We use the $\mathcal{D}_{\mathcal{S}} \overset{\leftrightarrow}{\cup} \mathcal{D}_{\mathcal{S}}'$ notation to merge the two dependency lists. The following rule provides the graphical representation of the state change.

Read-Spec-Nested

$$o_j : \Big[\ \langle \mathcal{D}_\mathcal{S}, V' \rangle\ ;\ \mathcal{K}\ \Big| V \Big]\ ;\ \Theta\ \ \ddagger$$

$$p_i : \Big[\ \mathcal{L}_\mathcal{S}\ \Big|\ \langle {\mathcal{D}_\mathcal{S}}'\ ,\ \Gamma'\ ,\ e' \rangle\ ;\ \mathcal{C}\ \Big| \Gamma \vdash \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v] \Big]\ ;\ \Pi$$

$$\Longrightarrow$$

$$o_j : \Big[\ \langle \mathcal{D}_\mathcal{S}, V' \rangle\ ;\ \mathcal{K}\ \Big| V \Big]\ ;\ \Theta\ \ \ddagger$$

$$p_i : \Big[\ \mathcal{L}_\mathcal{S}\ \Big| \langle \mathcal{D}_\mathcal{S} \overset{\leftrightarrow}{\cup} {\mathcal{D}_\mathcal{S}}'\ ,\ \Gamma\ ,\ \mathit{let}\ v = \mathit{read}(o_j)\ \mathit{in}\ e[v] \rangle\ ;\ \langle {\mathcal{D}_\mathcal{S}}'\ ,\ \Gamma'\ ,\ e' \rangle\ ;\ \mathcal{C}\ \Big|$$
$$\Gamma, v : V \vdash e[v]\ ]\ ;\ \Pi$$

3.5 Writing Data to a Shared Object

The process is inside a speculation and the object is outside any speculation. See rule Write-Spec-Proc**.** When a process that speculates writes a value to a shared object that is not part of any speculation the object is absorbed in the process's speculation. The system creates a checkpoint for the object, storing the value it had before becoming part of the speculation, which allows it to roll back if the speculation is aborted. The speculation id is also stored as part of the checkpoint.

Write-Spec-Proc

$$o_j : \Big[\langle\, \rangle \Big| V \Big]\ ;\ \Theta\ \ \ddagger\ \ p_i : \Big[\ \mathcal{L}_\mathcal{S}\ \Big|\ \langle \mathcal{D}_\mathcal{S}\ ,\ \Gamma'\ ,\ e' \rangle\ ;\ \mathcal{C}\ \Big| \Gamma \vdash \mathit{write}(V', o_j); e \Big]\ ;\ \Pi$$

$$\Longrightarrow$$

$$o_j : \Big[\langle \mathcal{D}_\mathcal{S}, V \rangle \Big| V' \Big]\ ;\ \Theta\ \ \ddagger\ \ p_i : \Big[\ \mathcal{L}_\mathcal{S}\ \Big|\ \langle \mathcal{D}_\mathcal{S}\ ,\ \Gamma'\ ,\ e' \rangle\ ;\ \mathcal{C}\ \Big| \Gamma \vdash e \Big]\ ;\ \Pi$$

The process is outside any speculation and the object is inside a speculation. See rule Write-Spec-Obj**.** A nonspeculative process that writes to a speculative object extracts the object from the speculation it belongs to. This behavior is expected, because regardless of the outcome of the speculation the object belongs to, the nonspeculative write would have been performed.

Write-Spec-Obj

$$o_j : \Big[\ \langle \mathcal{D}_\mathcal{S}, V' \rangle\ ;\ \mathcal{K}\ \Big| V \Big]\ ;\ \Theta\ \ \ddagger\ \ p_i : \Big[\ \mathcal{L}_\mathcal{S}\ \Big| \langle\, \rangle\ \Big| \Gamma \vdash \mathit{write}(V'', o_j); e \Big]\ ;\ \Pi$$

$$\Longrightarrow$$

$$o_j : \Big[\langle\, \rangle \Big| V'' \Big]\ ;\ \Theta\ \ \ddagger\ \ p_i : \Big[\ \mathcal{L}_\mathcal{S}\ \Big| \langle\, \rangle \Big| \Gamma \vdash e \Big]\ ;\ \Pi$$

The process and the object are inside different speculations; speculations are merged. See rule Write-Spec-Nested**.** The most interesting case for writing an object is when the process and the object are inside different speculations. In this case the object becomes speculative and a new checkpoint is created. Its checkpoint depends on both the speculations in the object's dependency list as well as the process's dependency list. Again, we use the $\mathcal{D}_{\mathcal{S}} \overset{\leftrightarrow}{\cup} \mathcal{D}_{\mathcal{S}}{}'$ notation to merge the two dependency lists.

Write-Spec-Nested

$$o_j : \left[\ \langle \mathcal{D}_{\mathcal{S}}{}', V' \rangle \ ; \ \mathcal{K} \ \middle| V \right] \ ; \Theta \ \ \ddagger$$

$$p_i : \left[\mathcal{L}_{\mathcal{S}} \ \middle| \ \langle \mathcal{D}_{\mathcal{S}} \ , \ \Gamma' \ , \ e' \rangle \ ; \ \mathcal{C} \ \middle| \Gamma \vdash \mathit{write}(V'', o_j); e \right] \ ; \ \Pi$$
$$\Longrightarrow$$
$$o_j : \left[\ \left\langle \mathcal{D}_{\mathcal{S}} \overset{\leftrightarrow}{\cup} \mathcal{D}_{\mathcal{S}}{}', V'' \right\rangle \ ; \ \langle \mathcal{D}_{\mathcal{S}}{}', V' \rangle \ ; \ \mathcal{K} \ \middle| V'' \right] \ ; \Theta \ \ \ddagger$$

$$p_i : \left[\mathcal{L}_{\mathcal{S}} \ \middle| \ \langle \mathcal{D}_{\mathcal{S}} \ , \ \Gamma' \ , \ e' \rangle \ ; \ \mathcal{C} \ \middle| \Gamma \vdash e \right] \ ; \ \Pi$$

3.6 Aborting a Speculation

Processes and Aborted Speculations. A process that is inside a speculation that it owns is allowed to abort it. The reduction rule for the $\mathit{abort}()$ call substitutes the id of the aborted speculation with the **aborted** special constant and rolls back the process to the state it was before entering the speculation. Also, the speculation id is erased from the speculations environment. The substitution operation guarantees that once a speculation has been aborted no other process or object can be absorbed in that speculation, since its id has been replaced by the special constant **aborted**. The substitution of all occurences of speculation identifier s with the special constant **aborted** is represented in the operational semantics as follows: $\Theta[s] \ \ \ddagger \ \ \Pi[s] \Longrightarrow \Theta[\mathbf{aborted}] \ \ \ddagger \ \ \Pi[\mathbf{aborted}]$

The abort operation has in fact two steps in our operational semantics, as described below. This separation is not necessary in the implementation, its sole purpose being to clearly identify the actions that have to be performed for the system to behave correctly.

First, the process that aborts the speculation has to abort all the speculations that it started since it entered that speculation. By repeatedly applying rule AB-OWNER it aborts them and rolls back its state. The program does not change for the process during this stage.

Ab-Owner

$$\Theta[s_{j_1}] \ \ \ddagger \ \ p_i : \left[s_{j_1}; \mathcal{L}_{\mathcal{S}}; s_{j_q}; \mathcal{L}_{\mathcal{S}}{}' \ \middle| \mathcal{C}' \ ; \ \langle s_{j_1}; \mathcal{D}_{\mathcal{S}} \ , \ \Gamma' \ , \ e' \rangle \ ; \ \mathcal{C} \ \middle| \Gamma \vdash \mathit{abort}(s_{j_q}); e \right] \ ;$$
$$\Pi[s_{j_1}]$$
$$\Longrightarrow$$
$$\Theta[\mathbf{aborted}] \ \ \ddagger \ \ p_i : \left[\mathcal{L}_{\mathcal{S}}; s_{j_q}; \mathcal{L}_{\mathcal{S}}{}' \ \middle| \mathcal{C} \ \middle| \Gamma' \vdash \mathit{abort}(s_{j_q}); e' \right] \ ; \ \Pi[\mathbf{aborted}]$$

The second step is reached when the speculation that it wants to abort is the last one in its list $\mathcal{L}_\mathcal{S}$ (See rule Ab-Owner-Last. At this point it aborts the speculation and it continues execution on the abort branch, e'.

Ab-Owner-Last

$$\Theta[s_{j_q}] \ \ddagger\ p_i : \left[\ s_{j_q};\mathcal{L}_\mathcal{S} \ \middle|\ \mathcal{C}' \ ;\ \langle s_{j_q};\mathcal{D}_\mathcal{S}\ ,\ \Gamma'\ ,\ e'\rangle\ ;\ \mathcal{C} \ \middle|\ \Gamma \vdash abort(s_{j_q}); e \right] \ ;\ \Pi[s_{j_q}]$$

$$\Longrightarrow$$

$$\Theta[\mathbf{aborted}] \ \ddagger\ p_i : \left[\ \mathcal{L}_\mathcal{S} \ \middle|\ \mathcal{C} \ \middle|\ \Gamma' \vdash e' \right] \ ;\ \Pi[\mathbf{aborted}]$$

If the execution of a process depends on a speculation that has been aborted by another process, it has to roll back and restart the computation from the point where it was absorbed in the speculation. If it has itself started speculations during the aborted computation, it has to abort them as well. Again, we designed a two-step process to handle this. First, the process aborts all the speculations that it has started and rolls back, step by step, to the last speculation it started inside the remotely aborted one. For this the rule Ab-Force-Ab is applied repeatedly.

Ab-Force-Ab

$$\Theta[s] \ \ddagger\ p_i : \left[\ s;\mathcal{L}_\mathcal{S} \ \middle|\ \mathcal{C}' \ ;\ \langle s\ ;\ \mathcal{D}_\mathcal{S}\ ;\mathbf{aborted}\ ;\ \mathcal{D}_\mathcal{S}'\ ,\ \Gamma'\ ,\ e'\rangle\ ;\ \mathcal{C} \ \middle|\ \Gamma \vdash e \right] \ ;\ \Pi$$

$$\Longrightarrow$$

$$\Theta[\mathbf{aborted}] \ \ddagger\ p_i : \left[\ \mathcal{L}_\mathcal{S} \ \middle|\ \mathcal{C} \ \middle|\ \Gamma' \vdash e' \right] \ ;\ \Pi[\mathbf{aborted}]$$

When the process is inside the remotely aborted speculation and has no active speculations that it started itself, then it rolls back to the state it had before becoming part of that speculation as described by rule Ab-Proc.

Ab-Proc

$$\Theta \ \ddagger\ p_i : \left[\ \mathcal{L}_\mathcal{S} \ \middle|\ \mathcal{C}' \ ;\ \langle \mathcal{D}_\mathcal{S}\ ;\ \mathbf{aborted}\ ;\ \mathcal{D}_\mathcal{S}'\ ,\ \Gamma'\ ,\ e'\rangle\ ;\ \mathcal{C} \ \middle|\ \Gamma \vdash e \right] \ ;\ \Pi$$

$$\Longrightarrow$$

$$\Theta \ \ddagger\ p_i : \left[\ \mathcal{L}_\mathcal{S} \ \middle|\ \mathcal{C} \ \middle|\ \Gamma' \vdash e' \right] \ ;\ \Pi$$

$$\mathbf{when}\ \mathcal{L}_\mathcal{S} \cap \mathcal{D}_\mathcal{S} = \emptyset$$

Objects and Aborted Speculations. If an object is inside an aborted speculation it rolls back its state to the state saved in the checkpoint associated with the speculation in a manner similar to that presented for processes in rule Ab-Proc.

3.7 Commit a Speculation

Processes and Committed Speculations. Only processes that own a speculation may commit it. The checkpoint associated with the speculation is discarded and the speculation is marked as committed. This prevents other processes to become dependent on the speculation. The reduction rule COMM-OWNER illustrates this behavior.

Comm-Owner

$$\Theta[s] \;\ddagger\; p_i : \left[\mathcal{L}_\mathcal{S}; s; \mathcal{L}_\mathcal{S}{}' \;\middle|\; \mathcal{C}' \;;\; \langle s \;;\; \mathcal{D}_\mathcal{S} \;,\; \Gamma' \;,\; e' \rangle \;;\; \mathcal{C} \;\middle|\; \Gamma \vdash commit(s); e \right] \;\Pi$$
$$\Longrightarrow$$
$$\Theta[\mathbf{committed}] \;\ddagger\; p_i : \left[\mathcal{L}_\mathcal{S}{}' \;;\; \mathcal{L}_\mathcal{S} \;\middle|\; \mathcal{C}'[\mathbf{committed}] \;;\; \mathcal{C} \;\middle|\; \Gamma \vdash e \right] \;\Pi[\mathbf{committed}]$$

If a speculation has been aborted its identifier can be eliminated from the dependency list associated with the checkpoints that depended on it. The operational semantics rule corresponding to this behavior is COMM-PEER.

Comm-Peer

$$\Theta \;\ddagger\; p_i : \left[\mathcal{L}_\mathcal{S} \;\middle|\; \mathcal{C}' \;;\; \langle \mathcal{D}_\mathcal{S} \;;\; \mathbf{committed} \;;\; \mathcal{D}_\mathcal{S}{}' \;,\; \Gamma' \;,\; e' \rangle \;;\; \mathcal{C} \;\middle|\; \Gamma \vdash e \right] \;\Pi$$
$$\Longrightarrow$$
$$\Theta \;\ddagger\; p_i : \left[\mathcal{L}_\mathcal{S} \;\middle|\; \mathcal{C}' \;;\; \langle \mathcal{D}_\mathcal{S} \;;\; \mathcal{D}_\mathcal{S}{}' \;,\; \Gamma' \;,\; e' \rangle \;;\; \mathcal{C} \;\middle|\; \Gamma \vdash e \right] \;\Pi$$

Finally, if a checkpoint does not depend on any speculations it can be completely eliminated. The COMM-REMOVE rule shows this action.

Comm-Remove

$$\Theta \;\ddagger\; p_i : \left[\mathcal{L}_\mathcal{S} \;\middle|\; \mathcal{C}' \;;\; \langle \;,\; \Gamma' \;,\; e' \rangle \;;\; \mathcal{C} \;\middle|\; \Gamma \vdash e \right] \;\Pi$$
$$\Longrightarrow$$
$$\Theta \;\ddagger\; p_i : \left[\mathcal{L}_\mathcal{S} \;\middle|\; \mathcal{C}' \;;\; \mathcal{C} \;\middle|\; \Gamma \vdash e \right] \;\Pi$$

Objects and Committed Speculations. When speculations are committed by processes, the objects inside the speculation may have to discard their saved checkpoint in a way similar to how processes do it.

3.8 Equivalence to Nonspeculative Excution

Using the operational semantics presented in this paper we have been able to provide a proof of equivalence between this model and a nonspeculative, nondeterministic model. This result enables the use of existing tools, like model checkers and theorem provers to reason about speculative programs, rather than building new such tools that understand the speculative constructs that we introduce. Due to space constraints we are unable to provide a detailed description of the nonspeculative model or of the equivalence proofs. However, a detailed presentation can be found in [4].

4 Implementations of Speculations and Their Limitations

To validate the operational semantics presented in this paper we implemented two systems that provide primitives for speculative execution. We have found speculations to be a powerful and useful programming model; however, in the two implementations that we provide there are several limitations that potential users should consider. The limitations we discuss are inherent to our implementations of speculations and could be addressed in other systems based on our operational semantics.

We initially investigated speculative execution in conjunction with process migration as programming language primitives implemented through a compiler. The Mojave Compiler Collection (MCC) [5] is a heap-based multi-language compiler that enables speculative execution. This aproach requires runtime library or operating system support for speculative I/O. Furthermore, the programmer has to be aware that, in the existing environments, certain actions, like printing or screen output, are hard or impossible to undo. This is especially true if users observe these actions; thus they should be wisely used inside speculations. We have chosen not to simply prohibit the use of such actions inside speculations because in certain cases, like building speculative debuggers, the speculative output can be used to identify problems with the code.

To fully support distributed speculative execution, including speculative network communication, we implemented speculations at the operating system level [4]. By providing the entire functionality inside the Linux kernel, we eliminated the need for a special compiler. We export the speculative primitives as system calls. Furthermore, user level processes that are not aware of speculations can be transparently instrumented by the operating system to enable speculative behavior. This approach significantly broadens the use of speculations. Caution is still required when interacting with non-speculative remote processes. However, we have found out that this is not an issue in the case of closed environments, like clusters or highly parallel machines dedicated to scientific computing, or if the actions performed are idempotent.

4.1 Cost of Speculative Primitives

Due to lack of space we only present a brief collection of experimental results obtained using our kernel module implementation for speculative execution. The testbed was a dual-processor machine with 700Mhz CPUs. The results are only presented to show the relation between the cost of the speculative primitives and the cost of other operations intrinsic to the operating system.

We measured the cost of each of the speculative primitives: entering the speculation, performing the commit and the abort operations. The numbers were averaged over 100 runs and were measured for a program were data is created and accessed before the speculation is started, data is modified in its entirety inside the speculation, and the speculation is either committed or aborted. The results are shown in Figure 7.

For comparison we present in Figure 7 sample times for several system calls as well as the context switch time for various numbers of processes of the same size running in parallel. The results were collected using the LMbench [6] tool.

In conclusion, the cost of starting, aborting, and committing a speculation are significantly less than the context switch times for large size processes and only one or

Op. \ Size	100kb	32Mb	64Mb	128Mb
Speculate	708	751	857	1727
Commit	412	704	1334	1733
Abort	950	1786	1807	1866

Sys. call	getpid()	stat()	open()/close()
Time	0.2063	23.0667	28.6804

Context switch				
# proc \ Size	10kb	32Mb	64Mb	128Mb
2	2.05	17949	35958	72029
5	2.55	17989	35972	///////
10	3.30	17989	///////	///////

Fig. 7. The cost of speculative operations (in μs) vs. the cost of other system calls and that of the context switch time (also presented in μs). The grayed out entries were eliminated because they were significantly affected by swapping.

two orders of magnitude higher than regular system calls. Analysis of the performance benefits of using speculative execution should be done on a per case basis since the improvements depend on the frequency of the speculations and on their outcome.

5 Related Work

5.1 Transactions

Database transactions, known for their ACID (Atomic, Consistent, Isolated, Durable) properties, are a powerful concept in providing reliability and fault-tolerance.

While database transactions have been studied extensively, an operational semantics describing the behavior of speculations that could be applied to other domains has only been recently provided [7]. In their approach, Prinz and Thalheim discuss ACID transactions and do not take into consideration any relaxation of the properties.

Black et.al. [8] provide a very interesting equational theory of various types of transactions. They discuss lightweight transactions that deviate from traditional transaction by relaxing either the Consistency or the Durability property [8]. The theory they provide is presented using an equational calculus that has limited expressiveness, as it only analyzes actions and does not capture state. Furthermore, in their approach isolation is implicit.

Non-isolated transactions have been studied in the context of long-lived transactions that can hold on to database resources for long periods of time, delaying the termination of other transactions. Molina and Salem [3] introduced the concept of *saga*, which is a non-isolated, non-atomic transaction formed of a set of smaller isolated, atomic transactions and a set of compensating transactions that undo the actions of the smaller transactions if those have to be rolled back.

Our concept of speculations and traditional database transactions share many traits, but they are distinct in one significant way: speculations do not provide isolation. Thus, processes executing inside speculations expose their actions to the outside world and can absorb other processes in their speculation.

Furthermore, we push the concept of non-isolated transactions from databases to programming languages and distributed environments. This requires redefining the semantics to the new domain it is used in, which we do in this paper.

The Venari project[9] implements a transaction mechanism as part of Standard ML, utilizing a mutation log produced by a generational garbage collector to implement undoability.

Harris and Keir provide conditional critical regions implementation in Java and integrate it with transactional memory and atomic execution blocks [10]. Their approach has been very successful in moving away from locks and condition variables in writing concurrent application.

Recent related work includes the AtomCaml [11] project, which is an extension to Objective Caml that provides a synchronization primitive for atomic (transactional) execution of code to replace locks. In our approach we allow speculative programs to use communication to interact with other programs while executing inside a speculations.

These approaches consider only traditional transactions that require isolation. We relax isolation, thus increasing the parallelism of the programs.

5.2 Checkpoint and Recovery

Both theoretical and practical works have discussed various approaches and protocols that enable distributed applications to recover in a consistent state based on saved checkpoints. This is achieved by either optimistic or pessimistic message logging, and by coordinated or uncoordinated checkpoints. They usually assume the existence of stable storage that survives failures.

Strom and Yemini introduced in [12] the notion of optimistic recovery. They define it as a technique based on dependency tracking, which avoids the domino effect while allowing the computation, the checkpointing and the "committing" to proceed asynchronously.

The "Virtual Time" [13] paper introduces a mechanism for optimistic speculative execution in a distributed system. Processes exchange messages and assume that the messages they receive are in order. In case this assumption is violated the computation is rolled back, the messages in the receive queue are reordered and the computation continues by processing messages in the newly provided order. Processes are implicitly forced to checkpoint regularly. The specification of the system requires the state of each process to be saved after each send or receive operation.

While speculations are similar to the concept of lookahead-rollback introduced by the TimeWarp [13] mechanism, we extend the concept by allowing both explicit and implicit speculations through programming languages extensions. We also introduce shared objects as part of the speculative model.

The *Rx* [14] system uses checkpointing and rollback to enable applications to survive software bugs. The limitation of the Rx system is that it operates only on isolated applications.

The main differences between this area of research and our approach are:

- speculations can provide programs with alternate execution paths upon rollback,
- speculations are lightweight checkpoints that are stored in memory and can be coupled with real checkpointing mechanism for increased reliability, and

- we expose speculations as programming language primitives that have a semantics closer to that of transactions than that of checkpoints.

5.3 Speculative Execution

Concepts of optimistic execution similar to speculations are used to address optimizations of I/O operations [15], fault-tolerant networking [16], shared memory systems [17] and also to increase the performance of processors [18]. By introducing programming language primitives we extend the usability of speculations to a wider range of applications. BlueFs [19] uses speculations to improve the performance of NFS clients. However, it only use local, isolated speculations that are not controllable by the user. Our approach pushes speculative primitives to user level and provides an implementation that is able to handle distributed speculations and distributed rollback, making it more generic and more widely applicable.

The *angelic nondeterminism* [20] concept introduced by Hoare has a semantic that is similar to speculative execution. It defines nondeterminism ($P \sqcap Q$) as the "execution of both P and Q concurrently until the environment chooses an event which is possible for one but not the other." This implementation of nondeterminism has a high cost in terms of efficiency. In our speculative model we optimize the *angelic nondeterminism* implementation by setting a higher preference for one of the two execution branches, based on the assumption that we make. This permits a more efficient implementation. Furthermore, we consider communicating processes and the effects of rollback (abort) to the state of the entire distributed system.

6 Conclusion and Future Work

The speculative execution model introduced in this paper has the following properties:

1. it eliminates the need for error handling code in writing programs;
2. it provides a mechanism similar to the concept of exceptions from programming languages that extends to parallel and distributed applications, and enables parallel applications to do synchronized rollbacks;
3. it provides safe distributed recovery lines for distributed applications;
4. it may improve performance of distributed applications by allowing optimistic execution;
5. it enables applications to use a different execution path upon rollback, depending on how the assumption was invalidated;
6. it can be implemented in practice using a copy-on-write mechanism to build efficient, lightweight, incremental checkpoints of processes, and
7. our representation makes it easy to code the semantics in an automated theorem prover, like MetaPRL [21], to reason about the correctness of speculative programs.

These properties provide the following advantages:

1. checkpoints generated using speculations introduce less overhead than certain types of traditional checkpointing, and

2. if a speculation fails due to an error in the code, the alternate execution path used when rolling back may be used to execute corrected code generated subsequently.

We are currently investigating the integration of a model-checker, a logging mechanism and our prototype implementation of speculations to provide a generic debugging tool for distributed applications [22]. We are also implementing a distributed filesystem with support for speculations and a communication layer in the Linux kernel that would enable transparent speculative communication.

We believe that this work may open a large number of research opportunities in the areas of software reliability, of distributed software debugging tools, and of developing efficient, optimistic communication protocols, to name just a few.

References

1. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco (1994)
2. Ţăpuş, C., Smith, J.D., Hickey, J.: Kernel level speculative DSM. In: IEEE International Symposium on Cluster Computing and the Grid (CCGRID 2003), Tokyo, Japan. Workshop on Distributed Shared Memory (DSM) (2003)
3. Garcia-Molina, H., Salem, K.: Sagas. In: SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data, pp. 249–259. ACM Press, New York (1987)
4. Ţăpuş, C.: Distributed Speculations: Providing Fault-tolerance and Improving Performance. PhD thesis, California Institute of Technology, Pasadena, CA (June 2006)
5. Smith, J.D., Ţăpuş, C., Hickey, J.: The mojave compiler: Providing language primitives for whole-process migration and speculation for distributed applications. To be presented at the HIPS/TOPMoDelS workshop (at IPDPS 2007) (2007)
6. McVoy, L., Staelin, C.: lmbench: Portable tools for performance analysis. Usenix (1996)
7. Prinz, A., Thalheim, B.: Operational semantics of transactions. In: CRPITS'17: Proceedings of the Fourteenth Australasian database conference on Database technologies 2003, Australian Computer Society, Inc. pp. 169–179 (2003)
8. Black, A.P., Cremet, V., Guerraoui, R., Odersky, M.: An equational theory for transactions. In: FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, Australian Computer Society, Inc. pp. 38–49 (2003)
9. Haines, N., Kindred, D., Morrisett, J.G., Nettles, S.M., Wing, J.M.: Composing first-class transactions. ACM Transactions on Programming Languages and Systems Short Communication (November 1994)
10. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Object-Oriented Programming, Systems, Languages, and Applications, pp. 388–402 (October 2003)
11. Ringenburg, M.F., Grossman, D.: Atomcaml: first-class atomicity via rollback. In: ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming, pp. 92–104. ACM Press, New York (2005)
12. Strom, R., Yemini, S.: Optimistic recovery in distributed systems. ACM Trans. Comput. Syst. 3(3), 204–226 (1985)
13. Jefferson, D.R.: Virtual time. ACM Trans. Program. Lang. Syst. 7(3), 404–425 (1985)
14. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: treating bugs as allergies—a safe method to survive software failures. In: SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles, pp. 235–248. ACM Press, New York (2005)

15. Chang, F., Gibson, G.A.: Automatic i/o hint generation through speculative execution. In: OSDI '99: Proceedings of the third symposium on Operating systems design and implementation (1999)
16. Thain, D., Livny, M.: The ethernet approach to grid computing. In: HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)
17. Lai, A.C., Falsafi, B.: Memory sharing predictor: the key to a speculative coherent dsm. In: Proceedings of the 26th annual international symposium on Computer architecture, pp. 172–183. IEEE Computer Society Press, Los Alamitos (1999)
18. Oplinger, J., et al.: Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical report, Stanford, CA, USA (1997)
19. Nightingale, E.B., Chen, P.M., Flinn, J.: Speculative execution in a distributed file system. In: SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles, pp. 191–205. ACM Press, New York (2005)
20. Hoare, C.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
21. Hickey, J.J.: The MetaPRL Logical Programming Environment. PhD thesis, Cornell University, Ithaca, NY (January 2001)
22. Ţăpuş, C., Noblet, D.: Fixd: Fault detection, bug reporting, and recoverability for distributed applications. To be presented at the HIPS/TOPMoDelS workshop (at IPDPS 2007) (2007)

Actors That Unify Threads and Events

Philipp Haller and Martin Odersky

Programming Methods Lab (LAMP)
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract. There is an impedance mismatch between message-passing concurrency and virtual machines, such as the JVM. VMs usually map their threads to heavyweight OS processes. Without a lightweight process abstraction, users are often forced to write parts of concurrent applications in an event-driven style which obscures control flow, and increases the burden on the programmer.

In this paper we show how thread-based and event-based programming can be unified under a single actor abstraction. Using advanced abstraction mechanisms of the Scala programming language, we implemented our approach on unmodified JVMs. Our programming model integrates well with the threading model of the underlying VM.

1 Introduction

Concurrency issues have lately received enormous interest because of two converging trends: First, multi-core processors make concurrency an essential ingredient of efficient program execution. Second, distributed computing and web services are inherently concurrent. Message-based concurrency is attractive because it might provide a way to address the two challenges at the same time. It can be seen as a higher-level model for threads with the potential to generalize to distributed computation. Many message passing systems used in practice are instantiations of the actor model [1,19]. A popular implementation of this form of concurrency is the Erlang [3] programming language. Erlang supports massively concurrent systems such as telephone exchanges by using a very lightweight implementation of concurrent processes [2,27].

On mainstream platforms such as the JVM [26], an equally attractive implementation was as yet missing. Their standard concurrency constructs, shared-memory threads with locks, suffer from high memory consumption and context-switching overhead. Therefore, the interleaving of independent computations is often modeled in an event-driven style on these platforms. However, programming in an explicitly event-driven style is complicated and error-prone, because it involves an inversion of control [32,8].

In previous work [15], we developed *event-based actors* which let one program event-driven systems without inversion of control. Event-based actors support the same operations as thread-based actors, except that the receive operation cannot return normally to the thread that invoked it. Instead the entire continuation of such an actor has to be a part of the receive operation. This makes it possible to model a suspended actor by a continuation closure, which is usually much cheaper than suspending a thread.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 171–190, 2007.

In this paper we present a unification of thread-based and event-based actors. An actor can suspend with a full stack frame (`receive`) or it can suspend with just a continuation closure (`react`). The first form of suspension corresponds to thread-based, the second form to event-based programming. The new system combines the benefits of both models. Threads support blocking operations such as system I/O, and can be executed on multiple processor cores in parallel. Event-based computation, on the other hand, is more lightweight and scales to larger numbers of actors. We also present a set of combinators that allows a flexible composition of these actors.

This paper improves on our previous work in several respects. First, the decision whether an actor should be thread-less or not is deferred until run-time. An actor may discard its corresponding thread stack several times. Second, our previous work did not address aspects of composition. Neither a solution for sequential composition of event-based actors, nor an approach for the composition of thread-based and event-based actors in the same program was provided.

The presented scheme has been implemented in the Scala *actors* library[1]. It requires neither special syntax nor compiler support. A library-based implementation has the advantage that it can be flexibly extended and adapted to new needs. In fact, the presented implementation is the result of several previous iterations. However, to be easy to use, the library draws on several of Scala's advanced abstraction capabilities; notably partial functions and pattern matching [11].

The user experience gained so far indicates that the library makes concurrent programming in a JVM-based system much more accessible than previous techniques. The reduced complexity of concurrent programming is influenced by the following factors.

- Since accessing an actor's mailbox is race-free by design, message-based concurrency is potentially more secure than shared-memory concurrency with locks. We believe that message-passing with pattern matching is also more convenient in many cases.
- Actors are lightweight. On systems that support 5000 simultaneously active VM threads, over 1,200,000 actors can be active simultaneously. Users are thus relieved from writing their own code for thread-pooling.
- Actors are fully inter-operable with normal VM threads. Every VM thread is treated like an actor. This makes the advanced communication and monitoring capabilities of actors available even for normal VM threads.

Related work. Lauer and Needham [20] note in their seminal work that threads and events are dual to each other. They suggest that any choice of either one of them should therefore be based on the underlying platform. Almost two decades later, Ousterhout [28] argues that threads are a bad idea not only because they often perform poorly, but also because they are hard to use. More recently, von Behren and others [32] point out that even though event-driven programs often outperform equivalent threaded programs, they are too difficult to write. The two main reasons are: first, the interactive logic of a program is fragmented across multiple event handlers (or classes, as in the state design pattern [12]). Second, control flow among handlers is expressed implicitly through manipulation of shared state [6]. In the Capriccio system [33], static

[1] Available as part of the Scala distribution at http://www.scala-lang.org/

analysis and compiler techniques are employed to transform a threaded program into a cooperatively-scheduled event-driven program with the same behavior.

There are several other approaches that avoid the above control inversion. However, they have either limited scalability, or they lack support of blocking operations. Termite Scheme [14] integrates Erlang's programming model into Scheme. Scheme's first-class continuations are exploited to express process migration. However, their system apparently does not support multiple processor cores. All published benchmarks were run in a single-core setting. Responders [6] provide an event-loop abstraction as a Java language extension. Since their implementation spends a VM thread per event-loop, scalability is limited on standard JVMs. SALSA [31] is a Java-based actor language that has a similar limitation (each actor runs on its own thread). In addition, message passing performance suffers from the overhead of reflective method calls. Timber [4] is an object-oriented and functional programming language designed for real-time embedded systems. It offers message passing primitives for both synchronous and asynchronous communication between concurrent *reactive objects*. In contrast to our programming model, reactive objects are not allowed to call operations that might block indefinitely. Frugal objects [13] (FROBs) are distributed reactive objects that communicate through typed events. FROBs are basically actors with an event-based computation model. Similar to reactive objects in Timber, FROBs may not call blocking operations.

Li and Zdancewic [25] propose a language-based approach to unify events and threads. By integrating events into the implementation of language-level threads, they achieve impressive performance gains. However, blocking system calls have to be wrapped in non-blocking operations. Moreover, adding new event sources requires invasive changes to the thread library (registering event handlers, adding event loops etc.).

The actor model has also been integrated into various Smalltalk systems. Actalk [5] is an actor library for Smalltalk-80 that does not support multiple processor cores. Actra [30] extends the Smalltalk/V VM to provide lightweight processes. In contrast, we implement lightweight actors on unmodified VMs.

In section 7 we show that our actor implementation scales to a number of actors that is two orders of magnitude larger than what purely thread-based systems such as SALSA support. Moreover, results suggest that our model scales with the number of processor cores in a system. Our unified actor model provides seamless support for blocking operations. Therefore, existing thread-blocking APIs do not have to be wrapped in non-blocking operations. Unlike approaches such as Actra our implementation provides lightweight actor abstractions on unmodified Java VMs.

Our library was inspired to a large extent by Erlang's elegant programming model. Erlang [3] is a dynamically-typed functional programming language designed for programming real-time control systems. The combination of lightweight isolated processes, asynchronous message passing with pattern matching, and controlled error propagation has been proven to be very effective [2,27]. One of our main contributions lies in the integration of Erlang's programming model into a full-fledged OO-functional language. Moreover, by lifting compiler magic into library code we achieve compatibility with standard, unmodified JVMs. To Erlang's programming model we add new forms of composition as well as *channels*, which permit strongly-typed and secure inter-actor communication.

The idea to implement lightweight concurrent processes using continuations has been explored many times [34,18,7]. However, none of the existing techniques are applicable to VMs such as the JVM because (1) access to the run-time stack is too restricted, and (2) heap-based stacks break interoperability with existing code. However, the approach used to implement thread management in the Mach 3.0 kernel [9] is at least conceptually similar to ours. When a thread blocks in the kernel, either it preserves its register state and stack and resumes by restoring this state, or it preserves a pointer to a continuation function that is called when the thread is resumed. Instead of function pointers we use closures that automatically lift referenced stack variables on the heap avoiding explicit state management in many cases.

There is a rich body of work on building fast web servers, using events or a combination of events and threads (for example SEDA [35]). However, a comprehensive discussion of this work is beyond the scope of this paper.

Our integration of a high-level actor-based programming model, providing strong invariants and lightweight concurrency, with existing threading models of mainstream VM platforms is unique to the best of our knowledge. We believe that our approach offers a qualitative improvement in the development of concurrent software for multicore systems.

The rest of this paper is structured as follows. In the next section we introduce our programming model and explain how it can be implemented as a Scala library. In section 3 we introduce a larger example that is revisited in later sections. Our unified programming model is explained in section 4. Section 5 introduces channels as a generalization of actors. By means of a case study (section 6) we show how our unified programming model can be applied to programming advanced web applications. Experimental results are presented in section 7. Section 8 concludes.

2 Programming with Actors

An actor is a process that communicates with other actors by exchanging messages. There are two principal communication abstractions, namely *send* and *receive*. The expression `a!msg` sends message `msg` to actor `a`. Send is an *asynchronous* operation, i.e. it always returns immediately. Messages are buffered in an actor's *mailbox*. The receive operation has the following form:

```
receive {
  case msgpat_1 => action_1
  ...
  case msgpat_n => action_n
}
```

The first message which matches any of the patterns $msgpat_i$ is removed from the mailbox, and the corresponding $action_i$ is executed. If no pattern matches, the actor suspends.

```
// base version
val orderMngr = actor {
  while (true)
    receive {
      case Order(sender, item) =>
        val o =
          handleOrder(sender, item)
        sender ! Ack(o)
      case Cancel(sender, o) =>
        if (o.pending) {
          cancelOrder(o)
          sender ! Ack(o)
        } else sender ! NoAck
      case x => junk += x
    }
}
val customer = actor {
  orderMngr ! Order(self, myItem)
  receive {
    case Ack(o) => ...
  }
}
```

```
// version with reply and !?
val orderMngr = actor {
  while (true)
    receive {
      case Order(item) =>
        val o =
          handleOrder(sender, item)
        reply(Ack(o))
      case Cancel(o) =>
        if (o.pending) {
          cancelOrder(o)
          reply(Ack(o))
        } else reply(NoAck)
      case x => junk += x
    }
}
val customer = actor {
  orderMngr !? Order(myItem) match {
    case Ack(o) => ...
  }
}
```

Fig. 1. Example: orders and cancellations

The expression `actor { body }` creates a new actor which runs the code in `body`. The expression `self` is used to refer to the currently executing actor. Every Java thread is also an actor, so even the main thread can execute `receive`[2].

The example in Figure 1 demonstrates the usage of all constructs introduced so far. First, we define an `orderMngr` actor that tries to receive messages inside an infinite loop. The `receive` operation waits for two kinds of messages. The `Order(sender, item)` message handles an order for `item`. An object which represents the order is created and an acknowledgment containing a reference to the order object is sent back to the sender. The `Cancel(sender, o)` message cancels order `o` if it is still pending. In this case, an acknowledgment is sent back to the sender. Otherwise a `NoAck` message is sent, signaling the cancellation of a non-pending order.

The last pattern `x` in the `receive` of `orderMngr` is a variable pattern which matches any message. Variable patterns allow to remove messages from the mailbox that are normally not understood ("junk"). We also define a customer actor which places an order and waits for the acknowledgment of the order manager before proceeding. Since spawning an actor (using `actor`) is asynchronous, the defined actors are executed concurrently.

Note that in the above example we have to do some repetitive work to implement request/reply-style communication. In particular, the sender is explicitly included in

[2] Using `self` outside of an actor definition creates a dynamic proxy object which provides an actor identity to the current thread, thereby making it capable of receiving messages from other actors.

every message. As this is a frequently recurring pattern, our library has special support for it. Messages always carry the identity of the sender with them. This enables the following additional operations:

`a !? msg`	sends `msg` to `a`, waits for a reply and returns it.
`sender`	refers to the actor that sent the message that was last received by `self`.
`reply(msg)`	replies with `msg` to `sender`.
`a forward msg`	sends `msg` to `a`, using the current `sender` instead of `self` as the sender identity.

With these additions, the example can be simplified as shown on the right-hand side of Figure 1.

Looking at the examples shown above, it might seem that Scala is a language specialized for actor concurrency. In fact, this is not true. Scala only assumes the basic thread model of the underlying host. All higher-level operations shown in the examples are defined as classes and methods of the Scala library. In the rest of this section, we look "under the covers" to find out how each construct is defined and implemented. The implementation of concurrent processing is discussed in section 4.

The send operation `!` is used to send a message to an actor. The syntax `a ! msg` is simply an abbreviation for the method call `a.!(msg)`, just like `x + y` in Scala is an abbreviation for `x.+(y)`. Consequently, we define `!` as a method in the `Actor` trait[3]:

```
trait Actor {
  private val mailbox = new Queue[Any]
  def !(msg: Any): unit = ...
  ...
}
```

The method does two things. First, it enqueues the message argument in the actor's mailbox which is represented as a private field of type `Queue[Any]`. Second, if the receiving actor is currently suspended in a `receive` that could handle the sent message, the execution of the actor is resumed.

The `receive { ... }` construct is more interesting. In Scala, the pattern matching expression inside braces is treated as a first-class object that is passed as an argument to the `receive` method. The argument's type is an instance of `PartialFunction`, which is a subclass of `Function1`, the class of unary functions. The two classes are defined as follows.

```
abstract class Function1[-a,+b] {
  def apply(x: a): b
}
abstract class PartialFunction[-a,+b] extends Function1[a,b] {
  def isDefinedAt(x: a): boolean
}
```

Functions are objects which have an `apply` method. Partial functions are objects which have in addition a method `isDefinedAt` which tests whether a function is defined for a

[3] A trait in Scala is an abstract class that can be mixin-composed with other traits.

given argument. Both classes are parameterized; the first type parameter `a` indicates the function's argument type and the second type parameter `b` indicates its result type[4].

A pattern matching expression `{ case` p_1`=>` e_1`; ...; case` p_n`=>` e_n`}` is then a partial function whose methods are defined as follows.

- The `isDefinedAt` method returns `true` if one of the patterns p_i matches the argument, `false` otherwise.
- The `apply` method returns the value e_i for the first pattern p_i that matches its argument. If none of the patterns match, a `MatchError` exception is thrown.

The two methods are used in the implementation of `receive` as follows. First, messages in the mailbox are scanned in the order they appear. If `receive`'s argument `f` is defined for a message, that message is removed from the mailbox and `f` is applied to it. On the other hand, if `f.isDefinedAt(m)` is `false` for every message `m` in the mailbox, the receiving actor is suspended.

The `actor` and `self` constructs are realized as methods defined by the `Actor` *object*. Objects have exactly one instance at run-time, and their methods are similar to static methods in Java.

```
object Actor {
  def self: Actor ...
  def actor(body: => unit): Actor ...
  ...
}
```

Note that Scala has different name-spaces for types and terms. For instance, the name `Actor` is used both for the object above (a term) and the trait which is the result type of `self` and `actor` (a type). In the definition of the `actor` method, the argument `body` defines the behavior of the newly created actor. It is a closure returning the unit value. The leading `=>` in its type indicates that it is an unevaluated expression (a *thunk*).

There is also some other functionality in Scala's actor library which we have not covered. For instance, there is a method `receiveWithin` which can be used to specify a time span in which a message should be received allowing an actor to timeout while waiting for a message. Upon timeout the action associated with a special `TIMEOUT` pattern is fired. Timeouts can be used to suspend an actor, completely flush the mailbox, or to implement priority messages [3].

3 Example

In this section we discuss the benefits of our actor model using a larger example. In the process we dissect three different implementations: an event-driven version, a thread-based version, and a version using Scala actors.

[4] Parameters can carry `+` or `-` variance annotations which specify the relationship between instantiation and subtyping. The `-a, +b` annotations indicate that functions are contravariant in their argument and covariant in their result. In other words `Function1[X1, Y1]` is a subtype of `Function1[X2, Y2]` if `X2` is a subtype of `X1` and `Y1` is a subtype of `Y2`.

```
class InOrder(n: IntTree)
  extends Producer[int] {
  def produceValues = traverse(n)
  def traverse(n: IntTree) {
    if (n != null) {
      traverse(n.left)
      produce(n.elem)
      traverse(n.right)
    }
  }
}
```

```
class InOrder(n: IntTree)
  extends Producer[int] {
  def produceValues = traverse(n, {})
  def traverse(n: Tree, c: => unit) {
    if (n != null) {
      traverse(n.left, produce(n.elem,
               traverse(n.right, c)))
    } else c
  }
}
```

Fig. 2. Producers that generate all values in a tree in in-order

We are going to write an abstraction of *producers* that provide a standard iterator interface to retrieve a sequence of produced values. Producers are defined by implementing an abstract `produceValues` method that calls a `produce` method to generate individual values. Both methods are inherited from a `Producer` class. As an example, the left-hand side of figure 2 shows the definition of a producer that generates the values contained in a tree in in-order.

In a purely event-driven style, there are basically two approaches to specifying traversals, namely writing traversals in continuation-passing style (CPS), and programming explicit FSMs. The left-hand side of figure 3 shows an event-driven implementation of producers where the traversal is specified using CPS. The idea is that the `produce` method is passed a continuation closure that is called whenever the next value should be produced. For example, the in-order tree producer mentioned earlier is shown on the right-hand side of figure 2. Produced values are exchanged using an instance variable of the producer.

The right-hand side of figure 3 shows a threaded version of the producer abstraction. In the threaded version the state of the iteration is maintained implicitly on the stack of a thread that runs the `produceValues` method. Produced values are put into a queue that is used to communicate with the iterator. Requesting the next value on an empty queue blocks the thread that runs the iterator. Compared to the event-driven version, the threaded version simplifies the specification of iteration strategies. To define a specific iterator, it suffices to provide an implementation for the `produceValues` method that traverses the tree in the desired order.

Figure 4 shows an implementation of producers in terms of two actors, a *producer* actor, and a *coordinator* actor. The producer runs the `produceValues` method, thereby sending a sequence of values, wrapped in `Some` messages, to the coordinator. The sequence is terminated by a `None` message. The coordinator synchronizes requests from clients and values coming from the producer. As in the threaded version, the `produce` method does not take a continuation argument.

The actor-based version improves over the event-driven version by not requiring to specify the traversal in CPS. Moreover, it supports concurrent iterators, since communication using mailboxes is race-free. For the same reason, there is no need for an explicit blocking queue as in the threaded version, since this functionality is subsumed by the

```
abstract class CPSProducer[T] {
  var next: Option[T] = None
  var savedCont: () => unit =
    () => produceValues
  def produce(x: T,
              cont: => unit) {
    next = Some(x)
    savedCont = () => {
      next = None; cont
    }
  }
  ...
}
```

```
abstract class ThreadedProducer[T] {
  val produced = new Queue[Option[T]]
  def next: Option[T] = synchronized {
    while (produced.isEmpty) {wait()}
    produced.dequeue
  }
  new Thread(new Runnable() {
    def run() {
      produceValues
      produced += None
    }
  }).start()
  def produce(x: T) = synchronized {
    produced += Some(x)
    if (produced.length == 1) notify()
  }
  ...
}
```

Fig. 3. Event-driven and threaded producers

```
abstract class ActorProducer[T] {
  def produce(x: T) {
    coordinator ! Some(x)
  }
  private val producer = actor {
    produceValues
    coordinator ! None
  }
  ...
}
```

```
private val coordinator = actor {
  loop { receive {
    case 'next => receive {
      case x: Option[_] => reply(x)
    }
  }}
}
```

Fig. 4. Implementation of the producer and coordinator actors

actors' mailboxes. We believe that the use of blocking queues for communication is so common that it is worth making them generally available in the form of mailboxes for concurrent actors.

4 Unified Actors

Concurrent processes such as actors can be implemented using one of two implementation strategies:

- Thread-based implementation: The behavior of a concurrent process is defined by implementing a thread-specific method. The execution state is maintained by an associated thread stack.
- Event-based implementation: The behavior is defined by a number of (non-nested) event handlers which are called from inside an event loop. The execution state of a concurrent process is maintained by an associated record or object.

Often, the two implementation strategies imply different programming models. Thread-based models are usually easier to use, but less efficient (context switches, memory requirements), whereas event-based models are usually more efficient, but very difficult to use in large designs [24,32,8].

Most event-based models introduce an *inversion of control*. Instead of calling blocking operations (e.g. for obtaining user input), a program merely registers its interest to be resumed on certain *events* (e.g. signaling a pressed button). In the process, *event handlers* are installed in the execution environment. The program never calls these event handlers itself. Instead, the execution environment dispatches events to the installed handlers. Thus, control over the execution of program logic is "inverted". Because of inversion of control, switching from a thread-based to an event-based model normally requires a global re-write of the program.

In our library, both programming models are unified. As we are going to show, this unified model allows programmers to trade-off efficiency for flexibility in a fine-grained way. We present our unified design in three steps. First, we review a thread-based implementation of actors. Then, we show an event-based implementation that avoids inversion of control. Finally, we discuss our unified implementation. We apply the results of our discussion to the case study of section 3.

Thread-based actors. Assuming a basic thread model is available in the host environment, actors can be implemented by simply mapping each actor onto its own thread. In this naïve implementation, the execution state of an actor is maintained by the stack of its corresponding thread. An actor is suspended/resumed by suspending/resuming its thread. On the JVM, thread-based actors can be implemented by subclassing the `Thread` class:

```
trait Actor extends Thread {
  private val mailbox = new Queue[Any]
  def !(msg: Any): unit = ...
  def receive[R](f: PartialFunction[Any, R]): R = ...
  ...
}
```

The principal communication operations are implemented as follows.

- *Send*. The message is enqueued in the actor's mailbox. If the receiver is currently suspended in a `receive` that could handle the sent message, the execution of its thread is resumed.
- *Receive*. Messages in the mailbox are scanned in the order they appear. If none of the messages in the mailbox can be processed, the receiver's thread is suspended. Otherwise, the first matching message is processed by applying the argument partial function `f` to it. The result of this application is returned.

Event-based actors. The central idea of event-based actors is as follows. An actor that waits in a receive statement is not represented by a blocked thread but by a closure that captures the rest of the actor's computation. The closure is executed once a message is sent to the actor that matches one of the message patterns specified in the receive.

The execution of the closure is "piggy-backed" on the thread of the sender. When the receiving closure terminates, control is returned to the sender by throwing a special exception that unwinds the receiver's call stack.

A necessary condition for the scheme to work is that receivers never return normally to their enclosing actor. In other words, no code in an actor can depend on the termination or the result of a receive block. This is not a severe restriction in practice, as programs can always be organized in a way so that the "rest of the computation" of an actor is executed from within a receive. Because of its slightly different semantics we call the event-based version of the receive operation `react`.

In the event-based implementation, instead of subclassing the `Thread` class, a private field `continuation` is added to the `Actor` trait that contains the rest of an actor's computation when it is suspended:

```
trait Actor {
  private var continuation: PartialFunction[Any, unit]
  private val mailbox = new Queue[Any]
  def !(msg: Any): unit = ...
  def react(f: PartialFunction[Any, unit]): Nothing = ...
  ...
}
```

At first sight it might seem strange to represent the rest of an actor's computation by a partial function. However, note that only when an actor suspends, an appropriate value is stored in the `continuation` field. An actor suspends when `react` fails to remove a matching message from the mailbox:

```
def react(f: PartialFunction[Any, unit]): Nothing = {
  mailbox.dequeueFirst(f.isDefinedAt) match {
    case Some(msg) => f(msg)
    case None      => continuation = f; suspended = true
  }
  throw new SuspendActorException
}
```

Note that `react` has return type `Nothing`. In Scala's type system a method has return type `Nothing` iff it never returns normally. In the case of `react`, an exception is thrown for all possible argument values. This means that the argument `f` of `react` is the last expression that is evaluated by the current actor. In other words, `f` always contains the "rest of the computation" of `self`[5]. We make use of this in the following way.

A partial function, such as `f`, is usually represented as a block with a list of patterns and associated actions. If a message can be removed from the mailbox (tested using `dequeueFirst`) the action associated with the matching pattern is executed by applying `f` to it. Otherwise, we remember `f` as the "continuation" of the receiving actor. Since `f` contains the complete execution state we can resume the execution at a later point when

[5] Not only this, but also the complete execution state, in particular, all values on the stack accessible from within `f`. This is because Scala automatically constructs a *closure* object that lifts all potentially accessed stack locations into the heap.

a matching message is sent to the actor. The instance variable `suspended` is used to tell whether the actor is suspended. If it is, the value stored in the `continuation` field is a valid execution state. Finally, by throwing a special exception, control is transferred to the point in the control flow where the current actor was started or resumed.

An actor is started by calling its `start` method. A suspended actor is resumed if it is sent a message that it waits for. Consequently, the `SuspendActorException` is handled in the `start` method and in the send method. Let's take look at the send method.

```
def !(msg: Any): unit =
  if (suspended && continuation.isDefinedAt(msg))
    try { continuation(msg) }
    catch { case SuspendActorException => }
  else mailbox += msg
```

If the receiver is suspended, we check whether the message `msg` matches any of the patterns of the partial function stored in the `continuation` field of the receiver. In that case, the actor is resumed by applying `continuation` to `msg`. We also handle `SuspendActorException` since inside `continuation(msg)` there might be a nested `react` that suspends the actor. If the receiver is not suspended or the newly sent message does not enable it to continue, `msg` is appended to the mailbox.

Note that the presented event-based implementation forced us to modify the original programming model: In the thread-based model, the `receive` operation returns the result of applying an action to the received message. In the event-based model, the `react` operation never returns normally, i.e. it has to be passed explicitly the rest of the computation. However, we present below combinators that hide these explicit continuations. Also note that when executed on a single thread, an actor that calls a blocking operation prevents other actors from making progress. This is because actors only release the (single) thread when they suspend in a call to `react`.

The two actor models we discussed have complementary strengths and weaknesses: Event-based actors are very lightweight, but the usage of the `react` operation is restricted since it never returns. Thread-based actors, on the other hand, are more flexible: Actors may call blocking operations without affecting other actors. However, thread-based actors are not as scalable as event-based actors.

Unifying actors. A unified actor model is desirable for two reasons: First, advanced applications have requirements that are not met by one of the discussed models alone. For example, a web server might represent active user sessions as actors, and make heavy use of blocking I/O at the same time. Because of the sheer number of simultaneously active user sessions, actors have to be very lightweight. Because of blocking operations, pure event-based actors do not work very well. Second, actors should be composable. In particular, we want to compose event-based actors and thread-based actors in the same program.

In the following we present a programming model that unifies thread-based and event-based actors. At the same time, our implementation ensures that most actors are lightweight. Actors suspended in a `react` are represented as closures, rather than blocked threads.

Actors can be executed by a pool of worker threads as follows. During the execution of an actor, *tasks* are generated and submitted to a thread pool for execution. Tasks are implemented as instances of classes that have a single `run` method:

```
class Task extends Runnable {
  def run() { ... }
}
```

A task is generated in the following three cases:

1. Spawning a new actor using `actor { body }` generates a task that executes `body`.
2. Calling `react` where a message can be immediately removed from the mailbox generates a task that processes the message.
3. Sending a message to an actor suspended in a `react` that enables it to continue generates a task that processes the message.

All tasks have to handle the `SuspendActorException` which is thrown whenever an actor suspends inside `react`. Handling this exception transfers control to the end of the task's `run` method. The worker thread that executed the task is then free to execute the next pending task. Pending tasks are kept in a task queue inside a global scheduler object.[6]

The basic idea of our unified model is to use a thread pool to execute actors, and to *resize* the thread pool whenever it is necessary to support general thread operations. If actors use only operations of the event-based model, the size of the thread pool can be fixed. This is different if some of the actors use blocking operations such as `receive` or system I/O. In the case where every worker thread is occupied by a suspended actor and there are pending tasks, the thread pool has to grow.

In our library, system-induced deadlocks are avoided by increasing the size of the thread pool whenever necessary. It is necessary to add another worker thread whenever there is a pending task and all worker threads are blocked. In this case, the pending task(s) are the only computations that could possibly unblock any of the worker threads (e.g. by sending a message to a suspended actor.) To do this, a scheduler thread (which is separate from the worker threads of the thread pool) periodically checks if there is a task in the task queue and all worker threads are blocked. In that case, a new worker thread is added to the thread pool that processes any remaining tasks.

Unfortunately, on the JVM there is no safe way for library code to find out if a thread is blocked. Therefore, we implemented a conservative heuristic that approximates the predicate “all worker threads blocked”. The approximation uses a time-stamp of the last “library activity”. If the time-stamp is not recent enough (i.e. it has not changed since a multiple of scheduler runs), the predicate is assumed to hold, i.e. it is assumed that all worker threads are blocked. We maintain a global time-stamp that is updated on every call to send, receive etc.

Example. Revisiting our example of section 3, it is possible to economize one thread in the implementation of `Producer`. As shown in Figure 5, this can be achieved by

[6] Implementations based on work-stealing let worker threads have their own task queues, too. As a result, the global task queue is less of a bottle-neck.

```
private val coordinator = actor {
  loop { react {
    // ... as in Figure 4
  }}}
```

Fig. 5. Implementation of the coordinator actor using `react`

simply changing the call to `receive` in the coordinator process into a call to `react`. By calling `react` in its outer loop, the coordinator actor allows the scheduler to detach it from its worker thread when waiting for a `Next` message. This is desirable since the time between client requests might be arbitrarily long. By detaching the coordinator, the scheduler can re-use the worker thread and avoid creating a new one.

Composing actor behavior. Without extending the unified actor model, defining an actor that executes several given functions in sequence is not possible in a modular way.

For example, consider the two methods below:

```
def awaitPing = react { case Ping => }
def sendPong = sender ! Pong
```

It is not possible to sequentially compose `awaitPing` and `sendPong` as follows:

```
actor { awaitPing; sendPong }
```

Since `awaitPing` ends in a call to `react` which never returns, `sendPong` would never get executed. One way to work around this restriction is to place the continuation into the body of `awaitPing`:

```
def awaitPing = react { case Ping => sendPong }
```

However, this violates modularity. Instead, our library provides an `andThen` combinator that allows actor behavior to be composed sequentially. Using `andThen`, the body of the above actor can be expressed as follows:

```
{ awaitPing } andThen { sendPong }
```

`andThen` is implemented by installing a hook function in the first actor. This hook is called whenever the actor terminates its execution. Instead of exiting, the code of the second body is executed. Saving and restoring the previous hook function permits chained applications of `andThen`.

The `Actor` object also provides a `loop` combinator. It is implemented in terms of `andThen`:

```
def loop(body: => unit) = body andThen loop(body)
```

Hence, the body of `loop` can end in an invocation of `react`.

5 Channels

In the programming model that we have described so far, actors are the only entities that can send and receive messages. Moreover, the receive operation ensures *locality*, i.e.

only the owner of the mailbox can receive messages from it. Therefore, race conditions when accessing the mailbox are avoided by design. Types of messages are flexible: They are usually recovered through pattern matching. Ill-typed messages are ignored instead of raising compile-time or run-time errors. In this respect, our library implements a dynamically-typed embedded domain-specific language.

However, to take advantage of Scala's rich static type system, we need a way to permit strongly-typed communication among actors. For this, we use channels which are parameterized with the types of messages that can be sent to and received from it, respectively. Moreover, the visibility of channels can be restricted according to Scala's scoping rules. That way, communication between sub-components of a system can be hidden. We distinguish input channels from output channels. Actors are then treated as a special case of output channels:

```
trait Actor extends OutputChannel[Any] { ... }
```

Selective communication. The possibility for an actor to have multiple input channels raises the need to selectively communicate over these channels. Up until now, we have shown how to use `receive` to remove messages from an actor's mailbox. We have not yet shown how messages can be received from multiple input channels. Instead of adding a new construct, we generalize `receive` to work over multiple channels.

For example, a model of a component of an integrated circuit can receive values from both a control and a data channel using the following syntax:

```
receive {
  case DataCh ! data => ...
  case CtrlCh ! cmd => ...
}
```

Our library also provides an `orElse` combinator that allows reactions to be composed as alternatives. For example, using `orElse`, our electronic component can inherit behavior from a superclass:

```
receive {
  case DataCh ! data => ...
  case CtrlCh ! cmd => ...
} orElse super.reactions
```

6 Case Study

In this section we show how our unified actor model addresses some of the challenges of programming web applications. In the process, we review event- and thread-based solutions to common problems, such as blocking I/O operations. Our goal is then to discuss potential benefits of our unified approach. Advanced web applications typical pose at least the following challenges to the programmer:

- *Blocking operations.* There is almost always some functionality that is implemented using blocking operations. Possible reasons are lack of suitable libraries (e.g. for non-blocking socket I/O), or simply the fact that the application is built on top of a large code basis that uses potentially blocking operations in some places. Typically, rewriting infrastructure code to use non-blocking operations is not an option.

- *Non-blocking operations.* On platforms such as the JVM, web application servers often provide some parts (if not all) of their functionality in the form of non-blocking APIs for efficiency. Examples are request handling, and asynchronous HTTP requests.
- *Race-free data structures.* Advanced web applications typically maintain user profiles for personalization. These profiles can be quite complex (some electronic shopping sites apparently track every item that a user visits). Moreover, a single user may be logged in on multiple machines, and issue many requests in parallel. This is common on web sites, such as those of electronic publishers, where single users represent whole organizations. It is therefore mandatory to ensure race-free accesses to a user's profile.

Thread-based approaches. VMs overlap computation and I/O by transparently switching among threads. Therefore, even if loading a user profile from disk blocks, only the current request is delayed. Non-blocking operations can be converted to blocking operations to support a threaded style of programming: after firing off a non-blocking operation, the current thread blocks until it is notified by a completion event. However, threads do not come for free. On most mainstream VMs, the overhead of a large number of threads–including context switching and lock contention–can lead to serious performance degradation [35,10]. Overuse of threads can be avoided by using bounded thread pools [21]. Shared resources such as user profiles have to be protected using synchronization operations. This is known to be particularly hard using shared-memory locks [23]. We also note that alternatives such as transactional memory [16,17], even though a clear improvement over locks, do not provide seamless support for I/O operations as of yet. Instead, most approaches require the use of compensation actions to revert the effects of I/O operations, which further complicate the code.

Event-based approaches. In an event-based model, the web application server generates events (network and I/O readiness, completion notifications etc.) that are processed by event handlers. A small number of threads (typically one per CPU) loop continuously removing events from a queue and dispatching them to registered handlers. Event handlers are required not to block since otherwise the event-dispatch loop could be blocked, which would freeze the whole application. Therefore, all operations that could potentially block, such as the user profile look-up, have to be transformed into non-blocking versions. Usually, this means executing them on a newly spawned thread, or on a thread pool, and installing an event handler that gets called when the operation completed [29]. Usually, this style of programming entails an inversion of control that causes the code to loose its structure and maintainability [6,8].

Scala actors. In our unified model, event-driven code can easily be wrapped to provide a more convenient interface that avoids inversion of control without spending an extra thread [15]. The basic idea is to decouple the thread that signals an event from the thread that handles it by sending a message that is buffered in an actor's mailbox. Messages sent to the same actor are processed atomically with respect to each other. Moreover, the programmer may explicitly specify in which order messages should be removed from its mailbox. Like threads, actors support blocking operations using implicit thread pooling as discussed in section 4. Compared to a purely event-based approach, users are relieved from writing their own ad-hoc thread pooling code. Since the internal thread pool can be global to the web application server, the thread pool controller can leverage more information for its decisions [35]. Finally, accesses to an actor's mailbox are race-free. Therefore, resources such as user profiles can be protected by modeling them as (thread-less) actors.

7 Preliminary Results

We realize that performance across threads and events may involve a number of non-trivial trade-offs. A thorough experimental evaluation of our framework is therefore beyond the scope of this paper, and will have to be addressed in future work. However, the following basic experiments show that performance of our framework is at least comparable to those of both thread-based and event-based systems.

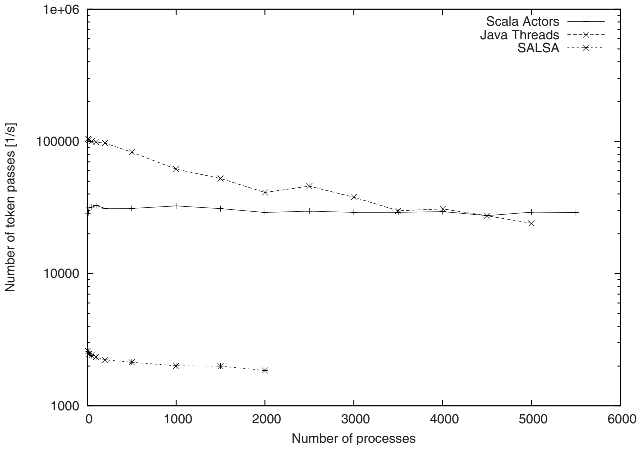


Fig. 6. Throughput (number of token passes per second) for a fixed number of 10 tokens

Message Passing. In the first benchmark we measure throughput of blocking operations in a queue-based application. The application is structured as a ring of n producers/consumers (in the following called *processes*) with a shared queue between each of them. Initially, k of these queues contain tokens and the others are empty. Each process loops removing an item from the queue on its right and placing it in the queue on its left.

The following tests were run on a 1.80GHz Intel Pentium M processor with 1024 MB memory, running Sun's Java HotSpot™VM 1.5.0 under Linux 2.6.15. We set the JVM's maximum heap size to 512 MB to provide for sufficient physical memory to avoid any disk activity. In each case we took the median of 5 runs. The execution times of three equivalent implementations written using (1) our actor library, (2) pure Java threads, and (3) SALSA (version 1.0.2), a state-of-the-art Java-based actor language [31], respectively, are compared. Figure 6 shows the number of token passes per second (throughput) depending on the ring size. Note that throughput is given on a logarithmic scale. For less than 3000 processes, pure Java threads are between 3.7 (10 processes) and 1.3 (3000 processes) times faster than Scala actors. Interestingly, throughput of Scala actors remains basically constant (at about 30,000 tokens per second), regardless of the number of processes. In contrast, throughput of pure Java threads constantly decreases as the number of processes increases. The VM is unable to create a ring with 5500 threads as it runs out of heap memory. In contrast, using Scala actors the ring can be operated with as many as 600,000 processes (since every queue is also an actor this amounts to 1,200,000 simultaneously active actors.) Throughput of Scala actors is on average over 13 times higher than that of SALSA.

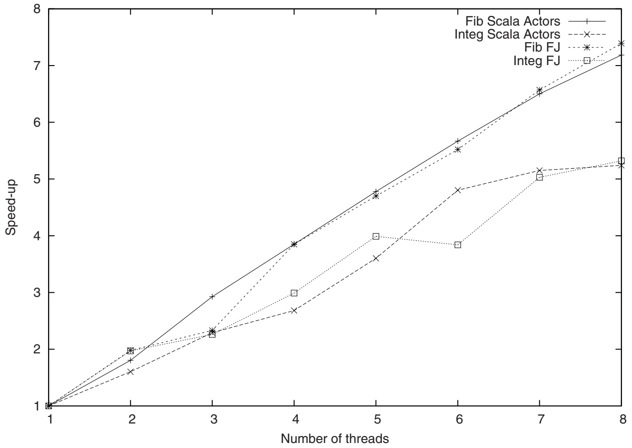


Fig. 7. Speed-up for Fibonacci and Integration micro benchmarks

Multi-core scalability. In the second experiment, we are interested in the speed-up that is gained by adding processor cores to a system. The following tests were run on a multi-processor with 4 dual-core Opteron 64-Bit processors (2.8 GHz each) with 16 GB memory, running Sun's Java HotSpot™64-Bit Server VM 1.5.0 under Linux 2.6.16. In each case we took the median of 5 runs. We ran direct translations of the Fibonacci (Fib) and Gaussian integration (Integ) programs distributed with Doug Lea's high-performance fork/join framework for Java (FJ) [22]. The speed-ups as shown in figure 7 are linear as expected since the programs run almost entirely in parallel.

8 Conclusion

In this paper we have shown how thread-based and event-based models of concurrency can be unified under a single abstraction of actors. While abstracting commonalities, our approach allows programmers to trade-off efficiency for flexibility in a fine-grained way. Scala's actor library provides a common programming model that permits high-level communication through messages and pattern matching. We believe that our work closes an important gap between message-passing concurrency and popular VM platforms.

Acknowledgments. We would like to thank our shepherd, Doug Lea, and the anonymous reviewers for their helpful comments.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, Massachusetts (1986)
2. Armstrong, J.: Erlang — a survey of the language and its industrial applications. In: Proc. INAP, pp. 16–18 (October 1996)
3. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. Prentice-Hall, Englewood Cliffs (1996)
4. Black, A., Carlsson, M., Jones, M., Kieburtz, R., Nordlander, J.: Timber: A programming language for real-time embedded systems (2002)
5. Briot, J.-P.: Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In: Proc. ECOOP, pp. 109–129 (1989)
6. Chin, B., Millstein, T.D.: Responders: Language support for interactive applications. In: Proc. ECOOP, pp. 255–278 (July 2006)
7. Cooper, E., Morrisett, G.: Adding Threads to Standard ML. Report CMU-CS-90-186, Carnegie-Mellon University (December 1990)
8. Cunningham, R., Kohler, E.: Making events less slippery with eel. In: Proc. HotOS. USENIX (June 2005)
9. Draves, R.P., Bershad, B.N., Rashid, R.F., Dean, R.W.: Using continuations to implement thread management and communication in operating systems. Operating Systems Review 25(5), 122–136 (October 1991)
10. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - A lightweight and flexible operating system for tiny networked sensors. In: LCN, pp. 455–462 (2004)
11. Emir, B., Odersky, M., Williams, J.: Matching Objects with Patterns. LAMP-Report 2006-006, EPFL, Lausanne, Switzerland (December 2006)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, London (1995)
13. Garbinato, B., Guerraoui, R., Hulaas, J., Monod, M., Spring, J.: Frugal Mobile Objects. Technical report, EPFL (2005)
14. Germain, G., Feeley, M., Monnier, S.: Concurrency oriented programming in Termite Scheme. In: Proc. Workshop on Scheme and Functional Programming (September 2006)
15. Haller, P., Odersky, M.: Event-based Programming without Inversion of Control. In: Lightfoot, D.E., Szyperski, C.A. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 4–22. Springer, Heidelberg (September 2006)
16. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA, pp. 388–402 (2003)
17. Harris, T., Marlow, S., Jones, S.L.P., Herlihy, M.: Composable memory transactions. In: Proc. PPOPP, pp. 48–60. ACM, New York (June 2005)
18. Haynes, C.T., Friedman, D.P.: Engines build process abstractions. In: Symp. Lisp and Functional Programming, pp. 18–24. ACM, New York (August 1984)
19. Hewitt, C.E.: Viewing controll structures as patterns of passing messages. Journal of Artificial Intelligence 8(3), 323–364 (1977)

20. Lauer, H.C., Needham, R.M.: On the duality of operating system structures. Operating Systems Review 13(2), 3–19 (1979)
21. Lea, D.: Concurrent Programming in Java. Addison-Wesley, London (1996)
22. Lea, D.: A java fork/join framework. In: Java Grande, pp. 36–43 (2000)
23. Lee, E.A.: The problem with threads. Technical Report UCB/EECS-2006-1, University of California, Berkeley (January 2006)
24. Levis, P., Culler, D.: Mate: A tiny virtual machine for sensor networks. In: Proc. ASPLOS (October 2002)
25. Li, P., Zdancewic, S.: A language-based approach to unifying events and threads. Technical report, University of Pennsylvania (April 2006)
26. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, London (1996)
27. Nyström, J.H., Trinder, P.W., King, D.J.: Evaluating distributed functional languages for telecommunications software. In: Proc. Workshop on Erlang, pp. 1–7. ACM, New York (August 2003)
28. Ousterhout, J.: Why threads are A bad idea (for most purposes). Invited talk at USENIX (January 1996)
29. Pai, V.S., Druschel, P., Zwaenepoel, W.: Flash: An efficient and portable Web server. In: Proc. USENIX, pp. 199–212 (June 1999)
30. Thomas, D.A., Lalonde, W.R., Duimovich, J., Wilson, M., McAffer, J., Berry, B.: Actra: A multitasking/multiprocessing Smalltalk. ACM SIGPLAN Notices 24(4), 87–90 (April 1989)
31. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Notices 36(12), 20–34 (2001)
32. von Behren, J.R., Condit, J., Brewer, E.A.: Why events are a bad idea (for high-concurrency servers). In: Proc. Hot OS,USENIX, pp. 19–24 (May 2003)
33. von Behren, J.R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.A.: Capriccio: scalable threads for internet services. In: Proc. SOSP, pp. 268–281 (2003)
34. Wand, M.: Continuation-based multiprocessing. In: LISP Conference, pp. 19–28 (1980)
35. Welsh, M., Culler, D.E., Brewer, E.A.: SEDA: An architecture for well-conditioned, scalable internet services. In: Proc. SOSP, pp. 230–243 (2001)

Generalized Committed Choice

Joxan Jaffar[1], Roland H.C. Yap[1], and Kenny Q. Zhu[2]

[1] School of Computing
National University of Singapore
{joxan,ryap}@comp.nus.edu.sg
[2] Department of Computer Science
Princeton University
kzhu@cs.princeton.edu

Abstract. We present a generalized committed choice construct for concurrent programs that interact with a shared store. The generalized committed choice (GCC) allows multiple computations from different alternatives to occur concurrently and later commit to one of them. GCC generalizes the traditional committed choice in Dijkstra's Guarded Command Language to handle *don't know* non-determinism and also allows for speculative computation. The main contribution of the paper is to introduce the GCC programming construct and the associated semantics framework for formalizing GCC. We give some experimental results which show that the power of GCC can be made practical.

1 Introduction

Nondeterminism means that a computation may need to choose between two or more choices [9]. *Don't care* non-determinism, or *committed choice*, is the most commonly used form of nondeterminism in concurrent programming systems, e.g. Occam [10] and Concurrent Prolog [17]. It's also the basis of many nondeterministic programming constructs such as guarded commands [5], CSP [9], and π-calculus [13]. The original form of committed choice is the *guarded command set* (Dijkstra's guard) [5], $G_1 \rightarrow S_1 \;\Box\; G_2 \rightarrow S_2 \;\Box\; \cdots \;\Box\; G_n \rightarrow S_n$ where G_i is a logical expression, the *guard*, and S_i is a list of statements. The meaning of Dijkstra's guard is that one can choose any S_i to execute so long as its guard G_i is true. Otherwise if all guards are false, then it aborts. Thus the choice gives rise to a form of *don't care* non-determinism in contrast with the *don't know* non-determinism used in OR-parallel logic programming [7] which explores a search space non-deterministically to find solutions.

In a concurrent programming setting, a meaningful program contains operations that manipulate its environment allowing it to interact with other running programs. A key characteristic of Dijkstra's guard (also guards in Concurrent Logic/Constraint Programming [16]) is that guard G_i is meant as a test to choose an alternative to commit to before performing any other operations which modify the environment. We call this, *early committed choice*.

In this paper, we propose a new choice construct which allows the commit to occur anywhere within the choice. We call this, *generalized committed choice*

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 191–210, 2007.

(GCC), since it generalizes the idea of early committed choice. We assume that all processes are operating in a shared environment in which all variables are global and shared (similar to [11]). We call the environment a *store*. The processes don't interact with each other directly, but instead communicate through modifying the values of the variables in the store. The only operations allowed are global variable assignments.

Next we will give a simple motivating example which explains why GCC is interesting along with some related work. The rest of the paper is organized as follows. Section 2 presents a small programming language which we embed GCC. Section 3 introduces the basic runtime structure for GCC. The operational semantics of GCC is given in Section 4. The meaning and alternatives of commit is discussed in section 5. Section 6 discusses how to make GCC practical and gives some experimental results.

1.1 A Motivating Example

The following example motivates the kinds of application of non-deterministic choice which are ideal applications for GCC. Imagine two people, among others, participate in an online automated second hand product trading system.

Bob is a photographer who wants to upgrade his equipment. He has two choices: either sell his old camera and buy a better one; or, keep the old camera but sell his old lens and buy a better lens. To avoid ending up with two cameras or selling all his equipment but unable to upgrade, only one scenario should occur. Jill wants to downgrade and either sell her good camera or her good lens. Using the proceeds from one of the above sales, she can now buy an average camera. To maximize buying and selling opportunities, we assume that the buying and selling of items can happen in any order.

We can program the requirements of Bob and Jill as follows. Exclusive choice is written as `XOR`. We assume that the market has a clearing function, which matches a buy action with a sell action, and vice versa. Thus both the buy and sell operations are synchronized and block if the corresponding action is not present.

```
 Bob:                               Jill:
(buy(goodlens); sell(averagelens)) (sell(goodlens); buy(averagecam))
           XOR                                 XOR
(buy(goodcam); sell(averagecam))   (sell(goodcam); buy(averagecam))
```

It is easy to see that there is a perfect match between Bob and Jill as Bob can buy Jill's good camera and then sell Jill his average camera. Thus there is a way in which both parties can be satisfied. However, since each party is not directly aware of the other, in this setting, we want them to be able to act independently.

Bob could choose one of the following two non-speculative strategies. The first is for Bob to take a bet on one of the choices and commit to that choice. For example, choose on the first choice that gets to make some progress. That is, if a good lens is for sale, then buy the good lens and take a risk by waiting for a buyer for the average lens. This ignores the second possibility to buy a better camera. Such a "bet-and-risk-it" strategy has obvious pitfalls. What happens if one makes the wrong choice? In the example above, if Bob's program finds a match with Jill's choice of selling good lens first, then the lens (1^{st}) choice of Bob will be committed and the camera (2^{nd}) choice will be eliminated. But as it turns out, Jill wants to buy an average camera and not an average lens after buying a good lens. As such, Bob and Jill are deadlocked, both waiting to complete their trade.

The second and more conservative strategy is for Bob to wait in both of his choices until the conditions for both buying and selling actions are met, and then do both actions atomically. While this is a safer option than the previous one, Bob will certainly miss the trading opportunity with Jill as Jill will not buy his average camera until she has sold her good camera. Both parties will be blocked even when there is a potential solution available.

Although our simple example has only two players, in a real life marketplace, much larger dependency cycles involving more parties may exist. These cyclic dependencies cannot be resolved through the use of the above early commit strategies. As we shall see next, the late commit in GCC solves this problem.

1.2 The Generalized Committed Choice Model

The generalized committed choice allows a new strategy which increases the probability of getting a solution. As Bob has two choices, to maximize his chances, he would like to be able to attempt both choices simultaneously and non-deterministically, and choose the one which succeeds. This leads to a form of speculative computation. We achieve this by having the computation in each choice operate in its own independent "world" containing an independent store. So one world does not effect the other. Now, when Jill comes to the system, her program will join the two existing worlds Bob's program has created. Since Jill also has two choices, her program will further split each world it is living in, and this creates four worlds altogether, each of which represents a possible interaction between Bob's and Jill's choices.

Here, everybody is given the full opportunity to complete their actions: while Bob may take Jill's good lens and get stuck in one world since he cannot sell his good lens; he may able to buy Jill's good camera and sell his average camera to her in another world and then eventually complete the transaction. Thus we also need a way of removing unwanted possibilities which represent other worlds and computations. In every world, Bob's and Jill's computation operate in their own independent reality. They can buy and sell items as if there is no speculation.

The model of GCC enables a programming paradigm where a computation can have a number of distinct possibilities. For simplicity, let us say there are two choices, α and β. We assume that the computations operate on a shared store which can change over time due to external events or through actions of a program. In a choice construct, we allow both α and β to proceed concurrently but isolated from each other. Although there is a form of isolation among choices within a program, an important property is that multiple programs (e.g. Bob and Jill's programs) interact with each other through different versions of the store, where each store represent one possibility. After some computation, one of the choices, say α can choose to commit. This has the effect as if the other possibility β never existed. When we have more than one user, all user choices are multiplied to form a number of worlds. Allowing speculation means that computation results in multiple rather than one store/world.

This paper proposes a programming model for speculation in a new don't-know non-determinism and concurrency context. The central contribution is a new programming construct GCC which generalizes early committed choice. We formalize the complicated semantics of GCC. The main challenge of the semantics is to deal with the notion of commit in the context of multiple worlds. We demonstrate in some experiments that the expressive power can be made practical. In our producer-consumer experiments, the growth in the number of worlds, total size of the store in all the worlds, and the number of program instances executing can be contained.

1.3 Related Work

A database transaction provides *atomicity* and *isolation* [15]. This can be used in the second strategy depicted earlier. The drawback is that it is unlikely to give the desired result since that means all blocking conditions are satisfied at once. Also, relational databases and SQL do not handle non-determinism, so Bob and Jill cannot specify their choices.

Long-lived transactions such as “Sagas” [6] do away with the isolation property of transactions. Partial changes to the database are visible to other transactions. Only one level of nested transactions is allowed. Operations in Saga can be unsafe, i.e. if one process cannot go through, then the whole saga needs to be compensated. Since there is concurrency, sometimes no compensation operations are possible, and hence the system becomes irreparably inconsistent. Saga does not handle our example as it does not provide non-deterministic choice. Even if Saga is extended, multiple choices are operating in the same environment, which means once Bob has bought the good lens for example, he will not have the money to pay for good camera, even if it is available. Furthermore, the compensating transactions have to be provided by the user program rather than being resolved by the system.

Dijkstra's guards [5,14] and concurrent constraint programming (CCP) [16] techniques such as GHC [19] and Oz [18] use *early committed choice* to handle non-determinism. Early committed choice can be used to implement the *bet-and-risk-it* strategy, but as we have shown, this may lead to a blocked/deadlocked computation. Furthermore, CCP languages require monotonic stores and is thus not applicable in our context. Perhaps the closest relative to GCC is the *deep guards* in GHC and some other CCP languages, where recursion is allowed in the guards. However, although commit can be postponed with deep guards only reads and no writes are allowed in the guards and hence the store is not changed. This is in contrast with the GCC model where updates to the store are permitted in the choices before commit.

Transaction Logic ($\mathcal{TR}$) [1] is an algebra that offers a logical framework to model traditional database transactions. It supports the sequence of transactions (with updates) through the $\otimes$ operator (similar to our ";") and non-determinism through the $\oplus$ operator. The non-determinism here is equivalent to an early-committed choice. $\mathcal{TR}$, however, does not support long-lived transactions. Concurrent Transaction Logic ($\mathcal{CTR}$) [2] adds the concurrent conjunction to $\mathcal{TR}$, which gives parallel interleaving of different sequences. It is different from our notion of late choice where each choice runs in isolation.

In composable memory transactions (CMT), the `orElse` construct [8] provides a way to state several *alternatives* in memory transactions. The transaction `s1 'orElse' s2` first runs `s1`; if it blocks (retries), then `s1` is abandoned with no effect, `s2` is run. If `s2` is also blocked (retries), then the whole transaction retries. In a way, the `orElse` construct offers a don't know type of non-determinism as it attempts the choices one by one until a satisfying one is found. The semantics of `orElse` differs from GCC, in the former the choices are attempted sequentially, while GCC allows all choices in parallel. In addition, CMT requires isolation of transactions, whereas GCC allows interactions between the programs during the resolution of don't-know choices. Hence, CMT cannot be used to code our Bob/Jill example directly.

In more recent development, transactional events [4] introduced a choice construct `chooseEvt` in a synchronous message passing setting. This construct is similar to the `orElse` in that there is no isolation between the choices, though the two choices are executed concurrently. Therefore it is not able to solve Bob and Jill's problem, either. Moreover, transactional event does not offer an explicit commit operation, therefore `chooseEvt` is not able to commit to a branch before the entire branch has been successfully completed, whereas GCC programs can commit a branch anywhere a commit point is reached.

The *Event-Condition-Action* rules in active databases [20,3] provide some degree of reactivity to the users, it is still early committed choice. In addition, the actions carried out in ECA rules are either simple database read/write operations or user-defined procedure calls. But in all cases, these actions are done atomically and in isolation, hence interleavings are not possible.

2 Programming with GCC

We illustrate the programming paradigm of GCC with the following simple setting. There are a number concurrent or parallel programs interacting with a common runtime system, providing a global computation environment or a global memory we call a *store*. Without loss of generality, we assume programs do not use any local variables. Synchronization is achieved by use of the common store and the use of a blocking guarded action. For now, we simply assume the store is a piece of shared memory which contains variable-value mappings. The main operation on the store are variable assignments which are atomic.

We now present simple programming constructs for programming with GCC. Although we have chosen a simple setting, it should be clear that GCC can be easily integrated into more complex programming languages and concurrent/parallel systems. For simplicity, we have used used a simple unstructured shared memory store. In the context of actual applications, a database store is consistent with programming in GCC and the semantics presented here.

We introduce a minimal concurrent language with GCC for programs r defined as follows:

$$\begin{array}{rcll}
r & ::= & \texttt{noop} & \text{no operation} \\
 & | & x := v & \text{atomic assignment} \\
 & | & \texttt{if } c \texttt{ then } r_1 \texttt{ else } r_2 & \text{conditional} \\
 & | & \texttt{while } c \texttt{ do } r_1 & \text{loop} \\
 & | & c \Rightarrow \delta & \text{guarded atomic action} \\
 & | & r_1;\ r_2 & \text{sequence} \\
 & | & r_1 \oplus r_2 & \text{GCC} \\
 & | & \texttt{cm} & \text{commit this choice} \\
 & | & \texttt{cu} & \text{commit other choice}
\end{array}$$

The first four constructs are rather standard and thus require little explanation. The assignment operation assigns a value v to a global variable x in the store atomically. Boolean condition c is tested in both the conditional, loop and guard constructs. An example of c is $x + y \leq 10$.

Guarded atomic action, or guard in short, is provided to allow for reactive behavior and enable synchronization among the programs. $c \Rightarrow \delta$, blocks until condition c is true w.r.t. the store and then atomically executes δ. δ is an action such as noop, assignment, `cm` and `cu`, all w.r.t. the store. A guarded sequence of actions can be decomposed as:

$$c \Rightarrow (\delta_1;\ \delta_2; \ldots;\ \delta_n) \equiv c \Rightarrow \delta_1;\ c \Rightarrow \delta_2; \ldots;\ c \Rightarrow \delta_n.$$

The choice construct, $r_1 \oplus r_2$, defines two computations r_1 and r_2 which are to be executed speculatively. For simplicity, we only deal with binary choices, and it is straightforward to extend to an arbitrary number of choices. Both r_1 and r_2 execute concurrently with any updates isolated from each other. Nested choices are allowed. Unless otherwise stated, in the remainder of this paper, we refer to generalized committed choice simply as *choice*.

Within a choice, there are two special operations, namely `cm` and `cu`, which stand for "commit me" and "commit you". These can only be used within the scope of a choice where they refer to the *innermost* enclosing choice structure. `cm` expresses the intention to *commit* to this branch of a choice and to remove the other branch as if it didn't exist; `cu` expresses the intention to *terminate* this branch of a choice and commit to the other branch. Notice that `cm` and `cu` are *not* symmetrical since after executing `cm` in a branch, the program can continue, whereas in the case of `cu`; the program instance is *halted.* If `cm` and `cu` are used outside the scope of a choice, they have no effect. Furthermore, only the first use of `cm` or `cu` has any effect within a choice branch. For example, in this program,

$$(r_1 \oplus_1 ((r_2; \texttt{cm}; \texttt{cm}) \oplus_2 r_3)),$$

only the first `cm` after r_2 is effective and refers to the left branch of choice $\oplus_2$. The second `cm` is ignored, and it is *not* referring to the right branch of choice $\oplus_1$.

Note we numbered the choice structures by subscripts just for the succinctness of explanation.

Static scoping is used for simplicity with `cm` and `cu`. Furthermore, we require that every choice branch must have a commit operation, so that a choice must eventually be "committed". This can be achieved by systematically adding `cm` to the end of every choice.

We remark that we have not defined a parallel composition construct here because our setting is of an open system where independent programs running in parallel can be introduced from the outside. One could, for example, implement parallel composition as the birth of a new program (see Section 4.1). Thus, one could add "fork-like" constructs to the language. We haven't done so for the interest of simplicity.

The Bob example can be re-written below in our simple programming language as follows:

```
(goodlens ≥ 1 ⇒                          (goodcam ≥ 1 ⇒
      goodlens := goodlens - 1;     ⊕          goodcam := goodcam - 1;
averagelens := averagelens + 1); cm    averagecam := averagecam + 1); cm
```

For simplicity, we have not dealt with the details of the transactions and have only expressed the requirements of the example as availabilities of the items.

Time variables are useful to write conditions involving external time. For example, this can be used to express *timeouts*. Consider a program fragment r which is to be executed as long as the timeout hasn't expired. This is expressed as, $r \ \oplus ((\mathit{time} \geq t) \Rightarrow \texttt{cm})$, so that when time has reached t, the right branch can commit and the effect is that r is aborted.

3 Worlds and Multi-worlds

We present informally a suitable runtime structure for GCC. We define a world, a store and programs and extend the definitions to multi-worlds.

3.1 World

The basic computation space for GCC is a *world*, and is denoted by w_i, where i is a unique identifier for a world. One can think of a world as a shared memory computer running multiple processes. Associated with a world is its memory, we call this a *store*, and is denoted by Δ. The store contains, in its simplest form, a set of variable-value mappings, such as $\{x = 2,\ y = -1,\ z = 0\}$.

Every world executes a dynamic number of processes or programs in an interleaving model of concurrency. New programs may enter the world by *birth*, or depart the world by *death*, or completion. Because of such dynamism, we say the world is *open*. Programs execute when the system picks one continuation from the set, and advances it by executing an instruction, known as a program step. Initially, *cids* is empty.

We define a *world* as a pair $(\Delta,\ \Sigma)$, where Δ is a store and Σ is a set of continuations of all the programs interacting with Δ. We denote the store as a triangle (△), annotated with an id of the world (and also the store itself). The set of continuations in that world is denoted by Σ below the triangle. Fig. 1 shows some worlds.

3.2 Multi-world

The runtime structure of GCC in general consists of a collection of worlds organized in a tree structure. We call this structure a *multi-world*. The leaves of the tree are worlds, and the intermediate nodes are choice nodes $\oplus_{id}$, where *id* is a unique identifier. The multi-world is used to represent different possible runtime scenarios for programs to interact. Program continuations associated with one world are executed in isolation from that of other worlds.

The operational semantics of the GCC and the programming language is centered around the creation, evolution and deletion of a number of worlds.

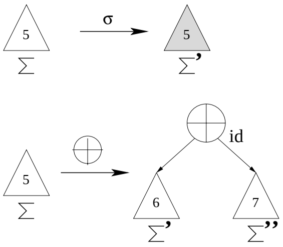


Fig. 1. Evolution of a world to a multi-world

Given a world, the execution of one program step by a continuation either evolves the world to a new state (store + continuations), or it can split the world into two new worlds using a choice, i.e. a choice point. This is illustrated in Fig. 1. The upper transition is when a continuation issues an update from a continuation σ, e.g. x := 10, which changes store Δ_5 to its new state (denoted by the gray color) and updated continuations Σ'. The lower transition shows the splitting of one world into two, when a continuation issues a choice $\oplus$. This creates two new stores, Δ_6 and Δ_7, each of which is a copy of the original store Δ_5, and also two sets of continuations Σ' and Σ'' for each branch in the choice point. Later, we show how `cm` and `cu` help reduce the number of worlds by "chopping" sub-trees from the multi-world. We can think of $\oplus_{id}$ as a logical XOR.

4 Operational Semantics

We now describe the operational semantics of GCC together with the simple programming language using state transitions on multi-worlds.

Let Δ be a store or a database which is a finite mapping between an infinite set of variables and values: $\Delta : x \mapsto v$. We write $\Delta[x \mapsto v]$ to denote the change of the value of x in Δ to v.

Let p be a program, pc be the program counter and $cids$ be a sequence of choice id's where each choice id is a unique number. An instance of a program, or a *continuation* σ is a triple: $\langle p,\ pc,\ cids\rangle$. Let $next(p, pc)$ be the pc of the next instruction, $next_l(p, pc)$ be the next instruction after σ which corresponds to the left branch of a choice, and $next_r(p, pc)$ be the next instruction which corresponds to the right branch of a choice.

Let w be a world. We recursively define a multi-world $\mathcal{W}$ to be:

$$\mathcal{W} = w \mid \mathcal{W}_1 \oplus_{id} \mathcal{W}_2$$

We can treat a multi-world as a tree, and a world as a leaf of the tree. Given a multi-world $\mathcal{W}$, the function $worlds(\mathcal{W})$ returns all worlds (or leaves) of $\mathcal{W}$.

We define δ to be a function that maps a world to a store, i.e. $\delta : w \mapsto \Delta$, and define θ to be a function that maps a world to a set of continuations currently associated with that world, i.e. $\theta : w \mapsto \Sigma$, where Σ is a set of continuations.

The state of a GCC system is a triple, $(\mathcal{W}, \delta, \theta)$, where $\mathcal{W}$ is a multi-world, and δ and θ are functions defined above. The system evolves by updating some or all of these parameters.

Let $view(\Delta) = \{(x = v) \mid \Delta(x) = v\}$, i.e. the view of a store is a set of all variable-value equations induced by the store. We further define the *conjunctive view* and *disjunctive view* of a multi-world as follows:

$$\begin{aligned}
\mathcal{CV}(w) &= view(\delta(w)) \\
\mathcal{CV}(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2) &= \{(x = v) | (x = v) \in \mathcal{CV}(\mathcal{W}_1) \wedge \\
&\quad\ (x = v) \in \mathcal{CV}(\mathcal{W}_2)\} \\
\mathcal{DV}(w) &= view(\delta(w)) \\
\mathcal{DV}(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2) &= \{(x = v) | (x = v) \in \mathcal{DV}(\mathcal{W}_1) \vee \\
&\quad\ (x = v) \in \mathcal{DV}(\mathcal{W}_2)\}
\end{aligned}$$

The conjunctive view of a multi-world represents the variables which hold identical values across all worlds in this multi-world; the disjunctive view gives a set of all possible values of all the variables in these worlds.

4.1 The GCC System

A GCC system is triple $(\mathcal{W},\ \delta,\ \theta)$. In general, the GCC system advances by the following transition rule:

$$(\mathcal{W}, \delta, \theta) \longrightarrow (\mathcal{W}', \delta', \theta') \qquad (1)$$

At each time step, the system non-deterministically chooses to take either a program step or a system step. To take a program step, the system picks a continuation in one of the worlds and executes its current instruction by one of the following program step rules: assignment, test, choice or commit. To take a system step, the system evaluates certain conditions of the multi-world, and makes changes such as pruning the multi-world tree, adding new programs or deleting completed programs from the system by selecting a suitable system step rule. The way multi-worlds evolve is that program steps grows the multi-world while system steps shrink it. We use '+' below to mean set union, '−' to mean set difference where appropriate, and '∗' to mean any value (wildcard).

Program Step Rules. We now define the transitions of (1) attributed to a user program. In what follows, we shall be selecting a continuation $\sigma = \langle p, pc, cids \rangle$ from $\mathcal{W}$, and describe its one-step execution. We write w to denote the world associated with σ, and Δ to denote the corresponding store $\delta(w)$. We write pc' to denote $next(p, pc)$ except for test or choice. For a test, we write pc_1 and pc_2 for the next pc after the test. For a choice, we write pc_l and pc_r for the next pc advancing to the left and the right branch in a choice.

We now proceed by case analysis on the construct corresponding to the current program point $p[pc]$:

Assignment $x := v$
$\Delta' = \Delta[x \mapsto v]$, $\delta' = \delta[w \mapsto \Delta']$, and
$\theta' = \theta[w \mapsto (\theta(w) - \{\sigma\} + \{\sigma'\})]$ where $\sigma' = \langle p, pc', cids \rangle$.

Test or Guard $c?$
$\theta' = \theta[w \mapsto (\theta(w) - \{\sigma\} + \{\sigma'\})]$
where $\sigma' = \langle p, pc_1, cids \rangle$ if $view(\delta(w)) \models c$;
$\sigma' = \langle p, pc_2, cids \rangle$ if $view(\delta(w)) \not\models c$.

Choice $\oplus$
$\mathcal{W}' = \mathcal{W}[w \leftarrow w' \oplus_{id} w'']$,
$id = \langle p, pc, s \rangle$ where s is a unique number,
$\delta' = \delta - \{w \mapsto \delta(w)\} + \{w' \mapsto \delta(w)\} + \{w'' \mapsto \delta(w)\}$, and $\theta' = \theta - \{w \mapsto \theta(w)\} + \{w' \mapsto (\theta(w) - \{\sigma\} + \{\sigma_l\})\} + \{w'' \mapsto (\theta(w) - \{\sigma\} + \{\sigma_r\})\}$
where $\sigma_l = \langle p, pc_l, cids.append(id) \rangle$ and $\sigma_r = \langle p, pc_r, cids.append(id) \rangle$.

Commit `cm`
$\delta' = \delta[w \mapsto (\delta(w) \cup \{\mathtt{cm}_{id} \mapsto 1\})]$ and $\theta' = \theta[w \mapsto \theta(w) - \{\sigma\} + \{\sigma'\}]$
where $id = cids.last()$ and $\sigma' = \langle p, pc', cids.droplast() \rangle$.

Commit `cu`
$\delta' = \delta[w \mapsto (\delta(w) \cup \{\mathtt{cu}_{id} \mapsto 1\})]$ and $\theta' = \theta[w \mapsto \theta(w) - \{\sigma\} + \{\sigma'\}]$
where $id = cids.last()$ and $\sigma' = \langle p, pc', cids.droplast() \rangle$.

In an assignment step coming from a continuation of a world w, the store of that world is updated and the continuation steps forward.

In a test step, no store is changed, except the continuation being activated advances to a new pc. In case of a guard, that pc is not changed if the test failed (which can be thought of as a busy loop[1]); in case of conditionals or while loop, the pc changes to different values depending on the result of the test.

If a continuation σ issues a choice in world w, that world is split into a multi-world $w' \oplus_{id} w''$, with the store of w copied to w' and w'', and the σ of w' advances to the left branch while the σ of w'' advances to the right branch. A

[1] In practice, a triggering mechanism which indexes all the blocked guards and only fires guards which could be enabled can be used [11,12].

new choice *id* is dynamically generated. We use *cids.append*(*id*) to denote that the new *id* is appended to the end of the original *cids* sequence.

The rules for `cm` and `cu` simply updates the affected world's store with a new variable and value: $\texttt{cm}_{id} = 1$ or $\texttt{cu}_{id} = 1$ to indicate that a previously issued choice has been committed on one of the branches. We use *cids.last*() to denote that the selected *id* is the last one in *cid* and *cids.droplast*() denotes that the new *cids* has the last element removed.

System Step Rules. We now complete the definition of (1), this time by defining transitions attributed to the system.

Birth $(\mathcal{W}, \delta, \theta) \xrightarrow{b(q)} (\mathcal{W}, \delta, \theta')$

where q is a new program, and $\theta' = \theta[w \mapsto \theta(w) + \langle q, 0, []\rangle]$ for all $w \in worlds(\mathcal{W})$.

Death $(\mathcal{W}, \delta, \theta) \xrightarrow{d(q)} (\mathcal{W}, \delta, \theta')$

where for all continuations $\langle q, pc, *\rangle$ in $\mathcal{W}$, $q[pc] = \texttt{end}$, and
$\theta' = \theta[w \mapsto (\theta(w) - \{\langle q[pc] = \texttt{end}, *\rangle\})]$ for all $w \in worlds(\mathcal{W})$.

Pruning

$$(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2, \delta, \theta) \xrightarrow{prune_r} (\mathcal{W}_1, \delta', \theta') \tag{2}$$

where $\mathcal{CV}(\mathcal{W}_1) \models cm_{id} = 1$,
$\delta' = \delta - \{w \mapsto \delta(w) \mid w \in worlds(\mathcal{W}_2)\}$,
$\theta' = \theta - \{w \mapsto \theta(w) \mid w \in worlds(\mathcal{W}_2)\}$.

$$(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2, \delta, \theta) \xrightarrow{prune_l} (\mathcal{W}_2, \delta', \theta') \tag{3}$$

where $\mathcal{CV}(\mathcal{W}_2) \models cm_{id} = 1$,
$\delta' = \delta - \{w \mapsto \delta(w) \mid w \in worlds(\mathcal{W}_1)\}$,
$\theta' = \theta - \{w \mapsto \theta(w) \mid w \in worlds(\mathcal{W}_1)\}$.

$$(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2, \delta, \theta) \xrightarrow{prune_l} (\mathcal{W}_2, \delta', \theta') \tag{4}$$

where $\mathcal{CV}(\mathcal{W}_1) \models cu_{id} = 1$,
$\delta' = \delta - \{w \mapsto \delta(w) \mid w \in worlds(\mathcal{W}_1)\}$,
$\theta' = \theta - \{w \mapsto \theta(w) \mid w \in worlds(\mathcal{W}_1)\}$.

$$(\mathcal{W}_1 \oplus_{id} \mathcal{W}_2, \delta, \theta) \xrightarrow{prune_r} (\mathcal{W}_1, \delta', \theta') \tag{5}$$

where $\mathcal{CV}(\mathcal{W}_2) \models cu_{id} = 1$,
$\delta' = \delta - \{w \mapsto \delta(w) \mid w \in worlds(\mathcal{W}_2)\}$,
$\theta' = \theta - \{w \mapsto \theta(w) \mid w \in worlds(\mathcal{W}_2)\}$.

The birth and death of programs formalize the launch and completion of an agent program. The birth of a new program q, denoted by $b(q)$, adds instances of this program to every world in the multi-world.[2] For simplicity, we treat every program as being unique even when they are identical. The only way programs are copied is through the execution of choice (see section 4.1). The death of a program q is happens when it has reached the end of the program in every instance of its appearance in the multi-world, then all its instances are removed from the multi-world.

There are four pruning rules, two due to `cm`, and two due to `cu`. Rules (2) and (3) chop off the right (left) subtree rooted at $\oplus_{id}$, if the conjunctive view of the left (right) subtree has implied that $\mathtt{cm}_{id} = 1$. Symmetrically, Rules (4) and (5) check if the left (right) subtree of $\oplus_{id}$ has a conjunctive view that implies $\mathtt{cu}_{id} = 1$. In other words, if a choice identified by id has committed to its left branch in all the worlds in the left sub-tree of a multi-world rooted at $\oplus_{id}$, then all other worlds under $\oplus_{id}$ in which the right branch of choice id is attempted are discarded. If the choice has given up (commit you) left branch in all worlds of the left sub-tree of the multi-world rooted at $\oplus_{id}$, then all worlds in the left sub-tree are discarded. Note that there can be more than one multi-worlds rooted at $\oplus_{id}$, but they are treated separately under the pruning rules.

4.2 An Example of Multi-world Transitions

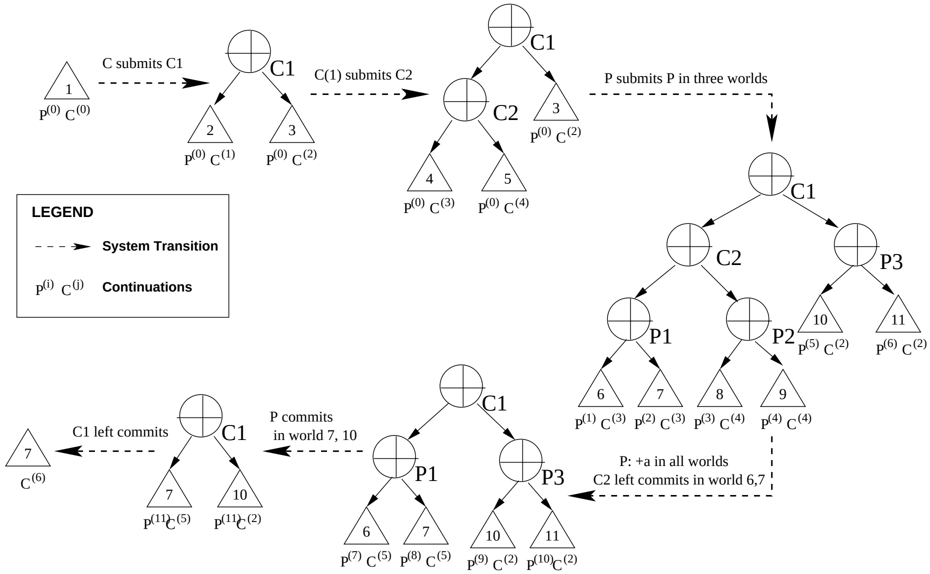
Consider the following producer/consumer example.

$$\begin{aligned} P ::= \; & (a := a+1;\; b := b+1;\; \mathtt{cm}) \oplus \\ & (a := a+1;\; c := c+1;\; \mathtt{cm}) \end{aligned} \qquad \begin{aligned} C ::= \; & (((a \geq 1 \Rightarrow a := a-1;\; \mathtt{cm}) \;\oplus_2 \\ & (b \geq 1 \Rightarrow b := b-1;\; \mathtt{cm}));\; \mathtt{cm}) \oplus_1 \\ & ((\mathit{time} \geq 10) \Rightarrow \mathtt{cm}) \end{aligned}$$

Fig. 2 illustrates the dynamic evolution of the multi-world with the two concurrent programs P and C. The initial store is $\Delta_1 = \{a = 0, b = 0, c = 0, \mathit{time} = 0\}$. We number the choice nodes of P and C as $P1, P2, P3, C1, C2$. $P^{(i)}$ and $C^{(i)}$ denote the continuations with the details given at the bottom.

Suppose C gets to make its choices first, the result is three worlds with the two choice nodes $C1$ and $C2$ in the multi-world. Then P starts to issue a choice which multiplies to six different worlds. As P produces a and b in one branch and a and c in another, before P commits in any of its branches, choice $C2$'s left branch can consume a and commit. The commit adds the information $\mathtt{cm}_{C2} = 1$ to both worlds, hence the prune step can be applied to prune off the worlds of w_8 and w_9. In the $P3$ subtree, P commits in w_{10} first and deletes w_{11}. P can also commit in the the right branch of the $P1$ subtree, using a prune step to prune off the left subtree of $P1$, namely w_6. At this point, there are only two worlds left, w_7 and w_{10}. Since P now has committed in all worlds (w_7 and w_{10}), it exits from the system. Finally, the left branch of $C1$ commits, which kills world w_{10} so there is only one remaining world w_7.

[2] The source of new programs here is assumed to be from the external environment, although, its also possible to enhance the programming language here to create new agents.



Continuations Listing:

$P^{(0)}$: $\langle P,\ 0,\ []\rangle$	$P^{(1)}$: $\langle P,\ 1,\ [P1]\rangle$	$P^{(2)}$: $\langle P,\ 5,\ [P1]\rangle$	$P^{(3)}$: $\langle P,\ 1,\ [P2]\rangle$
$P^{(4)}$: $\langle P,\ 5,\ [P2]\rangle$	$P^{(5)}$: $\langle P,\ 1,\ [P3]\rangle$	$P^{(6)}$: $\langle P,\ 5,\ [P3]$	$P^{(7)}$: $\langle P,\ 2,\ [P1]\rangle$
$P^{(8)}$: $\langle P,\ 6,\ [P1]\rangle$	$P^{(9)}$: $\langle P,\ 2,\ [P3]\rangle$	$P^{(10)}$: $\langle P,\ 6,\ [P3]\rangle$	$P^{(11)}$: $\langle P,\ 9,\ []\rangle$
$C^{(0)}$: $\langle C,\ 0,\ []\rangle$	$C^{(1)}$: $\langle C,\ 1,\ [C1]\rangle$	$C^{(2)}$: $\langle C,\ 10,\ [C1]\rangle$	$C^{(3)}$: $\langle C,\ 2,\ [C2,\ C1]\rangle$
$C^{(4)}$: $\langle C,\ 5,\ [C2,\ C1]\rangle$	$C^{(5)}$: $\langle C,\ 8,\ [C1]\rangle$	$C^{(6)}$: $\langle C,\ 12,\ []\rangle$	

Fig. 2. A worked example

5 On the Semantics of Commit

The key issue of commit is when a `cm` or `cu` is executed in some world, which other worlds should be deleted and when to delete them. For our discussion, we reuse one of the multi-words from Fig. 2 as Fig. 3. Fig. 3 shows that the same syntactic choice in P can be issued multiple times in different parts of the multi-world. These choice nodes have different id's, as they are created in different worlds and hence belong to different *scopes*. Subsequently other programs executing in worlds w_6 to w_{11} can also issue choices, further expanding the tree.

The *coordinated commit* works as follows. A commit operation not only has to match syntactically with the choice structure it belongs to (which is identified by the program code p and the program counter pc, but also maps itself to the scope in which the choice was first launched into. For example, if a commit is executed in one of the worlds under node $P1$ in Fig. 3, then only some of the worlds under $P1$ should be deleted and not worlds under $P2$ or $P3$ in the tree. This is accomplished by using the choice id sequence, *cids*, in each program continuation. A commit always uses the id of the inner-most choice construct that surrounds this commit operation. Since the id's are generated dynamically

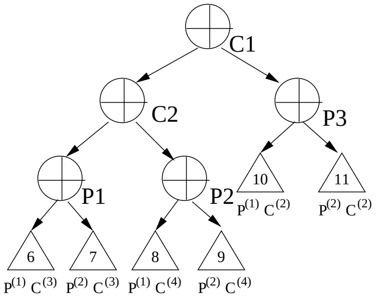


Fig. 3. A multi-world

as choices are issued, the same choice construct will obtain different id's in different scopes. Thus the matching of commits with the correct choice nodes is done automatically.

We now discuss other alternatives for the semantics of commit, and argue why the given semantics in section 4 is selected. We will illustrate these semantics in Fig. 4, 5 and 6. For simplicity, the continuations associated with the worlds are omitted. Where commit has been executed to a store Δ_i, we write $\texttt{cm}_{id}$ under Δ_i. The stores with the commits of interest are highlighted.

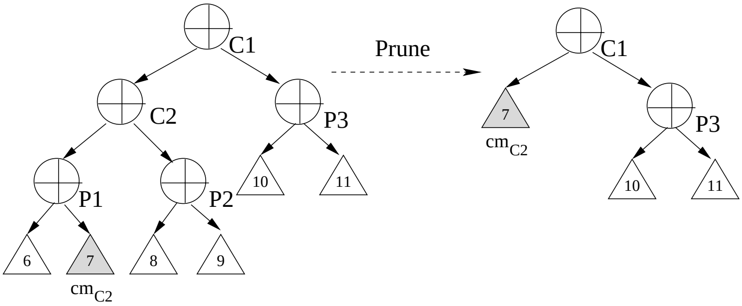


Fig. 4. Absolutely eager commit

Absolutely eager commit prunes in an eager fashion. If a cm whose id is *id* is reached in any world, this world is committed and all other worlds within the scope of this cm, that is, under the subtree rooted at $\oplus_{id}$, are killed immediately, leaving just one world under $\oplus_{id}$. That effectively removed all the intermediate nodes including $\oplus_{id}$ in that subtree (see Fig. 4).

This form of commit does not seem very useful since it allows for minimal inter-play of choices, and the chances of achieving a useful speculation is small since other possibilities are eliminated immediately. In addition, with this semantics, cu does not make sense as cu may refer to a choice currently in many worlds and the system does not know which world to commit to.

A "less eager" kind of commit is *eager coordinated* commit. For a program $r ::= r_1 \oplus r_2$, if cm is reached in one of the worlds where r_1 is executed, then

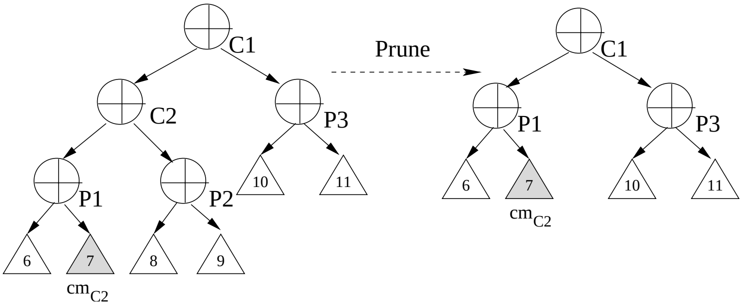


Fig. 5. Eager coordinated commit

all worlds associated with r_2 are deleted immediately (see Fig. 5). The idea here is that syntactically r_2 is not a viable choice any more. Conversely, if `cu` is reached by r_2 in any world, then all worlds associated with r_2 are deleted immediately. However, in either cases, r returns only after r_1 has committed in *all* the remaining worlds.

This type of commit is "eager" because commit in one world kills the alternative choice in all other worlds; it is "coordinated" as the program only returns after the choice can commit in all remaining worlds. The following example of two programs P and C shows a drawback of this semantics,

$$\begin{aligned} P &::= \ (+a; \texttt{cm}) \oplus (+b; \texttt{cm}) \\ C &::= \ (?a \Rightarrow -a; \texttt{cm}) \oplus (?b \Rightarrow -b; \texttt{cm}) \end{aligned}$$

Four worlds are created with these two programs. We denote the world in which the left branch of P and left branch of C interact as P_lC_l and likewise for other worlds. Assume nothing exists in the store initially, P produces a and b in all four worlds but hasn't committed. Now suppose C consumes a in P_lC_l and commits, which kills worlds P_lC_r and P_rC_r. However, as it turns out, P now commits in world P_rC_l first and kills P_lC_l, which renders C in a blocked state. Had C not killed P_lC_r and P_rC_r early, both P and C may commit in world P_rC_r.

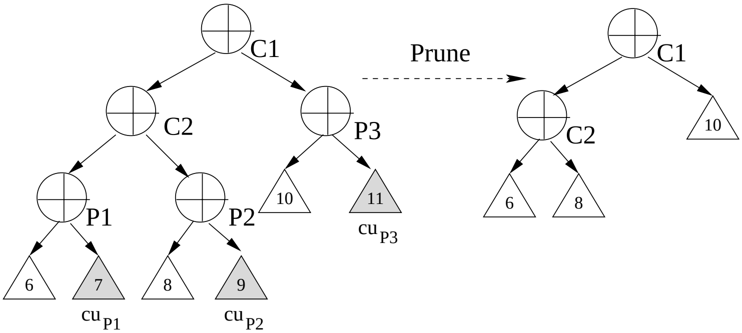


Fig. 6. Late coordinated commit

A more lazy form of commit is the *late coordinated* commit. Here the system only starts removing worlds when a program $r ::= r_1 \oplus r_2$ with a choice construct has committed its r_1 (or r_2) branch in *all* the worlds with which the r_1 (or r_2) branch continuations are associated. In other words, commits are not only coordinated within a scope, but also across all scopes this program was associated with. For example, in Fig. 6, worlds 7, 9 and 11 are only removed when a `cu` in program P has been executed in all these worlds. Note that worlds 7, 9, 11 are the only worlds in which the right branch of P exists.

While this semantics increases the possibility of getting a solution and decreases the chances of deadlock from a system point of view, it is unduly conservative in when to prune the tree which may cause the multi-world to be too large. One drawback is when one branch continuation of r cannot proceed to commit due to blocking, no worlds due to r can be removed from the multi-world.

The coordinated commit introduced in Section 4 can be considered as a compromise between the eager coordinated commit and the late coordinate commit. It does not kill the alternative choice until `cm` of this choice has been reached in all worlds in the scope of the choice construct. This solves the deadlock problem caused by eager coordinated commit, depicted in the example above. At the same time, it does not over-delay the removal of the worlds so that the size of the multi-world can be contained more effectively.

Finally, we remark that it is possible to have a mixed semantics for commit. Though we have presented one fixed commit semantics in Section 4, it is straightforward to extend it to several possible semantics. In a closed system, concurrently interacting programs can be determined and controlled, hence it makes sense for programs to specify their desired version of commit. In this paper, we consider the more general setting of an open system where agents can dynamically submit arbitrary programs. Here, having multiple commit semantics makes less sense because unpredictable behavior of the system can negate the extra program control associated with the mixed semantics. For example, a late committed choice could be killed by an eager commit issued elsewhere.

6 Empirics

The late choice in GCC allows for more kinds of reactive interactions which involve speculation. This extra expressive power naturally comes at a cost. When there are choices, this means that the choices in one program are multiplied with choices from other programs. This means that when choices do not or cannot commit for a long time, the multi-world can grow exponentially large. More worlds in the multi-world also mean more concurrency since there are more continuations. Furthermore, the space for the store might also increase with more worlds. We have developed the following implementation ideas which address the space and computation requirements.

Reducing storage — We define a *system root* to be the largest multi-world in the run-time system. Then a *differential view*, $\mathcal{DFV}$, of a non-system-root multi-world $\mathcal{W}_i \oplus_{id} \mathcal{W}_j$ is, $\mathcal{DFV}(\mathcal{W}_i) = \mathcal{CV}(\mathcal{W}_i) - \mathcal{CV}(\mathcal{W}_i \oplus_{id} \mathcal{W}_j)$. And for a system root $\mathcal{W}_0$, $\mathcal{DFV}(\mathcal{W}_0) = \mathcal{CV}(\mathcal{W}_0)$. This optimization re-organizes the multi-world so that, instead of having the data at the leaves, portions of the stores can be materialized in the internal nodes in the tree by using a differential view. In essence, this optimization stores common data as high as possible in the tree, to reduce storage redundancy. One can use a strategy that periodically materializes the $\mathcal{DFV}$ at respective nodes. This makes pruning more efficient as the common view can be assessed higher in the tree, without going to the leaves to compute the conjunction. For efficient evaluation of guard conditions, we can make use of materialized disjunctive view $\mathcal{DV}$ at internal nodes. $\mathcal{DV}$ can be used as an indexing condition to approximate whether a change in the view might wake up a blocked guard. However, disjunctive views are large and can be too expensive to materialize. We instead store a *common property*, $\mathcal{CP}$, of that view $\mathcal{DV}$, such that $\mathcal{DV} \models \mathcal{CP}$.

Reducing the number of continuations — Under some safety conditions, we can collect continuations which are identical copies (due to other programs splitting the worlds) from the leaves, treat them as one copy, which we call the *synchronous continuation* and execute it at a higher node in the multi-world tree. In other words, instead of running many copies of the same program on the leaves, we run just one copy in a higher internal node, and thus save computation. For example, in program

$$p ::= \texttt{while}(w > 0)\ \texttt{do}\ w := w - 1$$

if variable w is currently not speculated (i.e. has one value) and will not be updated by any other programs in future, then executing p at a node higher in the tree has the same effect as executing multiple instances of p at the leaves of the sub-tree.

Reducing the number of worlds — We keep sub-trees generated by each independent program in a "chain" form so long as the programs don't have data dependency on each other. If and when a data dependency is required, the linear tree can be expanded partially and on demand. The following property describes an ideal case of structure sharing.

Property 1. Given a set of programs R whose data requirements are *disjoint* from each other, the structure-shared GCC runtime structure is a linear ordering of $|R|$ subtrees, where each subtree represents the runtime structure of a respective program.

To investigate the implementation tradeoffs in applications with speculation, we experimented with simulations based on the producer and consumer problem. For simplicity, there are n_t types of resources being produced and consumed. The producer programs do not involve a choice, but just produce up to 3 items of the same type at a time. The consumer programs have one GCC choice, where each branch attempts to consume two items of the same type provided they are available. The consumer blocks until the items become available.

This simple setup replicates a basic model of an economy where agents can bundle long transactions and speculate on different possibilities. It is similar to what the wish list in real time transactions such as booking a holiday, i.e. buying the appropriate combination of air tickets, hotel rooms, ground transportation, etc.

Different choices arise due to different costs, routing possibilities, availability of the item, etc. For example, a different flight routing might necessitate a hotel room.

In the experiment, computations are executed by clock ticks. At each clock tick, either a producer program or a consumer program or no program is launched. At the same tick, every world in the system advances one step by picking one program (either producer or consumer) attached in this world and executes an instruction.

We identify two dimensions of simulation metrics:

- the level of overlapping interest by the producers and the consumers: high overlapping (HO) where $n_t = 5$ and low overlapping (LO) where $n_t = 100$.
- the relative rate of production against the consumption high production (HP), low(LP) and balanced production (BP)

We used six datasets (combinations of the above two dimensions, i.e. HO/HP, HO/BP, ...) with 50 producers and consumers randomly launched to the system with a random delay from 0 to 9 ticks between two consecutive program submissions. Due to lack of space, we only show two of the experiments, which are representative of the good cases where GCC does not incur much additional cost and the bad cases where GCC does lead to more overhead. Fig. 7 and Fig. 8 shows the resulting traces of three factors, i.e. total number of worlds in the multi-world, total number of program continuations and total number of data items stored, over a period of 500 time steps.

We can see that the optimizations we described are successful in controlling the size of the multi-worlds and the storage requirements. For high overlapping cases such as Fig. 7, the data storage is consistently low, and all programs run

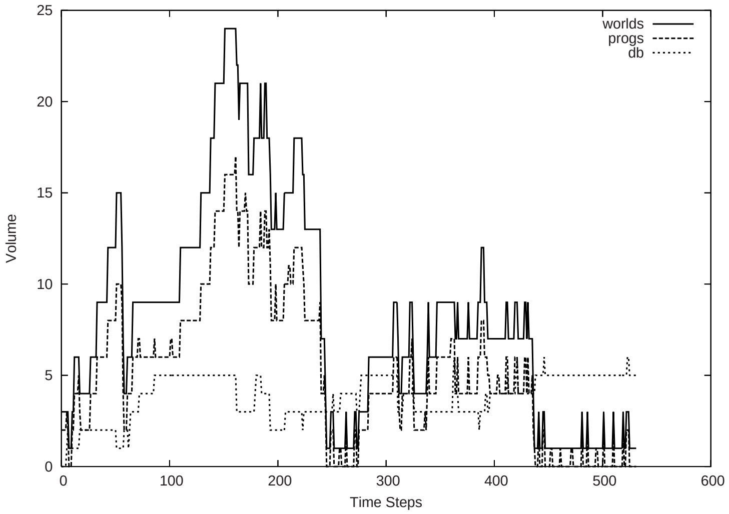


Fig. 7. Trace plot for HO/HP

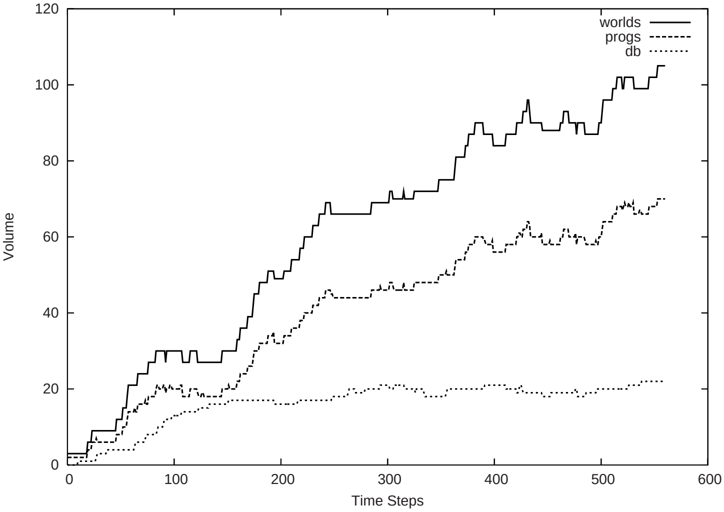


Fig. 8. Trace plot for LO/LP

to completion within 500 ticks given high and balanced production. In other more difficult cases like LO/LP in fig. 8, some of the programs were not able to complete due to scarcity of resources. The results demonstrate that in the expensive cases which might lead to exponential costs, the optimizations are effective in reducing the overheads. In the bad cases in our experiments, all three factors actually grow sub-linearly.

Table 1 records the maximum number of tree nodes, maximum number of program continuations and the maximum size of the data storage (in terms of total number of (variable, value) pairs stored) in each experiment.

The variation in the numbers in Table 1 has an easy and intuitive explanation due to the different nature of the datasets. The results show that balanced production/consumption gives the smallest tree size, and smallest number of programs. Excessive consumption and low overlapping interests result in larger tree and more computation since termination becomes less likely. When too many types of resources are produced, the mismatch between production and consumption also becomes more likely. These preliminary results suggest that

Table 1. Simulation results

	Max Worlds	Max Cont's	Max Data Storage
HO/HP	24	17	6
HO/LP	66	45	6
HO/BP	15	10	7
LO/HP	57	39	51
LO/LP	111	75	63
LO/BP	105	70	22

GCC with optimizations gives reasonable performance under realistic circumstances. They also demonstrate that even though GCC is potentially expensive, a "pay only when you use" principle is achievable.

References

1. Bonner, A.J., Kifer, M.: Transaction logic programming. Intl. Conf. on Logic Programming (1993)
2. Bonner, A.J., Kifer, M.: Concurrency and communication in transaction logic. Joint Intl. Conf. and Symp. on Logic Programming (1996)
3. Dayal, U., Hsu, M., Ladin, R.: Organizing long-running activities with triggers and transactions. ACM SIGMOD Conf. on Management of Data (1990)
4. Donnelly, K., Fluet, M.: Transactional events. In: Proceedings of ICFP (2006)
5. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
6. Garcia-Molina, H., Salem, K.: Sagas. ACM SIGMOD Conf. on Management of Data (1987)
7. Gupta, G., Pontelli, E., Ali, K.A.M., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: a survey. ACM Trans. on Programming Languages and Systems 23(4), 472–602 (2001)
8. Harris, T., Marlow, S., Jones, S.P., Herlihy, M.: Composable memory transactions, ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (2005)
9. Hoare, C.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1985)
10. Hull, M.E.C.: Occam - a programming language for multiprocessor systems. Comput. Lang. 12(1), 27–37 (1987)
11. Jaffar, J., Yap, R.H., Zhu, K.Q.: Coordination of many agents. Intl. Conf. on Logic Programming (2005)
12. Jaffar, J., Yap, R.H.C., Zhu, K.Q.: Indexing for dynamic abstract regions. Intl. Conf. on Data Engineering (2006)
13. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, part I/II. J, of Information and Computation 100, 1–77 (1992)
14. Nelson, G.: A generalization of dijkstra's calculus. ACM Trans. on Programming Languages and Systems 11(4), 517–561 (1989)
15. Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill, New York (2002)
16. Saraswat, V.A.: Concurrent Constraint Programming. MIT Press, Cambridge (1993)
17. Shapiro, E.Y., Takeuchi, A.: Object oriented programming in concurrent prolog. New Generation Comput. 1(1), 25–48 (1983)
18. Smolka, G.: The Oz programming model. Computer Science Today, pp. 324–343 (1995)
19. Ueda, K.: Guarded horn clauses. Intl. Conf. on Logic Programming (1986)
20. Vlahavas, I., Bassiliades, N.: Parallel, Object-Oriented, and Active Knowledge-Base Systems. Kluwer Academic Publishers, Dordrecht (1998)

Combining Formal Methods and Aspects for Specifying and Enforcing Architectural Invariants

Slim Kallel[1,2], Anis Charfi[1], Mira Mezini[1], and Mohamed Jmaiel[2]

[1] Software Technology Group
Darmstadt University of Technology, Germany
{kallel,charfi,mezini}@st.informatik.tu-darmstadt.de
[2] ReDCAD Laboratory
National Engineering School of Sfax, Tunisia
{slim.kallel@isecs,mohamed.jmaiel@enis}.rnu.tn

Abstract. Several types of invariants should be maintained when the architecture of a software application evolves. To specify these invariants in a reliable way, formal methods are used. However, current approaches suffer from two limitations. First, they support only certain types of invariants. Second, checking and enforcing the invariants is generally done by adding appropriate logic to the application implementation in a manual way, which is error-prone and may lead to architectural erosion.

In this paper, we combine the Z notation and Petri nets to specify formally architectural invariants in distributed object-oriented software applications. Moreover, we use a generative aspect-based approach to checking and enforcing these invariants. Thus, we bridge the gap between the formal specification and the implementation. Our approach brings several other benefits as the code that checks and enforces invariants is generated automatically and well-modularized in aspects.

1 Introduction

Architectural styles describe families of architectures by laying down the component/connector vocabulary, the topology for structuring components and connectors, and constraints to be satisfied by any (re-)configuration of the architecture.

The focus of this paper is on expressing and enforcing architectural constraints for distributed object-oriented software architectures. That is, the components in the subject systems are objects, whereby connectors can be method calls or events. More concretely, we target four kinds of semantic constraints:

1. Constraints on objects and object cardinality, e.g., constraints on the possible values of a certain object attribute, or on the total number of objects of a certain class in a system.
2. Constraints on object relationships and their cardinality, e.g., constraints on the number of objects of a class A that may be connected to an object of a class B.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 211–230, 2007.

3. Constraints on methods that modify the software architecture.
4. Method call protocols, i.e., constraints on the order in which methods that modify the architecture should be called.

We call constraints of the first two types architectural invariants whereas we call constraints of the third type pre- and post-conditions of architecture reconfiguration operations because they constrain operations that modify the software architecture. We call constraints of the fourth type coordination constraints as they constrain the ordering of architecture reconfiguration operations.

For illustration, consider the class of software systems supporting collaborative authoring of structured documents. Typically such a system is organized in a client/server style. Documents are located on a server internally structured as a set of non-overlapping sections. Clients connect to the server to view and edit these documents, at the granularity level of a section. Clients can have two different roles: *writers* can modify, create, and delete sections of a document, whereas *reviewers* can correct a section by adding annotations to it or by modifying the section formatting (i.e., they cannot edit the text but can change text fonts and colors).

In this application, several important architectural constraints should be defined and enforced. For example, two writers should not be allowed to connect simultaneously to the same section. Another constraint is that the number of writers that are connected simultaneously to the same document should not exceed a predefined value. A third constraint is that a reviewer can only connect to documents that were modified by at least one writer since the last review. Other constraints affect the order in which re-configuration operations are executed. For example, a writer should not be able to edit a document that was changed by another writer before a reviewer corrects that document.

Similar architectural constraints appear also in other applications with other architectural styles, e.g., a patient monitoring system defined according to the publish/subscribe architectural style. It allows nurses to control their patients remotely within a certain hospital department. In this application, a nurse can see the patient data (e.g., blood pressure, temperature, etc.) by periodically receiving events from the respective bed monitor. Moreover, a bed monitor can raise an alarm to the responsible nurse if the patient state becomes abnormal. In this application, several architectural constraints should be addressed. For instance, a nurse should not be assigned to more than five patients, otherwise, she will not be able to do her job appropriately, a patient should not be controlled by more than one nurse to avoid having several nurses called by the bed monitor when an alarm is raised, etc.

We propose an approach for expressing and enforcing architectural constraints that combines the strengths of Aspect-Oriented Programming (AOP) [1] with those of formal methods. On the formal methods side, we use Z [2] together with the Z-EVES tool [3] and Petri nets [4]. At the architectural specification level, formal methods are often used to specify such constraints and to ensure their consistency by proving theorems on the specified architecture. The particular use of Z and Petri nets that we propose has two main advantages over other formal specification methods as far as the particular needs of the kinds of architectural constraints that we address are concerned (we refer to Sec. 6 for a more elaborate discussion).

First, it allows us to cover all three kinds of architectural constraints in architecture specification: invariants, pre- and post-conditions of reconfiguration operations, and coordination constraints. Due to its set theoretic and predicate logic foundations, Z is well-suited for not only expressing constraints on the structure of individual components and on relations between objects in an object-oriented system [5]; it can also effectively be used to specify invariants that should hold for any dynamic re-configuration of the system. To the best of our knowledge, the latter feature of Z has not been exploited so far for architectural specifications. One of our contributions is using Z for covering both the invariants and the pre- and post-conditions of reconfiguration operations, which allows us to verify that reconfiguration operations do not break the architectural invariants. On the other side, Petri nets are well-suited to specify method protocols [6].

In addition to the particular combination of formal specification techniques to specify semantic architectural constraints on distributed object-oriented applications, this paper makes another important contribution: It presents a general schema for automatically generating AspectJ aspects [7] that enforce the architectural constraints.

Integrating logic for checking and enforcing constraints on object relations and coordination protocols is generally done by adding corresponding code to the affected components [8,9]. These approaches exhibit several limitations, which may ensue in a disconnection between the application implementation and its formal architecture specification, leading to architectural erosion [10].

The code that checks and enforces the architectural constraints is written manually in these approaches; besides being a tedious activity, such a manual translation has the more severe problem that there is no guarantee that the result actually conforms to the specification: The code that implements the constraints may contain contradictions that did not exist in the formal specification. This is accentuated especially by the fact that this code is not well-modularized, as it is tangled with the code implementing each component's core functionality and scattered across the implementation of different components. If the constraints change, e.g., to accommodate some changes in the requirements, all places in code affected by the implementation of the constraint enforcement logic need to be localized and accordingly changed, which is a tedious and error-prone process when the code to be changed is not well-localized.

A generative approach that compiles the formal specifications of the semantic architectural constraints into the component's code has the potential to avoid the risk of architectural erosion. In this paper, we propose an approach to compile Z and Petri net specifications into AspectJ code. Aspect-oriented programming is gaining popularity as a programming approach that supports the modularization of crosscutting concerns. The crosscutting nature is a feature that aspects share with specifications of architectural invariants in the focus of this paper, which makes them a natural choice for compiling such specifications into code. We propose generic compilation schemes from Z and Petri net specifications to AspectJ code. This process is automatic, whereby user input is expected for mapping high-level operation names from Z specifications to points in the implementation of the application.

Compared to other generative approaches that would compile formal specifications directly into individual component's code, the aspect-based generation has several advantages. The generation process is more direct and the resulting code is more reusable as it is well-modularized in aspects. Due to aspect generation, the approach as a whole can profit from advances in aspect-oriented language technology targeting faster aspect compilation time and runtime efficiency [11,12]. Also, by being kept explicit, the crosscutting structure in which the architectural invariants manifest themselves in code becomes subject to advanced tooling technology for aspect-oriented systems. Examples are the AspectJ development tooling (AJDT), which makes the crosscutting structure of aspects visible explicitly in the development environment, and aspect-oriented refactoring technologies [13].

On the other side, compared to writing architectural specifications directly as aspects [14], the combination of formal specifications and aspects has the advantage that the consistency of the specifications can be proved for correctness at the formal specification level, avoiding the risk of surprising aspect interactions at the code level [15].

The remainder of this paper is organized as follows. Section 2 provides some background information on the Z notation, Petri nets, and Aspect-Oriented Programming. Section 3 gives a short overview of our approach. In Section 4, we explain the formal specification of architectural invariants using Z and Petri nets. Section 5 describes the mapping of formal specification to aspect code and the automatic generation of aspects. Section 6 reports on related work and Section 7 concludes this paper.

2 Background

In this paper, we use Z and Petri nets for specifying software architectures and architectural invariants and Aspect-Oriented Programming for checking and enforcing these invariants.

2.1 The Z Notation and Petri Nets

The Z notation [2] is a formal specification language. Z defines a mathematical language, a schema language, and a refinement theory between abstract data types. The mathematical language is based on the set theory and on mathematical logic, i.e., first-order predicate logic. The schema language allows to describe the state of a system and how this state can change. The refinement theory allows to develop a system by building an abstract model from a system design. To edit and prove Z specifications, we use Z-EVES [3], which ensures the syntax and type checking, schema expansion, precondition calculation, and general theorem proving.

Petri nets [4] are a graphical and mathematical tool to model and analyze discrete systems. In Petri nets, the states of a system are modeled using *places* and *tokens*. The events are represented using *transitions* between places. To model coordination protocols with Petri nets, we use the tool *P3* [16], which supports the creation, the modeling of Petri net, and their export to XML.

2.2 Aspect-Oriented Programming

Aspect-Oriented Programming [1] is a programming paradigm, which supports the modularization of concerns that cut across the implementation of a software application, such as logging, persistence, and security.

AOP provides language means to separate the code that implements a crosscutting concern from the functional code of a software application. Using AOP, an application consists of two parts: The *base program*, which implements the core functionality, and the *aspects*, which implement the crosscutting concerns. Aspects are new units of modularity, which encapsulate crosscutting concerns in complex systems by using *join points*, *pointcuts*, and *advice*.

Join points are well-defined points in the execution of a program. In AspectJ [7], which is an aspect-oriented extension to Java, join points correspond to e.g., method calls, constructor calls, field read/write, etc. The pointcut allows to select a set of join points, where some crosscutting functionality should be executed.

The advice is a piece of code that implements a crosscutting functionality, which can be associated with a pointcut. The advice is executed whenever a join point in the set identified by the pointcut is reached. It may be executed before, after, or instead of the join point at hand; this corresponds respectively to the advice types *before*, *after* and *around* in AspectJ. With an around advice, the aspect can integrate the further execution of the intercepted join point in the middle of some other code using the keyword *proceed*.

3 The Approach in a Nutshell

Our approach presumes a three-step process to developing distributed object-oriented applications, as schematically shown in Figure 1.

In the first step, the developer specifies formally the architecture of the application in terms of components and relations as well as constraints that should be satisfied when the architecture evolves.

In the second step, the developer provides the functional code of the object-oriented application in Java. This code provides only the core functionalities of the application and does not enforce any architectural constraints. For instance, the functional code of the collaborative authoring application provides methods for opening a document or for changing the font of a certain section but it does not include code that enforces architectural constraints such as avoiding overlaps between sections.

In the third step, the developer defines a mapping between the formal specification of the different architecture reconfiguration operations and the implementation of the application. This mapping will be used by a generator, which emits a set of AspectJ aspects that enforce the formally specified invariants.

The functional code should conform to the formal specification of the components and their relations. For example, if the formal specification states that a certain component has some attribute, then the class that implements that component should have a field that matches that attribute. However, enforcing the formal specifications pertaining to individual components is out of the scope of this paper. Our focus is rather on the enforcement of architectural cross-component invariants.

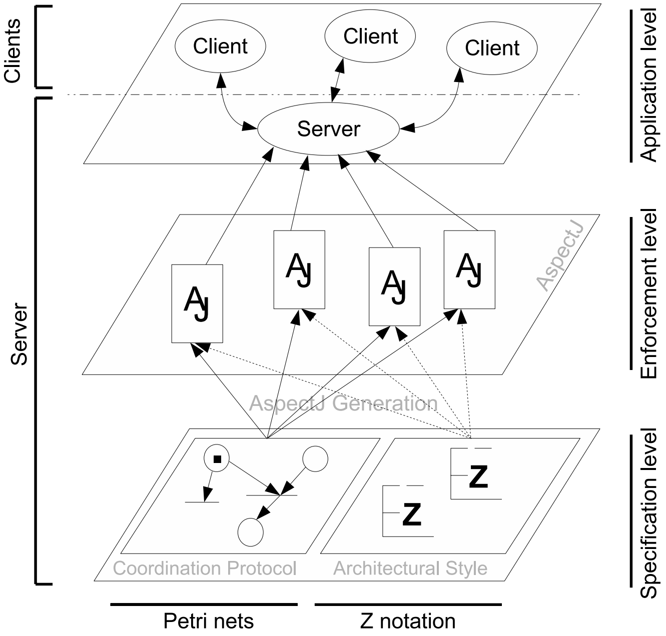


Fig. 1. Overview of the three-level architecture

4 Formal Specification Using Z and Petri Nets

This section presents the formal specification step using the collaborative authoring system shortly presented in Sec. 1 for illustration. Three kinds of specifications are defined and verified in this phase. First, the structure and the behavior of the individual components as well as the overall architecture of the system is specified and validated. Second, pre- and post-conditions for reconfiguration operations are defined and verified to ensure that the specified architectural style is maintained as the system evolves at runtime. The Z notation is used for both kinds of specifications so far. Finally, valid sequences of reconfiguration operations are specified using Petri nets. We shortly review the Z notation and Petri net when needed.

4.1 Overall System Specification and Verification

Predicate logic is used to specify static and dynamic properties of the individual components participating in the system following the component specification template shown below. In this template, att_i denotes an attribute, Spr_i and Dpr_i denote static, respectively dynamic properties of a component.

$$\begin{array}{|l}
\text{\_}\ Component_i\ \text{\_\_\_\_\_\_\_\_\_\_\_\_} \\
att_1 : Type_1, att_2 : Type_2, ..., att_n : Type_n \\
\hline
Spr_1, ..., Spr_n \\
Dpr_1, ..., Dpr_n \\
\hline
\end{array}$$

As an example, consider two kinds of components in our collaborative authoring system: shared documents and sections. A shared document is accessible to any client that is authorized either as a Writer or as a Reviewer. A section is defined by the position of its first and last characters in the whole document. In the schema below, the shared document is defined as a sequence of sections that do not overlap, as specified by the predicate in the lower part of the specification of a shared document.

$$\begin{array}{|l}
\text{\_}\ Section\ \text{\_\_\_\_\_\_\_\_\_\_\_\_} \\
firstCharacter : \mathbb{N} \\
lastCharacter : \mathbb{N} \\
\hline
lastCharacter \geq firstCharacter \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\text{\_}\ SharedDoc\ \text{\_\_\_\_\_\_\_\_\_\_\_\_} \\
sections : \operatorname{seq} Section \\
\hline
\forall\, i : \mathbb{N} \mid 1 \leq i < \#sections \\
\quad \bullet\ (section(i+1)).firstCharacter \\
\qquad = (section(i)).lastCharacter + 1 \\
\hline
\end{array}$$

The overall system specification defines a set of components, the relationships between them, and the architectural constraints that must be maintained when the system evolves. In the sample *System* schema shown below, c_i denotes a component instance, $Component_i$ denotes a component type, $relation_{ij}$ denotes a relation between the $Component_i$ and the $Component_j$ (as represented by the bidirectional arrow), and Apr_i denotes an architectural constraint. To verify the consistency of the system specification, it should be ensured that at least one valid initial state exists. Using Z-EVES, one can define an *InitialisationTheorem* and prove it, whereby *System* represents the system schema and *SystemInit* corresponds to a Z schema that describes the initial system state.

System

$c_i : Component_i;...$

$c_j : \mathbb{F}\ Component_j;...$

$relation_{ij} : Component_i \leftrightarrow Component_j;...$

$Apr_1, ...Apr_n$

Theorem *InitialisationTheorem*
$\exists\ System \bullet SystemInit$

For illustration, the system *CollaborativeAuthoringSystem* is specified in the schema below. It consists of finite sets ($\mathbb{F}$) of writers and reviewers, a shared document and relations between authorized writers/reviewers and sections of the shared document. Conditions on the relations are preserved by verifying the domain *dom* and the range *ran* of each relation. For illustration, also constraints C1 and C2 are given: C1 states that a writer or a reviewer can be connected to only one section at any point in time and C2 states that two actors (writers or reviewers) are never connected simultaneously to the same section. These constraints should be satisfied by any operation that changes the sets of writers or reviewers.

CollaborativeAuthoringSystem

$writers : \mathbb{F}\ Writer$

$reviewers : \mathbb{F}\ Reviewer$

$sharedDoc : SharedDocument$

$WriterSection : Writer \leftrightarrow Section$

...

$\mathrm{dom}\ WriterSection \subseteq writers$

$\mathrm{ran}\ WriterSection \subseteq \{s : Section \mid s \in \mathrm{ran}\ sharedDoc.section\}$

$\forall\ w : writers \bullet \#(WriterSection(\!|\ \{w\}\ |\!)) \leq 1$ [C1]

$\forall\ r : reviewers;\ w : writers;\ s : Section$
$\quad \mid s \in \mathrm{ran}\ sharedDoc.section$ [C2]
$\quad \bullet (r, s) \notin ReviewerSection \vee (w, s) \notin WriterSection$

....

To verify the consistency of the *CollaborativeAuthoringSystem* schema, we define an initial system state, *InitiCASystem*, shown below. The initial state consists of two writers $w1$, and $w2$, one reviewer $r1$, and a shared document sd that consists of three sections $s1, s2, s3$. The proof of the consistency theorem ensures that the specification of our collaborative authoring system is consistent and does not contain any contradictions.

InitCASystem

CollaborativeAuthoringSystem

$writers = \{w1, w2\}$

$reviewers = \{r1\}$

$sharedDoc = sd$

$WriterSection = \{(w1, s1), (w2, s3)\}$

$ReviewerSection = \{(r1, s2)\}$

Theorem *ConsistencyCASystem*
$\exists\ CollaborativeAuthoringSystem$
$\bullet\ InitCASystem$

4.2 Specification and Verification of Reconfiguration Operations

In this step, architectural reconfiguration operations are specified formally. Each reconfiguration operation is specified by means of a Z *operation schema*, which defines the input parameters ($c_i?$) as well as the pre- and post-conditions. These conditions are essential to verify that the evolution of the architecture preserves certain invariants. Reconfiguration operations are executed only if their pre-conditions are satisfied. In the operation schema below, $PreCond_i$ and $PostCond_i$ denote a pre- and a post-condition of the reconfiguration operation $Operation_i$.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Operation_i}} \\
\Delta \mathit{System} \\
c_i? : \mathit{Component_i}; ... \\
\hline
\mathit{PreCond_i}, ..., \mathit{PreCond_n} \\
\mathit{PostCond_i}, ..., \mathit{PostCond_n} \\
\hline
\end{array}$$

Theorem *PreCondTheorem*

$$\begin{array}{l}
\quad \forall \mathit{System} \land c? : \mathit{Component_i} \\
\qquad \mid \mathit{preConditions} \bullet \mathrm{pre}\ \mathit{Operation_i}
\end{array}$$

After specifying the reconfiguration operations formally, these operations need to be verified. To evaluate the impact of a reconfiguration operation on a constraint, we define and prove the theorem *PreCondTheorem* shown on the right-hand side of the specification above. This theorem states the pre-conditions that must initially be satisfied to guarantee that the constraints are preserved after the execution of the operation and verify that the execution of the reconfiguration operation preserves the architectural style.

Let us now illustrate the approach to specifying reconfiguration operations by means of our collaborative authoring system. We have specified and validated formally reconfiguration operations such as the insertion and connection of writers, reviewers, and sections. For illustration, the following schema specifies the operation *ConnectWriter*. The operation schema states that when a writer $w?$ is connected to a section of the shared document, then it should be one of the writers that are already present in the system and the section $s?$ should be already created. In order to validate the connection operation of a new writer, we use Z-EVES to prove the theorem *PreConnectWriter*, which ensures that the connection of a writer conforms to the system constraints described in the system schema *CollaborativeAuthoringSystem*. The theorem below states that the connection of a writer to a section requires that no reviewer is connected to that section.

$$\begin{array}{|ll}
\multicolumn{2}{l}{\mathit{ConnectWriter}} \\
\Delta \mathit{CollaborativeAuthoringSystem} & \\
w? : \mathit{Writer} & \\
s? : \mathit{Section} & \\
\hline
w? \in \mathit{writers} & [\mathsf{C3}] \\
s? \in \mathit{sections} & [\mathsf{C4}] \\
\mathit{WriterSection}' = & \\
\mathit{WriterSection} \cup \{(w?, s?)\} & \\
...... & \\
\hline
\end{array}$$

Theorem *PreConnectWriter*

$$\begin{array}{l}
\forall \mathit{CollaborativeAuthoringSystem}; \\
\quad w? : \mathit{Writer};\ s? : \mathit{Section} \\
\quad \mid w? \in \mathit{writers} \land s? \in \mathit{sections} \\
\qquad \land (\forall r : \mathit{reviewer} \\
\qquad\quad \bullet (r, s?) \notin \mathit{ReviewerSection}) \\
\qquad\quad ... \\
\qquad \bullet \mathrm{pre}\ \mathit{ConnectWriter}
\end{array}$$

4.3 Specification of Valid Protocols

In this step, Petri nets are used to define constraints on the execution order of reconfiguration operations that are already specified in Z. Typical coordinations in distributed object-oriented applications such as synchronization, mutual exclusion, conflicts, etc., can be naturally specified with Petri nets. We model each reconfiguration operation by a transition and the system state by a set of places and tokens. Enabling a transition in the Petri net means that the corresponding reconfiguration operation conforms to the constraints given the current system state. Consequently, the transition can be executed and its target places can be occupied by tokens.

In our collaborative authoring system, the writers can create, modify, and delete sections. Then, the reviewers can correct these sections and add annotations. To enforce the activity order described above, we define a coordination protocol, which requires that each section must be created or modified by a writer before it becomes accessible to reviewers for correction. In addition, after a section is corrected, the next reviewer cannot revise it before an author modifies it.

In the initial state that is shown in Fig. 2, the transitions *InsertWriter* and *InsertReviewer* are always enabled. Consequently, the transition *ConnectWriter* will be enabled. Thus, a writer can connect to a section. After the enabling of the transition *DisconnectionWriter*, the writer can be deleted but she cannot connect because there is no token in $P8$. However, a reviewer can still connect because the transition *ConnectReviewer* is enabled.

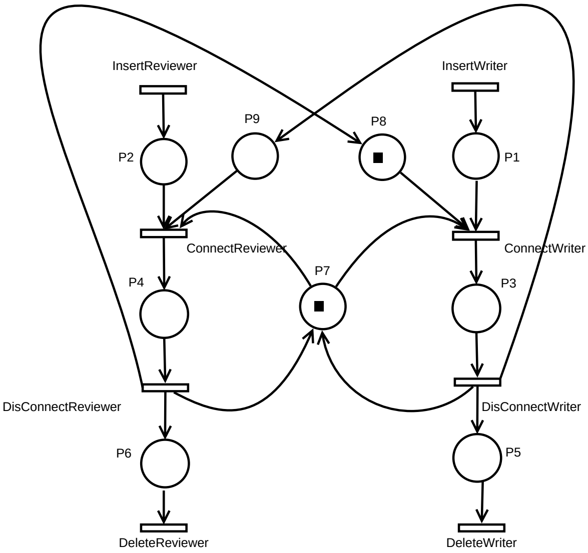


Fig. 2. A Petri net example

5 Mapping Formal Specifications to Code

This section describes how formal specifications are mapped to code. The mapping of the specifications pertaining to individual components and to the overall system architecture are done manually. The focus of this paper is on mapping cross-component specifications related to reconfiguration operations and their valid protocols. This mapping is done by translating specifications to aspects in the AspectJ language. In the following, we first present the structure of the generated aspects and subsequently describe the generation process.

5.1 Aspects That Implement Cross-Component Invariants

The layer that is responsible for checking and enforcing architectural constraints at runtime consists of a set of aspects that are generated from Z operation schemes and corresponding Petri net transitions. There is one aspect per reconfiguration operation, i.e., per each pair of Z operation schema and corresponding Petri net transition. For illustration, consider again our collaborative authoring system. Each reconfiguration operation, e.g., insert writer, connect reviewer, delete section, etc., corresponds to a Z operation schema in the formal specification of the architectural style and to a Petri net transition in a coordination protocol. For each reconfiguration operation, an AspectJ aspect is automatically generated. In other words, the aspects connect the formal level and the functional level in order to ensure that formally specified invariants are maintained when the architecture evolves.

The overall structure of these aspects is as follows. The pointcuts of the generated aspects intercept the execution of operations from the functional layer that correspond to architecture reconfiguration operations. For example, every call to the operation `insertW` in the public interface of the class `Writer` corresponds to the reconfiguration operation `InsertWriter`.

The advice of the generated aspects first checks and enforces the coordination protocols that are defined by Petri net transitions and then checks the constraints specified in Z. The first check verifies whether the transition in the Petri net is enabled based on the current system state. The second check verifies the constraints and pre-conditions corresponding to the reconfiguration operation that are specified within the style schema and the operation schema. If both checks are successful, the aspect executes the reconfiguration operation, and after that updates the state of the system. Otherwise, the aspect prohibits the execution of the reconfiguration operation.

For illustration, consider the aspect shown in Listing 1.1, which controls the connection of a writer to a section. This operation is specified by the Z operation schema *ConnectWriter* and the Petri net transition with the same name. The pointcut *ConnectWriterToSection* (line 3 and 4 in Listing 1.1) selects all calls to any public method *connectW* independently of its parameters and their types. The constructs *target* and *args* are used to bind the target, respectively the arguments, of the method call join points that are selected by the pointcut to the variables *w: Writer* and *s: Section* respectively.

```
 1 public aspect EnforceConstraintsForConnectWriter {
 2
 3      pointcut ConnectWriterToSection(Writer w,Section s):
 4                      call(public * connectW(..)) && target(w) && args(s);
 5
 6      void around(Writer w, Section s): ConnectWriterToSection(w,s)
 7      {
 8          ...
 9          if (isTransitionEnabled (CurrentMarking, ConnectWriterTransistion)) { ...}
10
11          ...
12          if (isMemberOf(wInput,SystemState.writers)) { ... } //Constraint C3
13          if (isMemberOf(sInput,SystemState.sections)) { ... } //Constraint C4
14          ...
15
16          if (checkConstraintC1()) { ... }
17          if (checkConstraintC2()) { ... }
18          ...
19
20          if (allConstraints) {
21             proceed(w,s);
22             updateSystemState(w,s);
23           }
24           else { ... }
25         }
26
27      boolean checkConstraintC1() { ... }                 //Constraint C1
28
29      boolean checkConstraintC2() {                       //Constraint C2
30           boolean constraints, result0, result1, result2 = true;
31           String [] Tab0 = SystemState.Newreviewers;
32           while ((Tab0.length!=0) && result0) {
33               String r = getFirstElement(Tab0);
34               String [] Tab1 = SystemState.Newwriters;
35           }
36           while ((Tab1.length!=0) && result1) {
37               String w = getFirstElement(Tab1);
38               String [] Tab2 = SystemState.Newsections;
39           }
40           while ((Tab2.length!=0)&& result2) {
41               String s = getFirstElement(Tab2);
42               constraints = Or(isNotMemberOf(w, s,SystemState.NewWriterSection),
43                                isNotMemberOf(r, s,SystemState.NewReviewerSection));
44               result2 = result2 && constraints;
45               result1 = result1 && constraints;
46               result0 = result0 && constraints;
47            }
48            return result0;
49       }
50
51       void updateSystemState(String NameW, String NameS) { ... }
52 }
```

Listing 1.1. Example of the AspectJ Code generated

The around advice of the aspect in Listing 1.1 controls the execution of the reconfiguration operation *ConnectWriter*. First, the advice contains Java code to ensure that the coordination protocols are respected. For example, we specified in the Petri net shown in Figure 2 that a writer cannot modify a section unless a reviewer has corrected it. The generated advice contains a call to the method *isTransitionEnabled* (line 9) to check whether the transition *ConnectWriter* is enabled given the current state of the system. Next, Z constraints are checked. For example, the constraints C3 and C4 from the Z operation schema *connectWriter* are translated to calls to the method *isMemberOf* of a Z operator package that we implemented (lines 12 and 13 in the Listing 1.1). Quantified constraints such as the constraint C1, which ensures that a writer or reviewer can modify only one section at a given point in time, and the constraint C2, which disallows overlaps between sections, are translated to Java code by using helper methods. For instance, a helper method *checkConstraintC*2 (lines 29–49) is generated to evaluate the constraint C2. This helper method is called in the advice (line 17).

Moreover, the advice contains a method *updateSystemState* (lines 51) that updates the system state that is stored in a generated class, which represents the components of the system, their relationships, and the marking of the Petri net. If all generated constraints are evaluated to true, the reconfiguration operation will be executed (using *proceed*) and the system state will be updated (lines 20–24).

5.2 The Aspect Generation Workflow

There are three inputs to the aspect generator. In addition to Z and Petri net specifications, the generation process also takes a mapping of the reconfiguration operations to points during the execution of the functional code as an input. This mapping is provided by the developer using an appropriate XML file. For illustration, Listing 1.2 shows the mapping of the reconfiguration operation *ConnectWriter*, defined in the formal specification, to calls to the method *connectW* (lines 5–7 in the Listing 1.2) from the functional level.

```
1  <Mapping Name="CollaborativeAuthoringSystem">
2  <PointCut Reference="InsertWriter">
3        InsertW(Writer w):
4            call(public * insertW(..))&& target(w); </PointCut>
5  <PointCut Reference="ConnectWriter" Component="Section">
6        ConnectWriterToSection(Writer w,Section s):
7            call(public * connectW(..)) && target(w) && args(s); </PointCut>
8  ...
9  </Mapping>
```

Listing 1.2. Mapping formal specification to code

To automatically translate Z specifications to aspects, the structure of the specifications must satisfy the following properties:

1. The system must be specified by a Z schema that consists of a finite set ($\mathbb{F}$) or a sequence of component instances (seq).
2. Each component must be specified by a Z schema that defines a unique identifier of the component and one or more constraints on it.
3. A component can be atomic or composite (composed by a set or sequence of sub-components).
4. Each component must be connected at least to another component.
5. Connections must be specified using Z relations ($\leftrightarrow$) and the domain (dom) and range (ran) of each relation must be specified in the system schema.
6. The reconfiguration operations must be specified using Z operation schemes.

These properties are specified in a meta-model for Z specifications available to the generation process as an XML Schema Definition (XSD). To verify that the Z specification is compliant with this meta-model, we translate the structure of the Z specification into an XML file and use DOM and SAX to verify that the XML representation of the Z specification satisfies the XSD of the meta-model.

The code generator gathers all necessary information to generate the aspects. It extracts the properties of the components, the relations between components, and the architectural constraints which are specified in the system schema and in the operation schemes. For each reconfiguration operation, which corresponds to a Z operation schema and/or to a transition in the Petri net, the pre-conditions are translated to *around advice*, which coordinate and control their execution, as schematically shown in line 5 of Listing 1.3. The generated advice is associated with a *pointcut* that is provided by the user in the XML mapping file (cf. line 2 in the Listing 1.3).

```
 1     //extract the pointcut from the mapping file
 2      public pointcut PointCutName (parameters1): set of join points
 3
 4     //The around advice for controlling the reconfiguration operation
 5      void around(parameters1): PointCutName(parameters1) {
 6
 7         // check if the Petri net transition is enabled
 8         if (isTransitionEnabled (CurentMarking, respectiveTransistion)) { ...}
 9
10         // evaluate the constraints defined in the operation schema
11         if (ZOperators(parameters2, SystemState)) { ... }
12
13         // evaluate constraints without quantification defined in the system schema
14         if (ZOperators(parameters3, SystemState)) { ... }
15
16         // call the auxiliary method for evaluating constraints with quantification
17          ...
18          if (checkConstraintCi()) { ... }
19          ...
20
21         // verify that all constraints hold and proceed if this is the case.
22          if (allConstraints) {
23              proceed(parameters1);
24              updateSystemState(parameters4);
25           }
26           else { ... }
27      }
28
29     // helper method for evaluating constraints with quantification
30     public boolean checkConstraintC_i() { ... }
31
32     // the method for updating the system state
33     void updateSystemState(parameters4) { ... }
34 }
```

Listing 1.3. Template of aspect generation

The workflow of the generated around advice is as follows. The advice first checks whether the Petri net transition for the corresponding reconfiguration operation is enabled. For that purpose, the generated advice contains a call to the method *isTransitionEnabled* (line 8 in the Listing 1.3). Next, the advice checks whether all pre-conditions of the corresponding reconfiguration operation are fulfilled. The constraints specified in the operation schema (line 11) and the constraints without quantification (line 14) specified in the system schema are evaluated first. Next, constraints that use quantification operators are evaluated by calling auxiliary methods generated for them (e.g., method *checkConstraintC$_i$* in Listing 1.3). If one of the generated pre-conditions is not fulfilled the operation will not be executed (lines 22-26). Otherwise, the operation will be executed by calling *proceed* (line 23).

The translation of the pre-conditions of reconfiguration operations into Java code makes use of a Java-based package of Z that we developed. This package contains classes representing the elements of the Z language such as operators, mathematical objects such as sets, relations, sequences, bags, etc.

In addition to aspects, the generator emits a Java class that stores the current state of the system in terms of components and relations between them. Each generated aspect implements a method *updateSystemState* (line 33), which updates the current state of the system, if the reconfiguration operation can be executed. This method is called in the aspect after *proceed* (line 24).

6 Related Work

In this section, we report on works that address the formal specification of software architectures as well as the generation of aspects and code from formal specifications.

6.1 Formal Specification of Software Architecture

Architecture description languages [17] emerged as modeling notations to support architecture-based development of software applications. However, ADLs support only the specification of architecture invariants as constraints on components and connections. They are not suited for describing the pre- and post-conditions of architecture reconfiguration operations as shown in [18]. In addition, most ADLs lack of formal foundation.

Formal methods were used to specify several constraints in software architecture such as invariants, pre- and post-conditions, and coordination constraints on architecture reconfiguration operations (i.e., specification of the execution order of reconfiguration operations). We classify works on formal specification of software architectures according to the used techniques into three classes: based on logic, on graphs, and on process algebras.

Some works proposed *logic-based* approaches such as first-order logic [19], Z [20], and temporal logic [21]. The first-order logic covers only the architecture invariants and pre- and post-conditions of architecture reconfiguration operations, whereas temporal logic can express at a very high level some coordination properties and temporal constrains. In [20], Gregory et al. propose using Z to analyze the architecture styles and the relation between them. However, this work does suports the expression of constraints on the evolution of the architecture. Based on the schema and mathematical language of Z, we specify in our approach both the the invariants and the pre- and post-conditions of reconfiguration operations. Thus, we can verify formally that the invariants are maintained when the architecture evolves.

Other approaches use *graph-based* formalisms such as graph grammars [22] and typed multi-graphs. In [22], Le Métayer uses a graph grammar, which is based on a mathematical model, to specify software architectures formally. This type of grammars does not support logical properties such as reasoning about the number of component instances and logical conditions such as absence of a communication link between two components. Generally, constraints on the evolution of the architecture are specified by means of graph rewriting rules to represent the reconfiguration, or by the chemical metaphor in the case of CHAM approach (Chemical Abstract Machine) [23]. Some other works used *process algebras*, e.g., CSP [24] and the π-calculus [25] to model architectural dynamism with mobile processes.

The approaches mentioned above do not support code generation from specifications. Moreover, they cover only one aspect in software architecture. To address this limitation, other proposals combined more than one formal method. In the following, we focus on work that combines Z and Petri nets.

Z and Object-Z[1] were combined with different process algebras, e.g., Z/CCS [26], Z/CSP [27], and OZ/CSP [28]. In these works, Z and Object-Z are used to describe the invariants and process algebras to describe the evolution of architecture. $\mu\mathcal{SZ}$ [29] combines Z with a variant of state charts for specifying the architecture evolution and a temporal logic (temporal discrete logical of intervals) for checking the safety properties and describing some duration constraints. The previous approaches do not address the coordination aspect in the evolution of software architectures (method protocols). Moreover, they do not propose any formal solution to check if the architectural constraints specified in Z are preserved when the architecture evolves. In our approach, we can specify and prove the pre-condition theorem by using the Z-EVES theorem prover. Thus, we can ensure that the reconfiguration operations preserve the architectural invariants that are specified in the system schema.

SAM [30] (Software Architecture Model) is a general software architecture development framework based on Petri nets and temporal logic. Petri nets are used to visualize the structure and model the coordination aspect of software architectures while temporal logic is used to specify the required properties of the software architecture. This work is related to ours as it uses temporal constraints to specify the system properties. However, Z, which is based on predicate logic and set theory allows lower level description of architectural invariants. To support the temporal properties in our approach, we plan to introduce a linear temporal logic extension of Z [31].

In [32], colored Petri nets[2] are combined with Past Predicate Temporal logic to specify the structure and behavior of a system. These formalisms are used to decide on when a system can run or when it can evolve depending on its function and structure in the past. That is, the evaluation of the pre-conditions is based

[1] Object-Z is an object-oriented extension of the formal specification language Z.

[2] Colored Petri nets is a high level extension of petri nets.

on the actions, which are saved in the system history. This approach covers partially the three aspects does not provide any tool to specify and validate specifications.

6.2 Code Generation from Formal Specification

The translation of formal specifications to code has been addressed in several works. In [33], Ramkerthik and Zhang discuss the generation of Java code with design contracts from an Object-Z Specification. This work translates the structure of Object-Z specifications to XML and then generates a Java skeleton by processing the XML representation. However, it translates only the simple predicates and the generated skeletons are not executable. Jia and al. [34] propose an approach to synthesizing functional code from UML and Z. They translate UML models into a Z specification, which is used to generate C++ code. This work is related to ours, but it generates the pre-conditions code in the functional code, which poses several modularity problems.

In more recent works, formal specifications were translated into aspect code. In [35], Bodden proposes a linear-time logic over join points to verify, during the program execution, the temporal properties of certain actions (e.g., a temporal relationship between two methods calls) by alternating a finite state whose transitions are triggered through generated aspects. Unlike our approach, the work of Bodden focuses only on temporal dependencies and does not target software architectures and their evolution.

In [36], multi-modal scenario-based specifications using Live Sequence Chart (LSC) are translated to Scenario Aspects that are implemented in AspectJ. This work specifies the mandatory, possible, and negative scenarios executed between inter-objects and enforces their execution using AspectJ aspects. This work specifies and enforces the inter-objects coordination constraints but it is not concerned with system constraints such as the ones expressed in our approach using Z.

7 Conclusion

In this paper, we presented an approach to compiling formal specifications of architectural invariants in Z and Petri nets into AspectJ code. The combination of Z and Petri nets makes our approach reliable and allows us to specify invariants, pre- and post-conditions, and coordination constraints on architecture reconfiguration operations. Nonetheless, the more important contribution of our work is the usage of an aspect-based enforcement layer, which allows for a modular implementation of the code that checks and enforces architectural constraints. The automatic generation of this code makes our approach especially user-friendly and bridges the gap between the implementation of the application and its formal specification.

However, our approach has some limitations, which we will address in future work. First, as we use petri nets, we support only applications with a finite state. To address this problem, we will extend Z with support for linear temporal logic. Second, the programming model introduced by our approach is relatively complex as we expect the user to know Java, Z, and Petri nets. We believe that the benefit make the increased programming model complexity worthwhile. Yet, a systematic investigation of the trade-off remains for future work. Third, the aspects resulting from generation are complex and extra code is needed to explicitly keep track of system state. This is because of the limited power of querying capabilities of AspectJ. More expressive pointcut languages such as Alpha [37] support a more declarative quantification over execution history and object heap. By using them, we hope to achieve a more direct mapping of formal specifications to aspect code. Such pointcut languages are also less fragile with respect to syntactic changes.

References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Weske, M., Liggesmeyer, P. (eds.) NODe 2004. LNCS, vol. 3263, pp. 220–242. Springer, Heidelberg (1997)
2. Spivey, M.: The Z notation: a reference manual, 2nd edn. Prentice Hall International Ltd, Hertfordshire, UK (1992)
3. Meisels, I., Saaltink, M.: The Z/EVES Reference Manual (for Version 1.5). Reference manual, ORA Canada (1997)
4. Petri, C.A.: Kommunikation mit Automaten. PhD thesis, Darmstadt University of Technology, Darmstadt, Germany (1961)
5. Jacky, J.: The Way of Z: Practical Programming with Formal Methods. Cambridge University Press, Cambridge (1997)
6. Agha, G.A., De Cindio, F., Rozenberg, G. (eds.): Concurrent Object-Oriented Programming and Petri Nets. LNCS, vol. 2001. Springer, Heidelberg (2001)
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–353. Springer, Heidelberg (2001)
8. Pellegrini, M.C., Riveill, M.: Component Management in a Dynamic Architecture. The Journal of Supercomputing 24, 151–159 (2003)
9. Georgiadis, I., Magee, J., Kramer, J.: Self-organising software architectures for distributed systems. In: Proceedings of the first workshop on Self-healing systems, pp. 33–38. ACM Press, New York (2002)
10. Medvidovic, N., Egyed, A., Gruenbacher, P.: Stemming Architectural Erosion by Coupling Architectural Discovery and Recovery. In: Proc. of the 2nd International Software Requirements to Architectures Workshop, Portland, Oregon (2003)
11. Bockisch, C., Kanthak, S., Haupt, M., Arnold, M., Mezini, M.: Efficient control flow quantification. In: Proc. of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 125–138. ACM Press, New York, NY, USA (2006)
12. Bockisch, C., Arnold, M., Dinkelaker, T., Mezini, M.: Adapting virtual machine techniques for seamless aspect support. In: Proc. of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 109–124. ACM Press, New York, NY, USA (2006)
13. Laddad, R.: Aspect Oriented Refactoring. Addison-Wesley Professional (2006)

14. Shomrat, M., Yehudai, A.: Obvious or not?: regulating architectural decisions using aspect-oriented programming. In: Proc. of the 1st international conference on Aspect-oriented software development, pp. 3–9. ACM Press, New York, NY, USA (2002)
15. Douence, R., Fradet, P., Sudholt, M.: A Framework for the Detection and Resolution of Aspect Interactions. In: Batory, D., Consel, C., Taha, W. (eds.) GPCE 2002. LNCS, vol. 2487, pp. 173–188. Springer, Heidelberg (2002)
16. Gasevic, D., Devedzic, D.: Software Support for Teaching Petri Nets: P3. In: Proc. of the 3rd IEEE International Conference on Advanced Learning Technologies, Athens, Greece, pp. 300–301. IEEE Computer Society, Washington, DC, USA (2003)
17. Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26, 70–93 (2000)
18. Kacem, M.H., Jmaiel, M., Kacem, A.H., Drira, K.: Evaluation and Comparison of ADL Based Approaches for the Description of Dynamic of Software Architectures. In: Proc. of the Seventh International Conference on Enterprise Information Systems, Miami, USA, pp. 189–195 (2005)
19. Endler, M., Wei, J.: Programming generic dynamic reconfigurations for distributed applications. In: Proc. of the International Workshop on Configurable Distributed Systems, pp. 68–79 (1992)
20. Abowd, G., Allen, R., Garlan, D.: Formalizing style to understand descriptions of software architecture. ACM Transactions on Software Engineering and Methodology 4, 319–364 (1995)
21. Aguirre, N., Maibaum, T.: A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In: Proc. of the 17th IEEE International Conference on Automated Software Engineering, Edinburgh, Scotland, UK, pp. 271–274. IEEE Computer Society, Los Alamitos (2002)
22. Métayer, D.L.: Describing software architecture styles using graph grammars. IEEE Transactions on Software Engineering 24, 521–553 (1998)
23. Berry, G., Boudol, G.: The chemical abstract machine. In: Proc. of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 81–94. ACM Press, New York, NY, USA (1990)
24. Hilderink, G.H.: Graphical modelling language for specifying concurrency based on CSP. IEE Proceedings - Software 150, 108–120 (2003)
25. Oquendo, F.: π-method: a model-driven formal method for architecture-centric software engineering. SIGSOFT Software Engineering Notes 31, 1–13 (2006)
26. Galloway, A., Stoddart, B.: An operational semantics for ZCCS. In: Proc. of the International Conference of Formal Engineering Methods, IEEE Computer Society Press, Washington, DC, USA (1997)
27. Heisel, M., Shl, C.: Formal specification of safety-critical software with Z and real-time CSP. In: Proc. of the 15th International Conference on Computer Safety, Reliability and Security, Vienna, Austria, pp. 31–46. Springer, Heidelberg (1996)
28. Smith, G.: A Semantic Integration of Object-Z and CSP for the Specification of Concurrent Systems. In: Fitzgerald, J., Jones, C.B, Lucas, P. (eds.) FME 1997. LNCS, vol. 1313, pp. 62–81. Springer, Heidelberg (1997)
29. Bussow, R., Geisler, R., Grieskamp, W., Klar, M.: The μSZ Notation Version 1.0. Technical report, TU Berlin, Gemany (1997)
30. He, X., Yu, H., Shi, T., Ding, J., Deng, Y.: Formally analyzing software architectural specifications using SAM. Journal System Software 71, 11–29 (2004)

31. Regayeg, A., Kallel, S., Kacem, A.H., Jmaiel, M.: ForMAAD Method: An Experimental Design for Air Traffic Control. International Transactions on Systems Science and Applications 1, 327–334 (2006)
32. Rodriguez-Fortiz, M., Parets-Llorca, J.: Using predicate temporal logic and coloured Petri nets to specifying integrity restrictions in the structural evolution of temporal active systems. In: Proc. of the international symposium on principles of software evolution, pp. 83–87 (2000)
33. Ramkarthik, S., Zhang, C.: Generating Java Skeletal Code with Design Contracts from Specifications in a Subset of Object Z. In: Proc. of the 5th IEEE/ACIS International Conference on Computer and Information Science, Honolulu, Hawaii, pp. 405–411. IEEE Computer Society, Washington, DC, USA (2006)
34. Jia, X., Skevoulis, S.: Code Synthesis Based on Object-Oriented Design Models and Formal Specifications. In: Proc. of the 22nd International Computer Software and Applications Conference, pp. 393–399. IEEE Computer Society, Washington, DC, USA (1998)
35. Bodden, E.: Efficient and Expressive Runtime Verification for Java. In: Proc. of the Grand finals of the ACM Student Research Competition, San Francisco, USA (2005)
36. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into aspectJ. In: Proc. of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 219–230. ACM Press, New York, NY, USA (2006)
37. Ostermann, K., Mezini, M., Bockisch, C.: Expressive Pointcuts for Increased Modularity. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 214–240. Springer, Heidelberg (2005)

Object-Oriented Coordination in Mobile Ad Hoc Networks

Tom Van Cutsem[⋆], Jessie Dedecker, and Wolfgang De Meuter

Programming Technology Lab
Vrije Universiteit Brussel
Brussels – Belgium
tvcutsem,jededeck,wdmeuter@vub.ac.be

Abstract. We introduce an object-oriented referencing abstraction to express coordination between objects hosted on mobile devices interconnected by a wireless ad hoc network. On the one hand, we notice that the most popular communication paradigms for mobile ad hoc networks, such as publish/subscribe and tuple space architectures, promote loose coupling of collaborating participants. On the other hand, the paradigm in which many applications are developed is object-oriented, and traditional object referencing abstractions typically lack the beneficial loose coupling properties of aforementioned paradigms. This paper proposes to close the paradigmatic gap between an object-oriented language and its distributed communication infrastructure by introducing *ambient references*: loosely-coupled remote object references designed for mobile ad hoc networks.

1 Introduction

The flourishing of research fields such as pervasive and ubiquitous computing [1] has lead to a tremendous increase in research on *mobile ad hoc networks* – networks composed of portable, mobile devices interconnected by wireless communication media. Such networks are often regarded as the ideal hardware infrastructure to support pervasive and ubiquitous computing scenarios [2]. The network's wireless capabilities, combined with the mobility of the devices, results in applications where software entities spontaneously detect one another, engage in various collaborations, and may disappear as swiftly as they have appeared. Example applications range from modest, already commonplace applications like collaborative text-editors, to more futuristic scenarios such as warehouses equipped with digital infrastructure allowing customers to interact with products, their shopping carts, etc.

This paper focuses on distributed programming language support for mobile networks. In distributed programming, communication paradigms based on loose coupling between the participants have been especially promoted in the context of mobile ad hoc networks [3,4,5,6]. Interestingly, none of these approaches is object-oriented in nature, while most mainstream programming languages in which applications are developed are. One of the reasons for this paradigm mismatch is that remote object references have not been as successful in achieving loose coupling between collaborating parties as other paradigms, such as publish/subscribe [7] and tuple space [8] architectures.

[⋆] Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 231–248, 2007.

The contribution of this paper is the proposition of a loosely coupled object-oriented coordination abstraction for mobile networks, named an *ambient reference*. This abstraction eliminates the paradigm mismatch between object-oriented applications and loosely coupled distributed coordination infrastructure, because it allows object-oriented programs to interact without leaving the paradigm, while keeping the benefits of loose coupling promoted by established collaboration paradigms. Ambient references have been implemented in a distributed object-oriented language called AmbientTalk, which we will briefly describe as well.

2 Motivation

Based on the fundamental characteristics of mobile hardware, we discern a number of phenomena that set mobile networks apart from their traditional, fixed counterparts. We show how these phenomena have motivated the choice of loosely coupled interaction paradigms for use in mobile networks. Next, we will highlight why traditional object-oriented distributed computing does not promote loose coupling and hence requires the use of other communication paradigms, leading to a paradigm mismatch.

2.1 Characteristics of Mobile Networks

There are two discriminating properties of mobile networks, which clearly set them apart from traditional, fixed computer networks: applications are deployed on *mobile* devices which are connected by *wireless* communication links with a limited communication range. Such networks exhibit two phenomena which are rare in their fixed counterparts, and which will be shown to be the main instigators for loosely-coupled interaction:

Volatile Connections. Mobile devices equipped with wireless media possess only a limited communication range, such that two communicating devices may move out of earshot unannounced. The resulting disconnections are not always permanent: the two devices may meet again, requiring their connection to be re-established. Quite often, such *transient* disconnections should not affect an application, allowing both parties to continue their collaboration where they left off. Although dealing with disconnection is not a new ingredient of distributed systems, these more frequent transient disconnections do expose applications to a much higher rate of partial failure than that which most distributed languages or middleware have been designed for.

Zero Infrastructure. In a mobile network, devices that offer services spontaneously join with and disjoin from the network. As a result, in contrast to stationary networks where applications usually know where to find collaborating services via URLs or similar designators, applications in mobile networks have to find their required services dynamically in the environment. Services have to be discovered on proximate devices, possibly without the help of shared infrastructure (e.g. a wireless base station), requiring a *peer-to-peer* network topology.

Any application designed for mobile ad hoc networks will have to deal with the above phenomena. Moreover, these phenomena are not easily hidden within a standard programming language or middleware because their effects pervade the entire application. In the following section, we show how dedicated communication paradigms can drastically ease the burden of dealing with these phenomena.

2.2 Loosely-Coupled Collaboration

In this section, we describe requirements for communication paradigms that, when adhered to, significantly reduce the impact of the above phenomena on software. The first three requirements pertain to *decoupling* the communicating parties along three dimensions as explained in detail in [7]. For each requirement, we state why it is critical in the context of mobile ad hoc networks.

Requirement 1 (Decoupling in Time). *The communicating parties do not need to be online at the same time.*

Requirement 1 states that a sender may send a message to a recipient that is offline, and a recipient may receive and process a message from a sender that is offline. This makes it possible for communicating parties to interact across volatile connections. Decoupling in time is directly inspired by the need to deal with the intermittent disconnections inherent to mobile ad hoc networks.

Requirement 2 (Decoupling in Space). *The communicating parties do not need to know each other beforehand.*

Requirement 2 states that communicating parties do not necessarily need to know one another's exact address or location. It implies that communicating parties can rely on some mechanism other than precise addresses or URLs to get to know one another. Decoupling in space is an important property in mobile ad hoc networks because they have a minimum of shared infrastructure, making reliance on servers to mediate collaborations impractical.

Requirement 3 (Synchronization Decoupling). *The control flow of communicating parties is not blocked upon sending or receiving.*

Requirement 3 states that a sending party can employ a form of *asynchronous* message passing, such that the act of message *sending* becomes decoupled from the act of message *transmission*. Likewise, allowing recipient parties to process messages asynchronously decouples the act of message *reception* from the act of message *processing*. Message transmission and reception require a connection between sender and receiver, but message sending and processing can be decoupled, allowing communicating parties to abstract over the fact whether the other party is online or not. This requirement is again directly derived from the volatile connections phenomenon in mobile networks. It allows parties to perform useful work while being disconnected.

Requirement 4 (Connection Awareness). *Communicating parties must be able to keep an up-to-date view of which participants are (dis)connected.*

At first glance, requirement 4 seems to somewhat contradict the above three requirements, because it seems to state that a process is no longer able to abstract over the state of the connection with communicating parties. However, this is not necessarily the case if the aspect of communication can be separated from the aspect of failure handling by means of orthogonal mechanisms. Being aware of the state of the connection of a participant is important because due to the limited infrastructure in mobile ad hoc networks, delivery guarantees for exchanged messages are often very weak. Hence, communicating parties must sometimes take explicit action when a participant disconnects.

2.3 A Paradigm Mismatch

In object-oriented distributed computing, objects distributed across several machines may refer to and communicate with one another by means of *remote object references*. A remote object reference represents a communication channel to a particular remote object. In its most simple form, distributed message passing is a straightforward adaptation of local message passing, known as remote method invocation (RMI). Using RMI, distributed request/response interaction is very easily expressed. Unfortunately, RMI does not decouple objects in time, space or synchronization [7]. However, asynchronous adaptations of RMI (e.g. Rover's Queued RPC [9]) have achieved decoupling in time and synchronisation.

Other communication paradigms have been more successful at achieving loose coupling between participants. For example, in *publish/subscribe* communication publishers asynchronously publish *events* on channels which leads to the asynchronous notification of registered subscribers [7]. Quite often, an *event service* acts as a middle man between publishers and subscribers, allowing them to be decoupled in space. Publishers may publish events even if no subscribers are registered on a channel and vice versa, making them decoupled in time. Tuple spaces, discussed in more detail in section 6.2, achieve a similar decoupling between participants.

In practice, object-oriented programs that require loosely coupled distributed communication abandon the remote reference and message passing abstractions in favour of paradigms such as publish/subscribe and tuple spaces. This requires object-oriented code to be adapted to the communication paradigm. For example, rather than sending messages to remote objects, publishers publish *event objects* on an event channel [10] or processes insert *tuple objects* into a shared space. Method invocation is replaced by subscribing event handlers on channels or by querying a tuple space using a template object, as in JavaSpaces [11].

These adaptations achieve better decoupling of objects, but at the price of giving up on the advantage of remote references to easily express request/response interactions. For example, messages sent via a remote reference have an explicit receiver, so multiple messages sent via the same reference are processed by the same receiver. Without additional programming, this property no longer holds when broadcasting events or publishing tuples. Also, messages invoke methods which have a return value. In contrast, matching an event or tuple that represents a request with its corresponding response event or tuple must be done explicitly in the code.

The contribution of this paper lies in an integration of the above requirements in an object-oriented language, such that distributed communication can still be expressed in terms of objects sending messages to one another. Before introducing ambient references, we first introduce the object-oriented programming language in which they have been developed.

3 The AmbientTalk Language

Ambient references have been implemented as part of the AmbientTalk programming language. AmbientTalk is an object-oriented distributed programming language specifically designed for distributed programming in mobile ad hoc networks [12]. The language has been implemented as an interpreter written on top of the Java Virtual Machine. A J2ME version exists which can be deployed on PDAs.

We will use a typical collaborative ad hoc networking application to illustrate the language and the ambient reference abstraction. After a short description of this running example, we describe standard, sequential programming in AmbientTalk to familiarise the reader with the language's syntax and semantics. Subsequently, we cover concurrent and distributed programming.

3.1 Running Example

Consider a music player running on mobile devices such as PDAs or cellular phones. The music player contains a library of songs. When two people running the music player enter one another's personal area network (delineated by e.g. the bluetooth communication range of their cellular phones), the music players exchange their music library's index (not necessarily the songs themselves). After the exchange, the music player can calculate the percentage of songs both users have in common. If this percentage is high enough, the music player can e.g. warn the user that someone with a similar taste in music is nearby and display those songs in the other user's library which are not in its user's library.

3.2 Sequential Computation

AmbientTalk is a dynamically typed object-oriented language. Computation is expressed in terms of objects sending messages to one another. The following code excerpt shows the definition and use of a simple `Song` object in AmbientTalk:

```
def Song := object: {
  def artist := nil;
  def title := nil;
  def timesPlayed := 0;
  def init(artist, title) {
    self.artist := artist;
    self.title := title;
    self.timesPlayed := 0;
  };
  def play() { timesPlayed := timesPlayed + 1; /* play the song */ };
};
def s := Song.new("U2", "One");
s.play();
```

In this example, a prototypical song object is assigned to the variable `Song`. A song object has three fields, a constructor (always called `init` in AmbientTalk), and a method `play` to play the song. Sending `new` to an object creates a copy of that object, initialised using its `init` method.

3.3 Distributed Computation

AmbientTalk's concurrency model is based on that of *communicating event loops* as featured by the E programming language [13]. This concurrency model has its roots in the actor model of computation [14] and its incarnation in stateful active objects in languages such as ABCL/1[15]. In the model, regular objects are associated with at most one actor (a *vat* in E terminology) and each actor has an associated message queue. Every actor is associated with exactly one thread, the *event loop* which perpetually takes messages from its message queue and invokes the corresponding methods on its associated objects. Within the confines of an actor, computation happens sequentially and objects communicate using sequential message sending, as in Java or Smalltalk. AmbientTalk actors process incoming messages in a serial manner, to ensure that no race conditions can occur on the internal state of their associated objects.

Asynchronous Message Passing. An object a owned by one actor can acquire a reference to an object b owned by another actor. In that case, a can only send messages to b *asynchronously*. When a sends a message to b, the message is placed in the incoming message queue of b's actor. Only when the actor processes the message at a later point in time is b's method invoked.

In the example scenario, each music player is modelled as an actor. Each such music player actor contains a music library, represented as a set of Song objects. When two such actors discover one another in the local ad hoc network, they exchange their music library index. Before a music player downloads the library index, it first asks for the size of the remote library. Given that remotePlayer denotes a reference to a remote music player (see section 4), this can be expressed as follows:

```
def sizeFuture := remotePlayer<-getSizeOfLibrary();
```

AmbientTalk borrows from the E language the syntactic distinction between sequential message sends (expressed as o.m()) and asynchronous message sends (expressed as o<-m()). An asynchronous message send always immediately returns a *future*, which is a placeholder for the actual return value. Once the return value is computed, it "replaces" the future object; the future is then said to be *resolved* with the value. Futures (also known as promises) are a frequently recurring abstraction in concurrent languages (e.g. in ABCL [15], Argus [16], E [13] and recently also in Java).

Futures are objects which can in turn be sent asynchronous messages. Those messages are accumulated within the future as long as it is unresolved. When the future is resolved, these messages are then forwarded to the resolved value. In the E language, it is possible to register a block of code with a future, which is executed asynchronously when the future becomes resolved. AmbientTalk also allows the expression of such "inline event handlers", which are very useful when access to the actual return value of a message send is required. For example, if the user must be informed of how many songs another user is sharing, the size of the other user's music library must be printed on the screen. This can only happen when the sizeFuture from the previous example is resolved to an integer value:

```
when: sizeFuture becomes: { |size|
  // execution of this code is postponed until the future is resolved
  system.println("User is sharing ", size, " songs.");
} catch: { |exception| ... };
// code following when: is processed normally
```

If the asynchronously invoked method raises an exception, rather than returning a result, the corresponding future is resolved with the exception and the `catch` clause rather than the `when` clause of the above code is executed. This enables applications to catch asynchronously invoked exceptions in a way similar to the well-known `try-catch` abstraction of sequential languages.

Exporting Objects. In order to make some objects available to remote actors and their objects, an actor can explicitly *export* objects that represent certain services. Because remote objects do not necessarily know the name or address of the exported service object, a service object is always exported together with a *service type*. A service type is a subtype of one or more other service types. Service types are not associated with a set of methods and are merely used to categorise which objects export what kinds of services[1].

In the music player example, each music player actor exports an `interface` object that can be used by other music players to start a communication session to exchange libraries. This object is exported with the service type `MusicPlayer`, as follows:

```
deftype MusicPlayer;

def interface := object: {
  def openSession(remotePlayer) {
    // return a session object (explained later)
  };
  def getSizeOfLibrary() { ... };
};

export: interface as: MusicPlayer;
```

From the moment an object is exported by its actor, it is discoverable by other actors by means of ambient references via its service type. This is explained in detail in the following section.

4 Ambient References

An ambient reference is a remote object reference that transparently discovers and binds to a remote object by means of a service type. For example, to discover a proximate music player, one creates an ambient reference initialised with the `MusicPlayer` service type, as follows:

```
def musicPlayerFuture := ambient: MusicPlayer;
```

[1] Service types are best compared with empty Java interface types, like the typical “marker” interfaces used to merely tag objects. Example interfaces are `java.io.Serializable` and `java.lang.Cloneable`.

The expression `ambient: MusicPlayer` initiates a service discovery request for a remote object exported as a `MusicPlayer` and immediately returns a future. When a matching object has been discovered, the future is resolved with an ambient reference bound to the discovered object. As usual, objects can start sending messages to the future before it is resolved, causing the future to accumulate those messages until a remote object has been discovered. One can regard this future as a dangling or unbound remote reference. When the future becomes resolved with an ambient reference, we refer to the remote object to which the ambient reference is bound as the ambient reference's *principal*.

Figure 1 depicts the situation where an ambient reference is asked for, but where no matching principal has yet been found. It shows two actors A and B. The wireless communication links of their host devices are represented as dotted circles which delimit their communication range. Each actor hosts a number of objects (white circles). B has exported an object using a service type symbolized as a diamond. A contains a future (gray circle) for an ambient reference that will bind to objects whose service type "matches" the diamond shape. Although the ambient reference does not yet exist, conceptually the future represents a dangling remote reference. Any messages sent to this future will be accumulated by the future until it is resolved.

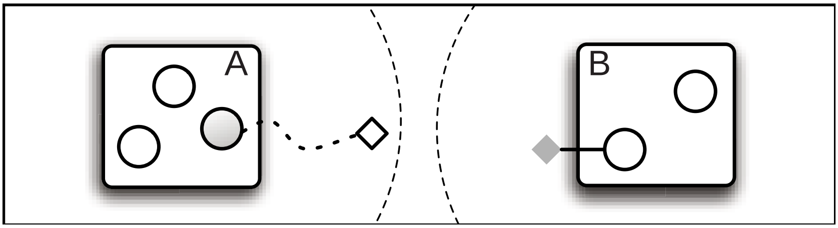


Fig. 1. A future for an ambient reference

Figure 2 depicts the situation where both devices move into one another's communication range. Because a matching service object has been found, A creates an ambient reference bound to this remote object and resolves the outstanding future with the bound ambient reference. Any messages that were accumulated in the future are forwarded to the ambient reference.

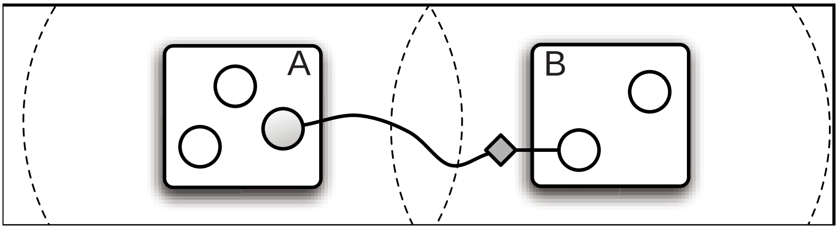


Fig. 2. A connected ambient reference

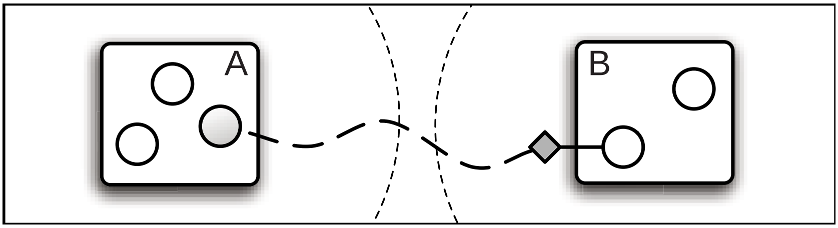


Fig. 3. A disconnected ambient reference

When the two host devices move back out of one another's communication range, the ambient reference does *not* break. Rather, it maintains the bond with the remote service, as depicted in figure 3. It follows that an ambient reference can be in two states: it can either be *connected* to its principal or *disconnected* from its principal. The influence of these states on message passing is explained in the following section.

As explained in section 3.3, the resolved value of a future can be awaited using a `when` block. Because the discovery mechanism immediately returns a future for the ambient reference, objects can take explicit action when proximate services appear in their environment by attaching a `when` block to the future for the ambient reference:

```
def musicPlayerFuture := ambient: MusicPlayer;
when: musicPlayerFuture becomes: { |ambientReference|
  system.println("discovered new music player: ", ambientReference);
};
```

It is important to note that the code that exports the `interface` object, and the code above that creates an ambient reference is executed by all music player actors in the network. This enables music players to engage in true peer-to-peer communication: when a music player `A` and a music player `B` enter one another's communication range, `A` will discover `B`'s interface object via its ambient reference and `B` will discover `A`'s interface object via its ambient reference. The discovery is successful because the service type of the ambient references, `MusicPlayer`, matches (i.e. is a subtype of) the corresponding service type of the exported `interface` object.

An ambient reference created by an actor will not bind to an object exported by that same actor. Indeed, if the object is local to the actor, it can be passed around by means of regular message passing without the need for a decoupled communication channel such as an ambient reference. Hence, in the example above, the ambient reference created by a music player will never bind to its own interface object. If multiple matching remote objects are available when an ambient reference is created, the reference binds to one single arbitrary matching object. Ambient references that may bind to multiple matching objects are not considered in this paper and are left as future work.

4.1 Message Passing

Ambient references follow the rules for inter-actor message passing and operate asynchronously. When a client object sends a message to an ambient reference, it does not wait for the message to be forwarded by the ambient reference to its principal. If the ambient reference is connected to its principal upon message reception, it forwards the message to the principal; if it is disconnected upon message reception, it accumulates the message internally and forwards it whenever it becomes reconnected at a later point in time. Hence, messages sent to ambient references are never lost, regardless of the internal state of the reference. Messages are guaranteed to be forwarded to a principal in the same order as they were received by the ambient reference. Recall that the principal is associated with an actor which ensures that incoming messages (sent by one or more ambient references) are executed serially.

In the music player example, once one music player has a reference to the `interface` object of another music player, it can ask the remote player to open a library exchange session by sending it the `openSession` message. The `interface` object implements this message as follows:

```
def openSession(remotePlayer) {
  def senderLib := Set.new(); // to store sender's music library
  object: {
    def downloadSong(artist, title) {
      senderLib.add(Song.new(artist, title));
      "ok"; // tell sender that song was successfully received
    };
    def endExchange() {
      // myLib and THRESHOLD are instance variables of this actor
      def matchRatio := (myLib.intersect(senderLib)).size() / myLib.size();
      if: (matchRatio >= THRESHOLD) then: {
        system.println("Found user with similar taste in music.");
      };
    };
  };
};
```

Note that the `openSession` method returns a new object which implements two methods which are used by a remote music player to send song information (`downloadSong`) and to signal the end of the library exchange (`endExchange`). A music player sends all of its own songs one by one to this session object after it has discovered a music player:

```
def musicPlayerFuture := ambient: MusicPlayer;
when: musicPlayerFuture becomes: { |ambientReference|
  system.println("discovered new music player: ", ambientReference);
  def session := ambientReference<-openSession(self);
  def iterator := myLib.iterator(); // to iterate over own music library
  def sendSongs() { // auxiliary function to send each song
    if: (iterator.hasNext()) then: {
      def song := iterator.next();
      when: session<-downloadSong(song.artist, song.title) becomes: { |ack|
          sendSongs(); // recursive call to send the rest of the songs
      } catch: { |exception| /* stops exchange (see section 4.2) */ };
    } else: {
      session<-endExchange();
    };
  };
  sendSongs();
};
```

The `session` object is again a future which will be resolved with an ambient reference that is bound to the object returned by the `openSession` method. The auxiliary function `sendSongs` sends the music player's songs one by one to the remote `session` object. This serial behaviour is guaranteed, because each subsequent `downloadSong` message is only sent after the previous one returned an acknowledgement (the simple `"ok"` string returned by the `downloadSong` method defined above).

4.2 Partial Failure Handling

The example application described above illustrates how the use of a loosely coupled communication abstraction (in this case an ambient reference) allows the application developer to abstract over transient disconnections: once the music players have established a library exchange session, they can disconnect from and reconnect to the network without hampering the control flow of exchanged messages. Note that the `catch:` clause in the previous code excerpt is normally not triggered when the ambient reference disconnects, it is only triggered if the invoked method raised an exception. Below, we describe how to trigger this `catch:` clause upon disconnection, such that it can also be used to perform failure handling if necessary.

Although an ambient reference allows a client object to safely abstract from its internal connection state, it is often useful for an application to be informed when a connection with a remote object is lost or reconnected. To this end, it is possible to install observers on an ambient reference which are triggered when the reference becomes disconnected or reconnected. The code below shows how a music player can notify the user whenever a proximate music player moves in and out of communication range:

```
when: musicPlayerFuture becomes: { |ambientReference|
  ...
  when: ambientReference disconnects: {
    system.println("music player disconnected: ", ambientReference);
  };
  when: ambientReference reconnects: {
    system.println("music player reconnected: ", ambientReference);
  };
};
```

The behaviour of ambient references is designed to allow the developer to abstract over transient network failures. However, a developer may want to perform failure handling from the moment an ambient reference has been disconnected for longer than a certain timeout period. The question then becomes how the developer can reasonably deal with all of the messages that have accumulated in the ambient reference while it was disconnected.

To deal with failures, ambient references support one final operation: a developer may *rebind* an ambient reference to point to another principal object. This object may be another remote object, but often it will be a local object that acts as a failure handler for all of the messages that were accumulated by an ambient reference *and* for all of the messages sent to the ambient reference from the moment it has been rebound.

In order to adapt the music player to terminate the library exchange upon disconnection, the ambient reference can be rebound to a failure handler object by means of a `disconnects:` observer (perhaps only after a certain timeout period). This failure handler can then reply to every message by raising an exception. This will resolve each message's future with that exception, which in turn triggers the `catch:` clause of any registered `when` blocks on that future. In the second code excerpt of the previous section, this would trigger the `catch:` clause for the `downloadSong` message, which enables the library exchange protocol to terminate smoothly.

5 Evaluation

Now that ambient references have been properly introduced, we can evaluate them with respect to the requirements postulated in section 2.2.

Requirement 1. Ambient references decouple the communicating objects in time. When a client object first requests an ambient reference, it will immediately get a future for the reference, allowing it to continue its computation until a suitable service object has been found. Moreover, clients are not obliged to send messages via an ambient reference only when it is connected, because an ambient reference properly accumulates messages while it is disconnected.

Requirement 2. Ambient references decouple the communicating objects in space by means of service types. Objects address one another by means of the services they describe and do not know or need to know the address of the hosting device. An exported service object also does not necessarily know which or how many client objects refer to it via an ambient reference. Thanks to the use of futures, a service can easily reply to its clients without referring to them explicitly simply by returning values from its invoked methods. These return values implicitly resolve the futures held by clients, allowing them to process replies asynchronously.

Requirement 3. Ambient references decouple the control flow of client and service objects. Client objects send messages asynchronously and can await results asynchronously by registering blocks of code with the futures. Exported service objects are hosted by an actor, whose incoming message queue ensures that messages can be received even while the service object is busy processing another message.

Requirement 4. Via the registration of dedicated observers which trigger upon the disconnection or reconnection of a principal, an application can have an up-to-date view of the internal state of an ambient reference without affecting other application code that sends messages and receives replies via that ambient reference. Failure handling can be performed by rebinding the ambient reference to a dedicated failure handler object. Any undelivered messages accumulated by the ambient reference are then forwarded to that object.

Because they adhere to the first three requirements, ambient references form a loosely coupled communication channel between objects, without sacrificing the remote object referencing abstraction. The contribution of the ambient reference abstraction lies in the combination of:

1. An abstract type-based discovery mechanism that immediately returns a future when a discovery request is made. The future represents a “not yet discovered” object. This enables a client object that needs to interact with a remote object to immediately interact with the future as if that future already *is* the remote object.
2. Asynchronous message passing semantics which allows one to abstract over the state of the connection with the remote object. This is achieved by implicitly accumulating messages within the remote reference itself while it is disconnected.
3. Using observers to keep track of changes in the state of the connection of the reference, such that failure handling can be performed separately from the main use of an ambient reference as a time- and space-decoupled communication channel.

While none of these mechanisms are by themselves novel, the contribution of ambient references lies in the combination of service discovery and asynchronous communication into one coherent language construct and its application to mobile ad hoc networks.

6 Related Work

We categorise related work into **1)** object-oriented referencing abstractions, **2)** tuple space architectures and **3)** publish/subscribe architectures. For each approach, we summarise their applicability to mobile networks and how they resemble or differ from ambient references.

6.1 Object-Oriented Referencing Abstractions

M2MI. The design of ambient references has been inspired by the notion of a *handle* in the many-to-many invocations (M2MI) paradigm [17]. M2MI is a paradigm for building collaborative systems deployed on wireless proximal ad hoc networks. M2MI handles use Java interfaces to decouple remote objects in space. M2MI handles also employ asynchronous message passing.

Although M2MI has influenced the design of ambient references, there are some important differences. First, M2MI handles do not decouple participants in time: if a message is sent to an object which is not in communication range at that time, the message is lost. Second, M2MI invocations require that asynchronous messages do not return a value or throw an exception. This makes it more cumbersome to express request/response interactions due to the lack of futures.

Actors. In the actor model of computation [14], actors refer to one another via *mail addresses*. When an actor sends a message to a recipient actor, the message is placed in a mail queue and is guaranteed to be eventually delivered by the actor system. One can regard a mail address as a “remote actor reference”. Although such a reference decouples actors in time and in synchronisation (actors communicate strictly asynchronously), it does not decouple them in space. A mail address represents a unique actor and does not allow actors to discover one another by means of an abstract description.

E. The E language [13,18] is designed for writing secure peer-to-peer distributed programs in open networks. Our notion of distinguishing intra-actor communication (synchronous message passing) from inter-actor communication (asynchronous message passing) is directly derived from E's similar message passing semantics. E pioneered the `when` construct to deal with the resolution of futures (or promises) in an entirely non-blocking, event-driven manner.

E was designed for open distributed systems, but not specifically for mobile ad hoc networks. This shows in a number of important differences with respect to the semantics of remote references in AmbientTalk. First, a network disconnection in E immediately *breaks* the remote reference: any message sent after the disconnection is not stored, and the message's promise is resolved with an exception. Hence, E's remote references do not decouple participants in time and are not designed to express communication over volatile connections. E does feature a hook similar to the one introduced in AmbientTalk to enable the programmer to react upon the disconnection of a remote reference. There is no corresponding hook for reconnection in E, because once broken, a remote reference in E remains broken.

To regain connectivity after a network failure, E features special references, known as *sturdy references*, which do survive network failures. Sturdy references, however, are created by means of an explicit address (in the form of a URL) and are meant to denote specific objects, so they do not decouple objects in space.

Jini. The Jini architecture for network-centric computing [19,20] is a platform for service-oriented computing built on top of Java. Jini introduces the notion of lookup services. Services may advertise themselves by uploading a proxy to the lookup service. Clients search the network for lookup services and may launch queries for services they are interested in. Clients can download the advertised proxy of a remote service and may interact with the remote service through the proxy. Java interface types are used to describe and discover services. Our use of service types to describe to which kinds of objects an ambient reference may bind has been inspired by this mechanism.

Jini is primarily a framework for bringing clients and services together in a network with minimal administrative infrastructure. Once a client has downloaded a service proxy, the proxy is the communication channel to the service. This proxy may be implemented however the service sees fit. For example, it is possible to construct proxy references which e.g. accumulate messages when the remote service is disconnected to achieve decoupling in time. Hence, Jini's *architecture* is flexible enough to accomodate ambient references. However, to the best of our knowledge, Jini does not by default offer any advanced remote "service" references. By default, the proxies advertised by services communicate synchronously with their service.

6.2 Tuple Spaces

Linda and LIME. Tuple spaces as originally introduced in the coordination language Linda [8] have received renewed interest by researchers in the field of mobile computing. Adaptations of tuple spaces for mobile computing, such as LIME [4], feature tuple spaces local to each device which are merged into a *federated transiently shared tuple space* when joining the network. In the tuple space model, processes communicate by inserting and removing tuples from the shared tuple space, which acts like a globally shared memory. Decoupling in time is achieved because processes can insert and retract tuples independently. Decoupling in space is achieved because the publisher of a tuple does not necessarily specify, or even know, which process will consume the tuple. Synchronisation decoupling is not adhered to in the original Tuple space model: although tuple insertion is asynchronous, there exist synchronous (blocking) operations to extract tuples from the tuple space.

As the need for total synchronization decoupling became apparent for mobile networks, extensions of the model such as LIME provide *reactions* which are callbacks that trigger asynchronously when a matching tuple becomes available in the tuple space. LIME adheres to requirement 4, connection awareness, by introducing a read-only, system-maintained tuple space whose tuples represent metadata, such as the hosts that are currently connected. Registering reactions on such tuples achieves a connection awareness strategy similar to one using the observers introduced in section 4.2.

The main difference between LIME and ambient references lies in their employed communication paradigm. Ambient references foster a more object-oriented programming style because communication is one-to-one rather than one-to-many and happens by means of asynchronous message sends (which capture the communication of both request *and* reply in one single abstraction).

ActorSpace. The inability of mail addresses to represent unknown, undiscovered actors have been addressed in the ActorSpace model [21]. This model is a unification of concepts from both the actor model and the tuple space model of Linda. Callsen and Agha note that, on the one hand, the actor model provides a secure model of communication as an actor may only communicate with actors whose mail address it has been explicitly given via message passing. On the other hand, this disallows actors to get acquainted with other actors in a time- and space-decoupled manner.

The ActorSpace model augments the actor model with *patterns*, denoting an abstract specification of a group of actors. The actor model's `send` primitive, which normally takes a receiver mail address and a message and sends the message to the corresponding mail address, is changed such that `send` now also accepts a pattern rather than a mail address. For example, `send("MusicPlayer", "getSizeOfLibrary")` can be received by any actor whose own name matches the pattern within the context of a so-called *actorspace*. The `send` primitive delivers the message to a non-deterministically chosen matching actor. Although this behaviour is good when it does not matter to the sender which specific actor receives the message (e.g. when the receiver is a replicated file server), it is not similar to an ambient reference in the sense that multiple messages sent to the same pattern may be received by different actors.

6.3 Publish/Subscribe Architectures

LPS. Location-based Publish/Subscribe (LPS) [6] is a publish/subscribe architecture designed specifically for the collaboration of mobile ad hoc applications. The main difference between LPS and traditional publish/subscribe architectures is that event dissemination and reception is bounded in physical space: a publisher defines a *publication range* and a subscriber defines a *subscription range*. Only when the publication range of the publisher and the subscription range of the subscriber overlap is an event disseminated to the subscriber.

STEAM. Scalable Timed Events and Mobility (STEAM) is an event-based middleware designed for supporting collaborative applications in mobile ad hoc networks [22,2]. STEAM builds upon the observation that the physically closer an event consumer is located to an event producer, the more interested it may be in those events. It allows events disseminated by producers to be filtered based on geographical location using *proximities*. Proximities are first-class representations of a physical range, which may be absolute or relative (i.e. a relative proximity denotes a surrounding area relative to a mobile node).

Both LPS and STEAM are publish/subscribe middleware and have no notion of remote object references. Applications are structured as a suite of event handlers and do not use the message passing abstraction to engage in distributed communication. As publish/subscribe architectures, they naturally decouple participants in time, space and synchronization. It is not immediately clear how the models allow applications to perform failure handling when publishers or subscribers disconnect.

7 Research Status and Future Work

Ambient references have been implemented as part of the AmbientTalk programming language[2]. The mobile music player used as a running example in this paper is also available for download as an example AmbientTalk program.

We are currently investigating a family of ambient reference abstractions with slight variations on the semantics presented here. For example, we are experimenting with ambient references that bind to all available matching services (rather than a single one). Such ambient references form a group communication channel which broadcast messages to all matching objects. Other kinds of ambient references vary in their binding semantics with their principal. As explained in section 4, when the remote principal becomes disconnected, the ambient reference remains bound to it. Sometimes it is more appropriate to clear the binding when a disconnection occurs, such that the ambient reference can *rebind* to other available matching objects (e.g. in the case of a replicated service where the identities of the replicated exported objects themselves are not important).

Another aspect of ambient references which has currently not yet been thoroughly addressed is the garbage collection of exported objects. For the purposes of this paper, it is assumed that exported service objects are long-lived objects which have to be unexported explicitly in order to be reclaimed. Once an exported object is advertised on the network, it can no longer be reclaimed automatically because at any point in time an ambient reference may bind to it. Moreover, in mobile ad hoc networks where relationships between devices are short-lived, traditional cooperative distributed garbage collection approaches become impractical. As illustrated by networking technology such as Jini, the notion of a *leased* reference provides more robust garbage collection in the face of both transient and permanent disconnections [19]. In future work, we would like to integrate leasing with ambient references. Using leases, exported objects can be unexported when their lease expires, while ambient references that still refer to the exported object are responsible for renewing the lease in time.

[2] The language is available at `http://prog.vub.ac.be/amop/at/download`

8 Conclusions

This paper put forward ambient references as a loosely coupled object-oriented coordination abstraction for mobile ad hoc networks. The conception of this abstraction is motivated by the observation that: **a)** mobile networks require loosely coupled communication abstractions and **b)** traditional distributed object-oriented computing abstractions do not fit these requirements which **c)** requires object-oriented programs to leave the object-oriented paradigm when performing distributed communication.

The contributions of this paper are: **a)** an analysis of the requirements for coordination abstractions for mobile ad hoc networks and **b)** the introduction of a coordination abstraction for mobile ad hoc networks in the guise of a language construct which is both object-oriented (it is an object reference carrying messages) *and* loosely coupled. We have exemplified ambient references by means of a typical collaborative application, developed in the AmbientTalk programming language.

Acknowledgments. The authors would like to thank Stijn Mostinckx, Elisa Gonzalez and Jorge Vallejos for the many discussions that have contributed to the ideas presented here. The authors also thank the anonymous referees for their valuable comments and suggestions for improvement.

References

1. Weiser, M.: The computer for the twenty-first century. Scientific American. pp. 94–100 (1991)
2. Meier, R., Cahill, V., Nedos, A., Clarke, S.: Proximity-based service discovery in mobile ad hoc networks. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 115–129. Springer, Heidelberg (2005)
3. Mascolo, C., Capra, L., Emmerich, W.: Mobile Computing Middleware. In: Gregori, E., Anastasi, G., Basagni, S. (eds.) NETWORKING 2002. LNCS, vol. 2497, pp. 20–58. Springer, Heidelberg (2002)
4. Murphy, A., Picco, G., Roman, G.C.: LIME: A middleware for physical and logical mobility. In: Proceedings of the The 21st International Conference on Distributed Computing Systems, pp. 524–536. IEEE Computer Society Press, Los Alamitos (2001)
5. Mamei, M., Zambonelli, F.: Programming pervasive and mobile computing applications with the TOTA middleware. In: PERCOM ’04: Proceedings of the Second IEEE International Conference on Pervasive Computing and Communications, IEEE Computer Society, Washington, DC, USA, p. 263 (2004)
6. Eugster, P., Garbinato, B., Holzer, A.: Location-based publish/subscribe. Fourth IEEE International Symposium on Network Computing and Applications, pp. 279–282 (2005)
7. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. 35(2), 114–131 (2003)
8. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems 7(1), 80–112 (1985)
9. Joseph, A.D., de Lespinasse, A.F., Tauber, J.A., Gifford, D.K., Kaashoek, M.F.: Rover: a toolkit for mobile information access. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP ’95), Colorado, pp. 156–171 (1995)

10. Eugster, P.T., Guerraoui, R., Damm, C.H.: On objects and events. In: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, pp. 254–269. ACM Press, New York, NY, USA (2001)
11. Freeman, E., Arnold, K., Hupfer, S.: JavaSpaces Principles, Patterns, and Practice, Essex, UK. Addison-Wesley Longman Ltd, London (1999)
12. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented Programming in Ambienttalk. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 230–254. Springer, Heidelberg (2006)
13. Miller, M., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in E as plan coordination. In: De Nicola, R., Sangiorgi, D. (eds.) TGC 2005. LNCS, vol. 3705, pp. 195–229. Springer, Heidelberg (2005)
14. Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
15. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Conference proceedings on Object-oriented programming systems, languages and applications, pp. 258–268. ACM Press, New York (1986)
16. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, pp. 260–267. ACM Press, New York (1988)
17. Kaminsky, A., Bischof, H.P.: Many-to-many invocation: a new object oriented paradigm for ad hoc collaborative systems. In: OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 72–73. ACM Press, New York (2002)
18. Miller, M.: Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control. PhD thesis, John Hopkins University, Baltimore, Maryland, USA (2006)
19. Waldo, J.: The Jini Architecture for Network-centric Computing. Commun. ACM 42(7), 76–82 (1999)
20. Arnold, K.: The jini architecture: Dynamic services in a flexible network. In: 36th Annual Conference on Design Automation (DAC'99), pp. 157–162 (1999)
21. Callsen, C.J., Agha, G.: Open heterogeneous computing in ActorSpace. Journal of Parallel and Distributed Computing 21(3), 289–300 (1994)
22. Meier, R., Cahill, V.: Exploiting proximity in event-based middleware for collaborative mobile applications. In: Stefani, J.-B., Demeure, I., Hagimont, D. (eds.) DAIS 2003. LNCS, vol. 2893, pp. 285–296. Springer, Heidelberg (2003)

Coordinating Workflow Allocation and Execution in Mobile Environments

Rohan Sen, Gregory Hackmann, Mart Haitjema, Gruia-Catalin Roman,
and Christopher Gill

Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{rohan.sen, ghackmann, mart.haitjema, roman, cdgill}@wustl.edu

Abstract. Workflows have been used successfully to model collaborative activities that have a well-defined structure. Workflow management systems today can execute workflows that range from a simple sequence of tasks to complex business processes, but have a common restriction in that they can only function in settings where the network is stable. This paper represents an initial investigation into the possibility of using workflows in a challenging new domain - that of an ad hoc mobile network - and for a wider purpose - that of supporting arbitary collaborations among groups of people. Moving to a mobile setting introduces many challenges, as the mobility of the participants in a workflow imposes constraints on the allocation of workflow tasks, coordination among participants, and the marshaling of results. We present an algorithm that heuristically allocates tasks to participants based on their capabilities *and* mobility, and a system that uses spatiotemporal coordination to control and manage workflow execution in a mobile environment.

1 Introduction

Workflows have proved to be useful for specifying complex activities that have a well defined structure. The *activity* described by a workflow is broken up into a set of smaller *tasks* with an *ordering* imposed among tasks to preserve the overall stucture of the activity. A workflow can be conceptualized as a directed acyclic graph with the nodes representing the tasks and the edges imposing the order among them. The workflow model supports multiple *candidate* hosts or agents working on a single workflow, which makes it ideally suited to describe activities that require collaboration and coordination among multiple participants.

Currently, the workflow model is used in a wide range of applications of varying complexities, e.g., the Automator application [1] in OS X uses workflows to describe multi-step scripts that are capable of tasks such as cropping and resizing photos, renaming files according to a rule set, etc. Workflow Management Systems (WfMSs) such as ActiveBPEL [2], YAWL [3], and BPWS4J [4] manage enterprise level business processes from mortgage loan processing, insurance claims processing, and payroll generation to managing shopping cart and

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 249–267, 2007.

check-out systems at online stores. These applications have two key aspects in common – 1) there are multiple participants in the workflow and 2) the workflow is executed in an environment where connectivity is not a concern.

The systems described have proved to be successful for the applications for which they were designed, but are restrictive in that they require a wired network of servers on which to execute. However, the workflow model does not inherently require its participants to be attached to a wired network. In fact, the workflow model is agnostic to the non-functional characteristics of the participants as long as they meet the functional requirements for their assigned tasks. As such, workflows can be used to specify broader types of collaborations.

Consider a construction site in a remote area, where workers must work together in a small area to build a bridge. Each worker is required to report completion of his assigned tasks to facilitate scheduling and tracking. The tasks throughout a day form a workflow which is administered by the site supervisor. The workflow is a simple sequence of tasks which entail building the frame for a support, delivering concrete to the work site, and then pouring the concrete. The WfMS on the supervisor's PDA assigns tasks based on skill, e.g., the delivering concrete task is assigned to a truck driver, building the framework for the supports is assigned to an iron worker whose PDA can render CAD drawings, while the task of pouring the concrete is assigned to a cement worker. The workflow execution starts with the iron worker building the support as per the directions from the CAD drawing on his PDA. When completed, he notifies the driver, again using his PDA. The notification is received on the driver's PDA. The driver delivers the concrete, and notifies the cement worker, who then pours the concrete.

In this example, we have used workflows to describe collaborative activities in the physical world among groups of individuals that involve computational tasks as well as tasks involving a physical activity by a human participant. In such collaborations, we assume that each participant 1) carries a mobile device that can run a small set of programs, 2) is physically mobile, and that 3) communicates over a mobile ad hoc network (MANET) in the interest of flexibility. The dynamics of the workflow execution environment create a new set of design challenges that are not faced when executing in a wired network. These challenges can be split into two broad categories – 1) *allocation*, i.e., determining which participant is responsible for a task in the workflow, becomes especially significant and must be determined *a priori*, and 2) *coordination* among the various participants becomes significantly more complex due to the dynamic topology of a MANET which often allows only transient connectivity among hosts.

In this paper, we describe a novel workflow management system designed for mobile environments that is built on top of coordination technology. The contributions of this paper can be summarized as follows: 1) an allocation algorithm that is used by a leader to determine how tasks are allocated to mobile participants that are only transiently within communication range of each other, 2) the architecture of a workflow management system (WfMS) for mobile environments, and 3) an execution model for a WfMS that uses CAST [5], a previously developed model for coordination across space and time to handle the coordination among the participants in the workflow.

2 Background

A WfMS operates as follows. A workflow specification is supplied to the WfMS. The WfMS chooses a suitable candidate to perform the first task in the workflow, and notifies that candidate, passing any input if applicable. When the task is completed, it notifies the WfMS, which then repeats the process for subsequent tasks until the workflow is completed. This approach is well suited to wired networks since it is possible to have a centralized WfMS and candidates are always available and accessible. In mobile networks, however, the emphasis is on a different set of issues. Choosing a suitable candidate comes to the fore since all candidates may not be in range of the host executing the WfMS at all times, as in a centralized, fully-connected setting. It is thus preferrable to allocate tasks *a priori* based on qualifications and future windows of communication– intervals of time when direct communciation between host pairs are possible due to them being in communication range of each other. Notification of completion and transmission of output also assumes a different character. Rather than routing such communication through the WfMS, such notifications and data should be routed directly to the candidate chosen for the next task so that the management of the overall workflow execution is handled in a distributed fashion. In addition, issues associated with MANETs, such as transient connectivity, decoupled computing, and lack of communication guarantees must be addressed.

Before we present our solutions to these problems, we describe the computational model used in this paper and provide an overview of CAST, a model for spatiotemporal coordination that we exploit both for allocation purposes and for handling low level message passing.

2.1 Computational Model

Our computational model consists of heterogenous, physically mobile computing devices which we refer to as *hosts*. We assume that these mobile hosts are carried on the person of a human participant in the workflow. Hosts have sets of *capabilities*, which are used to determine if they are suitable candidates for particular tasks. These capabilities may be *computational* in nature, i.e., a software service executing on the mobile device, or *physical*, i.e., an activity performed by the human participant associated with the device. A host also has an associated motion profile [6], which describes its location in physical space as a function of time, and is subject to a maximum possible speed that it is capable of achieving. This information is stored in a local knowledge base along with similar information about other hosts.

The workflow is specified as a directed acyclic graph, with the nodes representing *tasks* and the edges imposing the ordering. Each task has a set of *requirements*, including a start time, an end time, a location, and a set of *host* *constraints* (described later). Hosts whose capabilities are a superset of a given task's requirements and do not violate the host or spatiotemporal constraints are considered to be candidates for performing the task. The set of tasks from which a task must receive inputs is called its *preset* since their execution must precede its own. A task is considered ready for execution when all the tasks in its *preset* have been completed. The preset of a task x is defined as the set of tasks from which x has incoming edges. The set of tasks that immediately follow the execution of a task are said to form that task's *postset.* When the task has been completed, all tasks in its postset are notified. The postset of a task x is defined as the set of tasks to which x has outgoing edges. There is a single exception to these rules. Certain edges in the workflow may be marked as *optional.* When such edges appear in the preset of a task, the task at the source of that edge need not be completed in order for the workflow to proceed.

In this paper, we assume that all participants in a workflow meet at the beginning of the day. A group leader who has the workflow specification on his device initiates a planning process which allocates tasks in the workflow to participants based on their qualifications and spatiotemporal behavior. The group then disbands and the user with the first task begins executing it. When he completes his work, he notifies the users who have a task that appears in the postset of the first task, at which point they begin executing. This continues until the workflow is completed. Further details of the planning process appear in the next section.

2.2 CAST Overview

Since our WfMS is required to operate across a MANET, we cannot rely on standard abstractions such as sockets and streams for communication as they are too rigid, providing connectivity only between fixed points, and allowing a very restricted form of addressing. The use of ad hoc routing is precluded as well since those protocols require an end-to-end connected route, which too, is relatively inflexible. For increased flexibility, we chose to develop our WfMS on top of CAST, a model for coordination across space and time. CAST is designed to work in a MANET setting and provides abstractions based on a generative communication model. We provide a brief overview of CAST here.

The key idea in CAST is to make space and time an explicit component of the communication model. This is especially useful in MANETs since the physical motion of the hosts imposes spatiotemporal constraints on workflow management. In CAST every host has a *knowledge base*, a portion of which is a repository of information about the motion of other hosts in the MANET. CAST uses this knowledge to identify a series of *windows of communication* between host pairs which represent intervals of time during which communication among them is possible due to being within direct communication range. Data in CAST is tagged with a sequence of hosts and forwarded from one host to another using a hold and forward pattern. Each host stores the data locally until a window of communication becomes available to the next host in the sequence of required transmissions and then forwards it. Thus, by chaining multiple communication windows between host pairs, CAST forms a disconnected route between hosts that may never be directly in contact.

These disconnected routes are used to expand the notion of reachability in a MANET. Previously, a host was considered reachable if it was directly connected to the source host or had an end-to-end connected multihop route built using a MANET routing protocol such as AODV [7], DSDV [8], etc. With CAST, the notion of reachability is extended to include hosts to which a fully connected route is never established. This expanded notion of reachability is used to define *spatiotemporal* operations that execute not only on particular hosts, but also at specific points in physical space or over a geographic region.

It should be observed that all communication in CAST occurs when two hosts are directly connected to each other. This communication occurs in a generative fashion using *tuples* and *templates*. Tuples can be conceptualized as packets that contain data. Templates are sets of constraints. When the data in a tuple meets the constraints specified in a template, the tuple is said to *match* the template and vice versa. Tuples and templates are stored in a special section of the knowledge base on each host as is described in [5] and are transferred from host to host using a gossiping protocol.

CAST offers three important operations– `out`, `in`, and `rd`. The `out` operation is used to place data at a target location which may be an explicitly specified host, a physical location, or a region of space. For clarity, we refer to these operations as `out_host`, `out_loc`, and `out_reg` respectively. When `out_host` is used, a tuple containing the data is placed in the knowledge base of the target host. When `out_loc` is used, the tuple is placed in the knowledge base of the host that occupies that location. If no hosts are in that location, the data is held on proximal hosts and then transferred when a host enters the target location. An `out_reg` operation places tuples in the knowledge base of all hosts within the region. A monitoring scheme ensures that the data is removed from hosts that leave the target location or region and is added when new hosts enter the target area.

The `in` operation is used to remove data from a host, location, or region in space and has the same variants as the `out` operation, namely `in_host`, `in_loc`, and `in_reg`. The operations take a template as a parameter. For the `in_host` operation, a matching tuple (if it exists) is returned from the knowledge base of the target host. For `in_loc`, a matching tuple is returned from the host that occupies that location, if there is such a host, while for the `in_reg` operation, a matching tuple is returned from any host that is within the target region. If multiple candidate tuples are available, one is selected non-deterministically. The third type of operation is the `rd` operation, which is similar to the `in` operation except that it returns a copy of the data and does not remove it.

Further details about CAST can be found in [5]. It should be noted that for the work in this paper, we have made one modification to the CAST model, which is that we have exposed the knowledge-base at the API level so that applications may query it directly for planning purposes. We offer two kinds of queries on the knowledge base– 1) `is_spatiotemporal_constraint` takes in two location parameters, a time parameter, and a host and returns whether a host can travel between the two locations in the specified period of time and 2) `is_meeting_possible` takes in a start time, end time, and two host names and returns whether the hosts are within communication range for a sufficient interval of time that falls between the start time and the end time.

3 Allocation Algorithm

In this section, we describe the allocation algorithm for our WfMS. This algorithm is different from those employed by workflow systems in wired settings in the following ways: 1) the allocation of tasks is done *a priori* and in a batch (all tasks are allocated before the workflow execution begins) as opposed to in an on-demand fashion at runtime, 2) hosts are evaluated on the basis of their functional capabilities *as well as* their spatiotemporal behavior, and 3) the allocation process is partitioned into sub-problems with backtracking capabilities built in. It should be noted that in mobile environments, the allocation process assumes greater significance since a poor allocation can result in the workflow not executing to completion due to non-functional circumstances such as situations where a host has completed a task but cannot communicate the completion to the next host in the workflow because that host is not reachable.

Before we can begin the allocation of tasks to hosts, we must take into account any relevant constraints that ensure that an undesirable allocation is not computed. Two types of constraints are possible– *host allocation constraints* prevent certain hosts from being allocated to a task, or require a particular host to be allocated to a task, e.g., that a host must be allocated to both task X and task Y, or that task X cannot be allocated to the same host as task Y, etc. Such constraints form part of the workflow specification. *Spatiotemporal constraints* prevent allocations that are in conflict in the spatiotemporal domain, e.g., a host should not be allocated to two tasks whose start and end times overlap, a host should not be allocated to two tasks if they are separated by time t and distance d, if the host's maximum speed is lower than $\frac{d}{t}$, and two different hosts should not be allocated to two sequential tasks if the host executing the second task cannot receive the results of the first task (either directly or via a disconnected route) before the second task begins. The existence of spatiotemporal constraints for tasks and hosts can be determined by calls to the `is_spatiotemporal_constraint` and `is_meeting_possible` operations on the knowledge base associated with CAST. Spatiotemporal constraints are especially important because they abstract the effects of mobility and represent them as a simple constraint set to the allocation algorithm.

Constraints are represented as 3-tuples of the form $< t_1, A, t_2 >$ which indicates that host A cannot be allocated to task t_2 if it has been allocated to task t_1. Note that our algorithm is agnostic to the cause of the constraint which may be host-driven or spatiotemporally driven. For simplicity, we assume constraints to be symmetric. Once all constraints are established, we move on to build the data structures that represent the initial state of our allocation algorithm. In this phase, we create a table for every task in the workflow as shown in Figure 1.

The first column in each row represents a host that has the *functional* capability to perform the task and is not subject to a host constraint associated with that task. The matching of the functional capability of a host with the requirements of a task can be easily computed [9] if hosts' capabilities and tasks requirements are expressed using a uniform ontology such as OWL-S [10]. The second column in each row is a list of tasks to which we could not allocate the host (in the first column) if we were to allocate that host to the current task. This information can be obtained from the constraints assembled previously– for each constraint $< t_1, A, t_2 >$, we check the table for task t_1 to see if it has a row for host A. If it does, then we add t_2 to the corresponding list in the second column.

task 1	
host A	2, 3
host B	3, 4
host D	3, 4, 5

task 2	
host A	1, 4
host B	3, 4
host C	4, 5

Fig. 1. Example constraint tables

The tables for each task give us two pieces of information– the list of hosts to which that task can be allocated, and the list of future allocations made impossible by that decision, e.g., according to the first table in Figure 1, we can allocate task 1 to hosts A, B, or D. Allocating task 1 to host A, makes it impossible to allocate tasks 2 or 3 to host A.

Allocation Algorithm. The tables assembled in the previous step form the input to our core allocation algorithm, the pseudocode for which is shown in Figure 2. The algorithm first sorts the tables in ascending order of the number of rows in the table. Then, within each table, it sorts the rows in ascending order of the number of elements in the list in the second column. Thus the tables are sorted according to the number of hosts that can perform a task, and the rows are sorted according to the number of allocation conflicts the allocation decision causes.

The algorithm begins with the task represented by the first table. It selects the first host in that table. If this host has no conflicts, then it can allocate that host without any conflicts, i.e., this host can be allocated without affecting *any* other allocation decisions. Hence, the host is allocated to that task and the algorithm proceeds with the next task. If the first host (call it host A) has at least one conflicting task, then by allocating the task to the first host in the table, we are making some future allocations impossible. To establish whether this decision is the correct one, the algorithm tries recursively to resolve the conflicts. For this, it creates a stack to keep track of its allocation decisions as shown in Figure 3. It marks host A's row in the table, and pushes a token onto the stack reflecting this marking. Next, it collects the list of conflicting tasks from the table. For each conflicting task, its grays out the row with Host A in the corresponding task's table, and pushes onto the stack a marker that represents this change. When it later visits these tasks, it will disregard all the rows that have been

```
boolean heuristicAllocatetask(tasks, A, allocation)
    for each row (G, conflicts) in A, ordered by |conflicts|
        if |conflicts| = 0
            allocation := allocation $\cup$ (A, G)
            return true

        myToken := new AllocationToken(A, G)
        push(stack, myToken)
        allocation := allocation $\cup$ (A, G)

        for each C in conflicts
            if C $\notin$ allocation
                push(stack, new GreyRowToken(C, G))
                disableRow(C.table, G)

        for each C in conflicts, ordered by |conflicts|
            if C $\notin$ allocation
                if not heuristicAllocatetask(tasks, C, allocation)
                    do
                        token := pop(stack)
                        undo(token)
                    until token = myToken

                    push(stack, new GreyRowToken(A, G))
                    next row

        return true
    return false

map enhancedAllocate(tasks, hosts)
    allocation := $\emptyset$
    createConstraintTables(tasks, hosts)

    for each A in tasks, ordered by |A.table|
        if A $\notin$ allocation
            heuristicAllocatetask(tasks, allocation, A)
    return allocation
```

Fig. 2. Psuedo-code for heuristic allocation algorithm

grayed out, since they reflect decisions that would violate a constraint. This process continues until all conflicting tasks have been recursively allocated at which point it returns to the original list of tasks and continues allocating them sequentially as before.

At some point, the algorithm may encounter a task with no capable hosts left (due to them having been grayed out as an effect of previous allocation decisions). This means that one of the earlier decisions was undesirable, and that it must roll its state back to that decision point. It does this by popping elements off the stack, undoing the changes that they represent, until it reaches a change to a table that marked one of at least two remaining rows. This indicates a place where it made a decision that may have been incorrect. It un-marks the host chosen at this point, and grays out its row so that it doesn't try that host again (it also pushes a token onto the stack for the row that has just been grayed out). Finally, the algorithm attempts to re-allocate the task to the next un-grayed host in the table.

This algorithm has two key features. First, rather than allocating tasks in an arbitrary order, it first allocates the tasks that are hardest to satisfy and allocates them to hosts that will cause the fewest conflicts later. This reduces the amount

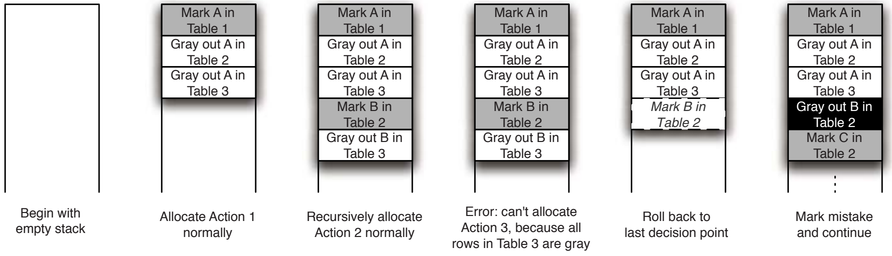

Mark A in Table 1
Gray out A in Table 2
Gray out A in Table 3
Mark B in Table 2
Gray out B in Table 3
Gray out B in Table 2
Mark C in Table 2
Begin with empty stack
Allocate Action 1 normally
Recursively allocate Action 2 normally
Error: can't allocate Action 3, because all rows in Table 3 are gray
Roll back to last decision point
Mark mistake and continue

Fig. 3. Using an allocation stack to track and roll back changes

of backtracking that the algorithm must do, since it will first consider the paths that are least likely to cause irresolvable conflicts. Second, the algorithm recurses through hosts' conflict lists, effectively dividing the allocation of the workflow into sub-problems. Due to this recursive process, it is guaranteed first to consider the entire "conflict closure" of a task, i.e., all the tasks that recursively conflict with it. Since by definition tasks in one closure cannot conflict with tasks in another closure, they are allocated completely independently of each other. So, once a closure has been fully allocated, the algorithm will never revisit any of the tasks in it, which greatly reduces the cost of backtracking.

Note that the algorithm does not consider the actual data flow when computing a well-formed allocation. This has two implications. First, the constraint that two hosts must "meet up" before exchanging data becomes more complex to describe when one host must receive results from multiple predecessors. This constraint can be simplified by requiring that all tasks corresponding to nodes that join to a common node in the graph must take place in the same physical location. This behavior can be enforced by adding "move to a common location" tasks to all the paths immediately before the join point. Second, our allocation is conservative: we assume that all tasks in the workflow will be executed, even though the workflow may split into multiple, mutually-exclusive paths. Thus, valid allocations may exist which do not execute all tasks, and which our algorithm will not find. This shortcoming can be worked around by enumerating all possible traces through the workflow and attempting to allocate each trace individually until one feasible allocation is found. As we show in Section 5, the cost of running our algorithm is low enough to make this approach feasible. Nevertheless, in future work we may consider ways to incorporate data flow information into our algorithm's decisions.

Before we conclude this section, we present a brief discussion of the complexity of this algorithm. Since our algorithm involves backtracking, in theory its worst case complexity is exponential. However, in practical use this is rarely likely to occur. Consider a workflow of 'n' tasks which must be allocated to 'h' hosts. Theoretically, there are 'h' options for each of the 'n' tasks leading to a worst case complexity of n^h. However, by saying that there are 'h' options for each task, we are in effect saying that all 'h' hosts are capable of doing any of the 'n' tasks. If this were true however, it would be much easier to find suitable hosts, and the amount of backtracking would be reduced significantly, reducing the overall complexity. Another possibility is that the number of hosts that are able to do a task is significantly less than ‘h’. Therefore ‘a’, the average number of options << ‘h’ which means that $n^a << n^h$. This complexity assumes a brute force approach of trying all options in a random order as in the naïve algorithm.

In our algorithm, we do not allocate tasks in a random order. Rather we allocate the task that has the least number of possible hosts to service it. In the process of allocating this task, we cancel out other opportunities for hosts to perform a task (due to conflicts) which brings down the value of ‘a’ as the algorithm progresses, making it more efficient at the tail end. Finally, once we allocate a task, we follow that by allocating its conflict closure. This ensures that the backtracking we do is over a subset of the tasks rather than the entire workflow because a task and its conflict closure by definition do not cause conflicts with other tasks and hence once allocated, need not be revisited as part of any backtracking. As such our algorithm makes it possible to allocate tasks to hosts while taking into account their mobility patterns at a lesser computational cost than a naïve approach.

4 Execution Support

Once the planning process is completed, the execution of the workflow must begin. In this section, we describe our system architecture, followed by details of how workflow operations are translated into CAST operations during execution.

4.1 System Architecture

Given that our WfMS must run on relatively resource poor devices, we kept the architecture of our system minimalist. The complete architecture of our system can be seen in Figure 4. The bottom most layer of our architecture is the physical network which supports TCP/IP over 802.11b/g or a similar wireless protocol. Immediately above this layer sits the knowledge base and the CAST middleware.

The knowledge base is a repository for both functional and non-functional information about the local host as well as other hosts in the network. Non-functional information includes the motion profiles, and maximum speeds of hosts, while functional information includes the capabilities of hosts as well as data that is in transit from one host to another over a disconnected route. When two hosts are within communication range, they synchronize their knowledge bases by gossipping, trading non-functional information as well as actual data (the data may be targeted towards one of these hosts or simply may be in transit to a third party). Details of this scheme are given in [5].

CAST uses the knowledge base to compute (possibly disconnected) routes to other hosts in the network and to physical locations. It offers a standard set of coordination operations, namely {`out, in, rd`}_{`host, loc, reg`}, which we discussed in Section 2. When an operation is invoked, CAST places the data or operation request in the local knowledge base (CAST uses the knowledge base for all coordination operation management), which then transmits it to

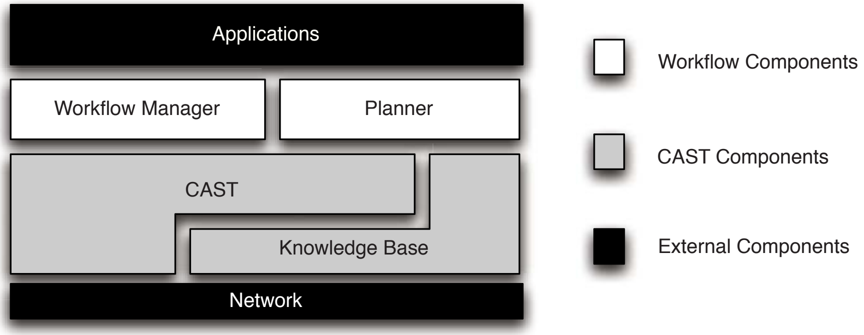


Fig. 4. Architecture of our WfMS

its destination via the gossiping protocol mentioned above. Any callbacks or responses are received directly from the network layer.

Above CAST and the knowledge base sit the two components that manage workflow planning and execution. The planner is responsible for allocating tasks in the workflow. To do this, it queries the knowledge base to get a list of spatiotemporal constraints using the operations described in Section 2. These spatiotemporal constraints are easily computed by the knowledge base as it has access to the motion profiles and velocities of hosts. For more information about these computations, please see Section 3 of [5]. Once the constraints are computed, the planner uses the algorithm described in the previous section to allocate tasks in the workflow. Once the allocation is completed, it informs the hosts of their assignments by issuing CAST operations. It should be noted that the planner is typically active only on the group leader's device, whereas other hosts have the planner available but it is not active. In future work, we intend to use the other host planners for run-time replanning in case of errors.

The Workflow manager is the component that takes the place of the centralized WfMS seen in traditional wired systems, and in combination with managers on other hosts represents a distributed WfMS. The workflow manager is responsible for accepting task assignments, starting each task at the appropriate time after making sure that all inputs are available, and disbursing the result of the task or a notification of completion to the postset of the task. A detailed explanation of the workflow manager's operations appears in the next subsection.

Above the planner and the Workflow manager lie external applications of two types– 1) applications that initiate a workflow execution by injecting a workflow specification into the planner, which then allocates the workflow and hands it over for execution, and 2) applications that represent the capabilities of the host and are invoked by the workflow manager to complete tasks in a workflow.

4.2 CAST Support for Workflow Operations

The workflow manager and the planner support the following operations– 1) distributing task allocations to hosts, 2) registering interest in task allocations,

3) registering interest in results from the preset of each task, 4) performing each assigned task, and 5) distributing the result of eac task to its postset. We now describe how these three operations are supported by CAST.

Distributing task allocations to hosts. Once a workflow has been allocated by the planner, individual hosts must be informed of their task assignments. This is done via a unicast of each task (along with the list of tasks in their preset and postset) in the workflow to the host to which it has been allocated. The data is placed in a tuple which is then sent to the target host using the `out_host` operation. The tuple contains the name of the destination host in a data field so that the hosts can identify the tuples that are intended for them.

Registering interest in plan allocations. While the distribution of the allocated plan ensures that the data is available to all hosts, the workflow manager on each host needs to register an interest in such data with CAST so that when the data becomes available, CAST propagates it to the workflow manager. This is done using the `rd_reg` operation over the physical region where the tasks in the workflow are being performed. The template passed to this operation specifies the name of the local host and the rules that will match the plan allocation tuples. Since the entire region is specified, the existence of any allocated workflow tuples in the region with the local host's name in a data field will generate a match and return the task assignments for that host.

Registering interest in inputs from the preset. Once the hosts know their task assignments, they must wait on the inputs to their task to be available before they can begin execution. Much like registering interest in the allocation, hosts must register interest in inputs to their allocated task. For every input to their task, a host issues an `in_host` operation targeted to the local host (as described below, all tuples are sent to the recipient host, and the recipients only need to recover it from the local knowledge base). The template contains the name of the source task and a wildcard for the result it generates. This ensures that it only receives results from the tasks in which it is interested.

Performing the assigned task. Once the inputs have been received, the task can be performed in two ways. If the task involves executing a piece of software, it is invoked automatically by the workflow manager and once the process has finished, the result is propagated to the postset automatically. If the task involves some physical action by the user, the workflow manager displays an alert on the device with the details of the task to be performed. Once the user has completed the task, he can input a notice of its completion and any relevant results into the device. This input is then propagated to the postset by the workflow manager.

Propagating results to the postset. Once the task has been completed, hosts must propagate the results to the postset of the task. For this, it uses an `out_loc` operation with the location being the location of the task in the postset. The data sent is the name of the originating task (i.e., the task sending the data) and the actual results from the task. The `out_loc` operation places the tuple in the knowledge base of a host that occupies the target location. This is repeated for all tasks in the postset. As described above, these tuples are retrieved by downstream hosts and the workflow execution continues until the last task is finished. Here it should be observed that CAST's spatiotemporal addressing scheme is especially beneficial to our WfMS. By propagating results by location, individual hosts do not need to know the specific hosts to which other tasks are allocated, i.e., each host is allocation agnostic. Additionally, this scheme can be extended easily to accomodate re-planning at runtime since tasks can be reassigned to alternate hosts without affecting the other task allocations.

4.3 Discussion

The key features of our architecture are its simplicity and the decoupled manner in which the workflow execution can proceed. Once the allocation has been completed, hosts are responsible only for completing the tasks that they have been assigned to perform and do not need to be concerned with how the rest of the workflow is allocated. The use of CAST ensures that results can be propagated effectively without requiring an explicit host address and without having to deal with MANET communication issues at the WfMS level. The decoupled style provides a basic level of flexiblity that allows us to execute workflows on well-behaved hosts. In reality, hosts may sometimes fail, or not move according to their advertised motion profiles, which may cause errors during the execution of workflows. To handle such errors, sections of the workflow need to be replanned. Replanning is a large area of research which we cannot cover here due space constraints. However, some schemes for error handling are described in [11].

5 Evaluation of Allocation Performance

We evaluated our allocation algorithm by designing a simulator written in Java. We generated a series of random plans and measured the time taken to find an allocation. For comparison, we implemented a naïve algorithm that allocates tasks in a random order while our algorithm employs the heuristic of allocating the tasks with the most constraints first.

Randomly generating a set of *realistic* plans is difficult, mainly because the "realism" of plans is hard to quantify. Instead, our random plan generator generates a *diverse* range of plans based on several parameters:

- r, the number of requirements that actions may draw from
- a, the number of actions in the plan
- g, the number of agents in the system
- p_r, the probability that an action has a specific requirement
- p_c, the probability that an agent has a specific capability
- p_o, the probability of an agent having a constraint between actions

By varying these parameters, we can determine the effect that certain properties of plans have on allocation performance. For the sake of simplicity, we do not

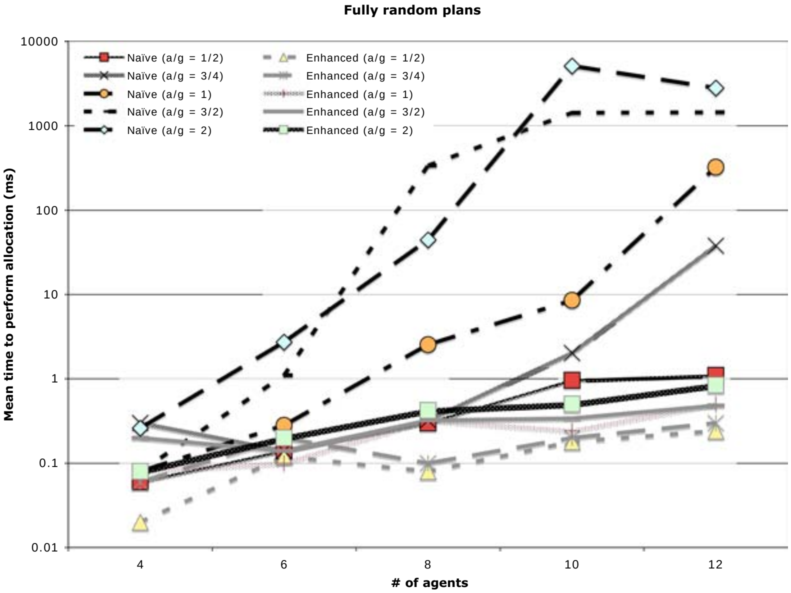


Fig. 5. Algorithm performance when $p_o = 0.1, r = 8, p_r = 0.1, p_c = 0.1$ for a mix of allocatable and unallocatable plans

subdivide spatiotemporal constraints from agent constraints. Rather, we generate a set of constraint tuples directly, without first generating a set of causes for those conflicts.

We performed 50 allocations of fully random plans using a wide range of values for these parameters, and recorded the time taken for each version of the algorithm either to find an allocation or to determine that the plan was impossible to allocate. For comparison, we repeated this procedure with 50 more random plans that were first filtered to ensure that an allocation existed. Since the decision space that the naïve algorithm traverses quickly becomes intractable as the number of actions and agents increases, we enforced an upper-bound of 30 seconds to find an allocation. In the interest of space, we will not present here the results of all combinations of parameters. However, we will note that the parameters that had the greatest effect on algorithm performance were the number of actions in the graph, the ratio of actions to agents, and the probability of conflicts. Figures 5 and 6 show the effect of varying the first two of these parameters with $p_o = 0.1, r = 8, p_r = 0.1$, and $p_c = 0.1$; Figures 7 and 8 show the effect of repeating these experiments with $p_o = 0.3$ and the other parameters unchanged. We also note that our algorithm required no more than 10 ms to allocate any plan of up to 24 actions, whereas the naïve algorithm frequently required more than 30 seconds to allocate the same plans.

The algorithm shows a significant performance improvement over naïve approaches because in the event of a wrong decision, our heuristic algorithm only has to explore the relatively small decision space of that sub-plan before

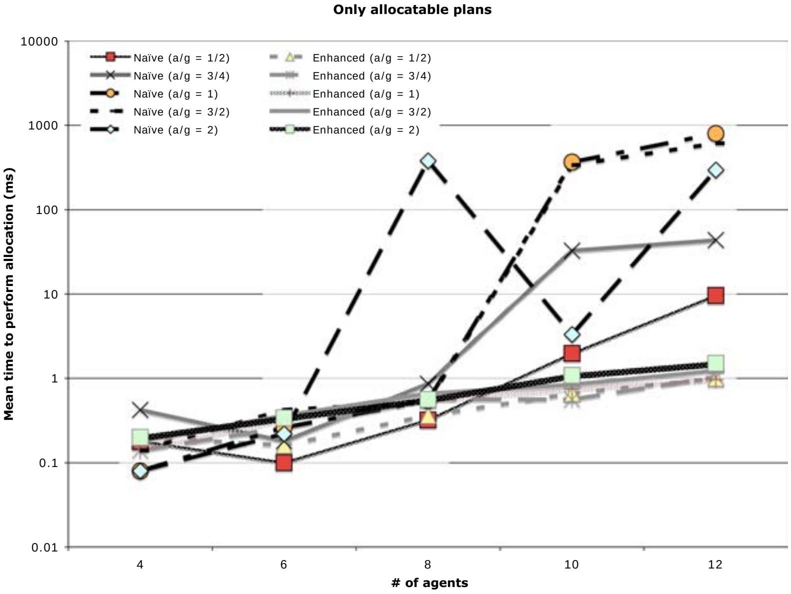


Fig. 6. Algorithm performance when $p_o = 0.1, r = 8, p_r = 0.1, p_c = 0.1$ for allocatable plans only

revisiting the incorrect decision. Furthermore, our algorithm does not need to traverse the entire decision space before it can conclude that a plan is not allocatable.

6 Related Work

Workflows are a powerful model for describing collaborative tasks which traditionally have been popular in business contexts. Owing to this popularity, a workflow commonly has been defined as the automation and management of a business process where a business process is the sequence of tasks which must be done to achieve a certain business goal [12]. While a workflow defines how individuals collaborate by describing the ordered series of tasks that must be performed, the WfMS is the software system that supports the execution of the workflow.

Workflow Management Systems have evolved from isolated legacy systems designed to automate processes for individual businesses, to systems that support workflows in a broader scope. In particular, the Web services community, fueled by the e-commerce revolution, has used the concept of a workflow as a tool for composing existing Web service infrastructure into orchestrated or choreographed distributed applications. Several standardized workflow specification languages such as WS-CDL [13], Wf-XML [14], and BPEL [15] have surfaced to allow tasks to be defined in terms of Web service descriptions. Commercial [16] and open source [2] workflow management engines match Web services to

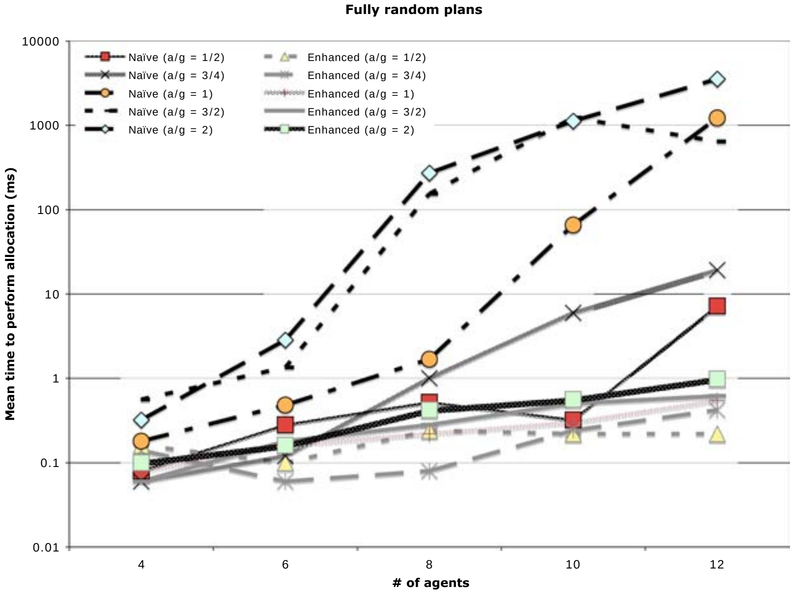


Fig. 7. Algorithm performance when $p_o = 0.3, r = 8, p_r = 0.1, p_c = 0.1$ for a mix of allocatable and unallocatable plans

tasks and execute the workflow. While these WfMSs allow tasks to be executed across geographic and organizational domains, Web services depend on reliable connections and are not designed with mobility in mind.

Recently some systems have been developed to address workflows in mobile settings explicitly. A series of systems such as Exotica/FMDC [17], DOORS [18] and ToxicFarm [19], adapt workflow models for mobility by supporting workflows in the face of network disconnections. Clients in these systems hoard the needed data from a centralized server before they disconnect from the network. Clients may then continue to perform their task(s) while disconnected and the server merges any changes upon reconnection. These systems, therefore, rely on some fixed network infrastructure and assume disconnections are temporary. They also do not exploit the potential for collaboration among clients which are not connected to a central server but which may communicate directly with each other. Another approach to workflows in mobile settings has been through the use of mobile agent technology. The Agent-based Workflow Architecture (AWA) [20] consists of mobile Task Agents which can migrate to mobile devices to execute workflow tasks. The task execution may occur while the device is disconnected provided the Task Agent eventually has the opportunity to migrate back to a Workflow Agent which oversees the execution of the workflow. This agent-based approach is more flexible and appropriate for dynamic settings, but its single point of failure (the Workflow Agent) makes it undesirable for MANETs.

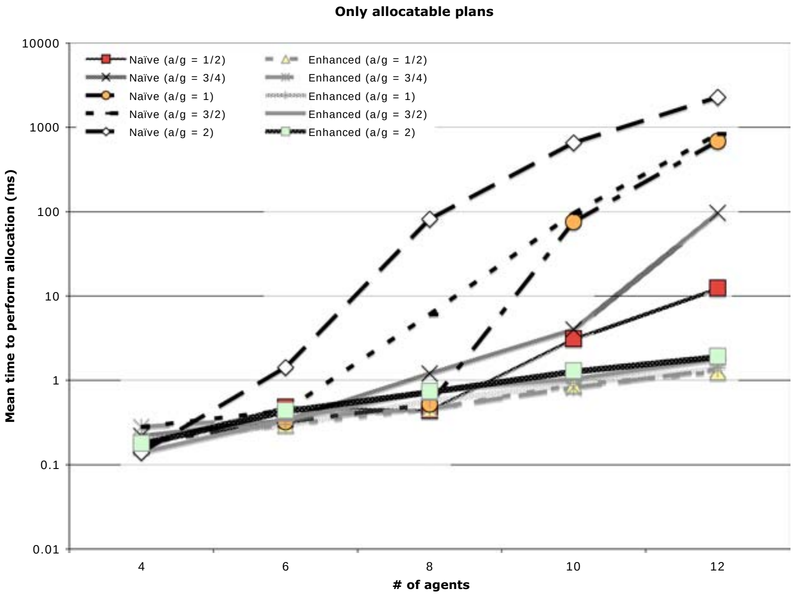


Fig. 8. Algorithm performance when $p_o = 0.3, r = 8, p_r = 0.1, p_c = 0.1$ for allocatable plans only

One emerging system is WORKPAD [21], which supports workflows in MANETs and is designed specifically for emergency/disaster scenarios. WORKPAD operates by centrally coordinating the activities of small teams which perform various disaster recovery activities. These teams operate in separate MANETS but the system assumes that hosts in each MANET operate in tight proximity with a central coordinating host that predicts disconnections and reallocates tasks to other agents or replans the workflow when disconnections occur. Additionally, WORKPAD assumes that the coordinator in each MANET maintains a reliable satellite connection to a central system of P2P servers. Our approach is to build a WfMS for a MANET setting without requiring any of the nodes to have an external connection. As demonstrated in this paper, we have exploited coordination technology to realize this goal.

7 Conclusion

Designing a WfMS targeted to mobile ad hoc networks (MANETs) poses a set of challenges that are very different from those faced when developing such a system for wired networks. There is increased emphasis on the allocation process, and the model of execution must support the dynamic nature of mobile platforms. In this paper, we described an allocation algorithm which uses a heuristic for efficient allocation of tasks and showed how its average case performance is significantly better than a naive allocation. For execution of the workflow, we have adopted a coordination approach for handling the communication and coordination between the participants in the workflow. We exploited the powerful spatiotemporal operations offered by CAST to simplify the design of our WfMS. The result is a lightweight, distributed WfMS that can address many of the demands of mobile environments.

Acknowledgements. This research was supported in part by the National Science Foundation under Grant No. IIS-0534699. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Inc., A.: Automator. http://www.apple.com/macosx/features/automator/
2. Active Endpoints: ActiveBPEL engine. `http://www.activebpel.org/`
3. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: Yet another workflow language. Information Systems 30(4), 245–275 (2005)
4. Curbera, F., et al: Bpel web services for java (bpws4j), `http://www.alphaworks.ibm.com/tech/bpws4j`
5. Roman, G.C., Handorean, R., Sen, R.: Tuple space coordination across space and time. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 266–280. Springer, Heidelberg (2006)
6. Sen, R., Handorean, R., Roman, G.C., Hackmann, G.: Knowledge-driven interactions with services across ad hoc networks. In: Proc. of ICSOC 2004, pp. 222–231 (November 2004)
7. Perkins, C., Royer, E.: Ad-hoc on-demand distance vector routing. In: Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications, pp. 90–100 (February 1999)
8. Perkins, C., Bhagwat, P.: Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In: ACM SIGCOMM'94 Conference on Communications Architectures, Protocols and Applications, pp. 234–244 (1994)
9. Li, L., Horrocks, I.: A software framework for matchmaking based on semantic web technology. In: WWW '03: Proc. of the 12th international conference on World Wide Web, pp. 331–339. ACM Press, New York (2003)
10. Martin, D. et al.: OWL-S: Semantic markup for web services. `http://www.w3.org/Submission/OWL-S/` (November 2004)
11. Sen, R., et al.: Supporting collaborative behavior in manets using workflow. Technical Report WUCSE-06-08, Washington University Dept. of Computer Science (2006)
12. Marshak, R.T.: Workflow: Applying Automation to Group Processes. In: Groupware: Technology and Applications, pp. 71–97. Prentice-Hall, Englewood Cliffs (1995)
13. Kavantzas, N., et al.: Web services choreography description language version 1.0. `http://www.w3.org/TR/ws-cdl-10/` (November 2005)
14. Workflow Management Coalition: Wf-XML 2.0. `http://www.wfmc.org/standards/wfxml_demo.htm`
15. Consortium, O.: OASIS web services business process execution language (WS-BPEL) TC. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel`

16. Inc, O.: Oracle BPEL process manager. `http://www.oracle.com/technology/products/ias/bpel/index.html`
17. Alonso, G., et al.: Exotica/FMDC: Handling disconnected clients in a workflow management system. In: Proc. 3rd International Conference on Cooperative Information Systems, pp. 99–110 (May 1995)
18. Preguiça, N., et al.: Integrating synchronous and asynchronous interactions in groupware applications. In: Fukś, H., Lukosch, S., Salgado, A.C. (eds.) CRIWG 2005. LNCS, vol. 3706, pp. 89–104. Springer, Heidelberg (2005)
19. Godart, C., et al.: The toxicfarm integrated cooperation framework for virtual teams. Distributed and Parallel Databases 15(1), 67–88 (2004)
20. Stormer, H., Knorr, K.: Pda- and agent-based execution of workflow tasks. In: Proceedings of the Informatik 2001, pp. 968–973 (2001)
21. Mecella, M., et al.: Workpad: an adaptive peer to peer software infrastructure for supporting collaborative work of human operators in emergency/disaster scenarios. In: IEEE International Symposium on Collaborative Technologies and Systems (2006)

Fact Spaces: Coordination in the Face of Disconnection

Stijn Mostinckx*, Christophe Scholliers, Eline Philips,
Charlotte Herzeel*, and Wolfgang De Meuter

Programming Technology Laboratory
Vrije Universiteit Brussel, Belgium
{smostinc, cfscholl, ephilips, caherzee, wdmeuter}@vub.ac.be

Abstract. Coordination languages for ad hoc networks with a fluid topology do not offer adequate support to detect and deal with device disconnection. Such a disconnection is particularly relevant if the device provided context information rather than emitting messages, as such context information then becomes invalid. This paper proposes the Fact Space Model which establishes a logic coordination language on top of Lime's federated tuple space. In the model, the federated space offers applications a consistent view of their environment over which they can reason using logic rules. These rules encode which conclusions may be drawn from the presence of particular facts, and are similarly used to ensure the consistency of these conclusions when devices go out of range. By allowing applications to add application-specific hooks to these rules, the application programmer is offered a general-purpose mechanism to respond to the discovery and disconnection of devices.

1 Introduction

The increasing popularity of cellular phones, PDAs and numerous other mobile devices heralds the realisation of the ubiquitous computing and ambient intelligence research visions [1,2]. A crucial aspect of the various scenarios put forward as part of these visions is that mobile users can reap the benefits of being surrounded by a cloud of small, interconnected computing devices. These benefits come in two distinct flavours: providing either additional or smarter behaviour. An example of the former is offering printing facilities to users in the proximity of a printer. A classical example of providing smarter behaviour is signalling a user's cellular phone that it is in a meeting room, allowing it to forward incoming calls to the user's voice mail.

As managing the interplay of services in an ad hoc network with a fluid topology and transient connectivity can become quite complex, this task is typically entrusted to a dedicated coordination language. A well-established coordination language for mobile ad hoc networks is provided by the Lime [3] middleware.

* Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 268–285, 2007.

Lime allows publishing information that steers the coordination of a set of applications in a tuple space that is *transiently shared* between all devices that are currently in range. Devices can respond to the appearance of such tuples by registering *reactions*. Unfortunately, Lime keeps no record of the causal link between a tuple and the reactions the tuple provoked. For the examples described above, this implies that there is no automatic mechanism to remove the printing facilities nor to signal that the cellular phone should stop forwarding calls when the user is no longer in the proximity of the printer respectively when he is no longer in the meeting room.

This paper proposes the Fact Space Model which conceives the federated space as a distributed knowledge base. The transiently shared facts describe the current environment of a device and may be used to customize the behaviour of applications accordingly. These customizations are achieved using a logic coordination language whose rules record the causal link between (a set of) facts and the conclusions that may be drawn from them. As facts are retracted when the device that published them goes out of earshot, the strict enforcing of these causal links – invalidating conclusions when the supporting facts are retracted – is the basis for the fine-grained support to deal with the effects of disconnection offered by the Fact Space Model.

The remainder of the paper is organised as follows: the next section introduces the relevant elements of the Fact Space Model in more detail. Subsequently, section 3 describes Crime (Consistent Reasoning in a Mobile Environment), a prototypical implementation of the Fact Space Model in which we have conducted our experiments. A selection of these experiments is then presented in section 4. Finally, we provide an overview of related work and present our conclusions.

2 The Fact Space Model

The Fact Space Model is a coordination model offering applications deployed on a mobile ad hoc network a consistent view on their environment. According to this view of their environment, the application may offer additional or adapted functionality to its user. An application's view of its environment consists of facts published in a *federated fact space*. Concretely every application can locally publish facts and transparently shares the facts of all nearby devices as long as they remain within communication range. Applications may react to the appearance of facts, using rules specified in a *logic coordination language*. These rules map (a combination of) facts onto a conclusion, which may involve adding new facts to the fact space or triggering application-specific actions. The Fact Space Model allows applications to intercept the retraction of these actions, at which point a compensating action will be performed. Given that facts are retracted automatically when devices disconnect, this mechanism provides fine-grained control over the effects of disconnection. The remainder of this section will explain both the federated fact space and the logic coordination language in more detail.

2.1 Federated Fact Spaces

Applications in the Fact Space Model contribute to the shared view of the environment by publishing facts. Such facts can uniformly represent various types of information ranging from context information over service descriptions up to tasks to be performed. Ensuring these facts are transparently distributed is done using a federated space, as was originally proposed in Lime [3]. With regards to the distribution architecture, the only difference with Lime is that the federated space represents a knowledge base containing facts. Consequently, both the assertion and the retraction of facts are meaningful events which may have repercussions on how the applications behave.

Applications are equipped with at least two fact spaces, a private one to store application-specific facts and one or more *interface fact spaces* for facts that need to be disseminated. In the cell-phone example, the private fact space could contain user preferences detailing which behaviour to adopt when in a particular (kind of) room. The location of a device is typically derived by the proximity to a predefined device, which published location information in an interface fact space, as illustrated by figure 1. The federated fact space consists of all host level fact spaces for devices which are currently in range. The aggregation of all interface fact spaces on a single host allows reusing context information that was previously computed by another application (e.g. a printer is available at a given ip-address) as well as coordination within the boundaries of a single device.

The Fact Space Model handles the discovery of devices in much the same way as Lime; whenever a host discovers the presence of a new device, it will *engage* the fact space of that device. This implies that all facts in the fact space of that device are atomically and transparently *asserted* in the host's fact space.

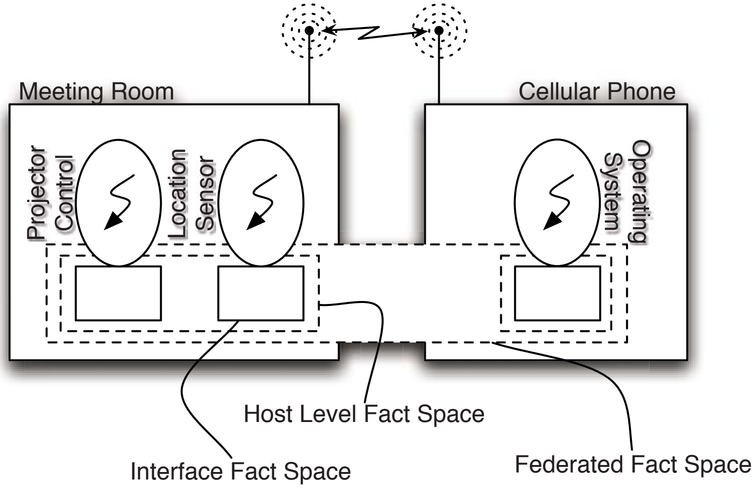


Fig. 1. A Federated Fact Space

At this point each application may adapt to its new environment using the logic coordination language described in the next section. When a device goes out of earshot, its fact space is *disengaged* implying that the facts published by that device are *retracted* from the host's fact space. The Fact Space Model offers applications provisions to respond to such disconnection, as will be explained in the next section.

2.2 Logic Coordination Language

The federated fact space described in the previous section provides each application with a consistent view of their environment. Applications can react to changes in their environment using rules in the logic coordination language of the Fact Space Model. A rule could specify for instance that upon detecting a printer in the environment, it should be added to the list of available printers. Figure 2 illustrates how such a rule could be written in the Fact Space Model. The rule can be read as follows: the application-specific action `addToPrinterList` should be performed (with a given set of arguments) *if and only if* the `public` federated fact space contains a `printer` fact (with the same arguments) whose `dpi` is at least 300.

```
:addToPrinterList(?name, ?ip) :-
  public -> printer(?name, ?dpi, ?ip),
  ?dpi >= 300.
```

Fig. 2. Rule to add printer functionality to an application

For consistency, the rule in figure 2 uses the syntax of Crime, our implementation of the Fact Space Model. As can be seen from the example, logic variables in the rules are prefixed with a question mark. Furthermore, a colon prefix is used to differentiate between application specific actions and logic facts. The former denote implicit method invocations on the `Action` subclasses described below. Finally, facts can be quantified to denote the fact space in which they should be found or asserted.

The main difference between the rule specified above and a similar *reaction* in Lime, is that the former implicitly provides a hook to respond to the disappearance of the fact. This is achieved by enforcing that any custom actions inherit from a common abstract class `Action`. Subclassing from this class implies that application developers need to implement an `activate` method which describes how to respond when the action is derived, as well as a `deactivate` method which describes a compensating action to be performed when the action is retracted as a response to changes in the environment. Note that compensating actions are not required to restore the application's state prior to the execution of the `activate` method.

Figure 3 exemplifies how the Fact Space Model can be used to switch the profile of a cellular phone depending on its location. The example shows a set

```
room(meetingRoom, silent).
room(office, general).

:switch(?profile), profile(?profile) :-
  public -> location(myID, ?room),
  room(?room, ?profile).

:switch(default) :- not profile(?p).
```

Fig. 3. Facts and rules to change the profile of a cellular phone

of private `room` facts detailing the user's preferred profile for a particular room. Furthermore, it contains a rule which triggers the `switch` application-specific action as well as adding a private `profile` fact, upon detecting that the cellular phone is located in a particular room for which the user has configured a preferred profile. Finally, a rule is provided to switch to the `default` profile when no explicit `profile` is prescribed[1].

3 Crime : Implementing the Fact Space Model

Crime (Consistent Reasoning in a Mobile Environment) is an experimental implementation in which we have explored the advantages of the Fact Space Model in building context-aware software. This section highlights some of the key points of the implementation before moving on to the examples presented in the next section.

3.1 Federated Fact Spaces

Crime's implementation of the federated fact space is achieved by building on top of Lime, which is possible due to the fact that the underlying distribution architecture of both systems are identical. This means that the interface fact spaces of an application are in fact wrappers around interface tuple spaces in Lime. Whenever a fact is added to an interface tuple space – either directly by the application or through derivation – this corresponds to adding the fact to the interface tuple space using the `out` operation. Similarly, the atomic engagement and disengagement of fact spaces is transparently handled by the underlying Lime implementation.

Achieving correct behaviour requires that the reasoning engine be informed of the appearance and disappearance of facts which are relevant to its rule base. This is achieved by installing *once-per-tuple* reactions whenever compiling a rule that depends on public facts. These reactions will trigger the reasoning engine and inform it that new facts have become available. The disappearance of hosts is observed in the system tuple space of Lime, which posts a `_host_gone` tuple

[1] The observant reader may notice that this rule is not strictly necessary as similar behaviour could be achieved using the compensating action of `switch`.

upon such an event. The reasoning engine is notified of this event (by means of a reaction) and will subsequently handle the disconnection as described in the next section.

3.2 Logic Coordination Language

In correspondence with the event-driven nature of the systems Crime caters for, it uses a forward-chaining inference engine to trigger rules when fact are asserted to and retracted from the federated fact spaces. The major benefit of forward-chaining is that rather than answering queries, all valid conclusions from a given set of facts are automatically derived. This implies that applications never have to query their context explicitly, instead they are automatically notified of all relevant changes in their environment.

Crime optimizes the derivation of valid conclusions for all available fact using the rete algorithm [4], which we briefly explain below. The gist of the rete algorithm is to combine the actual derivation of conclusions with an optimized caching of intermediate results in a so-called rete network. This caching strategy minimizes the set of rules to be re-evaluated whenever new facts are asserted. Dealing with the frequent retraction of facts – caused by devices which routinely go out of earshot – requires additional support. Whereas the rete algorithm can inherently deal with the retraction of facts, this requires the negated fact to be propagated through the network. This process can be optimized by explicitly keeping track of the dependencies between conclusions and facts using a justification-based truth maintenance system. The remainder of this section will briefly describe both components which jointly implement the logic coordination language prescribed by the Fact Space Model.

Inference Engine. The rete algorithm allows for the efficient derivation of valid conclusions given a set of facts by compiling these rules into a so-called rete network [4] with built-in caches for intermediate results. We exemplify the compilation of such a rule using the example given in figure 3, which dealt with changing the profile of a cellular phone according to the room it was in. The corresponding network for this rule is shown in figure 4.

Deciding when to change the profile of the cellular phone requires the presence of multiple facts. First of all, context information from the `public` interface fact space is needed to determine in which room the phone currently resides. Secondly, a private fact is needed which states the user's preferred profile for that room. Compiling the right-hand side of the rule, involves two steps :

1. Every fact is represented in the network by a *filter node* which will be triggered whenever a new fact is asserted with the correct name. When the fact is qualified, the filter node ensures it will be notified by registering a *reaction* for facts of that type in the underlying Lime distribution layer. As part of filtering out relevant events, the filter nodes also check simple constraints with respect to constants. In the example, the left-most filter node is used to filter out any `location` facts which are not related to the cellular phone.

2. Facts are subsequently combined pairwise using *join nodes* which ensure that constraints spanning multiple facts are upheld. A typical example of such a constraint is that variables with the same name have the same value. Similarly, if a rule contains explicit constraints on two variables (using relational operators such as <=, >= or =\=), these are checked by the first join node which has access to both variables.

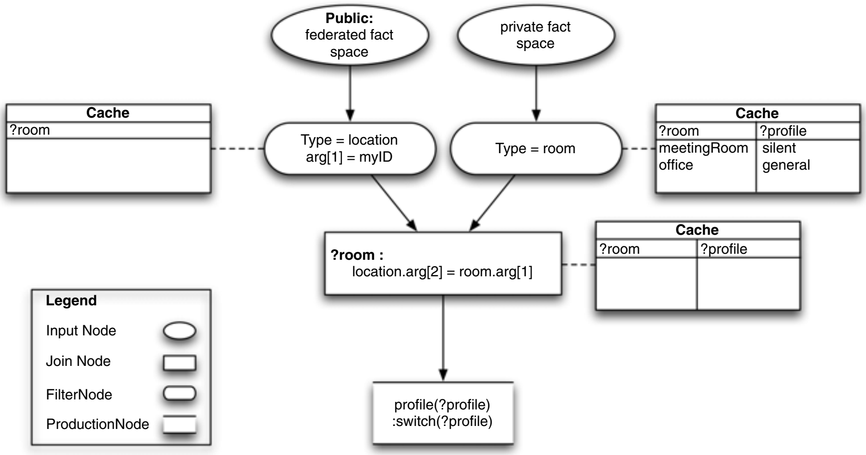


Fig. 4. RETE network to switch the profile of a cellular phone based on location

Having built a network which filters out the relevant combinations of facts that may trigger a rule, compiling its left-hand side is simply adding a *production node* as the child of the lowest join node for that rule. Upon triggering a production node, facts are added to their designated fact spaces (recall that if no qualification is given the fact is considered private) and application-specific actions need to be performed (invoking the `do` method). It is critical that a production node encapsulates an atomic activation, implying that the inference engine may only re-evaluate rules after the production node completed its task. If this constraint is not upheld, inconsistent behaviour may be triggered by mutually exclusive rules with negated clauses in the right-hand side (e.g. `A and !B`, and `A and B`).

Truth Maintenance System. The Fact Space Model introduces a logic coordination language which offers applications the ability to reason over a consistent view of their environment. Given that this environment is a mobile ad hoc network with a fluid topology and transient connectivity, this implies that suites of facts will be frequently retracted as devices move out of range. Such retraction cannot be delayed as this would compromise the consistency of the view an application has of its environment. Moreover, as the connectivity between the devices in a mobile ad hoc network is transient, it is quite likely that the same suite of facts will be reasserted later on as the communication is restored. This particular set of constraints necessitates the introduction of additional support to handle the retraction of facts in an acceptable way.

The RETE algorithm can handle the retraction of facts using rematch-based removal which involves propagating the dual of the removed fact through the network. However, this is a rather costly operation as all checks in the filter and join nodes need to be recomputed, and more importantly, it also removes useful information which could be used when the same fact is reasserted later on.

To optimize the retraction of facts, we use the scaffolding technique described by Perlin [5]. Concretely, this implies that a truth maintenance system is used which keeps track of the causal links between facts and conclusions. These causal links are said to be the *justification* for believing the conclusion. When a fact is subsequently retracted, it is straightforward to identify precisely those conclusions that need to be undone. Moreover, rather than deleting the information upon retracting, the causal link is preserved yet marked to be currently deactivated. If the same fact is reasserted shortly afterwards, this information can be used to reactivate the tuple.

4 Building Context-Aware Applications

We have employed CRIME to build a suite of context-aware applications all of which rely on location-based information. Throughout this paper we have already illustrated a simple application which allows switching the profile of a cellular phone. We have built similar light-weight applications such as an in/out board and a messenger service [6] which are presented in more detail at our website[2]. In this paper we present a slightly more involved application, namely a context-aware jukebox [7,8]. The precise scenario is described first, followed by a discussion of the location-based support for the application before we describe the rules which comprise the juke-box application itself.

4.1 A Context-Aware Jukebox

Alice, Bob and Carol are students which share an apartment. When they are not attending classes or studying, a great deal of their life is all about music. As a consequence, when one of them is relaxing in the joint living room of their apartment, it is quite common to find their jukebox playing music. Unfortunately, the students do not always share one another's taste in music. Whereas this might be a recipe for endless quarrels in any other situation, there is no arguing over who is in charge of choosing the music being played. This is due to the fact that the jukebox is in fact a small computer (a Mac Mini in our setup) which combines information regarding the presence of its users with their respective musical preference to construct a playlist which is acceptable for all present users. Moreover, If Alice, Bob and Carol invite some friends, their musical preferences can be taken into account as well. Finally, the jukebox can also stop playing automatically when it detects that no users are present.

[2] `http://prog.vub.ac.be/amop/research/crime`

A variety of issues are important to build a similar system. First of all, a fairly reliable mechanism is needed to detect when users are in a particular room. Second, when a user enters the room, his musical preferences should be queried and possibly the jukebox should be turned on. Thirdly, when a guest is invited, he may not have the appropriate software to cast his vote in the music selection and this software should be offered to him. Finally, the music selection should be designed to take into account the musical preferences of the present users. We will present the necessary building blocks for this setup in the remainder of this section.

4.2 Detecting a User's Location

Over the past few years, a large quantity of systems to derive a person's location have been developed. Positioning users indoor is usually achieved by giving them a small device which is tracked using wireless communication. The technology used may vary from infrared signals (as used in the seminal work on the Active Badge system [9]) up to the recently developed rfid tags [10]. Whereas the choice of which technology to use may be critical as it influences the size and battery consumption of the device that needs to be carried around, the effect on the proposed application is rather minimal. While developing the context-aware jukebox we have therefore adopted a conservative stance and employed the bluetooth connectivity offered by the jukebox to detect users (by means of their bluetooth-equipped cellphones).

Our concrete setup consists of an event-driven application which is used to detect all reachable bluetooth devices. Upon detecting a new device, it triggers an embedded Crime process by asserting a private fact (e.g. `observed ("Alice's Phone")`). The application further consists of a trivial mapping of such facts onto public `location` facts, as we have used in previous examples in the paper.

4.3 Deploying Applications

We have identified the deployment of applications, in this case to specify one's musical preference, as an essential aspect of the scenario at hand. Obviously, the deployment of Crime applications requires the presence of a minimum of infrastructure. Therefore, every Crime engine is equipped with a minimal application which listens for facts describing available applications. These applications are presented to the user who can opt to deploy them. Figure 5 illustrates how a simple rule can be used to notify the user interface (using a custom action `offer`). The actual deployment (i.e. downloading the application from the url and running it) is done by the underlying application.

One notable property of this application is that the list of available applications can be dynamically updated as the topology of one's mobile network changes. As the service descriptions are represented as public facts, these facts are automatically retracted from the fact space. In response, the `undo` method of the `offer` action will be invoked. In our current implementation, this method will simply remove the application from the list of available applications. It is

```
:offer(?name ?url) :-
  application(?name ?url).

application(jukebox, "http://prog.vub.ac.be/amop/crime/jukebox.zip").
```

Fig. 5. Supporting the advertisement of available applications

also possible to treat applications which were deployed separately using a small set of additional rules. The gist of the system would be to let the deployment application assert a private fact whenever it deploys an application, such that the Crime engine can include this information in its reasoning.

4.4 Music Selection

One of the major advantages of using a coordination language is that it offers a separate medium to specify the coordination and distribution aspects of an application. Concretely this implies that a traditional, non-distributed application can be adapted to provide additional or smarter behaviour when employed in a mobile context, as long as it provides the adequate hooks. To illustrate the potential of the Fact Space Model to be employed in this context, we have opted to use an existing and highly scriptable music player in our jukebox example. The iTunes music player[3] is an established software artifact which provides a rich set of hooks through its AppleScript support.

The Crime support for determining which music to play is described by the rules in figure 6. The first rule uses an application-specific action `toggle` to switch the jukebox on as soon as it detects a person in the living room. This detection is based on the `location` facts which were published by the location sensing support described in section 4.2. The second rule will use `updateRating` to change the average preference attributed to a genre based on the amount of people preferring a particular genre and the total amount of people in the room. This is sufficient to ensure that the music played by the jukebox will be appreciated by the users presented in the living room, as iTunes offers a dedicated *party shuffle* playlist which plays a selection of highly-rated music.

The rating of a particular genre depends on the number of present users which prefer a particular genre, as well the total amount of users in the living room. To compute these numbers, Crime integrates support for two accumulation techniques borrowed from Prolog, namely `findall` and `bagof`. The former is used to find all values for the `?person` variable who are located in the living room and accumulate them in the `?persons` variable. To find out how many people like a particular genre, the `bagof` construct is used, which also accumulates all values for the `?person` variable, yet groups them according to the corresponding values for other free variables (the `?genre` variable in the example).

[3] Copyright 2000-2006 Apple Computer Inc.

```
:toggle() :-
  location(?person, "Living Room").

:updateRating(?genre, ?rating) :-
  category(?genre, ?absolute),
  total(?total),
  rating is ?absolute / ?total.

total(?quantity) :-
  findall(?person, (
    location(?person, "Living Room")),
    ?persons),
  length(?persons, ?quantity).

category(?genre, ?quantity) :-
  bagof(?person, (
    location(?person, "Living Room"),
    prefers(?person, ?genre)),
    ?persons),
  length(?persons, ?quantity).
```

Fig. 6. Rule set to customize jukebox playlist

5 Related Work

The main contribution of the Fact Space Model is that it introduces the notion of causality in a coordination language as a fundamental mechanism to offer applications fine-grained control over the effects of device disconnection in mobile ad hoc networks. The introduction of causality is achieved by using a logic language which essentially reasons over a distributed fact base in order to coordinate the different applications involved. Finally, we have illustrated the applicability of the Fact Space Model to develop context-aware application. Therefore, the Fact Space Model can be contrasted to previous work in three fields of research, namely *coordination languages*, *distributed reasoning systems* and *context-aware computing middleware*. Our discussion of related work will therefore highlight the contributions in each of these research fields which are the most closely related to the Fact Space Model.

5.1 Coordination Languages

Coordination languages can be categorized according to the primitive communication support they offer: communication can be based on *directed* channels which connect two processes (the endpoints of such channels may vary over time), or it can be achieved using the notion of a shared blackboard which provides processes with an *undirected* communication model. The Fact Space Model employs the latter model of communication, making it akin to tuple space-based coordination languages.

During the discussion of the Fact Space Model in section 2, we have already indicated that it borrows the notion of a federated space from LIME [3]. Moreover, through the introduction of an inference engine, our model allows responding to changes in the state of the federated space (rather than operations performed on them) allowing behaviour similar to LIME's reactions. At first sight, reacting to disconnection can be introduced in LIME by allowing such reactions to include a compensating action which is then triggered upon disconnection. To the best of our knowledge, such support has not been introduced thus far. Even with such support in place, some differences remain with respect the Fact Space Model. First of all, the model has innate support for dealing with disconnection, requiring no additional infrastructure to detect and propagate disconnection events. More importantly, by keeping track of the causal connection between events, hand-coding compensating actions is only necessary when the reaction is related to the application's behaviour rather than with how it interacts with other applications.

Tuple space-based communication can equally well be achieved without the notion of a transiently shared tuple space, as is exemplified by TOTA [11]. Instead of implicitly sharing tuples when the hosting devices come into one another's range, TOTA tuples are instrumented with behaviour dictating when they may be copied (or moved) from one tuple space to the next. As tuples are copied, these tuples are not automatically retracted when the emitting device goes out of range. Using TOTA's support to notify tuples of system events, tuples can be removed after a certain period is elapsed. Broadcasting such tuples periodically would then allow to detect disconnection. Another possibility would be to reify the transitive unreachability of a device as an event of its own. However, even if the tuple can be removed when necessary, TOTA suffers from the same problems as LIME, namely that for each reaction that needs to be undone the corresponding compensating action needs to be hand-coded.

5.2 Distributed Reasoning Systems

The Fact Space Model goes beyond traditional coordination languages in that it introduces the notion of an inference engine coupled to a truth maintenance system. It therefore embodies a distributed reasoning engine, with explicit support for dealing with a fluid topology of fact providers. Whereas abundant distributed reasoning engines exist, most of them were introduced purely for the sake of parallelism [12,13], rather than to support reasoning in and about a physically distributed context.

UbiES is an expert system for modelling context-aware applications in a nomadic network setting [14]. The lack of support for ad hoc networks shows in various parts of the proposal: First of all, context information is managed in a context database which is apparently centralized. Moreover, clients are equipped only with a web browser leaving the actual functionality on a server. Finally, UbiES focusses on a single application at a time rather than at coordinating the interplay of various applications.

The Fact Space Model is closely related to DJess [15], a distributed variant of the Jess forward chainer [16]. Similar to the Fact Space Model, DJess connects different inference engines allowing them to share contextual information and trigger one another's reactive behaviours. However, the fundamental difference between both systems is that DJess does not consider applications for a mobile ad hoc network with a fluid topology. A first indication is that DJess features a single centralized server which serves as an intermediate to communicate facts between the individual clients. Furthermore, concurrency control is pessimistic as all facts needs to be locked before executing the actions associated with a rule. Finally, DJess provides no support to detect nor respond to disconnection.

Cooperative artefacts [23] sport a reasoning engine which is specifically tailored for mobile ad hoc networks. Similar to the Fact Space Model, it offers devices the means to reason about a distributed knowledge base representing their immediate environment using Prolog-like rules. The chief difference between both systems lies in the type of applications they support. Cooperative artefacts are embedded devices with very limited resources designed to address one specific problem, whereas the Fact Space Model addresses the collaboration of different idiosyncratic applications on high-end mobile devices. One aspect where the difference in both the underlying hardware and the supported applications has a considerable influence is the inference strategy. Cooperative artefacts rely on a backward chaining inference engine whereas the Fact Space Model emphasizes the importance of a forward chaining inference engine. Section 6.2 explains in more detail why this difference is relevant.

5.3 Context-Aware Computing Middleware

As illustrated in section 4, the Fact Space Model lends itself quite well to the development of context-aware applications. Its forward-chained rules could be interpreted as a structured and declarative subscription to relevant context information publishers. However, comparing it to classical middleware such as the Context Toolkit [6], JCAF [17] and WildCAT [18], a few differences are immediately apparent. First of all, the distribution model underlying the Fact Space Model is based on a federated space rather than on event channels. This is essential as it permits detecting when facts are retracted, and to act upon this observation. Secondly, the Fact Space Model relies on a separate logic language to describe the influence of context, a trait it shares with the approaches discussed in the remainder of this section.

Chisel is a system which primarily focusses on describing the concrete effects of contextual changes by assigning meta-types to runtime objects [19]. As a coordination language, the Fact Space Model was not designed to support such advanced adaptation strategies. Nevertheless, Chisel and the Fact Space Model are quite complementary as the latter offers a declarative language with support to respond to a combination of events as well as a clear distribution model for context information. Both features are lacking in Chisel's event-condition-action language to specify when adaptations should be plugged in.

Gaia's active spaces [20] have been extended with a dedicated language based on first order logic to describe how a system should adapt according to context information [7]. The context model for Gaia's Active Spaces uses first order logic to describe how to adapt to context information. Whereas the use of first order logic results in a similar expressive power to our proposal, quite a few differences remain between Gaia and the Fact Space Model: First of all, Gaia relies on context providers which publish information on channels managed by a centralised infrastructure. Another notable difference is that Gaia uses a standard Prolog implementation, rather than a forward chainer, which necessitates a manual triggering of rules upon context changes. Most importantly, Gaia assumes reliable connections and does not specify how to respond when no context information can be read, nor how this affects previously made decisions.

The Fact Space Model bears the most similarity to the Cortex middleware [21] which uses clips [22], a production system to reason about context information and to trigger reactions using clips's foreign function interface. The chief differences between Cortex and the Fact Space Model is that the latter is based on the notion of a federated space to exchange information, rather than on the publish/subscribe paradigm and most importantly, that the Fact Space Model provides support to meaningfully deal with the retraction of information from the said federated space.

6 Discussion

The Fact Space Model is a logic coordination language which allows reasoning about a distributed knowledge base which functions as a view on the (physical) environment. Moreover, the Fact Space Model requires this view to be kept consistent to ensure that applications do not act upon stale facts. This requirement has a direct influence on two important aspects of the Fact Space model which will be the topic of discussion in this section.

Automatic Retraction. We have adopted the stance that facts whose provider has gone out of earshot are to be retracted from the knowledge base. This paradigmatic decision is evaluated with respect to the opportunities it leaves the programmer to encode persistent facts, *i.e.* facts which are not automatically retracted when its provider goes out of earshot.

Inference Strategy. The choice for a forward chaining inference engine also follows from the requirement that applications should not act upon stale facts. Its data-driven reasoning strategy aligns well with the fact that changes to the knowledge base should be promptly reflected in the application's behaviour. This section provides a more thorough analysis of the differences with a goal-driven backward chaining strategy such as the one used in cooperative artefacts [23].

6.1 Persistent Facts

A distinguishing characteristic of the Fact Space Model is that it reifies device disconnection by retracting all facts published by that device. This is crucial to ensure that applications have a consistent view of their environment where all facts in the knowledge base are guaranteed to be true. Without the automatic retraction of facts, applications could be presented with stale data which leads to a variety of problems. Consider the following example: a location sensor detects that Alice is in her office and publishes this information. Bob's PDA receives this information and subsequently goes out of range. If the location information would not be retracted, the PDA would need to assume that Alice stays put, as it is unable to receive any new information. Such assumptions are clearly a source for unadjusted behaviour such as calling Alice's office number rather than her cellular phone since we presume to know her location. In this case, having no information available is clearly better than having incorrect information.

The semantics of the Fact Space Model align with those of the underlying distribution model of federated spaces, and may seem quite natural in the location-based example given above. However, the retraction of facts is not always the desired semantics. For instance, when a specific printer is out of reach, the fact representing its *availability* should be retracted, yet it can be useful to keep a fact representing information about the printer such as its maximal resolution. At present, Crime offers no direct support to achieve such behaviour, though this behaviour can be conceived in the following way:

Persistent Facts. One can conceive a `:persistent` custom action which adds a fact in its `activate` method, yet does not remove the fact in its `deactivate` method. As exemplified in figure 6.1, this mechanism can be used in the example to add a fact which represents the information about a printer, whenever a (public) fact representing the *availability* of a printer is first detected in the environment. Using a similar custom action, such persistent facts can then also be removed from the system, for instance at the user's request.

```
:persistent(knownPrinter(?name, ?dpi)) :-
  public -> printerAvailable(?name, ?dpi, ?ip),
  not knownPrinter(?name, ?dpi).
```

Fig. 7. Adding persistent facts in Crime

6.2 Inference Strategy

At the heart of the Fact Space Model lies a logic-based coordination language reasoning over a distributed knowledge base. In principle, the interpretation of the rules in the coordination language can be achieved using two distinct inference strategies: namely backward and forward chaining. Backward chaining is a *goal-driven* strategy which attempts to prove queries supplied by the users, whereas forward chaining is a *data-driven* strategy which derives all valid conclusions from a given data set. This section briefly discusses the merits of both strategies and motivates Crime's use of a forward chaining inference engine.

Backward chaining is a commendable strategy to reason over a stable distributed knowledge base which is dedicated to a single application. The underlying reason for this restriction is that in order to respond to the availability of new data, the inference engine needs to be triggered explicitly. For instance, Strohbach *et al.* issue a new attempt to prove a predefined set of goals whenever a change in the distributed knowledge base is observed [23]. Each such attempt completely reconstructs the proof which can be both costly and time-consuming (*e.g.* it may require remote communication). It is therefore important to ensure that such attempts are only made when the result produced by the inference engine is likely to have changed. Hence, the use of backward chaining inference engines for the coordination of different device should be restricted to cases where all facts in the knowledge base are related to a single application. In these cases, the addition of facts in the knowledge base (which in turn triggers the inference engine) is the most likely to affect the outcome of the reasoning process.

Forward chaining on the other hand is a useful strategy to reason over a fluctuating distributed knowledge base which is shared by different idiosyncratic applications. Forward chaining is a data-driven reasoning strategy which implies that the inference engine is triggered whenever new data becomes available. The chief difference with the strategy outline above is that when new data becomes available, the inference engine derives the influence this fact has on previously derived information, rather than reconstructing the entire proof. This strategy is particularly beneficiary when changes to the knowledge base (which may originate from different applications) may not be relevant, as such changes can be filtered out in the first step of reasoning.

7 Conclusion

Starting from a traditional coordination language based on a Lime-like federated tuple space, this paper has explored how applications can be presented with a consistent view of their environment, allowing them to adapt their behaviour according to their current context. Crucial in such applications is that they are notified of *all* relevant changes in the environment, including when information becomes unavailable as a consequence of user mobility or transient disconnection. This has led us to propose the Fact Space Model, which differs from a classical tuple space in two regards: First of all, the federated space is treated as a knowledge base where both the *assertion* and the *retraction* of facts are relevant. Secondly, to provide a minimal and reasonable behaviour when retracting facts, the Fact Space Model is a fully reactive system where the causal relations between reactions and the triggering of facts is documented using logic rules.

The Fact Space Model combines a particular set of features, which clearly sets it apart from existing work on coordination languages, distributed reasoning engines and context-aware computing. First of all, using a federated space to manage context information rather than a publish-subscribe mechanism allows detecting the retraction of information. This feature allows the Fact Space Model to respond to the disconnection of a context provider, a meaningful event when dealing with mobile ad hoc networks with a fluid topology. Furthermore, the Fact Space Model is a fully reactive coordination language based on a forward chained logic language. This language offers a similar programming model as reactive tuple spaces, with the added benefit of delimiting the causal relation between facts and the resulting actions. This causal link allows undoing the effects of a context-dependent adaptation when the context is no longer valid. This feature is currently not offered by coordination languages for mobile ad hoc networks. Finally, the Fact Space Model differs from existing distributed reasoning systems as it uses a distribution model which does not rely on reliable communication or a centralized architecture.

Acknowledgements. The authors would like to thank Brecht Desmet, Kris Gybels, Tom Van Cutsem and the anonymous reviewers for their comments on earlier drafts of this paper. They have significantly improved the readability and quality of this paper.

References

1. Weiser, M.: The computer for the twenty-first century, pp. 94–100. Scientific American, USA (1991)
2. Ducatel, K., Bogdanowicz, M., Scapolo, F., Leijten, J., Burgelman, J.C.: Scenarios for ambient intelligence in 2010. Technical report, EC Information Society Technologies Advisory Group (ISTAG) (2001)
3. Murphy, A., Picco, G., Roman, G.C.: Lime: A middleware for physical and logical mobility. In: Proceedings of the The 21st International Conference on Distributed Computing Systems, pp. 524–536. IEEE Computer Society Press, Los Alamitos (2001)
4. Forgy, C.L.: Rete: a fast algorithm for the many pattern/many object pattern match problem. In: Expert systems: a software methodology for modern applications, pp. 324–341. IEEE Computer Society Press, Los Alamitos (1990)
5. Perlin, M.: Scaffolding the RETE network. In: Proceedings of the International Conference on Tools for Artificial Intelligence, IEEE Computer Society, pp. 378–385. IEEE Computer Society Press, Los Alamitos (1990)
6. Dey, A.K., Salber, D., Abowd, G.D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Human-Computer Interaction, vol. 16 (2001)
7. Ranganathan, A., Campbell, R.H.: An infrastructure for context-awareness based on first order logic. Personal Ubiquitous Comput. 7, 353–364 (2003)
8. Barron, P., Cahill, V.: Using stigmergy to co-ordinate pervasive computing environments. In: WMCSA '04: Proceedings of the Sixth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'04), pp. 62–71. IEEE Computer Society Press, Washington, DC, USA (2004)
9. Want, R., Hopper, A., Falcão, V., Gibbons J.J.: The active badge location system. ACM Transactions on Information Systems 10, 91–102 (1992)
10. Ni, L.M., Liu, Y., Lau, Y.C., Patil, A.P.: Landmarc: indoor location sensing using active rfid. Wirel. Netw. 10, 701–710 (2004)
11. Mamei, M., Zambonelli, F.: Self-maintained distributed tuples for field-based coordination in dynamic networks. In: SAC 2004, pp. 479–486. ACM Press, New York (2004)
12. Eliëns, A.: DLP: a language for distributed logic programming: design, semantics, and implementation. John Wiley & Sons, Inc, New York, NY, USA (1992)

13. Cunha, J., Medeiras, P., Carvalhosa, M., Pereira, L.: Deltaprolog: A distributed logic programming language and its implementation on distributed memory processors. In: Kacsuk, P., Wise, M. (eds.) Implementations of Distributed Prolog, pp. 335–356. John Wiley & Sons, New York, NY, USA (1992)
14. Kwon, O., Yoo, K., Suh, E.: ubiES: An intelligent expert system for proactive services deploying ubiquitous computing technologies. In: HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 3, IEEE Computer Society, Washington, DC, USA, 85.2 (2005)
15. Cabitza, F., Sarini, M., Seno, B.D.: Djess - a context-sharing middleware to deploy distributed inference systems in pervasive computing domains. In: International Conference on Pervasive Services, 2005, ICPS '05., pp. 229–238. IEEE Computer Society Press, Los Alamitos (2005)
16. Friedman-Hill, E.: Jess in Action: Java Rule-Based Systems. Manning Publications Co. (2003)
17. Bardram, J.E., Hansen, T.R.: The aware architecture: supporting context-mediated social awareness in mobile cooperation. In: CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work, pp. 192–201. ACM Press, New York (2004)
18. David, P.C., Ledoux, T.: Wildcat: a generic framework for context-aware applications. In: MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing, pp. 1–7. ACM Press, New York (2005)
19. Keeney, J., Cahill, V.: Chisel: A policy-driven, context-aware, dynamic adaptation framework. In: POLICY '03: Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks, pp. 3–14. IEEE Computer Society Press, Los Alamitos (2003)
20. Roman, M., Hess, C.K., Cerqueira, R., Ranganathan, A., Campbell, R.H., Nahrstedt, K.: Gaia: A middleware infrastructure to enable active spaces. IEEE Pervasive Computing 1, 74–83 (2002)
21. Sorensen, C.F., Wu, M., Sivaharan, T., Blair, G.S., Okanda, P., Friday, A., Duran-Limon, H.: A context-aware middleware for applications in mobile ad hoc environments. In: MPAC '04: Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing, pp. 107–110. ACM Press, New York (2004)
22. Giarratano, J.C., Riley, G.: Expert Systems: Principles and Programming. Brooks/Cole Publishing Co., Pacific Grove, CA, USA (1989)
23. Strohbach, M., Gellersen, H.W., Kortuem, G., Kray, C.: Cooperative artefacts: Assessing real world situations with embedded technology. In: Davies, N., Mynatt, E.D., Siio, I. (eds.) UbiComp 2004. LNCS, vol. 3205, pp. 250–267. Springer, Heidelberg (2004)

Component Connectors with QoS Guarantees[⋆]

Farhad Arbab, Tom Chothia, Sun Meng, and Young-Joo Moon

CWI, Kruislaan 413, Amsterdam, The Netherlands
{Farhad.Arbab,T.Chothia,Meng.Sun,Y.J.Moon}@cwi.nl

Abstract. Connectors have emerged as a powerful concept for composition and coordination of concurrent activities encapsulated as components and services. Compositional coordination models and languages serve as a means to formally specify and implement component and service connectors. They support large-scale distributed applications by allowing construction of complex component connectors out of simpler ones. Modelling, analysis, and ensuring end-to-end Quality of Service (QoS) represent key concerns in such large-scale distributed applications. In this paper we introduce a compositional model of QoS, called *Quantitative Constraint Automata*, that reflects the underlying architecture of component/service composition represented by the Reo connector circuits. These can support compositional reasoning about component/service connectors, modelled as Reo circuits with QoS properties.

Keywords: Coordination, Composition, Reo, Quality of Service, Quantitative Constraint Automata.

1 Introduction

Service-oriented computing (SOC) [14] is an emerging paradigm for the development of complex applications that may run on large-scale distributed systems. Such systems, which typically are heterogeneous and geographically distributed, usually exploit communication infrastructures whose topology frequently varies and components can, at any moment, connect to or detach from. Compositional coordination models and languages provide a formalisation of the “glue code” that interconnects the constituent components/services and organises the communication and cooperation among them in a distributed environment. They support large-scale distributed applications by allowing construction of complex component connectors out of simpler ones.

An example of such a model is Reo [2,6], which offers a powerful glue language for implementation of coordinating component connectors based on a calculus of channels. The work presented here adds Quality of Service to Reo, hence giving us a way of co-ordinating components that takes into account their costs and the costs of coordination.

Quality of Service (QoS) is a measure of the non-functional properties of services along multiple dimensions, such as reliability, security, scalability, response time, etc. Over the past few decades, several quantitative stochastic methods (e.g., stochastic Petri

[⋆] The work in this paper is supported by a grant from the GLANCE funding program of the Dutch National Organisation for Scientific Research (NWO), through project CooPer (600.643.000.05N12).

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 286–304, 2007.

Nets [15], interactive Markov Chains [12] and different kinds of stochastic process algebras [8,11,13]) have been proposed and used in a variety of application areas to study different QoS metrics, mainly on performance evaluation. In recent years we have observed the phenomenon of unbundling of complex monolithic IT and communication services causing large-scale distributed applications to cross intra- and inter-organisational borders. In such environments it becomes a great challenge to provide end-to-end QoS. A promising approach to meet this challenge advocates a compositional approach to provisioning QoS, which allows one to express connector QoS requirements as QoS requirements on the connector's constituent basic elements. However, extending connector specifications directly with information about non-functional concerns limits the reusability of the connector specification and hence any implementation of it. Thus there has been a great deal of interest recently in techniques to provide an effective separation of concerns for end-to-end non-functional requirements and the more stable functional requirements. Reo is a good candidate to serve as the base model for a calculus for compositional QoS, because it offers a fully compositional structure of architecturally meaningful user-defined primitives (i.e, channels) to construct systems with complex behavior, involving arbitrary combinations of synchronous and asynchronous protocols. Analytic or experimental QoS models for such structurally meaningful primitives are more likely to be available, robust, and application context independent (i.e., reusable).

QoS aspects have been handled by using constraint semirings (c-semirings for short) in [9,20]. Assuming a suitable level of abstraction for QoS constraints and a metric for the actual QoS values, c-semirings provide an algebraic structure with two operations, one to select among values and the other to combine values into a new QoS value. Thus compositionality of QoS values is guaranteed in this approach. Chothia and Kleijn extend c-semirings to an algebraic structure called Q-algebra [10], which have two operators for combining QoS values. In this way it becomes possible both to combine QoS values when they occur sequentially and also when they occur concurrently. Moreover, the QoS values can also be compared within the algebra. In general, QoS values are tuples where each component represents a particular aspect: the entries can be of different kinds (numerical to indicate latency, access rights of a service, memory usage, etc.) and a finite number of Q-algebras may be combined as a tuple with tuples as its values. The operators that calculate the combination of costs can be simple (e.g. addition or multiplication) giving an approximation of the true cost or much more complex, giving an accurate picture. The semiring framework, on which Q-algebras are based has been successfully used to model a range of real life situations [19]. Q-algebra is also used to build an automata model called Q-automata [10], which can be used to represent components or channels with costs.

The purpose of this paper is to introduce an operational model for reasoning about general QoS properties of the exogenous channel-based coordination language Reo [1,2,4]. An operational semantics of Reo connector circuits has been provided by constraint automata [6]. Extensions of constraint automata have been investigated for real-time constraints on behaviour of component connectors [3] and to study probabilistic and stochastic properties [5,7].

In this paper we consider the quality aspects of Reo circuits when the specification of channels and component interfaces can involve non-functional requirements. Because connectors, not components, are the primary concern in Reo, our primary interest here is with channels whose behaviour involves quality features and their composition. Remarkably, we do not refer to specific QoS aspects related to concrete services. On the contrary, we are concerned with a unifying theory of QoS model that might be used for different QoS measures. We introduce *Quantitative Constraint Automata* (QCA) as an extension to ordinary constraint automata with QoS values added as additional labels to the individual transitions indicating their use of resources, costs, reliabilities, etc. when executed. To support compositional reasoning, we provide semantic operators corresponding to Reo's main primitives (join and hiding) to model complex component connectors, and thus obtain a compositional framework to generate QCA from a given Reo circuit. Furthermore, we present several notions of simulation for QCA that serve to formalise the replacability of Reo circuits by means of refinement and cost reduction.

The rest of this paper is organised as follows: Section 2 contains a brief introduction to Reo and constraint automata. In Sections 3 and 4 we introduce QCA and show how Reo connectors are equipped with QoS characteristics. In Section 5 we present notions of simulation on QCA and their relationship. Section 6 concludes the paper.

2 Reo and Constraint Automata

In this section we provide a brief introduction to Reo [2], which is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are compositionally built out of simpler ones. We summarise only the main concepts in Reo and its constraint automata semantics here. Further details about Reo and its semantics can be found in [2,4,6].

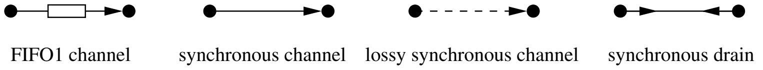


Fig. 1. Some Basic Channels in Reo

Complex connectors in Reo are organised in a network of primitive connectors, called *channels*, that serve to provide the protocol that controls and organises the communication, synchronisation and cooperation among the components/services that they interconnect. Each channel has two *channel ends*. There are two types of channel ends: *source* and *sink*. A source channel end accepts data into its channel, and a sink channel end dispenses data out of its channel. It is possible for the ends of a channel to be both sinks or both sources. Reo places no restriction on the behaviour of a channel and thus allows an open-ended set of different channel types to be used simultaneously together. Each channel end can be connected to at most one component instance at any given time. Figure 1 shows the graphical representation of some simple channel types that will be used in this paper. A *FIFO1 channel* (FIFO1) represents an asynchronous channel with one buffer cell which is empty if no data item is shown in the box (as the one in Figure 1). If a data element d is contained in the buffer of a FIFO1 channel then d is shown inside the box in its graphical representation. A *synchronous channel* (Sync) has a source and a sink end and no buffer. It accepts a data item through its source end iff it can simultaneously dispense it through its sink. A *lossy synchronous channel* (LossySync) is similar to synchronous channel except that it always accepts all data items through its source end. The data item is transferred if it is possible for the data item to be dispensed through the sink end, otherwise the data item is lost. A *synchronous drain* (SyncDrain) has two source ends and no sink end. It can accept a data item through one of its ends iff a data item is also available for it to simultaneously accept through its other end as well.

Connectors are constructed by composing simpler ones via the *join* and *hiding* operations. Channels are joined together in a node which consists of a set of channel ends. Nodes are categorised into source, sink and mixed nodes, depending on whether all channel ends that coincide on a node are source ends, sink ends or a combination of the two. The hiding operation is used to hide the internal topology of a component connector. The hidden nodes can no longer be accessed or observed from outside.

A component can write data items to a source node that it is connected to. The write operation succeeds only if all (source) channel ends coincident on the node accept the data item, in which case the data item is transparently written to every source end coincident on the node. A source node, thus, acts as a replicator. A component can obtain data items, by an input operation, from a sink node that it is connected to. A take operation succeeds only if at least one of the (sink) channel ends coincident on the node offers a suitable data item; if more than one coincident channel end offers suitable data items, one is selected non-deterministically. A sink node, thus, acts as a non-deterministic merger. A mixed node non-deterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends. Note that a component cannot connect to, take from, or write to mixed nodes. At most one component can be connected to a (source or sink) node at a time. The I/O operations are performed through interface nodes of components which are called ports.

Example 1. Figure 2 shows a Reo connector that implements the behaviour of a black-box that in the workflow literature is called a discriminator. The first item that arrives through one of the writers at nodes **A**, **B** or **C** is selected for output through the taker at

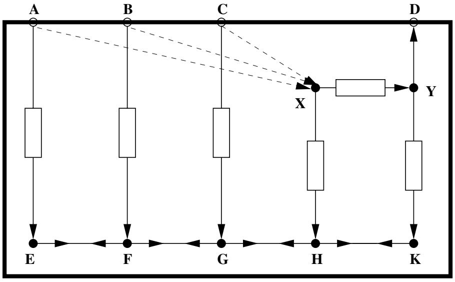


Fig. 2. Discriminator in Reo

node **D**. All three input values must arrive before the next cycle can start. Here, we use this figure to show the visual syntax for presenting Reo connector graphs. The enclosing thick box in this figure represents hiding: the topologies of the nodes (and their edges) inside the box are hidden and cannot be modified. Hiding yields a connector with a number of input/output ports, represented as nodes on the border of the bounding box, which can be used by other entities outside the box to interact with and through the connector. The basic channels used in the connector are synchronous channels (like the edge **YD** in Figure 2), lossy synchronous channels (e.g., **AX**, **BX** and **CX** in Figure 2), synchronous drain whose visual symbol appears as the edge **EF**, **FG**, **GH** or **HK** in Figure 2, and FIFO1 channel, like the edge **AE** in Figure 2.

Constraint automata (CA) [6] were introduced as a formalism to capture the operational semantics of Reo, based on timed data streams which also constitute the foundation of the coalgebraic semantics of Reo [4].

In the sequel, we assume a finite set $\mathscr{N}$ of nodes, and $Data$ as a fixed, non-empty set of data that can be sent and received via channels. A data assignment denotes a function $\delta : N \to Data$ where $N \subseteq \mathscr{N}$. We use $DA(N)$ for the set of all data assignments for the node-set N. CA use a symbolic representation of data assignments by data constraints which are propositional formula built from the atoms "$d_A \in P$", "$d_A = d_B$" or "$d_A = d$" and standard Boolean connectors, where $A, B \in \mathscr{N}$, d_A is a symbol for the observed data item at node A and $d \in Data$, $P \subseteq Data$. We write $DC(N)$ to denote the set of data constraints that at most refer to the observed data items d_A at node $A \in N$, and DC for $DC(\mathscr{N})$. Logical implication induces a partial order $\leq$ on DC: $g \leq g'$ iff $g \Rightarrow g'$.

A constraint automaton over the data domain $Data$ is a tuple $\mathscr{A} = (S, s_0, \mathscr{N}, \longrightarrow)$ where S is a set of states, also called configurations, $s_0 \in S$ is its initial state, $\mathscr{N}$ is a finite set of nodes, $\longrightarrow \subseteq \bigcup_{N \subseteq \mathscr{N}} S \times \{N\} \times DC(N) \times S$, called the transition relation. A transition fires if it observes data items in its respective ports/nodes of the component and according to the observed data, the automaton may change its state.

We write $s \xrightarrow{N,g} s'$ instead of $(s, N, g, s') \in \longrightarrow$ and refer to N as the node-set and g the guard for the transition. By an instance of $s \xrightarrow{N,g} s'$ we mean a transition of the form $s \xrightarrow{N,\delta} s'$ where δ is a data assignment for the nodes in N with $\delta \models g$.

The intuitive operational behaviour of a constraint automaton can be specified by its *runs*. A run in a constraint automaton is defined as a (finite or infinite) sequence of consecutive transition instances

$$r = s_0 \overset{N_0,\delta_0}{\longrightarrow} s_1 \overset{N_1,\delta_1}{\longrightarrow} s_2 \overset{N_2,\delta_2}{\longrightarrow} \dots$$

We require that runs are either infinite or finite runs where the last state s_n does not have any outgoing transition whose node set N consists of mixed nodes only. This requirement can be understood as a *maximal progress assumption* for the mixed nodes.

Figure 3 shows the constraint automata for the basic channels given in Figure 1. Here and in the following, we skip the trivial guard $true$.

Constructing complex connectors out of simpler ones is done by the join operation in Reo. Joining two nodes destroys both nodes and produces a new node on which all

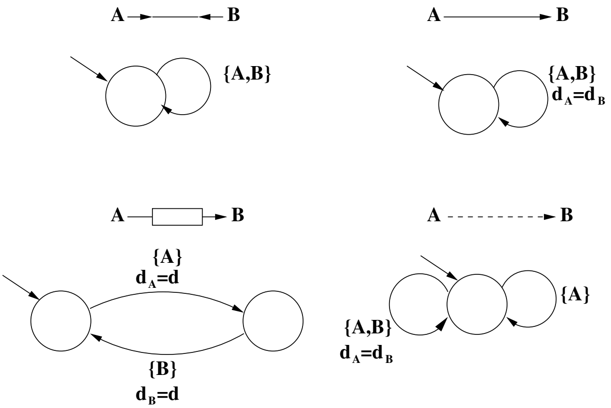


Fig. 3. Constraint Automata for Basic Reo Channels

of their coincident channel ends coincide. Each channel in Reo, as well as the merger nodes are mapped to their own corresponding constraint automata. Reo's join operation can be realised by the product construction of constraint automata.

The product for two given constraint automata $\mathscr{A}_1 = (S_1, s_{0,1}, \mathscr{N}_1, \longrightarrow_1)$ and $\mathscr{A}_2 = (S_2, s_{0,2}, \mathscr{N}_2, \longrightarrow_2)$ is defined as a constraint automaton $\mathscr{A}_1 \bowtie \mathscr{A}_2$ with the components

$$(S_1 \times S_2, (s_{0,1}, s_{0,2}), \mathscr{N}_1 \cup \mathscr{N}_2, \longrightarrow)$$

where $\longrightarrow$ is given by the following rules:

- If $s_1 \xrightarrow{N_1,g_1}_1 s_1'$, $s_2 \xrightarrow{N_2,g_2}_2 s_2'$, $N_1 \cap \mathscr{N}_2 = N_2 \cap \mathscr{N}_1$ and $g_1 \wedge g_2$ is satisfiable, then $\langle s_1, s_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2} \langle s_1', s_2' \rangle$.
- If $s_1 \xrightarrow{N,g}_1 s_1'$, where $N \cap \mathscr{N}_2 = \emptyset$ then $\langle s_1, s_2 \rangle \xrightarrow{N,g} \langle s_1', s_2 \rangle$.
- If $s_2 \xrightarrow{N,g}_2 s_2'$, where $N \cap \mathscr{N}_1 = \emptyset$ then $\langle s_1, s_2 \rangle \xrightarrow{N,g} \langle s_1, s_2' \rangle$.

The first rule applies when there are two transitions in the automata that can fire together. This happens only if there is no shared name in the two automata that is present on one of the transitions but not present on the other one. In this case the transition in the resulting automaton has the union of the name sets on both transitions, and the data constraint is the conjunction of the data constraints of the two transitions. The second rule applies when a transition in one automaton can fire independently of the other automaton, which happens when the names on the transition are not included in the other automaton. The third rule is symmetric to the second one.

Another operator that is helpful for abstraction purposes and can be used in Reo to build connectors from networks is the hiding operator which declares the internal topology of the network as hidden. Hiding takes as input a constraint automaton $\mathscr{A} = (S, s_0, \mathscr{N}, \longrightarrow)$ and a non-empty node-set $M \subseteq \mathscr{N}$. The result is a constraint automaton $\texttt{hide}(\mathscr{A}, M)$ that behaves as $\mathscr{A}$ except that data flow at the nodes $A \in M$ is made invisible. Formally, $\texttt{hide}(\mathscr{A}, M) = (S, s_0, \mathscr{N} \setminus M, \longrightarrow_M)$ where $s \xrightarrow{\bar{N},\bar{g}}_M s'$

iff there exists a transition $s \xrightarrow{N,g} s'$ such that $\bar{N} = N \setminus M$ and $\bar{g} = \exists M[g]$. Here $\exists M[g]$ stands short for $\bigvee_{\delta \in DA(M)} g[d_A/\delta.A | A \in M]$, where $g[d_A/\delta.A|A \in M]$ denotes the syntactic replacement of all occurrences of d_A in g for $A \in M$ with $\delta.A$. Therefore, $\exists M[g]$ formalises the set of data assignments for $\bar{N}$ that are obtained from a data assignment δ for N where g holds by dropping the assignments for the nodes in $N \cap M$.

3 Quantitative Constraint Automata

The basic channels presented in Section 2 assume a perfect behaviour of the channels. However, in real systems, certain channels might have different quality values, like the time of data transmission, reliability, throughput, etc. The behaviour of such channels cannot be captured by constraint automata. Several approaches have been developed to deal with different kinds of quality related properties in connectors [3,5,7]. In this section we introduce the notion of *Quantitative Constraint Automata* (QCA), which is an extension of constraint automata with Q-algebra and forms the basis for compositional specification and reasoning on Quality of Service (QoS) issues for connectors. A *Q-algebra* is an algebraic structure $R = (C, \oplus, \otimes, \oplus, \mathbf{0}, \mathbf{1})$ such that $R_{\otimes} = (C, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ and $R_{\oplus} = (C, \oplus, \oplus, \mathbf{0}, \mathbf{1})$ are both c-semirings. C is a set of QoS values and is called the *domain* of R. The operation $\oplus$ induces a partial order $\leq$ on C, which is defined by $c \leq c'$ iff $c \oplus c' = c'$. Much of the technical details of QCA is identical to or only slightly different than that of CA as presented in [6]. However, for completeness, we describe QCA in full detail here.

The states of a QCA stand for the network configurations, e.g., the contents of the buffers for FIFO channels. Each edge in a QCA is labelled with a tuple (N, g, c) where N is a set of ports (nodes) in a network where data-flow is observed simultaneously, g is a Boolean condition on the observed data items, c is a value in the domain C of a Q-algebra that denotes the QoS or cost value, e.g., the (shortest) execution time, probability, bandwidth, etc.

We consider the quality of service for the basic channels in Reo by three values: t (shortest time for data transmission), c (allocated memory cost for the message transmission) and p (reliability represented by the probability of successful transmission). The corresponding Q-algebras are given as:

- shortest time: $(\mathbb{R}_+ \cup \{\infty\}, max, +, max, \infty, 0)$
- memory cost: $(\mathbb{N}_+ \cup \{\infty\}, max, +, +, \infty, 0)$
- reliability: $([0, 1], min, \times, \times, 0, 1)$

Thus the corresponding QCA for the basic (quantitative) Reo channels SyncDrain, Sync, FIFO1 and LossySync used in this paper are given in Figure 4. For SyncDrain and Sync, the tuples labelled on the channels denote the corresponding quality of service for communication over these channels. For FIFO1, the tuple (t_3, c_3, p_3) presents the QoS values for the operation to input a data to the buffer, and (t_4, c_4, p_4) denotes the QoS values for taking the data from the buffer. For LossySync, the tuple (t_5, c_5, p_5) and (t_6, c_6, p_6) present the QoS values for successful communication over the channel and information lost respectively.

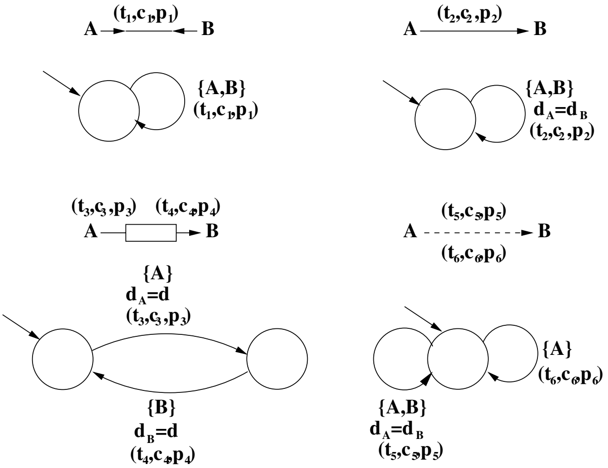


Fig. 4. QCAs for the basic Reo channels

Figure 5 shows on its left a quantitative Reo circuit consisting of three channels **AB**, **AC** and **BC** which are of type SyncDrain, FIFO1 and Sync, respectively. The behaviour of this circuit can be seen as imposing an order on the flow of data items written to **A** and **B**, through **C**. The picture on the right shows a QCA corresponding to this quantitative Reo circuit. In the QCA on the right hand side in Figure 5, location s_0 stands for the initial configuration where the buffer is empty, while location $s(d)$ represents the configuration where the buffer contains a data element d. If node **C** is already for I/O operations in location $s(d)$ then we assume that **C** takes an element d from the buffer and this corresponds to the transition labelled with the set $\{\mathbf{C}\}$, data constraint $d_C = d$ and the QoS values t_4, c_4, p_4 for the shortest time, memory cost and reliability values of the transition respectively. From the initial location s_0 we can input data from nodes **A** and **B** simultaneously, the data input at **A** will be stored in the buffer while the data input at **B** will be directly taken by **C** if the node **C** is ready to take it. The related QoS values are given as in the figure where (t_i, c_i, p_i) represents the related QoS values for the basic channels, which are specified in Figure 4.

Similar to the work on constraint automata model, in QCA we use a finite set $\mathscr{N}$ of nodes and we do not distinguish between write and read operations at the nodes. We assume a fixed, non-empty and finite data domain $Data$ consisting of the data items that can be transmitted through the channels. A data assignment denotes a function $\delta : N \rightarrow Data$ where $\emptyset \neq N \subseteq \mathscr{N}$. We use notations like $\delta = [A \mapsto \delta_A : A \in N]$ to describe the data-assignment that assigns the value $\delta_A \in Data$ to every node $A \in N$. Data constraints can be viewed as a symbolic representation of sets of data assignments. Formally data constraints are propositional formula built from the atoms "$d_A = d_B$", "$d_A = d$" and "$d_A \in P$" where A, B are nodes, d_A is a symbol for the observed data item at node A and $d \in Data$, $P \subseteq Data$. For $N \subseteq \mathscr{N}$, $DA(N)$ denotes the set of all data assignments for the node-set N and $DC(N)$ denotes the set of data constraints that at most refer to the terms d_A for $A \in N$.

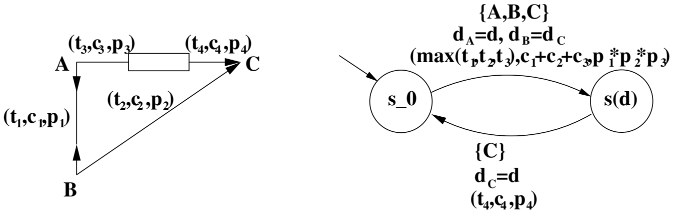


Fig. 5. A Reo Circuit and its Quantitative Constraint Automata

Definition 1. *A Quantitative Constraint Automaton is a tuple* $\mathscr{Q} = (S, s_0, \mathscr{N}, R, \longrightarrow)$ *where*

- S *is a set of states, also called configurations,*
- $s_0 \in S$ *is its initial state,*
- $\mathscr{N}$ *is a finite set of nodes,*
- $R = (C, \oplus, \otimes, \oplus\!\!\!\!\!\!\!\;\mid, \mathbf{0}, \mathbf{1})$ *is a labelled Q-algebra with domain* C *of costs,*
- $\longrightarrow \subseteq \bigcup_{N \subseteq \mathscr{N}} S \times \{N\} \times DC(N) \times C \times S$*, called the transition relation.*

We write $q \xrightarrow{N,g,c} p$ instead of $(q, N, g, c, p) \in \longrightarrow$ and refer to N as the node-set and g the guard. Transitions where the node-set N is non-empty are called *visible*, while transitions with the empty node-set are called *hidden*. In a hidden transition, none of the nodes is visible and the data constraints appear as unknown from outside, thus a hidden transition is witnessed only by its cost value. We denote hidden transitions by the label τ together with the cost. Moreover, a hidden transition cannot be carried out in cooperation with any other transitions. In effect, the ports of a hidden transition are no longer externally accessible to outside, but the costs of a hidden transition still affect the overall cost of a run. Each transition represents a set of possible interactions given by the transition instances that result by replacing the guard g with a data assignment δ where g holds. The QoS metric of a transition step is represented by the cost value c.

QCA composition is a binary function that maps two QCA with consistent Q-algebras into a new QCA. In the following definition we assume that the common nodes of the two QCAs are those where data flow must be synchronised.

Definition 2. *For two QCA* $\mathscr{Q}_1 = (S_1, s_{0,1}, \mathscr{N}_1, R, \longrightarrow_1)$ *and* $\mathscr{Q}_2 = (S_2, s_{0,2}, \mathscr{N}_2, R, \longrightarrow_2)$ *that have the same Q-algebra* R*, their product is defined as a QCA*

$$\mathscr{Q}_1 \bowtie \mathscr{Q}_2 = (S_1 \times S_2, (s_{0,1}, s_{0,2}), \mathscr{N}_1 \times \mathscr{N}_2, R, \longrightarrow)$$

where $\longrightarrow$ *is given by the following rules:*

- *If* $s_1 \xrightarrow{N_1, g_1, c_1}_1 s_1'$, $s_2 \xrightarrow{N_2, g_2, c_2}_2 s_2'$, $N_1 \cap \mathscr{N}_2 = N_2 \cap \mathscr{N}_1 \neq \emptyset$ *and* $g_1 \wedge g_2$ *is satisfiable, then* $\langle s_1, s_2 \rangle \xrightarrow{N_1 \cup N_2, g_1 \wedge g_2, c_1 \oplus\!\!\!\!\!\!\!\;\mid c_2} \langle s_1', s_2' \rangle$.
- *If* $s_1 \xrightarrow{N,g,c}_1 s_1'$*, where* $N \cap \mathscr{N}_2 = \emptyset$ *then* $\langle s_1, s_2 \rangle \xrightarrow{N,g,c} \langle s_1', s_2 \rangle$.
- *If* $s_2 \xrightarrow{N,g,c}_2 s_2'$*, where* $N \cap \mathscr{N}_1 = \emptyset$ *then* $\langle s_1, s_2 \rangle \xrightarrow{N,g,c} \langle s_1, s_2' \rangle$.

A basic constraint automata can be "lifted" to a QCA by placing the null $\mathbf{1}$ value on each of its transitions. Therefore we can define the product of a quantitative and a non-quantitative constraint automata by applying this lifting and then the product, as defined above. We also note the product on QCA is associative.

The effect of hiding a node that is internal to some connector in a Reo circuit is that data flow at that node is no longer observable from outside. However, the quality/cost value should remain the same whether or not the node is hidden. Therefore, the hiding operator for QCA is defined as follows:

Definition 3. *The hiding operator takes as input a QCA $\mathscr{Q} = (S, s_0, \mathscr{N}, R, \longrightarrow)$ and a non-empty node-set $M \subseteq \mathscr{N}$. The result is a QCA $\mathtt{hide}(\mathscr{Q}, M)$ that behaves as $\mathscr{Q}$ except that data flow at the nodes $A \in M$ is made invisible. Formally, $\mathtt{hide}(\mathscr{Q}, M) = (S, s_0, \mathscr{N} \setminus M, R, \longrightarrow_M)$ where*

- *$q \xrightarrow{\bar{N},\bar{g},c}_M p$ iff there exists a transition $q \xrightarrow{N,g,c} p$ such that $N \setminus M = \bar{N} \neq \emptyset$ and $\bar{g} = \exists M[g]$. Here $\exists M[g]$ stands short for $\bigvee_{\delta \in DA(M)} g[d_A/\delta.A | A \in M]$, where $g[d_A/\delta.A|A \in M]$ denotes the syntactic replacement of all occurrences of d_A in g for $A \in M$ with $\delta.A$.*
- *$q \xrightarrow{\tau,c}_M p$ iff there exists a transition $q \xrightarrow{N,g,c} p$ such that $N \setminus M = \emptyset$.*

The automaton starts in an initial state. If the current state is s then an instance $s \xrightarrow{N,\delta,c} s'$ of the outgoing transitions from s is chosen, the corresponding I/O operations are performed and the next state is s'. If there are several outgoing transitions from state s the next transition is chosen non-deterministically. A formalisation of the possible (finite or infinite) observable data flow of a QCA is obtained by the notion of a run.

Definition 4. *Let $s \in S$ be a state of a QCA $\mathscr{Q}$. An s-run in $\mathscr{Q}$ denotes a (finite or infinite) sequence of consecutive transition instances*

$$r = s \overset{N_0,\delta_0,c_0}{\longrightarrow} s_1 \overset{N_1,\delta_1,c_1}{\longrightarrow} s_2 \overset{N_2,\delta_2,c_2}{\longrightarrow} \dots$$

An s-run is called initial if $s = s_0$. For finite runs we require that the last state s' does not have an outgoing hidden transition. This can be understood as a maximal progress assumption for hidden transitions, i.e., steps that do not require any interaction with the environment. The cost of an s-run r is $cost(r) = c_0 \otimes c_1 \otimes c_2 \otimes \cdots$. A weak atomic run $r^{wa} = s \overset{N,\delta,c}{\twoheadrightarrow} s'$ exists iff there is a finite run

$$r = s_0 \overset{\tau_0,c_0}{\longrightarrow} s_1 \overset{\tau_1,c_1}{\longrightarrow} s_2 \overset{\tau_2,c_2}{\longrightarrow} \dots \overset{\tau_{n-2},c_{n-2}}{\longrightarrow} s_{n-1} \overset{N,\delta,c_{n-1}}{\longrightarrow} s_n$$

such that $s_0 = s$, $s_n = s'$ and $c_0 \otimes c_1 \otimes c_2 \otimes \cdots \otimes c_{n-1} = c$. A weak run r^w is defined analogously to a run as a sequence of weak atomic runs.

We use $R(\mathscr{Q})$ to denote the set of all runs of the QCA $\mathscr{Q}$, and $R_0(\mathscr{Q}) \subseteq R(\mathscr{Q})$ the set of all initial runs of $\mathscr{Q}$.

4 Quantitative Reo Circuits

Reo is a channel-based exogenous coordination model wherein complex coordinators, called connectors, are built from instances of basic channel types using certain composition operators. In this paper, as in [3,4,6], we concentrate on connectors that have graphical representations as Reo circuits which express the mechanisms that coordinate the data-flow through the channels connecting the input/output ports of some components, as well as the quality of service aspects of the behaviour.

Example 2. We now consider the example given in Figure 5 again. It depicts a connector which consists of three channels: AB, AC and BC. These channels are of type SyncDrain, FIFO1 and Sync, respectively. According to the definition of the product operator, the composition of the corresponding QCA is given in Figure 5. We define the QoS values for the whole connector as: $t = max(t_1, t_2, t_3) + t_4$, $c = \sum_{i=1}^{4} c_i$ and $p = \prod_{i=1}^{4} p_i$.

	provider1	**provider2**
SyncDrain	$(t_1 = 0, c_1 = 3, p_1 = 1)$	$(t_1 = 0.1, c_1 = 2, p_1 = 1)$
Sync	$(t_2 = 1, c_2 = 2, p_2 = 0.95)$	$(t_2 = 1, c_2 = 8, p_2 = 0.99)$
FIFO1	$(t_3 = 1, c_3 = 2, p_3 = 0.9,$ $t_4 = 0.5, c_4 = 2, p_4 = 0.9)$	$(t_3 = 1, c_3 = 5, p_3 = 1,$ $t_4 = 1, c_4 = 5, p_4 = 0.99)$
LossySync	$(t_5 = 1, c_5 = 2, p_5 = 0.95,$ $t_6 = 0.1, c_6 = 1, p_6 = 0.95)$	$(t_5 = 1, c_5 = 3, p_5 = 0.99,$ $t_6 = 0.2, c_6 = 0.5, p_6 = 0.99)$

Fig. 6. The QoS values for the basic channels

We may have different implementations for the Reo basic channels (and connectors). These implementations may have different QoS properties. For instance, the table in Figure 6 presents the QoS values for every basic channel in the Reo circuit of Figure 5 offered by two different service providers. Using these offerings, the QoS values for the Reo circuit are computed as follows:

- provider1: $t = 1.5, c = 9, p = 0.7695$;
- provider2: $t = 2, c = 20, p = 0.9801$.

Suppose now that a client's QoS requirement of this connector states that the cost of the whole connector (as defined above) should be no more than 15 units and the probability of successful transmission should be greater than 90 percent. Neither of the above two alternatives meet these requirements. However, we can choose the Sync offered by provider1, and the SyncDrain and the FIFO1 offered by provider2, and compose them together. Now the QoS value of the whole connector are: $t = 2$, $c = 14$, $p = 0.9405$, which satisfies the requirement.

Example 3. We now show a more interesting example: the discriminator connector in Reo as given in Figure 2. The basic channel types SyncDrain, Sync, FIFO1 and LossySync used in this example are all equipped with QoS labels. The corresponding

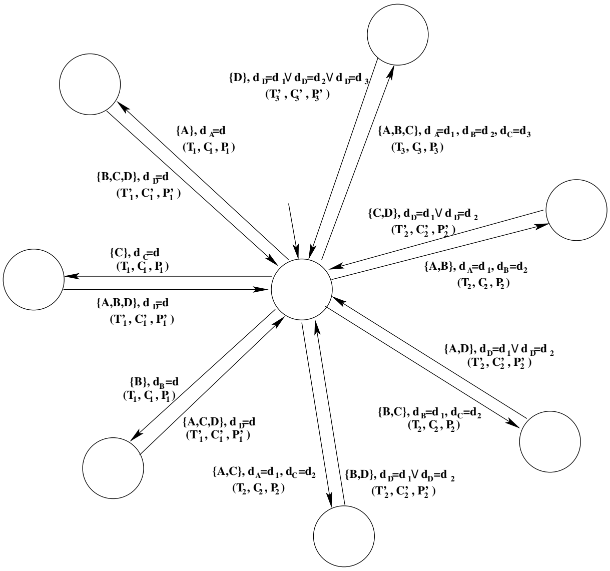


Fig. 7. The QCA for the discriminator connector in Reo

QCAs and the QoS values for basic channels offered by different providers are given in Figures 4 and 6, respectively. The resulting QCA after composition and hiding all of its internal nodes appears in Figure 7, where its QoS properties are given in Figure 8 [1].

The QoS value for the QCA in Figure 7 using the offerings of our two providers in Figure 6 are, for provider1:

$$\begin{aligned}
&T_1 = T_2 = T_3 = 1, T_1' = T_2' = T_3' = 2,\\
&C_1 = 8, C_2 = 12, C_3 = 16, C_1' = 34, C_2' = 33, C_3' = 30,\\
&P_1 \approx 0.69, P_2 \approx 0.48, P_3 \approx 0.5, P_1' \approx 0.33, P_2' \approx 0.39, P_3' \approx 0.41
\end{aligned}$$

and for provider2:

$$\begin{aligned}
&T_1 = T_2 = T_3 = 1, T_1' = T_2' = T_3' = 3.1,\\
&C_1 = 18, C_2 = 26, C_3 = 34, C_1' = 62, C_2' = 61.5, C_3' = 56,\\
&P_1 = 0.99, P_2 \approx 0.98, P_3 \approx 0.97, P_1' \approx 0.89, P_2' \approx 0.89, P_3' \approx 0.92
\end{aligned}$$

As in the previous example, we may have some QoS requirements for the connector that may not be satisfied by either of the above alternatives, but it may be possible to select appropriate basic channels from different providers separately and compose them together to satisfy them, we skip the details here.

[1] In fact the QCA given in Figure 7 is not exactly the QCA resulting from composition and hiding. There are many more states and transitions in the resulting QCA, but most of the extra transitions are τ-transitions. Therefore, we merge them with the visible transitions together as weak atomic runs of Definition 4 and represent them simply as single transitions in Figure 7.

$T_1 = max\{t_3, t_5\}$	$T_1' = max\{t_2, t_3, t_3 + t_4, t_6\} + max\{t_4, t_1 + t_4\}$
$C_1 = 3 \times c_3 + c_5$	$C_1' = 4 \times c_1 + c_2 + 3 \times c_3 + 6 \times c_4 + 2 \times c_6$
$P_1 = p_3^3 \times p_5$	$P_1' = p_1^4 \times p_2 \times p_3^3 \times p_4^6 \times p_6^2$
$T_2 = max\{t_3, t_5\}$	$T_2' = max\{t_2, t_3, t_3 + t_4, t_6\} + max\{t_4, t_1 + t_4\}$
$C_2 = 4 \times c_3 + 2 \times c_5$	$C_2' = 4 \times c_1 + c_2 + 2 \times c_3 + 7 \times c_4 + c_6$
$P_2 = p_3^4 \times p_5^2$	$P_2' = p_1^4 \times p_2 \times p_3^2 \times p_4^7 \times p_6$
$T_3 = max\{t_3, t_5\}$	$T_3' = max\{t_2, t_3 + t_4\} + max\{t_4, t_1 + t_4\}$
$C_3 = 5 \times c_3 + 3 \times c_5$	$C_3' = 4 \times c_1 + c_2 + c_3 + 7 \times c_4$
$P_3 = p_3^5 \times p_5^3$	$P_3' = p_1^4 \times p_2 \times p_3 \times p_4^7$

Fig. 8. The QoS values for the discriminator

5 Simulation on QCA

Simulation relations were first introduced by Milner [18] for the purpose of comparing programs, and widely used later to show abstraction and refinement between models and specifications. They provide a sufficient condition for language inclusion that can be established with low complexity, and their precongruence properties are suited for compositional reasoning. In [6] simulation relations for ordinary constraint automata were defined to verify if two automata are language equivalent or the language of one is contained in the language of the other. In this section we propose to use *quality improving simulation* as a way to guarantee not only the inclusion of languages induced by Reo circuits, but also a higher (or at least equal) quality. For example, we may ask a connector implementation to be always more reliable, or faster than what is required by the specification, where both the specification and the implementation are given as QCA.

Before the definition of simulation, we first define some notation. For a QCA $\mathscr{Q} = (S, s_0, \mathscr{N}, R, \longrightarrow)$, given a relation $\preceq \subseteq S \times S$ and a set of states $P \subseteq S$, P is called $\preceq$-upward closed iff for all states $p \in P$ and $p \preceq p'$, $p' \in P$. For a relation $\preceq$ and two $\preceq$-upward closed sets P_1 and P_2, if there exist $s_1 \in P_1$ and $s_2 \in P_2$, such that $s_1 \preceq s_2$, then $P_1 \cup P_2$ is also $\preceq$-upward closed. Furthermore, let $s \in S$ be a state in $\mathscr{Q}$, $N \subseteq \mathscr{N}$ and $P \subseteq S$, then

$$dc(s, N, P) = \bigvee \{g : s \xrightarrow{N,g,c} s' \text{ for some } s' \in P\}$$

$$dc_w(s, N, P) = \bigvee \{g : s \overset{N,g,c}{\twoheadrightarrow} s' \text{ for some } s' \in P\}$$

and

$$cost(s, N, g, s') = \{c : s \xrightarrow{N,g,c} s' \text{ for some } s' \in P\}$$

$$cost_w(s, N, g, s') = \{c : s \overset{N,g,c}{\twoheadrightarrow} s' \text{ for some } s' \in P\}$$

Note that by introducing upward closed set, we can make an abstraction on QCA and lead to a smaller state space. Here $dc(s, N, P)$ ($dc_w(s, N, P)$ respectively) denotes the weakest data constraint that ensures the existence of an N-transition (a weak atomic run with node set N respectively) from s to P. We now recall the definition of simulation for ordinary constraint automata [6] and give the following definition for QCA. Simulation of QCA considers only the functional aspects of connectors and ignores their quality aspects.

Definition 5. *Let $\mathscr{Q} = (S, s_0, \mathscr{N}, R, \longrightarrow)$ be a QCA and $\preceq$ a reflexive binary relation on S. $\preceq$ is called a simulation for $\mathscr{Q}$ if for all pairs $(s_1, s_2) \in \preceq$, all $\preceq$-upward closed sets $P \subseteq S$ and every $N \subseteq \mathscr{N}$:*

$$dc(s_1, N, P) \leq dc(s_2, N, P)$$

A state s_1 is simulated by another state s_2 (or s_2 simulates s_1), denoted as $s_1 \preceq s_2$, iff there exists a simulation relation $\preceq$ with $(s_1, s_2) \in \preceq$. A QCA $\mathscr{Q}_2$ simulates another QCA $\mathscr{Q}_1$ (denoted as $\mathscr{Q}_1 \preceq \mathscr{Q}_2$) iff the initial state of $\mathscr{Q}_1$ is simulated by the initial state of $\mathscr{Q}_2$[2].

The above notion of simulation is some kind of "strong relation" because it refers to the stepwise behaviour, but does not accumulate the effect of non-observable behaviour alternatives. Thus, we define a notion of weak simulation for QCA analogously to simulation over weak atomic runs as follows:

Definition 6. *For a QCA $\mathscr{Q} = (S, s_0, \mathscr{N}, R, \longrightarrow)$, and a reflexive binary relation $\preceq_w$ on S, $\preceq_w$ is called a weak simulation if for all pairs $(s_1, s_2) \in \preceq_w$, all $\preceq_w$-upward closed sets P and every $N \subseteq \mathscr{N}$,*

$$dc_w(s_1, N, P) \leq dc_w(s_2, N, P)$$

Next, we introduce our notion of *quality improving simulation* (QIS). We distinguish two classes of QISes: QIS and *Weak QIS*. Quality improving simulations impose conditions over the cost values corresponding to the quality of each transition separately. In contrast, weak quality improving simulations constrain the cost values corresponding to the weak atomic runs. The definitions are as follows:

Definition 7. *Let $\mathscr{Q} = (S, s_0, \mathscr{N}, R, \longrightarrow)$ be a QCA and $\lesssim \subseteq S \times S$ a reflexive binary relation on the state space of $\mathscr{Q}$. $\lesssim$ is called a quality improving simulation relation iff $\lesssim$ is a simulation and for all pairs $(s_1, s_2) \in \lesssim$,*

$$\forall c \in cost(s_1, N, g, s_1').\exists c' \in cost(s_2, N, g, s_2').s.t.s_1' \lesssim s_2' \wedge c \leq c'$$

A state s_1 is quality improving simulated by another state s_2, denoted as $s_1 \lesssim s_2$, iff there exists a quality improving simulation relation $\lesssim$ with $(s_1, s_2) \in \lesssim$. A QCA $\mathscr{Q}_2$ quality improving simulates another QCA $\mathscr{Q}_1$ (denoted as $\mathscr{Q}_1 \lesssim \mathscr{Q}_2$) iff the initial state of $\mathscr{Q}_1$ is quality improving simulated by the initial state of $\mathscr{Q}_2$.

Example 4. Let's go back to Example 2. For the two providers, we have two concrete QCAs $\mathscr{Q}_1$ and $\mathscr{Q}_2$ respectively for the QCA given in Figure 5. In other words, the two

[2] Here and in Definition 7 $\mathscr{Q}_1$ and $\mathscr{Q}_2$ should rely on the same set of names, and be combined into a "large" QCA obtained through the disjoint union of the state spaces of $\mathscr{Q}_1$ and $\mathscr{Q}_2$.

QCAs have same behavior but different QoS values. Suppose we only care about the timing performance on the communication with this connector, then we can get the following table focusing on the time value costs. In this table each state is renamed like sx_i where i represents the corresponding QCA and sx represents the derived state name in Figure 5.

$\mathscr{Q}_1$	$\mathscr{Q}_2$
$dc(s0_1, \{A, B, C\}, \{s(d)_1, s(d)_2\}) =$ $\{d_A = d \wedge d_B = d\}$	$dc(s0_2, \{A, B, C\}, \{s(d)_1, s(d)_2\}) =$ $\{d_A = d \wedge d_B = d\}$
$cost(s0_1, \{A, B, C\},$ $d_A = d \wedge d_B = d, s(d)_1) = \{1\}$	$cost(s0_2, \{A, B, C\},$ $, d_A = d \wedge d_B = d, s(d)_2) = \{1\}$
$dc(s(d)_1, \{C\}, \{s0_1, s0_2\}) = \{d_C = d\}$	$dc(s(d)_2, \{C\}, \{s0_1, s0_2\}) = \{d_C = d\}$
$cost(s(d)_1, \{C\}, d_C = d, s0_1) = \{0.5\}$	$cost(s(d)_2, \{C\}, d_C = d, s0_2) = \{1\}$

By defining relation $\lesssim = \{(s0_1, s0_2), (s(d)_1, s(d)_2)\}$, we can find that it is a quality improving simulation, since we have the condition

$$dc(s0_1, \{A, B, C\}, \{s(d)_1, s(d)_2\}) \leq dc(s0_2, \{A, B, C\}, \{s(d)_1, s(d)_2\})$$

and

$$dc(s(d)_1, \{C\}, \{s0_1, s0_2\}) \leq dc(s(d)_2, \{C\}, \{s0_1, s0_2\})$$

Furthermore, the condition in Definition 7 is also satisfied. Therefore, we can conclude that $\mathscr{Q}_1 \lesssim \mathscr{Q}_2$.

Definition 8. *Weak quality improving simulation is defined analogously to quality improving simulation over weak atomic runs. For a QCA $\mathscr{Q} = (S, s_0, \mathscr{N}, R, \longrightarrow)$, and a reflexive binary relation $\lesssim_{wq}$ on S, $\lesssim_{wq}$ is called a weak quality improving simulation iff $\lesssim_{wq}$ is a weak simulation and for all pairs $(s_1, s_2) \in \lesssim_{wq}$,*

$$\forall c \in cost_w(s_1, N, g, s_1').\exists c' \in cost_w(s_2, N, g, s_2').s.t.s_1' \lesssim_{wq} s_2' \wedge c \leq c'$$

The different types of simulation introduced in this section are ordered with respect to how closely they distinguish behaviour alternatives of QCA.

Proposition 1. *Simulation relations satisfy the following partial order:*

$$\begin{aligned}
\mathscr{Q}_1 \lesssim \mathscr{Q}_2 &\Rightarrow \mathscr{Q}_1 \preceq \mathscr{Q}_2 \\
\mathscr{Q}_1 \lesssim \mathscr{Q}_2 &\Rightarrow \mathscr{Q}_1 \lesssim_{wq} \mathscr{Q}_2 \\
\mathscr{Q}_1 \lesssim_{wq} \mathscr{Q}_2 &\Rightarrow \mathscr{Q}_1 \preceq_w \mathscr{Q}_2 \\
\mathscr{Q}_1 \preceq \mathscr{Q}_2 &\Rightarrow \mathscr{Q}_1 \preceq_w \mathscr{Q}_2
\end{aligned}$$

In the context of Reo circuits, the connectors of practical interest are mostly composed from a family of simpler connectors. A compositional approach to modelling and analysis of such a connector is based on the description of the sub-connectors, without further information about the composed connector. As usual, compositionality is captured by the following definition:

Definition 9. *A relation $\lesssim$ between automata is compositional iff*

$$\mathscr{P}_1 \lesssim \mathscr{Q}_1 \wedge \mathscr{P}_2 \lesssim \mathscr{Q}_2 \;\Rightarrow\; \mathscr{P}_1 \bowtie \mathscr{P}_2 \lesssim \mathscr{Q}_1 \bowtie \mathscr{Q}_2$$

The following theorem provides a congruence result for the first three simulation preorders with respect to the product operator. This result allows us to refine a QCA by another one that may improve its quality without affecting its behaviour. However, the compositionality does not hold for the weak quality improving simulation.

Theorem 1. *Simulation, quality improving simulation and weak simulation are compositional with respect to the product operation.*

Proof (sketched): We consider the relation $\mathscr{R} = \{(\langle s_1, s_2\rangle, \langle s'_1, s'_2\rangle) : s_1 \preceq s'_1, s_2 \preceq s'_2\}$ and show that it is a simulation (quality improving simulation and weak simulation respectively). For simulation, we have

$$dc(\langle s_1, s_2\rangle, N, P) = \bigvee(dc_{\mathscr{P}_1}(s_1, N_1, P) \wedge dc_{\mathscr{P}_2}(s_2, N_2, P))$$

and

$$dc(\langle s'_1, s'_2\rangle, N, P) = \bigvee(dc_{\mathscr{Q}_1}(s'_1, N_1, P) \wedge dc_{\mathscr{Q}_2}(s'_2, N_2, P))$$

where $N_1 \cup N_2 = N$. Since $s_1 \preceq s'_1, s_2 \preceq s'_2$, $dc_{\mathscr{P}_i}(s_i, N_i, P) \leq dc_{\mathscr{Q}_i}(s'_i, N_i, P)$, thus $dc(\langle s_1, s_2\rangle, N, P) \leq dc(\langle s'_1, s'_2\rangle, N, P)$ is satisfied.

For QIS, we only need to prove that for all $c \in \mathit{cost}(\langle s_1, s_2\rangle, N, g, \langle \hat{s}_1, \hat{s}_2\rangle)$, there exists $c' \in \mathit{cost}(\langle s'_1, s'_2\rangle, N, g, \langle \hat{s}'_1, \hat{s}'_2\rangle)$, such that $\langle \hat{s}_1, \hat{s}_2\rangle \lesssim \langle \hat{s}'_1, \hat{s}'_2\rangle$ and $c \leq c'$. Since $c \in \mathit{cost}(\langle s_1, s_2\rangle, N, g, \langle \hat{s}_1, \hat{s}_2\rangle)$, according to Definition 2, there exist c_1, c_2 such that $c = c_1 \oplus c_2$ and $s_1 \xrightarrow{N_1, g_1, c_1}_{\mathscr{P}_1} \hat{s}_1$ and $s_2 \xrightarrow{N_2, g_2, c_2}_{\mathscr{P}_2} \hat{s}_2$, where $N_1 \cup N_2 = N$, $g_1 \wedge g_2 = g$. Because $s_1 \lesssim s'_1$, $s_2 \lesssim s'_2$, there always exist transitions $s'_1 \xrightarrow{N_1, g_1, c'_1}_{\mathscr{Q}_1} \hat{s}'_1$ and $s'_2 \xrightarrow{N_2, g_2, c'_2}_{\mathscr{Q}_2} \hat{s}'_2$, where $c_1 \leq c'_1$ and $c_2 \leq c'_2$. Due to the distributivity of $\oplus$ over $\oplus$ in Q-algebra, we have $c \leq c'_1 \oplus c'_2$ where $c'_1 \oplus c'_2 \in \mathit{cost}(\langle s'_1, s'_2\rangle, N, g, \langle \hat{s}'_1, \hat{s}'_2\rangle)$.

The proof for weak simulation is similar like simulation. □

To see that the compositionality does not hold for weak quality improving simulation, consider the QCA given as follows where $R = (\mathbb{N}_+ \cup \{\infty\}, \mathit{max}, +, +, \infty, 0)$:

- $\mathscr{P}_1 = (\{s_0, s_1, s_2\}, s_0, N, R, \longrightarrow)$, in which $s_0 \xrightarrow{\tau, 12} s_1$ and $s_1 \xrightarrow{N, g, 5} s_2$,
- $\mathscr{P}_2 = (\{s'_0, s'_1, s'_2\}, s'_0, N, R, \longrightarrow)$, in which $s'_0 \xrightarrow{\tau, 2} s'_1$ and $s'_1 \xrightarrow{N, g, 9} s'_2$,
- $\mathscr{Q}_1 = (\{t_0, t_1, t_2\}, t_0, N, R, \longrightarrow)$, in which $t_0 \xrightarrow{\tau, 10} t_1$ and $t_1 \xrightarrow{N, g, 7} t_2$,
- $\mathscr{Q}_2 = (\{t'_0, t'_1, t'_2\}, t'_0, N, R, \longrightarrow)$, in which $t'_0 \xrightarrow{\tau, 2} t'_1$ and $t'_1 \xrightarrow{N, g, 9} t'_2$.

According to Definition 2 and 8 we have both $\mathscr{P}_1 \lesssim_{wq} \mathscr{Q}_1$ and $\mathscr{P}_2 \lesssim_{wq} \mathscr{Q}_2$. However, we have $\langle s_0, s'_0\rangle \overset{N, g, 23}{\twoheadrightarrow} \langle s_2, s'_2\rangle$ in $\mathscr{P}_1 \bowtie \mathscr{P}_2$ and $\langle t_0, t'_0\rangle \overset{N, g, 21}{\twoheadrightarrow} \langle t_2, t'_2\rangle$ in $\mathscr{Q}_1 \bowtie \mathscr{Q}_2$ respectively. Therefore, the weak quality improving simulation does not hold between $\langle s_0, s'_0\rangle$ and $\langle t_0, t'_0\rangle$.

On the other hand, the hiding operator preserves all kinds of simulation preorders.

Theorem 2. *If $\mathscr{P} \lessdot \mathscr{Q}$, where $\lessdot$ is any kind of simulation relation, then for a given set of nodes M,* $\mathtt{hide}(\mathscr{P}, M) \lessdot \mathtt{hide}(\mathscr{Q}, M)$.

Proof: To prove the result, it suffices to show that for a QCA $\mathscr{Q} = (S, s_0, \mathscr{N}, R, \longrightarrow)$, and M, any (weak / quality improving / weak quality improving) simulation relation $\mathscr{R}$ for $\mathscr{Q}$ is a (weak / quality improving / weak quality improving) simulation relation for $\mathtt{hide}(\mathscr{Q}, M)$. We first consider the simulation case. Let $\rightsquigarrow$ be the relation such that $s \rightsquigarrow s'$ iff there exists a finite path

$$s \xrightarrow{M,g_1,c_1} s_1 \xrightarrow{M,g_2,c_2} s_2 \cdots \xrightarrow{M,g_n,c_n} s'$$

where all the g_is are satisfiable. Then by considering the M-transitions in $\mathscr{Q}$, we obtain

$$(s_1, s_2) \in \mathscr{R} \wedge s_1 \rightsquigarrow s'_1 \Rightarrow s_2 \rightsquigarrow s'_2 \text{ for some state } s'_2 \tag{1}$$

where $(s'_1, s'_2) \in \mathscr{R}$. Let $(s_1, s_2) \in \mathscr{R}$, N a nonempty subset of $\mathscr{N} \setminus M$, and P an $\mathscr{R}$-upward closed subset of $\mathscr{Q}$. Then for all states s of $\mathscr{Q}$,

$$dc_{\mathtt{hide}(\mathscr{Q},M)}(s, N, P) = \bigvee_{s' \in s^*} (dc_{\mathscr{Q}}(s', N, P) \vee dc_{\mathscr{Q}}(s', N \cup M, P))$$

where $s^* = \{s' \in S : s \rightsquigarrow s'\}$. From (1) we obtain that for every $s'_1 \in s^*_1$ there exists a state $s'_2 \in s^*_2$ with $(s_1, s_2) \in \mathscr{R}$. Because

$$dc_{\mathscr{Q}}(s'_1, N, P) \leq dc_{\mathscr{Q}}(s'_2, N, P)$$
$$dc_{\mathscr{Q}}(s'_1, N \cup M, P) \leq dc_{\mathscr{Q}}(s'_2, N \cup M, P)$$

we get $dc_{\mathtt{hide}(\mathscr{Q},M)}(s_1, N, P) \leq dc_{\mathtt{hide}(\mathscr{Q},M)}(s_2, N, P)$.

The proof for weak simulation is similar like simulation.

Now we consider the case for quality improving simulation. If $(s_1, s_2) \in \mathscr{R}$, then

$$\forall c \in cost(s_1, N, g, s'_1).\exists c' \in cost(s_2, N, g, s'_2).s.t.(s'_1, s'_2) \in \mathscr{R} \wedge c \leq c'$$

Whenever $N \setminus M \neq \emptyset$, we can easily get

$$\forall c \in cost_{\mathtt{hide}(\mathscr{Q},M)}(s_1, N \setminus M, \exists M[g], s'_1).\exists c' \in cost_{\mathtt{hide}(\mathscr{Q},M)}(s_2, N \setminus M, \exists M[g], s'_2).$$
$$s.t.(s'_1, s'_2) \in \mathscr{R} \wedge c \leq c'$$

Therefore, by considering the proof for simulation, we know that for any transition $s_1 \xrightarrow{\hat{N},\hat{g},c} s'_1$ in $\mathtt{hide}(\mathscr{Q}, M)$ with cost c and $(s_1, s_2) \in \mathscr{R}$, there always exist a transition $s_2 \xrightarrow{\hat{N},\hat{g},c'} s'_2$ such that the condition for quality improving simulation is still staisfied in $\mathtt{hide}(\mathscr{Q}, M)$. So $\mathscr{R}$ is still a quality improving simulation in $\mathtt{hide}(\mathscr{Q}, M)$.

For weak quality improving simulation, suppose $(s_1, s_2) \in \mathscr{R}$, then we have

$$\forall c \in cost_w(s_1, N, g, s'_1).\exists c' \in cost_w(s_2, N, g, s'_2).s.t.(s'_1, s'_2) \in \mathscr{R} \wedge c \leq c'$$

If $N \setminus M = \emptyset$, suppose $s'_1 \overset{\hat{N},\hat{g},\hat{c}}{\twoheadrightarrow} s''_1$, and $\hat{N} \setminus M \neq \emptyset$, then we have $s'_2 \overset{\hat{N},\hat{g},\hat{c}'}{\twoheadrightarrow} s''_2$ where $(s''_1, s''_2) \in \mathscr{R}$ and $\hat{c} \leq \hat{c}'$. Therefore, in $\mathtt{hide}(\mathscr{Q}, M)$, there exist transitions

$s_1 \xrightarrow{\tau,c} s_1'$ and $s_2 \xrightarrow{\tau,c'} s_2'$, and thus weak atomic runs $s_1 \overset{\hat{N},\hat{g},c\otimes\hat{c}}{\twoheadrightarrow} s_1''$ and $s_2 \overset{\hat{N},\hat{g},c'\otimes\hat{c}'}{\twoheadrightarrow} s_2''$. According to the definition of Q-algebra, $\otimes$ distributes over $\oplus$, i.e.,

$$a \otimes (c \oplus d) = (a \otimes c) \oplus (a \otimes d)$$

So if $c \leq c'$, $\hat{c} \leq \hat{c}'$, then $c \otimes \hat{c} \leq c' \otimes \hat{c}'$. Therefore, the condition for weak quality improving simulation is still staisfied in $\texttt{hide}(\mathcal{Q}, M)$. □

6 Conclusion

In this paper we introduced QCA as an operational model for reasoning about component connectors with QoS guarantees, together with notions of simulation that are preserved by the QCA product. This result together with the relationship between weak and strong version of (quality improving) simulations provide the basis for analysis of both functional and non-functional aspects of Reo component connectors.

In our future activity, we will work on the development of appropriate logic and quantitative model checking algorithms based on the QCA model, and investigate the expressiveness of our model and its relationship with other extensions to constraint automata, such as in [3,5,7]. Reo and constraint automata have been successfully applied in web service composition [17], but whether the QCA model can be taken as a new, proper entry in the bazaar of web service composition with QoS constraints [16], is still an open research question.

Acknowledgements. The authors are indebted to R. D. van der Mei, C. G. Verhoef, and the members of SEN3 for helpful discussions. We are also grateful to the referees for their constructive criticism.

References

1. Arbab, F.: Abstract Behavior Types: A Foundation Model for Components and Their Composition. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2002. LNCS, vol. 2852, pp. 33–70. Springer, Heidelberg (2003)
2. Arbab, F.: Reo: A Channel-based Coordination Model for Component Composition. Mathematical Structures in Computer Science 14(3), 329–366 (2004)
3. Arbab, F., Baier, C., de Boer, F., Rutten, J.: Models and Temporal Logics for Timed Component Connectors. In: Cuellar, J.R., Liu, Z. (eds.) SEFM2004, 2nd International Conference on Software Engineering and Formal Methods, pp. 198–207. IEEE Computer Society Press, Los Alamitos (2004)
4. Arbab, F., Rutten, J.: A coinductive calculus of component connectors. In: Wirsing, M., Pattinson, D., Hennicker, R. (eds.) Recent Trends in Algebraic Development Techniques. LNCS, vol. 2755, pp. 34–55. Springer, Heidelberg (2003)
5. Baier, C.: Probabilistic Models for Reo Connector Circuits. Journal of Universal Computer Science 11(10), 1718–1748 (2005)
6. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Science of Computer Programming 61, 75–113 (2006)

7. Baier, C., Wolf, V.: Stochastic Reasoning About Channel-Based Component Connectors. In: Ciancarini, P., Wiklicky, H. (eds.) COORDINATION 2006. LNCS, vol. 4038, pp. 1–15. Springer, Heidelberg (2006)
8. Bernardo, M., Gorrieri, R.: A tutorial on EMPA: a theory of concurrent processes with nondeterminism, priorities, probabilities and time. Theoretical Computer Science 202, 1–54 (1998)
9. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint satisfaction and optimization. JACM 44(2), 201–236 (1997)
10. Chothia, T., Kleijn, J.: Q-Automata: Modelling the Resource Usage of Concurrent Components. In: Proceedings of FOCLASA 2006, ENTCS (2006)
11. Götz, N., Herzog, U., Rettelbach, M.: Multiprocessor and distributed system design: the integration of functional specification and performance analysis using stochastic process algebras. In: Donatiello, L., Nelson, R. (eds.) SIGMETRICS 1993 and Performance 1993. LNCS, vol. 729, pp. 121–146. Springer, Heidelberg (1993)
12. Hermanns, H.: Interactive Markov Chains And the Quest for Quantified Quality. LNCS, vol. 2428. Springer, Heidelberg (2002)
13. Hillston, J.: A Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge (1996)
14. Papazoglou, M.P., Georgakopoulos, D.: Service Oriented Computing. Comm. ACM 46(10), 25–28 (2003)
15. Marsan, M.A., Conte, G., Balbo, G.: A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. ACM Transactions on Computer Systems 2(2), 93–122 (1984)
16. Menascé, D.A.: Composing Web Services: A QoS View. IEEE Internet Computing 8(6), 88–90 (2004)
17. Meng, S., Arbab, F.: Web Services Choreography and Orchestration in Reo and Constraint Automata. In: Proceedings of SAC'07 (2007)
18. Milner, R.: An algebraic definition of simulation between programs. In: Cooper, D.C. (ed), Proceedings of the 2nd International Joint Conference on Artifiial Intelligence, London, UK. William Kaufmann, British Computer Society (1971)
19. Mohri, M., Pereira, F., Riley, M.: Weighted automata in text and speech processing. In: ECAI'96 Workshop on Extended Finite State Models of Language, pp. 46–50 (1996)
20. De Nicola, R., Ferrari, G., Montanari, U., Pugliese, R., Tuosto, E.: A process calculus for qos-aware applications. In: Jacquet, J.-M., Picco, G.P. (eds.) COORDINATION 2005. LNCS, vol. 3454, pp. 33–48. Springer, Heidelberg (2005)

Context-Based Adaptation of Component Behavioural Interfaces

Javier Cubo, Gwen Salaün, Javier Cámara,
Carlos Canal, and Ernesto Pimentel

Department of Computer Science, University of Málaga
Campus de Teatinos, 29071, Málaga, Spain
{cubo,salaun,jcamara,canal,ernesto}@lcc.uma.es

Abstract. In the development of component-based systems, components need to be adapted in most of the occasions to work under certain conditions which were not initially predicted by their developers. These conditions are likely to change at runtime, therefore it is very important to provide systems with the ability to alter their behaviour while they are running, depending on the changing conditions of the environment. This paper presents an expressive and graphically-based notation to specify flexible adaptation policies (or mappings) between the interfaces of two or more components to be integrated. In a second step, we propose an algorithm which automatically derives the resulting adaptor from a mapping, and a description of component interfaces. We illustrate our proposal using an *E-book* system.

1 Introduction

Building software systems as a combination of interacting software entities (components, services, etc.) is gaining momentum in the software engineering community, improving productivity as it enables the reuse of existing software entities. These can be adapted in order to fit specific needs within different systems. In such a way, application development is mainly concerned with the selection, adaptation and composition of different pieces of software rather than with the programming of applications from scratch. This approach to systems engineering, designated as Component-Based Software Engineering (CBSE) [24], promotes the use of the so-called *Commercial-Off-The-Shelf* (COTS) components. These are third-party, pre-existing software components which are selected and assembled in order to build a working system. Due to the black-box nature of these components, they must be equipped with external interfaces giving information about component functionality. However, since the interfaces of the constituent components of a system do not always fit one another, they require a certain degree of adaptation in order to avoid mismatching behaviours.

The need to automate these adaptation tasks has driven the development of Software Adaptation [11]. This discipline covers all the topics related to the management of communication between entities. It is characterised by highly dynamic runtime procedures that occur as devices and applications move from

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 305–323, 2007.

network to network, modifying or extending their behaviour, and enhancing the flexibility and maintainability of systems. Software Adaptation promotes the use of software adaptors. These are software entities capable of enabling components with mismatching behaviours to interoperate. They are automatically built from an abstract description of how mismatch can be solved (i.e. adaptation *mapping*), which is based on the description of component interfaces. Mismatch between components can be presented at four different levels. The *signature level* deals with the static aspects of component interoperability. Interfaces at this level provide names, type of arguments and return values, as well as exception types. This kind of adaptation implies solving syntactical differences between signatures. The *behavioural level* specifies the order in which the component messages are exchanged with its environment. The *service level* groups other sources of mismatch, usually related to non-functional properties like temporal requirements, security, etc. Finally, even if components present perfectly matching signatures, follow compatible protocols, and are also compatible at the service level, we must ensure that they are going to behave as expected. The *semantic level* is concerned about component functional specifications (i.e. what they actually do).

Currently, industrial platforms only provide some means to describe components at their signature level (e.g. CORBA's Interface Definition Language). However, most of the time mismatch occurs at the aforementioned behavioural level, due to an incompatibility in the order of the exchanged messages between components, which can lead to deadlock situations. Similarly to several recent proposals in CBSE and Software Adaptation [1,2,3,4,7,17,26], we focus on the behavioural interoperability level, extending interfaces with a description of their protocol and dealing with the different compositional issues between them.

Furthermore, during the last few years we have been witnessing a boost in the use of pervasive computing, where devices and applications dynamically find and use services from their environment. The situation or characteristics of these services are likely to change at runtime, thus an appropriate adaptation of the components or services involved in the system does need to consider all these variations which may affect their behaviour. In order to support this variability in the process of adaptation, this work advocates for the flexible interaction between an arbitrary number of components depending on the current state of the execution of the system (i.e. current context).

However, the support of such flexible adaptation policies can lead to a remarkable degree of complexity in the specification of adaptation, especially in the case of many interacting components. To this purpose, we propose an expressive and user-friendly graphical notation which reduces the complexity of specifying mappings. This is achieved through the incremental specification of the mapping focusing on the different aspects involved. Finally, from such an adaptation mapping, we propose an algorithm to automatically compute the resulting adaptor.

This paper is organised as follows: Section 2 gives a formal model of component interfaces, and presents an *E-Book* system that we use as running example throughout the paper. Section 3 introduces our mapping notation to define correspondences among component messages taking contextual information into account. In Section 4, we formalise the algorithm that generates in an automatic way an adaptor protocol from a contextual mapping. Section 5 compares our proposal with related works, namely context-aware computing, coordination, and component adaptation. Finally, Section 6 draws up the main conclusions and sketches some future works.

2 Component Model

This section presents a simple model to describe component interfaces, as well as an *E-Book* system as case study.

2.1 Component Interfaces

Component interfaces are given using a signature and a behavioural interface. A *signature* is a set of operation profiles. This set is a disjoint union of provided operations and required operations [19]. An operation profile is the name of an operation, together with its argument types, its return type and the exceptions it raises. In addition, we take into account *behavioural interfaces* through the use of process algebra notation. Many process algebras already exist and can be used to this purpose, for instance CSP [14], the π-calculus [20], or LOTOS [16]. In this proposal, we have chosen CCS [18] because it is a simple, concise, and provides the level of expressiveness we need to specify behavioural descriptions of components. Messages involved in processes correspond to the operations used in the signature. All the messages involved in this part of the component description constitute its alphabet. Since we focus on the behavioural interoperability level, we keep only for the signature level the message names appearing in the protocols, therefore we do not deal with operation arguments, return values or exceptions.

Definition 1 (CCS). *A process P in CCS is defined using the following operators, where $\hat{P}$ is a process identifier, and 'a' is a message name:*

$$\begin{aligned} P &::= 0 \mid \alpha.P \mid P+P \mid P \parallel P \mid \hat{P} \\ \alpha &::= a? \mid a! \mid \tau \end{aligned}$$

The specific process 0 denotes termination. Each process can be prefixed by an atomic action α, or composed with other processes using the parallel '$\parallel$' operator or the choice '+' operator. Atomic actions are given by the internal (or silent) action τ, or by input/output actions $a?/a!$. For any process identifier $\hat{P}$ there must be a single definition $\hat{P} = P$.

Automata-based languages such as UML state diagrams, Petri nets, or Labelled Transition Systems (LTSs) can be used as an alternative notation to process algebra to describe component interfaces. They are especially adequate to favour user-friendliness, and to make the graphical specification of interfaces possible. In the sequel, while presenting our context-based adaptation approach, we use LTSs as behavioural interfaces to simplify the writing of the underlying formal aspects. These LTSs can be automatically generated from algebraic processes using the operational rules of the process algebra.

Definition 2 (LTS). *A Labelled Transition System is a tuple* (A, S, I, F, T) *where: A is an alphabet (set of events),* $S \subseteq Id$ *is a set of states,* $I \in S$ *is the initial state,* $F \subseteq S$ *are final states, and* $T \subseteq S \times A \times S$ *is the transition function. Id stands for a set of identifiers.*

Most of the time, components cannot be reused as they are because interactions among them would lead to an erroneous execution, namely a mismatch. Formally, cases of mismatch lead the whole system viewed as a (global) LTS into deadlock states. For an LTS (A, S, I, F, T), a deadlock state is a state $s \in S$ which has no outgoing transition $(\nexists\, (s, l, s') \in T)$ and is not final $(s \notin F)$. In practice, mismatch situations may be caused by message names which do not correspond (a regular use of components makes them interact on the same names of messages), the order of messages which is not respected, a message in one component which has no counter part, or which matches with several messages (*e.g.*, in case of broadcast communication where one component is sending and several ones receiving). We show on the example in Section 2.2 some concrete situations of mismatch. In the remainder, adaptors generated by our approach aims at compensating these cases of mismatch by making the whole system communicate properly.

2.2 Running Example: An E-Book System

We introduce now a case study that we will use throughout the paper to illustrate our approach. It consists of an *E-Book* system, which is a Web service providing access to a database of books. The *E-Book* system is constituted by the *BookServer (BS)* and *BookProvider (BP)* components. Both of them can be accessed by clients. Here, we focus on a single session which corresponds to the connection and use by one client of the *E-Book* system. A comprehensive system handling any number of sessions can be easily derived from our simplified version of the *E-Book* system. A client can use the system to read a book online or to download it. Component interfaces are described below in CCS. We also show the corresponding LTS to favour the understanding. With initial and final states respectively marked using bullet arrows and darkened states. We start with the *Client* interface which is presented in Figure 1.

After a client has opened a session with the *E-Book* system, he/she may: read a book online (*read!*), download a book copy on his/her computer (*save!*), require the change of his/her access rights (*switch!*), receive a present (*gift?*) in gratitude for his/her fidelity, such as an unreleased book not yet included in the catalogue, and finally end the session (*exit!*).

Now, we present the behaviour specification of the *E-Book* system starting with the *BookServer* component in Figure 2.

$$\mathit{Client} = \mathit{user}!.\mathit{passwd}!.\mathit{ConnectedClient}$$

$$\begin{aligned}\mathit{ConnectedClient} = {} & \mathit{read}!.\mathit{ConnectedClient}\\ & +\mathit{save}!.\mathit{ConnectedClient}\\ & +\mathit{switch}!.\mathit{ConnectedClient}\\ & +\mathit{gift}?.\mathit{ConnectedClient}\\ & +\mathit{exit}!.0\end{aligned}$$

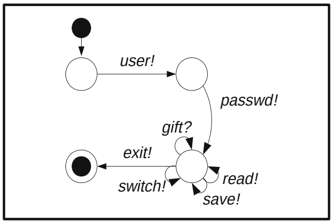


Fig. 1. *Client* Behavioural Interface

$$\mathit{BookServer} = \mathit{login}?.\mathit{ConnectedBS}$$

$$\begin{aligned}\mathit{ConnectedBS} = {} & \mathit{subscribe}?.\mathit{SubscribeBS}\\ & +\mathit{logout}?.0\end{aligned}$$

$$\begin{aligned}\mathit{SubscribeBS} = {} & \mathit{unsubscribe}?.\mathit{ConnectedBS}\\ & +\mathit{offer}!.\mathit{OfferBS}\\ & +\mathit{logout}?.0\end{aligned}$$

$$\begin{aligned}\mathit{OfferBS} = {} & \mathit{unsubscribe}?.\mathit{ConnectedBS}\\ & +\mathit{stopoffer}!.\mathit{SubscribeBS}\\ & +\mathit{logout}?.0\end{aligned}$$

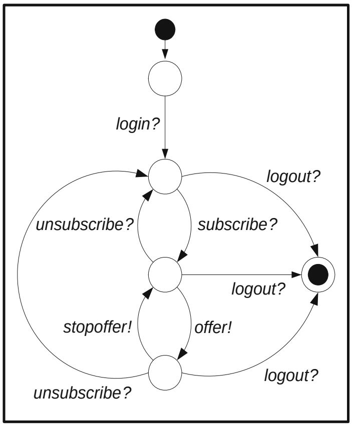


Fig. 2. *BookServer* Behavioural Interface

The *BS* component waits for an incoming connection (*login?*) which is necessary to open a session. Then, it can receive a subscription request. Note that a concrete implementation of the *subscribe?* message will distinguish between the first time the client subscribes (payment of fees), and the following accesses (identification as a *Subscriber* using a password). The subscription terminates with the *unsubscribe?* action. *BS* can decide to give a gift to the client (*offer!*), and to finish this special offer (*stopoffer!*). Finally, it may receive a *logout?* message at any moment.

Last, we give the *BookProvider* component specification depicted in Figure 3. The *on?* message opens a new session. During the session, the *BP* receives requests to access a book (*abstract?*, *full?*), or to download it (*download?*). When the *BP* receives the *offer?* message, then it sends one or several presents (*gift!*) to the client. When receiving the *endoffer?* message, then *BP* stops offering presents, and goes back to the regular book access. The *BP* ends its session with an *off?* message.

These three components cannot be directly used together, since mismatch occurs at several levels: (i) message names, e.g. *save!* in *Client* versus *download?* in *BP*, (ii) 3-party interactions to be explicited, e.g. *exit!/logout?/off?*, and (iii)

$$
\begin{aligned}
\mathit{BookProvider} &= \mathit{on}?.\mathit{ConnectedBP} \\
\mathit{ConnectedBP} &= \mathit{abstract}?.\mathit{ConnectedBP} \\
&\quad +\mathit{full}?.\mathit{ConnectedBP} \\
&\quad +\mathit{download}?.\mathit{ConnectedBP} \\
&\quad +\mathit{offer}?.\mathit{OfferBP} \\
&\quad +\mathit{off}?.0 \\
\mathit{OfferBP} &= \mathit{gift}!.\mathit{OfferBP} \\
&\quad +\mathit{endoffer}?.\mathit{ConnectedBP} \\
&\quad +\mathit{off}?.0
\end{aligned}
$$

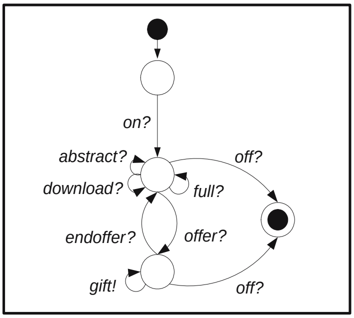


Fig. 3. *BookProvider* Behavioural Interface

correspondences between two messages and a single one, e.g. *user!.passwd!* in *Client* with *login?* in *BS* and *on?* in *BP*.

Furthermore, apart from solving cases of mismatch, we want adaptation to distinguish between the available *user profiles* when translating the messages among components. Using a non-contextual approach, message correspondences are fixed, which means that any client request is always associated to the same target message. This prevents to take changes in these connections into account, and motivates the need of new capabilities that our context-based adaptation approach provides in order to achieve message translation depending on contexts. Then, we distinguish two user profiles according to certain access rights. A client session running on the *E-Book* system can switch between these two profiles:

- *Guest*. These users are only allowed to read the abstract of a book available within the *E-Book* catalogue. A client session always starts with this profile.

- *Subscriber*. These users are those who have paid a subscription fee, and are allowed to access books from the catalogue to read or download a copy of them on their computer.

User profiles are not made explicit in the user interface because the client can execute exactly the same requests independently of its profile.

3 Mapping Notation

Here we define a mapping notation based on vectors expressing correspondences among component messages, and on transition systems to specify the evolution of every component depending on its contexts. To make the mapping writing easier, the graphical part of its specification focuses on one component after the other.

3.1 Contextual Mapping

A first goal of the mapping is to define a mapping between events in the component LTSs. These interactions are formalised through synchronisation vectors. They allow messages with different names to synchronise, and even different numbers of messages to synchronise. These communications may be global to all the involved components, or related to a specific context in which the component is (for instance, *Guest* or *Subscriber* in the *E-Book* system). To take this information into account, we will extend in a second step the mapping language with an LTS-based notation to distinguish these cases.

A *synchronisation vector* denotes a communication between several components. Each event appearing in one vector is executed by one component, and the overall result corresponds to a synchronisation between all the involved components. A vector may involve any number of components. In addition, a vector does not require interactions on the same names of events as it is the case in process algebra for instance.

Definition 3 (Synchronisation Vector). *A synchronisation vector (or vector for short) for a set of components* $C_i = (A_i, S_i, I_i, F_i, T_i)$*,* $i \in \{1, .., n\}$*, is a tuple* $\langle e_1, \ldots, e_n\rangle$ *with* $e_i \in A_i \cup \{\varepsilon\}$*,* ε *meaning that a component does not participate in a synchronisation.*

To identify component messages in a vector, their names are prefixed by the component identifier. Hence, in a vector, all the components which do not participate in an interaction may be removed to simplify the notation, *e.g.,* $\langle c_1 : comm!, c_2 : \varepsilon, c_3 : comm?\rangle$ will be written as $\langle c_1 : comm!, c_3 : comm?\rangle$.

A Contextual LTS (CLTS) is a part of the full mapping focusing only on one component, and making explicit how this component may solve mismatch occurring in its behavioural interface. A CLTS is an extended LTS where states are not only identifiers, but also defines a set of vectors: we refer to them as *contextual states* or *contexts*. Therefore, a CLTS defines the possible interactions that can be run with the other components involved in the system in each of its contextual states. According to its context, a message used in one component may be associated to a different message in another component.

Definition 4 (Contextual LTS). *A Contextual Labelled Transition System (CLTS) is a tuple* $(A_c, S_c, I_c, F_c, T_c)$ *specified over a set of vectors* V *where:* A_c *is an alphabet (set of labels),* $S_c \subseteq Id \times V$ *is a set of contextual states (or contexts for short),* $I_c \in S_c$ *is the initial state,* $F_c \subseteq S_c$ *are final states, and* $T_c \subseteq S_c \times A_c \times S_c$ *is the transition function.*

Now, we define the full mapping, namely *contextual mapping*, for all the components of the system. A contextual mapping is made up of several constituents. First, a set of global vectors correspond to the interactions that do not depend on any context in which the components are. Then, for each component, a CLTS is given defining interactions that are dependent of the context in which the component is. Vectors used in the contextual states are identified as contextual vectors. Last, a set of trigger vectors express how the different CLTSs are connected, and on which interactions the components switch from one context to another.

Definition 5 (Contextual Mapping). *A contextual mapping is defined as a tuple* $(V_g, V_c, CLTS_{i\in\{1,\ldots,n\}}, V_t)$ *built over components* $C_i = (A_i, S_i, I_i, F_i, T_i)$*,* $i \in \{1, .., n\}$ *where:*

- *V_g is a set of vectors global to all the components,*
- *V_c is a set of vectors used in contextual states,*
- *$CLTS_{i\in\{1,\dots,n\}}$ are contextual LTSs defined for every component C_i and built over the set of contextual vectors V_c, and*
- *V_t is a set of trigger vectors corresponding to CLTS interactions.*

Being given such an abstract mapping, we will propose in Section 4 an algorithm that automatically computes a correct adaptor.

3.2 Contextual Mapping for the E-Book System

Now we focus on the *E-Book* system and illustrate how the contextual mapping is written in practice. First of all, we present the different contexts involved in the current system. They are deduced from the component specifications but also from the kind of adaptation to be considered. Since we want to distinguish user profiles within the client behaviour, we use a *Guest* and a *Subscriber* context for it. The *BS* takes into account three contexts, *Guest*, *Subscriber*, and *Offer* in which gifts are sent. Finally, the *BP* interface starts in a *Running* context, and may evolve once in a while to the *Offer* context.

We define the contextual mapping corresponding to the *E-Book* system, by specifying the following elements: global vectors (V_g), contextual vectors (V_c), contextual LTSs ($CLTS_{i\in\{1,\dots,n\}}$), and trigger vectors (V_t).

In the remainder, component messages in the signature mapping are prefixed by the component identifier, namely c, s, and p respectively for the *Client*, *BookServer* and *BookProvider*, to uniquely identify all the messages involved in the system.

Global Vectors. These vectors do not belong to any context, and then are global information, common to all the components. To express the correspondences between two messages on one side (i.e. *user!.passwd!*) and one message on the other (i.e. *login?*, *on?*), we use two tuples (v_{g_1} and v_{g_2} below) where the first one is an independent evolution of the first component, and the second one corresponds to the effective synchronisation. These vectors will be applied successively due to the order of messages *user!* and *passwd!* in the client. Note that vector v_{g_2} is a 3-party communication which means that the three involved components have to synchronise on the messages defined in this vector. This corresponds to broadcast communication with one sender and two receivers.

$$\begin{aligned}
v_{g_1} &= \langle c\!:\!user!\rangle\\
v_{g_2} &= \langle c\!:\!passwd!, s\!:\!login?, p\!:\!on?\rangle\\
v_{g_3} &= \langle c\!:\!exit!, s\!:\!logout?, p\!:\!off?\rangle
\end{aligned}$$

Contextual Vectors. These vectors are used within contextual states, that are the states appearing in the CLTSs we introduce in the next step.

$$\begin{aligned}
v_{c_1} &= \langle c\!:\!read!, p\!:\!abstract?\rangle\\
v_{c_2} &= \langle c\!:\!read!, p\!:\!full?\rangle
\end{aligned}$$

$$v_{c_3} = \langle c : \mathit{save}!, p : \mathit{abstract}? \rangle$$
$$v_{c_4} = \langle c : \mathit{save}!, p : \mathit{download}? \rangle$$
$$v_{c_5} = \langle c : \mathit{gift}?, p : \mathit{gift}! \rangle$$

Contextual LTSs. A Contextual LTS ($CLTS_{i \in \{c,s,p\}}$) is defined for every component $C_{i \in \{c,s,p\}}$, and is built over the set of contextual vectors (V_c). Figure 4 shows $CLTS_c$, $CLTS_s$, and $CLTS_p$ respectively for *Client*, *BookServer*, and *BookProvider* components. It is worth noticing that all the $CLTS_i$ states are final (not depicted in the figure), because the vector v_{g_3} is global and can be applied at any moment.

The contextual vectors used in a specific CLTS correspond to the interactions of the component at hand with the rest of the system depending on its contexts. For instance, the client in its *Guest* context can synchronise with the *BP* with respect to vectors v_{c_1} and v_{c_3}, as well as the component *BP* in its *Running* state, but this latter can also interact on vectors v_{c_2} and v_{c_4}.

Trigger Vectors. They correspond to CLTS interactions, and make explicit on which synchronisations components switch from one context to another.

$$v_{t_1} = \langle c : \mathit{switch}!, s : \mathit{subscribe}? \rangle$$
$$v_{t_2} = \langle c : \mathit{switch}!, s : \mathit{unsubscribe}? \rangle$$
$$v_{t_3} = \langle c : \mathit{switch}!, s : \mathit{unsubscribe}?, p : \mathit{endoffer}? \rangle$$

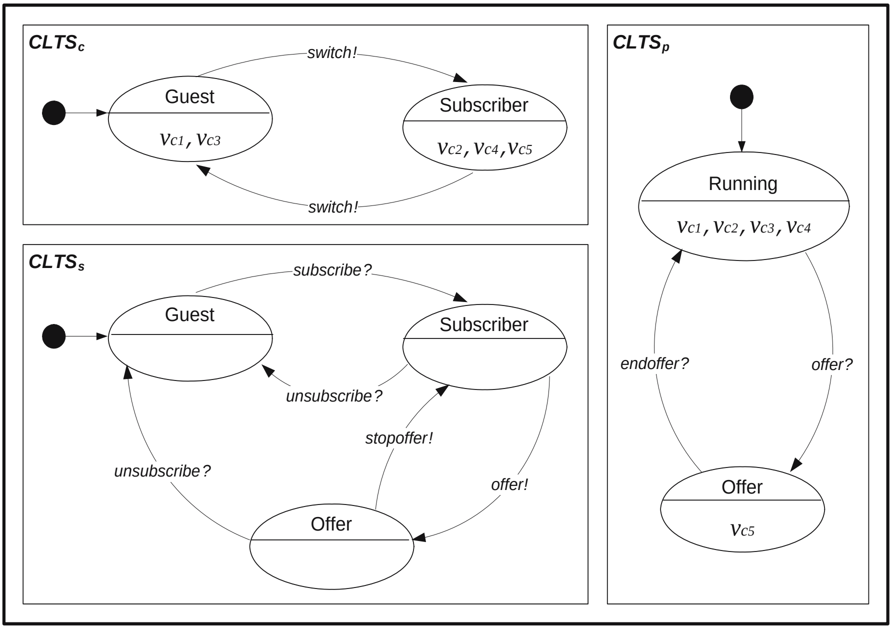


Fig. 4. CLTSs for *Client*, *BookServer*, and *BookProvider* Components

$v_{t_4} = \langle s : \mathit{offer}!, p : \mathit{offer}? \rangle$
$v_{t_5} = \langle s : \mathit{stopoffer}!, p : \mathit{endoffer}? \rangle$

The contextual mapping notation provides a compact representation of all the information required for adaptation, and supplies a flexible mechanism for it, specifying the conditions which will determine contextual changes. Therefore, this new technique intends not only to perform flexible adaptation by altering the behaviour of the system at runtime, but also to change message correspondences as well depending on some changing conditions of the environment.

4 Context-Based Adaptation

In this section, being given a set of components, and a mapping description as formalised respectively in Sections 2 and 3, we propose an algorithm to compute an adaptor component. Note that from this mapping which is defined manually by the designer, our approach generates automatically the resulting adaptor. An adaptor is a third-party component that is in charge of coordinating all the components involved in the system with respect to a set of interactions defined in the mapping. Consequently, all the components communicate through the adaptor as illustrated in Figure 5 where a component C_1 wants to activate a component C_2. The mapping for this simple example is given by the vector $\langle c_1 : on!, c_2 : \mathit{activate}? \rangle$. We emphasize that the adaptor interacts with the components using the same name of messages but the reversed directions, e.g. communication between *on!* in c_1 and *on?* in the adaptor. Furthermore, the adaptor always starts a set of interactions formalised in a vector by the receptions (*on?*), and next handles the emissions (*activate!*).

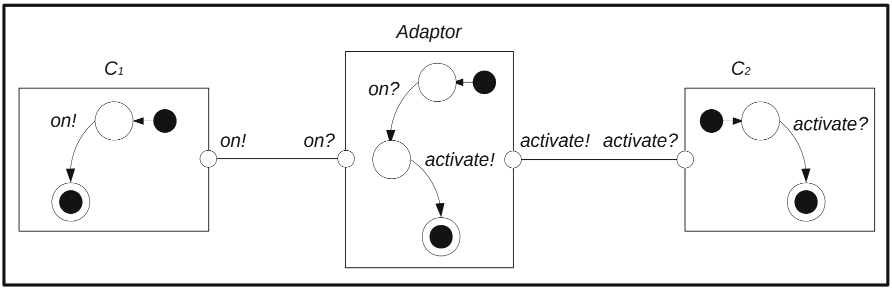


Fig. 5. A Simple Example of Adaptation

4.1 Contextual LTS Product

In Section 3, to make the writing of the mapping easier, we have specified one CLTS for each component. Now, for adaptor generation purposes, we need a single description of the mapping, then the first step is to compute a single

CLTS from a set of CLTSs. Such a computation is achieved computing a product between all the involved CLTSs which synchronise on events expressed in the trigger vectors V_t specified in the mapping.

Definition 6 (CLTS Product). *The CLTS product of n CLTSs defined as tuples $(A_{c_i}, S_{c_i}, I_{c_i}, F_{c_i}, T_{c_i})$, $i \in \{1, .., n\}$, with respect to a set of trigger vectors V_t, is the CLTS $(A_c, S_c, I_c, F_c, T_c)$ such that:*

- $A_c = A_{c_1} \times \ldots \times A_{c_n}$,
- $S_c = \{((s_1, \ldots, s_n), V_1 \sqcap \ldots \sqcap V_n) \mid \forall\, i \in \{1, \ldots, n\}, (s_i, V_i) \in S_{c_i}\}$,
- $I_c = ((o_1, \ldots, o_n), V_1 \sqcap \ldots \sqcap V_n)$ *with* $\forall\, i \in \{1, \ldots, n\}$, $I_{c_i} = (o_i, V_i)$,
- $F_c = \{((f_1, \ldots, f_n), V_1 \sqcap \ldots \sqcap V_n) \mid \forall\, i \in \{1, \ldots, n\}, (f_i, V_i) \in F_{c_i}\}$, *and*
- T_c *contains a transition* $(((s_1, \ldots, s_n), V), \langle a_1, \ldots, a_n \rangle, ((s'_1, \ldots, s'_n), V'))$, *with* $V' = V'_1 \sqcap \ldots \sqcap V'_n$, *iff there is a state* $((s_1, \ldots, s_n), V)$ *in* S_c, *and there is a vector* $\langle l_1, \ldots, l_n \rangle$ *in* V_t *such that for every* i *in* $\{1, \ldots, n\}$ *and* $(s_i, V_i) \in S_{c_i}$:
  - *if* $l_i = \varepsilon$ *then* $s'_i = s_i$, $V'_i = V_i$ *and* $a_i = \varepsilon$,
  - *otherwise there is a transition* $((s_i, V_i), a_i, (s'_i, V'_i))$ *with* $a_i = l_i$ *in* T_{c_i}.

The resulting CLTS automaton may be simplified keeping only states reachable from its initial state. Function $\sqcap$ computes the intersection of sets of vectors discarding empty sets: $\sqcap V_{i \in \{1, \ldots, n\}} = \{\, V_1 \cap \ldots \cap V_k \mid \forall\, j \in \{1, \ldots, k\}\ V_j \neq \varnothing\}$.

CTLS Product for the E-Book System. We will illustrate throughout section 4 the different steps in the construction of the adaptor for the *E-book* system. First, the CLTS product is computed using $CLTS_c$, $CLTS_s$, and $CLTS_p$, and the set of trigger vectors $\{v_{t_1}, v_{t_2}, v_{t_3}, v_{t_4}, v_{t_5}\}$ defined in Section 3.2. The resulting global CLTS is presented in Figure 6 with three different contextual states, and five transitions labelled with trigger vectors. All the states in the resulting CLTS are final. We recall that vectors in the resulting product contextual states are computed using an intersection on the non-empty contextual states of component CLTSs. Identifiers of the product contextual states are built as a tuple gathering the identifiers of the constituent contextual states.

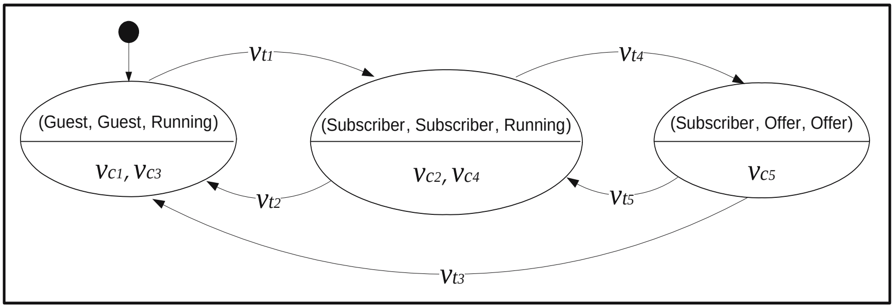


Fig. 6. CLTS Product for the *E-Book* System

4.2 Contextual Product

After the CLTS product, the next step, namely contextual product, aims at generating from a global CLTS obtained using Definition 6 and a set of global vectors defined in the mapping, a single mapping expression representing all the possible interactions between the different components. Intuitively, the contextual product flattens the contextual states of the global CLTS applying either contextual vectors or global vectors. The result of this product is an LTS with vectors on transitions that correspond to a high-level description of the adaptor. Section 4.3 will refine this LTS generating interactions (with the components) from vectors appearing on transitions to finally obtain the final adaptor.

Definition 7 (Contextual Product). *The contextual product of n LTS* $C_i = (A_i, S_i, I_i, F_i, T_i)$*,* $i \in \{1,..,n\}$*, with a CLTS* $(A_M, S_M, I_M, F_M, T_M)$ *and a set of global vectors* V_g*, is the LTS* (A, S, I, F, T) *such that:*

- $A = A_M \times A_1 \times \ldots \times A_n$*,*
- $S = S_M \times S_1 \times \ldots \times S_n$*,*
- $I = (I_M, I_1, \ldots, I_n)$*,*
- $F = F_M \times F_1 \times \ldots \times F_n$*, and*
- *(i) a transition* $(((s_m, V_m), s_1, \ldots, s_n), \langle a_m, a_1, \ldots, a_n \rangle, ((s'_m, V'_m), s'_1, \ldots, s'_n))$ *is in* T *iff there is a state* $((s_m, V_m), s_1, \ldots, s_n)$ *in* S*, and there is a transition* $((s_m, V_m), \langle l_1, \ldots, l_n \rangle, (s'_m, V'_m))$ *in* T_M*, or*
  (ii) a transition $(((s_m, V_m), s_1, \ldots, s_n), \langle a_m, a_1, \ldots, a_n \rangle, ((s_m, V_m), s'_1, \ldots, s'_n))$ *is in* T *iff there is a state* $((s_m, V_m), s_1, \ldots, s_n)$ *in* S*, and* $\langle l_1, \ldots, l_n \rangle \in V_m$ *or* $\langle l_1, \ldots, l_n \rangle \in V_g$*, and in both cases (i) and (ii), for every* i *in* $\{1, \ldots, n\}$*:*
  - *if* $l_i = \varepsilon$ *then* $s'_i = s_i$ *and* $a_i = \varepsilon$*,*
  - *otherwise there is a transition* (s_i, a_i, s'_i) *with* $a_i = l_i$ *in* T_i*.*

Similarly to Definition 6, the resulting LTS may be simplified keeping only reachable states. Moreover, elements of the CLTS used for convenience purposes to build the contextual product above have to be discarded after its computation. This is simply achieved removing all the first elements appearing in first places of states and transitions.

Contextual Product for the E-Book System. The next step is to compute the contextual product using the global CLTS in Figure 6, and the component behavioural interfaces. Figure 7 shows the LTS resulting of this product where contextual states are replaced by several transitions. Transitions hold vectors which correspond to all the possible interactions of the entities involved in the system at hand with respect to the given contextual mapping.

4.3 Adaptor Generation

We present in this section an algorithm which computes the adaptor from the component interfaces and a contextual mapping. The essential part of the algorithm is Step 1 where first the product of all CLTS appearing in the contextual

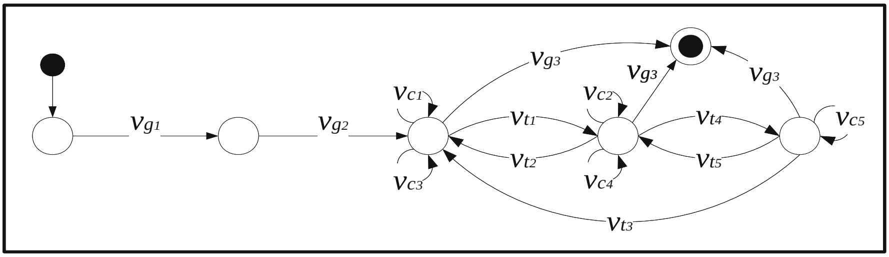


Fig. 7. Contextual Product for the *E-Book* System

mapping is computed (function *compute_CLTS_product*, Def. 6), and then the contextual product uses this global CLTS, components C_i and global vectors V_g to generate a contextual product (function *compute_contextual_product*, Def. 7). The rest of the algorithm corresponds to common processings to obtain the resulting adaptor [12]. The main idea is to derive labels from vectors appearing in the LTS computed in Step 1 so as to make communications with the involved components possible.

Removing deadlocks is important to suppress spurious interactions that will not leave the system in a stable (final) state. As an example, consider two components, a controller C and a device D. Suppose that the controller can turn on and off the device, and that it can be warned whether the device is removed or added back. Suppose also that the device can be turned on and off, and also be unplugged, which stops its execution. The process algebraic interfaces of these simple components are as follows:

$$C = on!.off!.C + removed?.added?.C$$
$$D = on?.off?.D + unplugged!.0$$

A possible adaptation mapping could be: $\langle c : on!, d : on?\rangle$, $\langle c : off!, d : off?\rangle$, and $\langle c : removed?, d : unplugged!\rangle$. Yet, a synchronisation on $\langle c : removed?, d : unplugged!\rangle$ could make the controller C engage in the second branch of its behaviour, and then it would wait indefinitely for an *added* event. This would produce a deadlock state in the corresponding adaptor that has to be removed.

Function *remove_deadlocks* in Step 2 of the algorithm is achieved recursively removing transitions and states yielding deadlocks: find a deadlock state s, remove s and any transition t with target s, and do this until there is no more such s in the LTS. For all vector transition, we apply reversal of direction messages and computation of all possible interleavings (function *compute_permutations*) starting by the receptions. Message directions are reversed (Steps 6-7) because all messages will go through the adaptor, and this latter has to synchronise with these messages using completementary directions (!/? or ?/!). Interleaving of messages (Steps 8-13) is important when vectors involve more than two messages in a communication (*e.g.,* in case of broadcast communication), then interleavings make the system non-deterministic on the order in which messages occur, and the adaptor accepts any possible combination.

Algorithm 1. *adaptor_generation*

constructs an adaptor for a set of components being given a contextual mapping
inputs components $C_1, \ldots, C_n$ with each $C_i = (A_i, S_i, I_i, F_i, T_i)$, and a contextual mapping $CM = (V_g, V_c, CLTS_i, V_t)$
output adaptor $CA = (A, S, I, F, T)$

```
$P := compute\_contextual\_product(C_i, compute\_CLTS\_product(CLTS_i, V_t), V_g)$
$P_{restr} := remove\_deadlocks(P)$
$S_{\mathrm{add}} := \varnothing$
$T_A := \varnothing$
for all $t = (s = (s_1, \ldots, s_n), (l_1, \ldots, l_n), s' = (s'_1, \ldots, s'_n))$ in $T_{P_{\mathrm{restr}}}$ do
    $L_{\mathrm{rec}} = \{l? \mid l! \in (l_1, \ldots, l_n)\}$
    $L_{\mathrm{em}} = \{l! \mid l? \in (l_1, \ldots, l_n)\}$
    $\mathrm{Seq}_{\mathrm{rec}} = compute\_permutations(L_{\mathrm{rec}})$
    $\mathrm{Seq}_{\mathrm{em}} = compute\_permutations(L_{\mathrm{em}})$
    for all $(R = (r_1, \ldots, r_i), E = (e_1, \ldots, e_p)) \in \mathrm{Seq}_{\mathrm{rec}} \times \mathrm{Seq}_{\mathrm{em}}$ do
        $T_A := T_A \cup \{s \xrightarrow{r_1} q_1, \ldots, q_{i-1} \xrightarrow{r_i} q_i, \ldots, q_{i+1} \xrightarrow{e_1} q_{i+2}, \ldots, q_{n-1} \xrightarrow{e_p} s'\}$
        $S_{\mathrm{add}} := S_{\mathrm{add}} \cup q_{k \in 1..n-1}$
    end for
end for
return $CA = (A_{P_{\mathrm{restr}}}, S_{P_{\mathrm{restr}}} \cup S_{\mathrm{add}}, I_{P_{\mathrm{restr}}}, F_{P_{\mathrm{restr}}}, T_A)$
```

The complexity of our algorithm lies mainly in the CLTS and contextual product construction, that is $\mathbf{O}(\mid S \mid^n)$ where S is the largest set of states and n the number of involved entities.

Proof (sketch). We prove that the resulting adaptor makes all the component LTSs terminate in one of their final states. First, by definition of the contextual product, the resulting automaton contains all the interactions possible among the components as formalised in the different sets of vectors given in the contextual mapping. Next, the removal of deadlocks suppresses all the deadlocks leading to non-final states, hence all the correct termination states are preserved. Last, directions of messages are reversed that means that all the component messages involved in correct synchronisations (labels of the product automaton) will have a matching event in the resulting adaptor. In addition, computation of permutations implies an application order (first receptions, then emissions) with all possible interleavings within these two steps. Hence, reversal of directions and permutations preserve all the simple transitions appearing in the product, hence they also preserve all the correct final states existing after the removal of deadlocks. □

Our algorithm does not take reordering of messages into account, which may be needed in some adaptation scenarios. Reordering of events is needed to ensure a correct interaction when two communicating entities have messages which are not ordered as required. However, in our proposal, such a reordering of messages can be specified making it explicit in the writing of the adaptation mapping. Let us illustrate these ideas on a simple example depicted in Figure 8. We consider two components, C_1 and C_2, exchanging login and request information, with messages that have to be reordered to make the communication possible:

$$C_1 = \mathit{log}!.\mathit{req}!.0$$
$$C_2 = \mathit{query}?.\mathit{id}?.0$$

An algorithm that might work the reordering out would accept a mapping defined by the two vectors $\langle c_1 : \mathit{log}!, c_2 : \mathit{id}?\rangle$, and $\langle c_1 : \mathit{req}!, c_2 : \mathit{query}?\rangle$ (this is the case in algorithms presented in [12] for instance). Our approach can reorder messages using the following vectors $\langle c_1 : \mathit{log}!, c_2 : \varepsilon\rangle$ equivalent to $\langle c_1 : \mathit{log}!\rangle$, $\langle c_1 : \mathit{req}!, c_2 : \mathit{query}?\rangle$, and $\langle c_1 : \varepsilon, c_2 : \mathit{id}?\rangle$ equivalent to $\langle c_2 : \mathit{id}?\rangle$, in which we specify that the interaction on *log* is desynchronised, temporarily memorised until its use for effective interaction on *id*. Both approaches (reordering messages at the level of the algorithm or at the level of the adaptation mapping) lead to the same adaptor that we show in Figure 8.

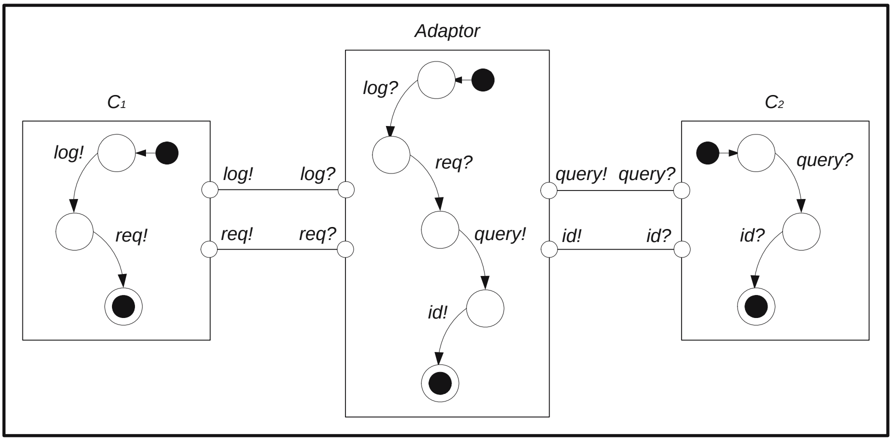


Fig. 8. An Adaptation Example with Reordering

Building the Adaptor for the E-Book System. Finally, vectors appearing in the LTS of Figure 7 are decomposed in basic interactions. Thus, Figure 9 presents the protocol of the adaptor which is a solution to the different mismatching situations originally present in the system, and also takes user profiles into account.

The top part of Figure 9 corresponds to the login stage and to the *Guest* execution of the *E-Book* system. In the middle, the left-hand side describes the logout information, whereas the right-hand side presents the switch from *Guest* to *Subscriber* in both directions. The bottom part of the figure presents the *Offer* behaviour on the left, and the *Subscriber* one of the *E-Book* system on the right.

To make interactions possible between components and the adaptor, the direction of messages is reversed. As an example, from the v_{g_1} vector, the single reception *c:user?* is derived. Moreover, interleavings may be introduced to consider all the possible orderings of messages in case a vector in Figure 7 was made up of more than two messages. This is the case for instance, for the 3-party communication expressed in v_{g_2}, resulting in Figure 9 in the reception *c:passwd?* followed by an interleaving of *s:login!* and *p:on!*.

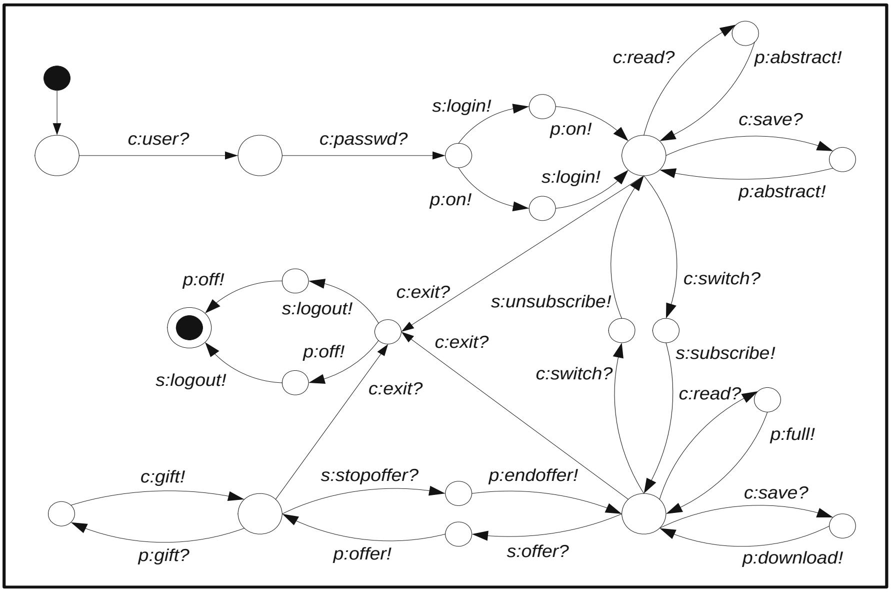


Fig. 9. Adaptor for the *E-Book* System

5 Related Work

Context-aware computing [22] is a mobile computing paradigm in which applications can discover and take advantage of contextual information (such as user location, time, nearby people and devices, etc.). Many researchers have studied this topic and built context-aware applications to demonstrate the usefulness of this technology [5,6]. Although there have been relevant achievements in the architectural support of context-aware applications [13,21], this paradigm does not explicitly deal with the adaptation of software entities within the system.

On one hand, the approach of Software Adaptation is reusing components and tackling the interoperability issues which exist at the different levels of component interaction. On the other hand, context-aware computing is concerned about the design and implementation of applications which are able to modify their functionality depending on context information which has been specified during the design of the system.

As regards coordination, adaptors generated by our approach are abstract descriptions of coordinator or orchestrator modelling complex interaction scenarios between the components involved in the system at hand. In [25], Wermelinger and Fiadeiro define component connections through morphisms between the actions involved in components. Nevertheless, these morphisms only tackle mismatch at the signature level, whereas the tuples handled in our mapping notation can work behavioural mismatch out. Braione and Picco [8] have proposed a calculus to specify contextual reactive systems separating the description of behaviours and the definition of contexts in which some actions are enabled or inhibited. Our goal here is slightly different, since we are taking contextual information into account while integrating components with mismatching interfaces.

Schmidt and Reussner present in [23] an adaptation approach as a solution to particular synchronisation problems between concurrent components, when for instance one component interacts with two other components. This approach is based on algorithms close to basic synchronous products. Inverardi and Tivoli [15] tackle the automatic synthesis of connectors in COM/DCOM environments, by guaranteeing deadlock-free interactions among components. They may also define properties that the resulting system should verify using liveness and safety properties expressed as specific processes. Compared to these proposals, we may match different name messages using our correspondences. In addition, our approach does not only restrict the adaptor to possible non-deadlocking behaviours [15] but may also address behavioural adaptation. That comes from the rich notation we have proposed that allows to deal with possibly complex adaptation scenarios, whereas these approaches do not use any mapping language for adaptor specification.

In [12], the authors propose a solution to behavioural adaptations using regular expressions of vectors as a mapping notation. Their work is supported by algorithms based on synchronous products and Petri nets encodings. Our mapping notation is as expressive as regular expressions of vectors since we use synchronisation vectors but also transition systems that can express the sequence, choice and iteration operators of regular expressions. Moreover, we propose a divide-and-conquer approach to tackle the complexity of the mapping, which makes its writing easier. This purpose is also achieved thanks to the automaton-based notation we have advocated.

Finally, in a previous work [9], we proposed a first solution to contextual adaptation that we have extended here in several directions: (i) an approach to deal with more than two components, (ii) a mapping notation based on synchronisation vectors and transition systems which is more expressive than the original one, and allows the separate description of the mapping on its constituent components, and (iii) an algorithm to construct adaptors between an arbitrary number of components.

6 Concluding Remarks

In this work we have presented an expressive and graphically-based notation to specify flexible adaptation policies between the interfaces of two or more components to be integrated. The separate specification of the mapping for each of the different interacting components simplifies the adaptation process. Then, we have proposed an algorithm which automatically derives the resulting adaptor from a mapping and a description of component behavioural interfaces. Our proposal has been illustrated using an *E-Book* system.

We plan to extend our context-based adaptation approach to open systems. Currently, our approach generates a global adaptor for all the components involved in the system, therefore all the messages go through the adaptor, and the whole system is closed from an external point of view. Our goal is to modify our approach to leave some ports opened to the environment. Through these, new components which can be added and removed dynamically, will be able to interact with the current system. This is particularly convenient when some systems cannot be shut down, such as banking or airport traffic control systems, but need to be dynamically reconfigured. In such a situation, we should enhance our approach to take into account the addition of new contexts or the removal of outdated ones, and also to adjust the current adaptor.

Another perspective is to connect our approach for software adaptation with programming platforms such as the BPEL orchestration language [3] or the .NET 3.0 framework. As an example, the recent .NET 3.0 platform relies on the implementation of components or business processes based on workflows (Windows Workflow Foundation [10]) which can be used as behavioural interfaces. Our proposal could be applied to this framework in order to support the design and development of .NET 3.0 software entities, such as the *E-Book* Web service we presented in this paper, and to generate code from obtained adaptor protocols in case of mismatch to be solved.

Acknowledgements. The authors would like to thank Michael Hicks and the anonymous reviewers for their comments that help us to prepare the final version of this paper. This work has been partially supported by the project TIN2004-07943-C04-01 funded by the Spanish Ministry of Education and Science (MEC), and project P06-TIC-02250 funded by the Andalusian local Government.

References

1. Alfaro, L., Henzinger, T.A.: Interface Automata. In: Proc. of ESEC/FSE'01, pp. 109–120. ACM Press, New York (2001)
2. Allen, A., Garlan, D.: A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology 6(3), 213–249 (1997)
3. Andrews, T., et al.: Business Process Execution Language for Web Services (WS-BPEL). BEA Systems, IBM, Microsoft, SAP AG, and Siebel Systems (February 2005)
4. Arbab, F., de Boer, F.S., Bonsangue, M.M., Scholten, J.V.G.: A Channel-based Coordination Model for Components. In: Proc. of FOCLASA'02, vol. 68(3) of ENTCS (2002)
5. Asthana, A., Cravatts, M., Krzyzanowski, P.: An Indoor Wireless System for Personalized Shopping Assistance. In: IEEE Workshop on Mobile Computing Systems and Applications (1994)
6. Bennett, F., Richardson, T., Harter, A.: Teleporting – Making Applications Mobile. In: Workshop on Mobile Computing Systems and Applications (1994)

7. Bracciali, A., Brogi, A., Canal, C.: A Formal Approach to Component Adaptation. The Journal of Systems and Software 74(1), 45–54 (2005)
8. Braione, P., Picco, G.P.: On Calculi for Context-Aware Coordination. In: De Nicola, R., Ferrari, G.L., Meredith, G. (eds.) COORDINATION 2004. LNCS, vol. 2949, pp. 38–54. Springer, Heidelberg (2004)
9. Brogi, A., Cámara, J., Canal, C., Cubo, J., Pimentel, E.: Dynamic Contextual Adaptation. In: Proc. of FOCLASA'06, ENTCS, Elsevier, Amsterdam (2006)
10. Bukovics, B.: Pro WF: Windows Workflow in .NET 3.0. Apress (2007)
11. Canal, C., Murillo, J.M., Poizat, P.: Software Adaptation. L'Objet. Special Issue on the 1st International Workshop on Coordination and Adaptation of Software Entities (WCAT'04) 12(1), 9–31 (2006)
12. Canal, C., Poizat, P., Salaün, G.: Synchronizing Behavioural Mismatch in Software Composition. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 63–77. Springer, Heidelberg (2006)
13. Chen, H., Finin, T., Joshi, A.: An Intelligent Broker for Context-Aware Systems. In: Proc. of UbiComp'03 (2003)
14. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Englewood Cliffs (1984)
15. Inverardi, P., Tivoli, M.: Deadlock-free Software Architectures for COM/DCOM Applications. The Journal of Systems and Software 65(3), 173–183 (2003)
16. ISO. LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. Technical Report 8807, International Standards Organisation (1989)
17. Magee, J., Kramer, J., Giannakopoulou, D.: Behaviour Analysis of Software Architectures, pp. 35–49. Kluwer Academic Publishers, Dordrecht (1999)
18. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
19. OMG. CORBA Component Model Specification, version 4.0. Object Management Group (2006)
20. Parrow, J.: An Introduction to the π-Calculus. In: Handbook of Process Algebra, Chapter 8, pp. 479–543, Elsevier (2001)
21. Salber, D., Dey, A.K., Abowd, G.D.: The Context Toolkit: Aiding the Development of Context-Enabled Applications. In: Proc. of CHI'99, pp. 434–441. ACM Press, New York (1999)
22. Schilit, B., Adams, N., Want, R.: Context-aware Computing Applications. In: Proceedings of IEEE Workshop on Mobile Computing Systems and Applications, pp. 85–90. IEEE Computer Society Press, Los Alamitos (1994)
23. Schmidt, H.W., Reussner, R.H.: Generating Adapters For Concurrent Component Protocol Synchronisation. In: Proc. of FMOODS'02, pp. 213–229. Kluwer Academic Publishers, Dordrecht (2002)
24. Szyperski, C.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Adisson-Wesley, London, UK (2003)
25. Wermelinger, M., Fiadeiro, J.: Connectors for Mobile Programs. IEEE Transactions on Software Engineering 24(5), 331–341 (1998)
26. Yellin, D.M., Strom, R.E.: Protocol Specifications and Components Adaptors. ACM Transactions on Programming Languages and Systems 19(2), 292–333 (1997)

Author Index