Lecture Notes in Computer Science 2786
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Flavio Oquendo (Ed.)

Software Process Technology

9th European Workshop, EWSPT 2003
Helsinki, Finland, September 1-2, 2003
Proceedings

Springer

Volume Editor

Flavio Oquendo
Université de Savoie
Ecole Supérieure d'Ingénieurs d'Annecy - LISTIC
Formal Software Architecture and Process Research Group
B.P. 806, 74016 Annecy Cedex, France
E-mail: Flavio.Oquendo@univ-savoie.fr

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): D.2, K.6, K.4.2

ISSN 0302-9743
ISBN 3-540-40764-2 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP Berlin GmbH
Printed on acid-free paper SPIN: 10931783 06/3142 5 4 3 2 1 0

Preface

During the last two decades we have seen tremendous developments within software engineering research and practice. The ever increasing complexity of software systems and the revolutionary development in the Internet have led to many interesting challenges and opportunities for new developments in Software Process Technology.

Started in 1990, the primary goal of the European Workshops on Software Process Technology is to achieve better understanding of the state-of-the-art on all issues related to software process technology, including (but not limited to): process modeling languages, computer-supported process description, analysis, reuse, refinement and enactment, process monitoring, measurement, management, improvement and evolution, process enactment engines, tools, and environments. Besides the technical viewpoint, the workshops have also taken into account human and social dimensions in software process enactment.

The 9th European Workshop on Software Process Technology (EWSPT-9) provided an international forum for researchers and practitioners from academia and industry to discuss a wide range of topics in the area of software process technology, and to jointly formulate an agenda for future research in this field.

The Program Committee met in Portland, Oregon, USA during ICSE 2003, to select the papers for inclusion in the proceedings. Twelve papers (eleven research papers and one position paper) were selected out of 25 submissions from 12 countries (Australia, Austria, Brazil, China, Finland, France, Germany, Mexico, Norway, Spain, UK, USA). All submissions were reviewed by three members of the Program Committee. Papers were selected based on originality, quality, soundness and relevance to the workshop. In addition, the program included an invited talk by Volker Gruhn (University of Leipzig, Germany). Credit for the quality of the proceedings goes to all authors of the papers.

I would like to thank the members of the Program Committee (Jean-Claude Derniame, Jacky Estublier, Carlo Ghezzi, Carlo Montangero, Leon Osterweil, Dewayne Perry, and Brian Warboys) for providing timely and significant reviews, and for their substantial effort in making EWSPT-9 a successful workshop.

This year the workshop was held in conjunction with the joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering. I would like to acknowledge the General Chair, Jukka Paakki, and the members of the Organizing Committee for their assistance during the organization of EWSPT-9 as a co-located workshop.

I would also like to acknowledge the prompt and professional support from Springer-Verlag, who published these proceedings in printed and electronic volumes as part of the Lecture Notes in Computer Science series.

June 2003 Flavio Oquendo

Program Committee

Program Chair:

- Flavio Oquendo
  Université de Savoie – LISTIC, Annecy, France
  flavio.oquendo@univ-savoie.fr

Committee Members:

- Jean-Claude Derniame
  LORIA/INPL, Nancy, France *derniame@loria.fr*
- Jacky Estublier
  IMAG – LSR, Grenoble, France
  estublier@imag.fr
- Carlo Ghezzi
  Politecnico di Milano, Italy
  ghezzi@elet.polimi.it
- Carlo Montangero
  Universita di Pisa, Italy
  cmontangero@di.unipi.it
- Leon J. Osterweil
  University of Massachusetts, USA
  ljo@cs.umass.edu
- Dewayne E. Perry
  University of Texas at Austin, USA
  perry@ece.utexas.edu

- Brian Warboys
  University of Manchester, UK
  brian@cs.man.ac.uk

Steering Committee

- Vincenzo Ambriola
  University of Pisa, Italy
- Reidar Conradi
  University of Science and Technology, Trondheim, Norway
- Volker Gruhn
  University of Leipzig, Germany
- Mehdi Jazayeri
  Technical University of Vienna, Austria

Previous Workshops of the EWSPT Series

EWSPT-8: Witten, Germany, June 19-21, 2001
Vincenzo Ambriola (Ed.): Software Process Technology
Proceedings: Lecture Notes in Computer Science 2077 Springer
ISBN 3-540-42264-1

EWSPT-7: Kaprun, Austria, February 21-25, 2000
Reidar Conradi (Ed.): Software Process Technology
Proceedings: Lecture Notes in Computer Science 1780 Springer
ISBN 3-540-67140-4

EWSPT-6: Weybridge, UK, September 16-18, 1998
Volker Gruhn (Ed.): Software Process Technology
Proceedings: Lecture Notes in Computer Science 1487 Springer
ISBN 3-540-64956-5

EWSPT-5: Nancy, France, October 9-11, 1996
Carlo Montangero (Ed.): Software Process Technology
Proceedings: Lecture Notes in Computer Science 1149 Springer
ISBN 3-540-61771-X

EWSPT-4: Noordwijkerhout, The Netherlands, April 3-5, 1995
Wilhelm Schäfer (Ed.): Software Process Technology
Proceedings: Lecture Notes in Computer Science 913 Springer
ISBN 3-540-59205-9

EWSPT-3: Villard de Lans, France, February 7-9, 1994
Brian Warboys (Ed.): Software Process Technology
Proceedings: Lecture Notes in Computer Science 772 Springer
ISBN 3-540-57739-4

EWSPT-2: Trondheim, Norway, September 7-8, 1992
Jean-Claude Derniame (Ed.): Software Process Technology
Proceedings: Lecture Notes in Computer Science 635 Springer
ISBN 3-540-55928-0

EWPM-1: Milano, Italy, May 30-31, 1990
Vincenzo Ambriola, Reidar Conradi and Alfonso Fuggetta (Eds.): Process Modelling
Proceedings: AICA Press

Table of Contents

Invited Talk

Research and Experience Papers

Position Paper

Process Landscaping: From Software Process Modelling to Designing Mobile Business Processes

Volker Gruhn

University of Leipzig, Computer Science, Applied Telematics,
Klostergasse 3,
04103 Leipzig, Germany
gruhn@ebus.informatik.uni-leipzig.de
http://www.lpz-ebusiness.de

Abstract. The method of process landscaping was initially focused on modelling and analysis of distributed software processes. Its strengths are that interfaces between processes and process parts are considered as first class entities and that analysis is focused on distribution issues. Recent experiments with distributed processes in insurance companies and in the logistics sector show that process landscaping can beneficially be used to identify process parts which can be made mobile.

1 Introduction

Process Landscaping is a method which pays particular attention to the question how processes should be distributed over various sites, how data and documents should be moved around between these sites and which telecommunication infrastructure is most appropriate for a modeled distribution scenario [5]. In [7] process landscaping has been applied to a real world example of a component based software process. The systematic analysis of its communication behavior and of the telecommunication costs induced by particular interface definitions showed improvement potential beyond those which were intuitively known right from the beginning.

2 Mobile Business Processes

More and more business processes are "mobilized". That means, parts of these processes are prepared for mobile execution. This is true for software processes, service processes and most sales oriented business processes. Generally speaking, to make a process part mobile has a substantial impact onto the structure of processes, onto the supporting technology and onto the question which data has to be available where. Most attempts to make processes mobile focus on single activities. More coarse-grained "mobilization" sometimes helps to reach more efficient processes.

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 1–3, 2003.

In particular for mobile processes, an efficient distribution of process parts and a clever identification of processes parts (mostly single activities) which are made mobile has an enormous impact onto the overall process costs structure. Mobile telecommunication, despite its decreasing costs, is still a hindrance to mobile solutions. In the insurance industry, for example, it would be an enormous functional improvement, to provide sales personnel with central applications available anywhere and at any time. For the time being, this is hardly realized. Most sales systems depend on replicated data and on redundant applications parts. Systems of this type (sometimes called offline systems) are faced by typical problems of redundancy:

- Data is not available, when spontaneously needed (because it has not been downloaded).
- Data is subject to lost updates (simply because replicated data has been changed at two or more sites without proper synchronization mechanisms).
- Offline applications provide similar (!) functionality as central systems (which might be considered by customers who are faced by prices and tariffs which slightly differ from those announced by the sales staff).
- Offline systems have to be installed at many workplaces (being run with different operating systems etc.). This usually is the origin for cumbersome quality management processes, manifold failures and unsatisfying robustness.

3 Process Landscaping for Mobile Business Processes and Further Impact of Software Process Work

A process landscape consists of hierarchically structured process models and their interfaces to each other [5]. Both can be described at different levels of detail. All refinements belong to this process landscape, but they can also be modelled as separate process landscapes. The main steps for the refinement of a given process landscape are:

- the identification of activities and interfaces between them (refinement of activities),
- the identification of interfaces between processes and process parts.

These steps can be repeated as long as detailed information has to be added to a process landscape. The level of detail needed depends on the purpose a process landscape is used for.

The Process Landscaping method (initially meant to support modelling and analysis of distributed software processes) turned out to be useful for identifying business process parts which easily can be made mobile. The focus on interfaces helps in identifying telecommunication costs.

Thus, once again, software process technology has had an impact onto non-software process technology. This phenomenon had been observed in at least the following contexts:

- The idea to model software process formally was initially related to software processes (compare to recent European and International Workshops on Software Processes [1,2]). Later on, vendors of business process modelling tools tried to provide a formal basis for their modelling languages.
- The idea of interacting software processes (being allocated to several software development teams) was raised in the software process community [3,4]) and has had an impact on the integration work carried out in the workflow management coalition.
- The idea to describe how software processes are spread over various sites was a key issue in a European Workshop on Process Technology [6] and arrived some years later in the arena of commercial process modelling tools.

References

1. Ambriola, V. (ed.), Software Process Technology , EWSPT-8: Witten, Germany, June 19–21, 2001, LNCS 2077, Springer, Berlin 2001
2. Boehm, B. (ed.): Proceedings of the 10th International Software Process Workshop, Ventron (France), June 17–19, 1996, IEEE Computer Society Press
3. Estublier, J., Dami, S., Process Engine Interoperability: An Experiment, in C. Montangero (ed.), Software Process Technology, 5th European Workshop, EWSPT '96, Nancy, France, October 1996, LNCS 1149, Springer, Berlin 1996, pp. 43–60
4. Groenewegen, L., Engels, G., Coordination by Behavioural Views and Communication Patterns, in W. Schäfer (ed.): Proc. of the Fourth European Workshop on Software Process Technology (EWSPT'95), Noordwijkerhout, The Netherlands, LNCS 913, Springer, Berlin 1995, pp. 189–192
5. Gruhn, V., Wellen, U., Structuring Complex Software Processes by „Process Landscaping“, in R. Conradi (ed.), 7th European Workshop on Software Process Technology, EWSPT 2000, Kaprun, Austria, February 2000, LNCS.1780, pp. 138–149
6. Schäfer, W. (ed.): Proc. of the Fourth European Workshop on Software Process Technology (EWSPT'95), Noordwijkerhout (The Netherlands), April 1995, LNCS 913, Springer Berlin
7. Wellen, U., Process Landscaping - Eine Methode zur Modellierung und Analyse verteilter Softwareprozesse (in German), PhD Thesis, University of Dortmund, June 2003

Empirical Validation of the Prospect Method for Systematic Software Process Elicitation

Ulrike Becker-Kornstaedt and Holger Neu

Fraunhofer Institute for Experimental Software Engineering (Fraunhofer IESE),
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
{becker, neu}@iese.fraunhofer.de

Abstract: This paper describes Prospect, a systematic approach to software process elicitation, and its validation in an industrial environment. Descriptive process models are important assets in software process improvement. A survey of the state of practice indicates that software process models are insufficient in quality, the models are developed in an uneconomic way, and the success of modeling activities depends largely on the Process Engineer's experience. We present the results of an empirical study to show that Prospect circumvents these problems.

1 Motivation

Descriptive software process models are important assets in software process improvement. They are used, for instance, for process enactment, as a basis for measurement, and for process redesign.

We conducted an internal survey at Fraunhofer IESE with the purpose of assessing the state of practice in process modeling in the context of process improvement programs. This survey revealed that process elicitation is often performed in an ad-hoc fashion, that is, in an unplanned way. Information sources are not selected according to information needs but randomly. Process Engineers do not have at their disposal techniques specifically aimed at process elicitation, and to systematically exploit information sources. The situation is aggravated, when process elicitation is done by inexperienced Process Engineers. This results in the following three main problems related to process elicitation:

- Quality of the models is insufficient because the information gathered is incorrect or incomplete.
- Effort is being wasted, for instance for detecting false information, finding and exploiting alternative information sources, and correcting errors late.
- The outcome of the process modeling activities is random in that it depends on the Process Engineer's personal experience.

Although these problems were reported in the context of IESE process modeling activities, they apply – at least in parts – to other contexts as well. To face these problems, we designed the Prospect method for systematic software process

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 4–15, 2003.

elicitation. Prospect integrates ideas from existing approaches to software process elicitation, and makes them operational.

This paper describes the Prospect method for systematic software process elicitation and its application and validation in an industrial setting.

This paper is structured as follows: Section 2 summarizes the Prospect method for systematic software process elicitation. Section 3 describes the validation of Prospect, the project context, the case study design, and the execution. Section 4 presents the empirical results and the lessons learned in the case study. Section 5 concludes the paper.

2 Prospect

This section sketches the Prospect method and its key features. Prospect stands for *Pro*cess-outline based *s*oftware *p*rocess *elicit*ation. At the outset Prospect requires the Process Engineer to define the process modeling goal. The process modeling goal states the requirements for the process model in an operational way. It follows the five-dimensional structure of a GQM goal [1] and describes what parts of the process are to be modeled and in what granularity, what aspects of the process are to be captured in the model, the intended usage, the intended users, and the environment (e.g., department) in which the model is to be valid. The purpose of the process modeling goal is to keep elicitation focused. This allows to avoid unnecessary effort for eliciting information that is not relevant to the process modeling goal.

In Prospect, the process model is developed incrementally in a number of iterations. Figure 1 describes an iteration step of Prospect. Each iteration i takes place in an iteration context isc_i. The iteration context consists of the process modeling goal as well as the partial process model PM_i and the information sources available at the outset of iteration step i. In each iteration, the first task is to determine the information deficit based upon the process modeling goal and the partial process model PM_i. The information deficit describes what information is missing in the current partial process model in order to reach the process modeling – it is the 'difference' between the current process model and the process modeling goal. Both the information deficit and the current process model are expressed with the help of a process modeling schema, where the current process model is a partially instantiated process model, and the information deficit is a set of schema elements. Prospect uses the Spearmint process modeling schema implemented in the tool of the same name [3].

The next task is to select an information source and a technique to exploit this information source. An information source can be a person with his role and department or a particular process document. Prospect techniques are defined according to a template. This template includes the preconditions for the application of a technique, the information source types (e.g., roles or document types) the technique can be applied to, and the information a technique can elicit in terms of process model elements. Based on three, Prospect's decision model selects applicable techniques. Once a technique and source are determined, the knowledge is elicited and integrated into the process model to form the next iteration context.

When designing the Prospect method we considered it essential to allow experience exchange at the level of elicitation techniques. Prospect allows to record in a common technique data base experience with the employed technique, including the context of the iteration step. The technique template reserves an explicit slot for the experience gained with a technique. For example, a Process Engineer may annotate that a technique did not yield the expected results under the given circumstances as described in the iteration step context.

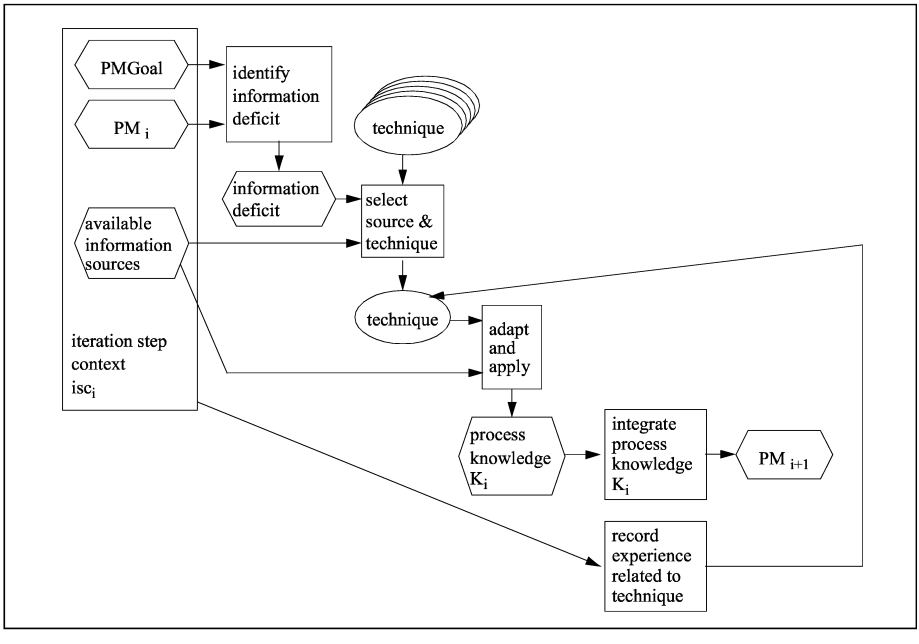


Fig. 1. Prospect iteration step

Many methods for software process elicitation, such as Elicit [5] or Process Discovery [4], suggest to obtain an initial understanding of the process prior to going into the details of the process. This understanding helps assess information gathered, in particular to fit more detailed process information into the big picture. In Prospect, this initial understanding is actually represented as a partial process model. It contains the major activities and artifacts, along with the product flow or control flow among them. To obtain such an initial understanding Prospect features two distinct phases, called process orientation and detailed elicitation. Each of these phases consists of a number of iterations as described above. The phases differ in the granularity specified in the process modeling goal and are linked by the partial process model.

The process outline is 'filled' in the subsequent detailed elicitation phase. Having an overview of the process prior to eliciting details also allows the Process Engineer assessing information sources with respect to their reliability. It facilitates the selection of interview partners and helps assess the completeness of the model. It helps the Process Engineer determine what information sources will be needed to

provide the details of the process and gives orientation on who can provide information on details.

Prospect currently features an initial technique data base, which includes techniques that are adapted from qualitative research techniques and knowledge elicitation techniques, based on personal experience with process elicitation. Prospect provides concrete guidance in the selection of information sources and techniques.

To facilitate experience exchange at the level of techniques, the Prospect technique templates follows the structure of a *reuse object* in the comprehensive reuse approach [2]. A concrete example technique designed for Prospect is *Product Flow Chaining*, which is depicted in Figure 2.

Example Technique Product Flow Chaining

Description: This technique can be used both with individuals and with a group. Conduct individual interviews or a group discussion with representatives of the relevant roles. Starting from the first product D_0 produced the in the process ask for each product D_i what activities A_i this document is input to. Ask for the roles performing A_i, the main output products D_{i+1} of activity A_i. If an activity has more than one output document, follow all threads. Iterate. For each entity captured, the Process Engineer should ask for a short description of the entity's purpose. As company-specific terminology may often be misleading, this information serves as further orientation for the Process Engineer and may avoid misunderstandings and misinterpretations. Especially when using this technique as a group technique it is important to visualize the process model to all participants of the discussion, for instance, on a white board.

Issues to cover:

- What is the first document to produce/what do you start with?

Iterate for all documents D

what activities is document D input to,

what are other input documents to this activity, description of these artifacts,

what are output documents of this activity, description of these artifacts

what roles are involved in performing A

until an activity or artifact beyond the scope of the process modeling goal is reached.

Sources: Project Manager, Quality Assurance (they get to assess intermediate and final documents, but do not necessarily know the steps in between),

Knowledge elicited: activities, artifacts, artifact flow, roles, role assignment. This technique yields the product flow graph at top level.

Dependencies: no knowledge required to start

Object quality: When used as a group interview, knowledge obtained through this technique usually has a higher confidence level, as it is confirmed by more than one person. If used as group interview technique, a white board (or similar) is recommended for visualization, product flow may be confounded with control flow.

Experience: none so far

Fig. 2. Prospect Example Technique Product Flow Chaining

3 Application and Validation of Prospect

A key requirement for the design of Prospect was its applicability in real-world settings (i.e., eliciting real-world processes from actual process documents used and Process Performers). To validate whether Prospect fulfills this requirement and circumvents the problems described above, we conducted a case study in a real-world project. Furthermore, we aimed for more detailed, qualitative experience with the method. This section describes the context, the design, and the execution of the case study.

A case study is more adequate to validating a comprehensive method, such as Prospect, than controlled experiments, as the number of influencing factors is very high. However, case studies have their limitations: a case study can demonstrate the effects of a technology or a method in a typical situation, but it cannot be generalized to every situation using that technology or method [6].

3.1 Context

Company X[1], where the case study took place, is involved in mechanical engineering and construction. In the context of the technology transfer project in cooperation with Fraunhofer IESE, a process model was needed that described the development of different releases for the control unit of a particular machine. The main functionality of the control unit of these machines is implemented in software, in other words, the process presents all difficulties related to software development for embedded systems. The overall process involves about 50 Developers. The foremost difficulty of the process is that it is iterative: a new release has to be developed every three months. These properties lead to a high complexity of the process.

According to the process modeling goal the process model had to cover the design, code, test, integration, and integration test activities. The process model had to describe all activities that had a distinct output; for the entities a short description was sufficient. The process model was built with the purpose of gaining an understanding of the process and the artifacts produced. The process model was to be used by Developers and the Software Engineering Process Group (SEPG). The Quality Assurance Group at Company X was to later integrate the process model into a new configuration management system to form a new reference process model.

In the context of the technology transfer project a high-level process model of the actual process was developed by two IESE Process Engineers. This model was expressed using the Spearmint [3] notation and was then handed over to the Quality Assurance at Company X. The subsequent versions of the model were reviewed by the IESE Process Engineers who gave feedback and recommendations on the model.

The IESE Process Engineers had a background in process modeling in general, but their previous experience with process elicitation was very low. One of them had been involved in one process elicitation project before, the second Process Engineer had no prior experience in process elicitation. Both had received a short training with

[1]Name withheld for confidentiality.

Prospect. Neither of the Process Engineers had been involved in the design of Prospect.

3.2 Evaluation Goal and Design of Study

In particular, the study was to show that Prospect alleviates the problems encountered in ad-hoc software process elicitation, such as insufficient quality of the resulting models, unnecessarily high effort for process elicitation, and a high degree of dissatisfaction among Process Engineer with process elicitation.

To actually see whether Prospect performed better than an ad-hoc approach to software process elicitation, data from the Prospect application in Company X was compared with data from an elicitation project that used an ad-hoc approach to software process elicitation (*sister project design* [7]). Like the validation project, the sister project deals with a software process in an embedded system. The granularity and the schema elements of the process models are approximately the same for both process models. The purpose and the intended users were comparable for both projects. The resulting process model in the sister project contained about 45 products and 58 activities. As a notation, natural language (a Word document), enhanced with graphics using IDEF0 [9] notation was selected. Modeling was performed by one Process Engineer who had no knowledge of Prospect but had been involved in five previous process modeling exercises.

The following hypotheses had been formulated for the validation:

- Hypothesis 1: Process Engineers using Prospect introduce fewer and less severe errors due to incorrect information than Process Engineers using an ad-hoc approach to process elicitation.
- Hypothesis 2: Process Engineers using Prospect for software process elicitation spend less rework effort due to incorrect or incomplete information than Process Engineers using an ad-hoc approach to process elicitation.
- Hypothesis 3: Prospect can compensate lack of Process Engineers' process elicitation experience.

To operationalize the validation goals, we had to derive measures. We faced the problem that for a case study taking place in the context of a technology transfer project, it is difficult to obtain data on the execution of the project itself, and in the required quality, without interfering with the project: project members cannot be expected to spend effort on data collection not related to the project itself. Thus, the measures had to be defined in a way that would allow data collection without interfering with the project. Another requirement on the used measures was that they did not refer to the size and complexity of the developed models. The factors that really influence size and complexity of a model are unknown. Since the two projects were concerned with distinct processes, any measure that would take size or complexity into account would not yield comparable results.

Hypothesis 1 deals with errors. Ideally, for each error one would gather data on rework effort, error type, and its severity, but this would require interfering with the project. Instead, we measured how many iterations were needed between presenting

the first version to the customer and the final model. The severity of errors was modeled by estimating the portion of the model affected by rework.

Hypothesis 2 deals with rework effort. To normalize rework effort and make it independent of size and complexity of the model, we captured rework effort in relationship to the overall effort such that rework effort is expressed as the ratio between the effort spent for rework and the overall effort. Rework effort was estimated by the Process Engineers.

Hypothesis 3 deals with the degree to which process elicitation was driven by personal experience. We used a subjective measure. The Process Engineers were asked directly whether they felt that using Prospect compensated a lack of experience in process elicitation.

After the high-level process model was completed and accepted by the customer, we conducted an informal interview with the Process Engineers to gather data on the case study. The interview followed an interview guide that had been designed based on the hypotheses. In particular, the Process Engineers were asked how they conducted their process modeling activities, about their qualitative experience with the method and the techniques employed, and on quantitative data in order to find out whether the hypotheses held.

3.3 Threats to Validity

When designing a study, possible threats to validity ought to be considered before hand. Wohlin et al. [10] give a list of possible threats to validity when doing experimentation in software engineering. The most relevant threats to be considered for the design of the case study are discussed in this section.

Threats to internal validity refer to threats that a relationship between a treatment and its outcome is detected where no causal relationship exists. A key validity threat in the validation of knowledge elicitation methods in general is that even if technique A has shown to perform better than Technique B in an experiment this does not necessarily mean that Technique A is generally better than B [8]. The successful application of a technique may not be rooted in the quality of the technique itself, but caused by other factors, such as the general skills of the Process Engineers or their experience with processes that are similar to the one to be elicited using Prospect, skills of the Process Performers (e.g., communication skills), or the overall complexity of the process to be elicited. To overcome threats related to this, during the case study not only data with respect to performance will be collected, but the Process Engineer doing process modeling in the case study will also be asked about their experience in the particular situation. In the case study the data collection relies on estimates (number of iterations, portion of the model affected by changes, etc.) instead of accurate measures. These measures do not yield detailed information. However, in the context of a case study conducted in the context of a technology transfer project, it is very difficult to collect reliable data at a detailed level of granularity. Thus, we decided to gather less detailed data, rather than an inaccurate estimation of more detailed measures given after the termination of the project. To avoid inaccuracies due to mono-method bias the overall data – especially the quantitative part intended to show the performance of Prospect – needs to be treated

with care. However, as performance is not the key issue to be evaluated, this is not a general threat to validity.

When interpreting the findings from the case study, the coarse level of detail has to be considered in the interpretation.

The way measures are constructed has a high impact on the validity. The number of errors in the process models was modeled using the number of iterations on the process model until the customer or sponsor approved the model. A threat here is that it is not clear what the quality criteria for passing the process model review were, as these were internal at the customers and could not be assessed. For instance, one customer may have had very strict criteria for the process model while the other only had weak criteria. Thus, the number of iterations in itself should not be seen as an absolute number, but ought to be seen rather as a tendency.

3.4 Process Modeling

This section describes how Prospect was applied in Company X. The description is based upon the oral accounts of the Process Engineers.

After obtaining an overview of the domain, the two Process Engineers scheduled a group interview with four Quality Managers and one Process Owner on site. The Process Engineers recurringly used Product Flow Chaining as a group interview technique. One Process Engineer led through the discussion while the other developed the model. The process model was visualized to all participants of the meeting using the Spearmint tool and a beamer. This technique was selected because it seemed most adequate to obtain a fast overview of the activities and artifacts involved, and the relationships among them. Furthermore, this technique did not require any background knowledge.

During the interview several discussions arose. These sometimes got very fast and were difficult to follow for the Process Engineers. These discussions were due to the fact that different participants had different views on the process. In particular, not all participants had the same picture of the inputs and outputs for the different activities. Thus, to the people from Company X these discussions increased the overall understanding of the process and helped reconcile different views. As a result from this stage, the Process Engineers obtained a consistent picture upon which all discussion participants could agree upon.

In some situations the discussion participants could not agree upon a common process. In such situations the views of the people who dominated the discussion was finally adopted by the discussion participants as the common process.

Sometimes the interviews drifted into details of the process, e.g., variations of activities, details of roles) before the overall process structure was clear (the technique employed suggests to only focus on the product flow structure). This often prolongated the overall discussion as a common understanding of the process had not yet been achieved. In these situations the Process Engineers tried to focus the discussion on the process structure again.

The meeting on site lasted one day. During this session five different views could be captured.

The resulting model contained activities and artifacts, each with a 'description' attribute, and product flow relationships. The model contained 25 activities and 36 artifacts and described the process on top level, that is, without refinements of activities. The overall for creation effort for this model was seven person days, which break up into one person day for preparation, two person days on site, and four person days for the modeling activities. The initial model was later handed over to a process group member of Company X, who refined the model and used it as a basis for an improved process. The later versions of the process model as refined by Company X were reviewed by the IESE Process Engineers. As compared to the first version of the process model, the refined versions contain the product flows of new sub processes. Discussions and reviews of the revised model later showed that the person responsible at Company X for process modeling had followed an approach similar to the one taken by the IESE Process Engineers: He had used a graphical representation of the process model as a basis for interviews with individuals in the company, and had walked sequentially through the product flow, asking for each product and activity how they could be streamlined in order to improve the overall process performance.

4 Empirical Results and Lessons Learned

For the evaluation, two types of data were collected: data to compare the performance of the case study with the sister project in order to confirm or refute the hypotheses, and concrete experience and lessons learned.

4.1 Comparison with Ad-Hoc Elicitation

One goal of the case study was to demonstrate that Prospect performed better than a reference project that used an ad-hoc approach to software process modeling with respect to the hypotheses stated.

For Hypothesis 1 we asked the Process Engineers for the numbers of elicitation/modeling iterations between when the first model version was presented to the customer and the final model. For each of these iteration, we asked the Process Engineers to estimate the portion of the model affected by rework, and the effort required for rework due to incorrect or incomplete information collected.

The Process Engineers in the validation project reported that they needed one additional iteration after the first one to obtain a version that was accepted by the customer. According to the Process Engineers, the portion of the model affected by the rework was very small, they estimated it to be about 5%. In contrast, five iterations were needed in the sister project, and rework was estimated to as high as 75%. Some of this the Process Engineer attributed to the lack of an overview of the process and the resulting misinterpretation of elicited information.

For Hypothesis 2 (portion of rework effort) we asked the Process Engineers to estimate the overall effort and rework effort. In the validation project, the rework effort was estimated to be half a person day out of four overall person days for modeling. The rework effort of the sister project was estimated to be 20 days for rework out of 29.

For Hypothesis 3, the Process Engineers who had been involved in the validation project confirmed that they found that Prospect had helped them overcome their low level of experience in process elicitation.

Although the data collected is rather coarse, the case study is a strong indication that Prospect performs better than an ad-hoc approach to software process elicitation.

To corroborate the results with respect to the performance of Prospect, we collected qualitative data regarding the personal experience of the Process Engineers with the method.

4.2 Experience with Method

A second aspect of the validation was to gather concrete experience with the method. In summary it can be said that the Prospect method helped the Process Engineers a great deal in their process modeling tasks. The Process Engineers considered the process modeling goal a good way to clearly set the focus of the process modeling activities. In particular, the explicit statement of the process modeling goal helped them to focus on those aspects relevant for Company X. After the model was refined by the Quality Assurance group the process modeling goal helped the IESE Process Engineers assess later versions of the model and made it easier to give concrete feedback and recommendations on the model to its maintainers.

The Process Engineers considered the product flow chaining a good way to focus on those aspects relevant in the early stages, and that the techniques helped to focus the discussion on the topic.

The Process Engineers found that the group discussion techniques suggested by Prospect are a good way to reach a common understanding of the process early on. This makes further elicitation tasks efficient, because the data obtained from this discussion is reliable as it is reconciles several people's view on the process. The Process Engineers pointed out that the visualization was particularly efficient, as it allowed to capture five views on the process in one consistent process model within one day. This would probably not have been possible by having five individual interviews. The group discussion increased the overall process understanding of the different participants, they got a better feeling of how 'their' process fragment fits into the overall picture of the process, and got a good understanding of the general dependencies within the process. Process Performers found the group discussions enlightening, as they had the side-effect to improve their understanding of the overall dependencies of their process fragment within the process. They learned how much their output affected the subsequent activities. This led to a net improvement for the organization.

Group discussions, on the other hand, may get very fast and difficult to follow for people who are not familiar with the company-specific terminology and processes. An inherent risk of group discussions is that individual participants may dominate the discussion. On the other hand, dominant participants may push results in otherwise lingering discussions.

The Process Engineers found that a number of six participants was a good group size: there were enough people to stimulate the discussion and to get several views of the process. On the other hand, a larger group may have resulted in Process

Performers loosing the overview. Furthermore, in a larger group it is difficult for participants to contribute to the discussion.

A risk of the Prospect method and this technique in particular is going into detail too fast, that is, discussing process details without having enough understanding of the overall process. This happened in the validation project and did not lead to results but only lengthened the discussion. To make this technique more efficient, details of the model should not be asked for before all inputs and outputs are clear. A lesson learned is that it is important to stress why an overview of the process structure – the process outline – is so important.

Although the Product Flow Chaining technique warns that there may be a risk in interviewees confounding product flow with control flow this was not the case in this project.

The Process Engineers emphasized that the model quality depends highly on the people interviewed. Having the 'right' interviewees is therefore crucial, and interviewee sampling has to be done very carefully.

The discussion of the model structure and modeling/visualization was a very good way to coach the Quality Assurance so that they could take over modeling tasks. Later on, the person responsible for process modeling at Company X reported that the method used had worked quite well for him.

5 Summary and Conclusion

This paper presented the Prospect method for systematic process elicitation and described an application of the method in an industrial technology transfer project. Data regarding the performance of the method was compared with process elicitation in a comparable project that used an ad-hoc approach to software process elicitation. The project where Prospect was employed performed better in the way that it used fewer iterations due to rework than the reference project. In the project where Prospect was employed the portion of the process model affected by rework was smaller than in the project that followed an ad-hoc approach. However, even more interesting, in particular for future application of Prospect, is the detailed qualitative experience the Process Engineers gained with respect to the method and the technique mainly employed.

The process modeling activities, and in particular the group discussion where the process model was visualized, increased the process understanding of the people involved in the process modeling activities. In particular, several different views on the process could be reconciled through a process model that had been developed in a group discussion within a rather short time frame.

In summary, it can be said that Prospect facilitates process elicitation and makes it more systematic, as demonstrated in the case study. However, case studies have their limitations as they cannot be generalized to every situation using that technology or method. Thus, more case studies would be helpful for the validation of Prospect. In particular, more experience with the overall method and with concrete techniques would be helpful.

Acknowledgments. The authors would like to thank the people involved in the case study – the Process Engineers as well as the people in Company X. We would like to thank Leif Kornstaedt and Jürgen Münch for their valuable comments on an ealier version of this paper.

References

[1] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal Question Metric Paradigm. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 528–532. John Wiley & Sons, 1994.

[2] Victor R. Basili and H. Dieter Rombach. Support for comprehensive reuse. *IEEE Software Engineering Journal*, 6(5):303–316, September 1991.

[3] Ulrike Becker-Kornstaedt, Dirk Hamann, Ralf Kempkens, Peter Rösch, Martin Verlage, Richard Webby, and Jörg Zettel. Support for the Process Engineer: The Spearmint Approach to Software Process Definition and Process Guidance. In Matthias Jarke and Andreas Oberweis, editors, *Proceedings of the Eleventh Conference on Advanced Information Systems Engineering (CAISE '99)*, pages 119–133, Heidelberg, Germany, June 1999. Lecture Notes in Computer Science 1626, Springer.

[4] Jonathan E. Cook and Alexander L. Wolf. Discovering Models of Software Processes from Event-Based Data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[5] Dirk Höltje, Nazim H. Madhavji, Tilman Bruckhaus, and Won Kook Hong. Eliciting Formal Models of Software Engineering Processes. In *Proceedings of the CAS Conference*. IBM Canada Ltd. and The National Research Council of Canada, October 1994.

[6] Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. Case studies for method and tool evaluation. *IEEE Software*, 12(4):52–62, July 1995.

[7] Barbara Ann Kitchenham. Evaluating software engineering methods and tools, part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes*, 21(1):11–15, January 1996.

[8] Nigel Shadbolt, Kieron O'Hara, and Louise Crow. The experimental evaluation of knowledge acquisisiton techniques and methods: history, problems and new directions. *International Journal on Human-Computer Studies*, 51:729–755, 1999.

[9] Softech Inc. Integrated Computer-Aided Manufacturing (ICAM) Architecture Part II. Volume IV - Function Modeling Manual (IDEF0). Technical Report AFWAL-TR-81-4023 Volume IV, AF Wright Aeronautical Laboratories (AFSC), Wright-Paterson AFB, Dayton, Ohio, USA, June 1981.

[10] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslen. *Experimentation in Software Engineering. An Introduction*, volume 6 of *The Kluwer International Series in Software Engineering*. Kluwer Academic Publishers, Boston, 2000.

Formalizing Rework in Software Processes

Aaron G. Cass, Stanley M. Sutton Jr., and Leon J. Osterweil

Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
+1 413 545 2013
{acass, ljo, suttonsm}@cs.umass.edu

Abstract. This paper indicates how effective software-process programming languages can lead to improved understandings of critical software processes, as well as improved process performance. In this paper we study the commonly mentioned, but seldom defined, rework process. We note that rework is generally understood to be a major software development activity, but note that it is poorly defined and understood. In this paper we use the vehicle of software-process programming to elucidate the nature of this type of process. In doing so we demonstrate that an effective language (i.e. one incorporating appropriate semantic features) can help explain the nature of rework, and also raise hopes that this type of process can be expedited through execution of the defined process. The paper demonstrates the key role played in effective rework definition by such semantic features as procedure invocation, scoping, exception management, and resource management, which are commonly found in programming languages. A more ambitious program of research into the most useful process-programming language semantic features is then suggested. The goal of this work is improved languages, for improved understandings of software processes, and improved support for software development.

1 Introduction

Rework is an ongoing problem in software development. Rework is generally considered to be potentially avoidable work that is triggered to correct problems or (sometimes) to tune an application [5,9]. The cost of rework can approach or exceed 50% of total project cost [19,9,5]. Rework cost rises dramatically the longer the delay, relative to the life cycle, between the occurrence of a problem and its remediation. For example, for a problem that occurs in requirements analysis, the cost of repair may rise by one to two orders of magnitude depending on how late in the life cycle it is caught [5].

Research on rework has focused on minimizing the amount of rework that a project may incur [19,9,5,26]. This is typically done through the introduction of earlier, more frequent, and more formal reviews, inspections, and tests; these aim to detect and enable the correction of problems as early in the life cycle as possible. A reduction in rework costs has also been shown for increases in the general level of process maturity [21,26]. Through such approaches the cost of rework has been reduced significantly, in some cases to less than 10% of total project cost [19,9].

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 16–31, 2003.

Despite successes in reducing rework, it is generally accepted that rework cannot be eliminated entirely. Moreover, not all rework-inducing problems can be detected as soon as they occur; some problems will only be caught some distance downstream. Thus, some amount of rework, including some expensive rework, is inevitable.

In this paper we propose that the problem of rework can be attacked from another angle, namely, that of software-process programming. Process programming, which is the specification of software processes using formal, usually executable, languages, can contribute to the solution of the rework problem in two ways: It can help with the prevention of rework, and it can help in facilitating rework that cannot be prevented. In this paper we address the latter opportunity.

Toward the facilitation of rework, process programming can contribute to the precise definition of rework processes, which in turn can contribute to the analysis and understanding of rework. Execution support can enhance the fidelity and effectiveness of rework processes, thereby improving their monitoring and reducing their costs. In combination, these measures further help to ensure the formal and actual correctness of rework processes, and they may indirectly lead to reductions in the amount of rework by reducing the need for rework.

The specification or programming of rework processes poses special challenges, however, that arise from an essential characteristic of rework, namely, that it entails repetition with variation. Typically in rework an activity must be repeated, a work product must be revised, or a goal must be reestablished. Thus, what was done or achieved before must be done or achieved again, but in a new context, one that reflects prior work, identified problems, dependent artifacts and activities, new resource constraints, and other ongoing efforts.

To take one example, the same basic activity to define a requirement may be invoked during the initial specification of requirements as well as when a problem with the requirements is later discovered during design. However, in the rework situation, developers with different qualifications may be required (say, expert versus intermediate), access to requirements documents may need to be coordinated with access by design engineers, dependent design elements may need to be identified, a specific sort of regression test (or review) of the revised requirements may be needed, and problems with the rework may require handling in ways specific to that context.

If rework activities are variations of initial activities, then it should be beneficial to specify those activities in such a way that the nature of the variations is identifiable, for example, as alternation, parameterization, or elaboration. This suggests that rework activities are essentially reinstantiations, perhaps with context driven parameterized modifications. We believe that this view can lead to cleaner, clearer, more understandable development processes that, nevertheless, are complete and faithful to the complexities of real development. This clarification should, moreover, expedite automated support of the performance of real development. Thus, while a deeper understanding of the notion of rework is a key goal of this work, so too is the provision of more effective automated support to real software development processes.

A key obstacle in achieving these goals is the ability to understand how to specify the contexts in which rework might occur in software development. This seems to us to entail the ability to capture precisely all of the relevant elements of that context, including

both those that are retained from earlier development and those that are new. However, the ability to specify the relevant elements of rework processes, including their contexts and activities, in turn depends on having process languages with appropriate constructs and semantics.

This research makes two primary contributions toward these problems. First, we show that some kinds of rework can indeed be formalized by process programming, especially focusing on details of the rework context. Second, we identify features that process languages should incorporate in order to support real-world software processes including rework. Conversely, we recommend that those who are interested in specifying rework processes should look for process languages with these features. With such languages we should be able to give more complete and precise definitions of rework processes. With such definitions, we should be able to better understand the nature of rework and its relationship to other aspects of the software life cycle. Furthermore, by applying the definitions to the execution of processes, we hope to further reduce rework costs and to maximize rework efficiency in practice.

The remainder of this paper is organized as follows. Section 2 describes our approach to formalizing rework processes using process programming. It gives an example of a software process that includes rework. This process is specified using the process-programming language Little-JIL [36,37]. This section also presents an analysis of the language features used to capture various aspects of rework. Section 3 discusses related work on defining rework processes. Section 4 then presents our recommendations regarding language features for capturing rework processes and discusses applications of process programming to defining, understanding, and executing rework. Finally, Section 5 presents our conclusions and indicates directions for future work.

2 Approach

Our approach to the formalization of rework is to represent rework using process programming languages in which the activities to be performed and especially the context of those activities can be precisely defined. This should allow us to be clear about how rework is to be performed, for example, when a flaw in the requirements specification is only discovered during design or coding and must be repaired at that later stage. It should also allow us to be clear about what happens when a design fails to pass review and must be revised still within the design phase but perhaps using a technique different from the one that gave rise to the problem.

2.1 Basis in Programming

We see significant analogies between common rework situations and typical programming language constructs. Rework activity definitions are analogous to procedures or methods. The context of a rework activity is analogous to the scope in which a procedure or method is invoked. The particular features associated with these programming-language notions suggest benefits for the definition of rework processes.

For example, a procedure allows a flow of actions to be defined. Additionally, a procedure can be explicitly parameterized and may rely on additional information or

services defined externally in the scope from which it is invoked. Mechanisms such as these enable the definition of activities that are adaptable to their contexts, in particular to factors that characterize and differentiate initial development from rework.

Similarly, a scope in a programming language may contain data and control declarations, initializations and bindings, assertions, exception handlers, and so on. Additionally, the construct that defines the scope will typically invoke some activities as subroutines, controlling whether, when, and with what parameters a particular invocation occurs, and also determining how the results, whether normal or abnormal, are handled. These sorts of mechanisms enable the definition of contexts appropriate to both initial development and rework.

The analogies to general-purpose programming-languages concepts are thus suggestive of many benefits for the specification of rework processes. However, as with general-purpose programming languages, the particular design of a process-programming language affects the kinds of activities and contexts that can be readily described and automated. To illustrate the formalization of rework through process programming, we will use a particular process-programming language, Little-JIL [36,37], applied to an example of a phased software-development process that includes explicit rework.

The process we describe addresses requirements specification and design at a high level. This process description is intended as an example of how rework can be incorporated accurately into a software process. It is NOT intended to represent an ideal software process or even to imply that this is the only way that rework could be handled in this process. The approach we present can be applied to processes both more and less complex or restrictive than the example we give – an effective process language should enable a corresponding range of process characteristics to be addressed.

2.2 Example

Consider the initial portion of a phased software development process. After requirements specification activities are completed, the developers proceed to design activities. During the requirements specification activity, as requirements elements are developed, they are reviewed, both independently and for inter-requirement consistency.

As design proceeds, design reviews are also conducted, including ones that check the conformance of the design with requirements. As a result, it might be discovered that there are design elements based on design decisions that have no basis in the stated requirements. It might be further determined that the requirements specification needs some additional requirements to address the concerns reflected in these particular design decisions. At this point, the developers will engage in a rework activity. While still in the design phase, some requirements specification activities must be performed.

We believe that many aspects of this scenario will be easily described using a process-programming language that incorporates constructs (or variations thereof) from general-purpose programming languages, such as scoping, procedure invocation, exception handling, and so on. To investigate this hypothesis, we have used our process-programming language, Little-JIL [36,37], to address the above scenario.

Figure 1 shows a Little-JIL program for a part of the phased software development process described above. A Little-JIL program is a hierarchy of steps, each of which has an interface and defines a scope in which a variety of additional elements are structured.

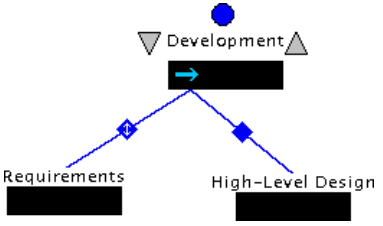


Fig. 1. A phased software development process

Little-JIL is a visual language in which every step is represented by a named black bar that is decorated by a variety of graphical badges. The step interface is represented by annotations on a filled circle above the name of the step. The interface represents the view of the step as seen from the caller of the step, including parameters, resources, messages, and exceptions that may cross the interface. Little-JIL passes parameters as value, result, or value-result. In Little-JIL a step is defined as either a root step or a substep. However, a step defined in one part of a program can be referenced from other parts of the program. Such a reference represents a reinstantiation and invocation of the step in a new scope. Both the ability to construct process contexts and the ability to invoke steps in multiple contexts are important for describing rework.

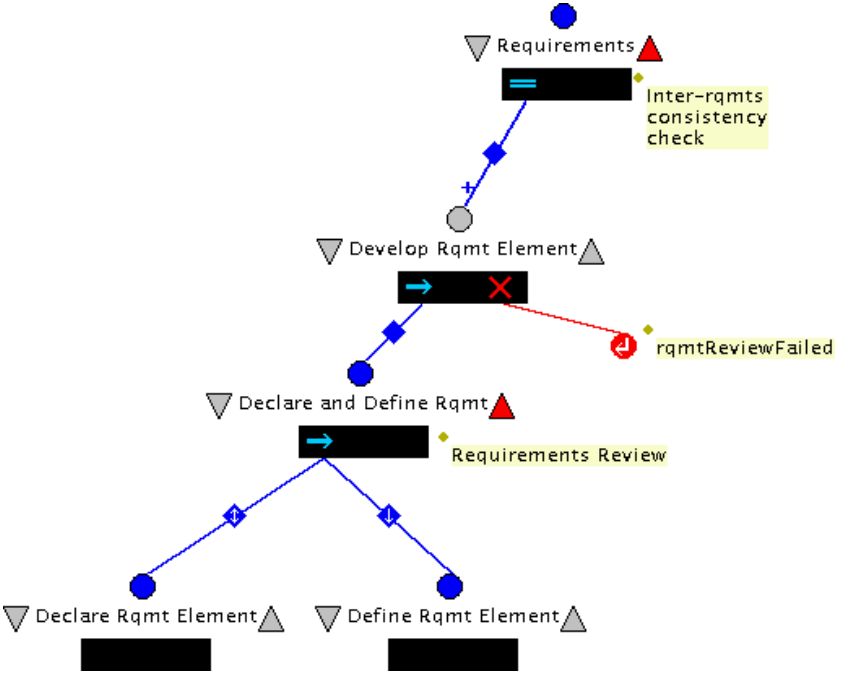


Fig. 2. Requirements activities

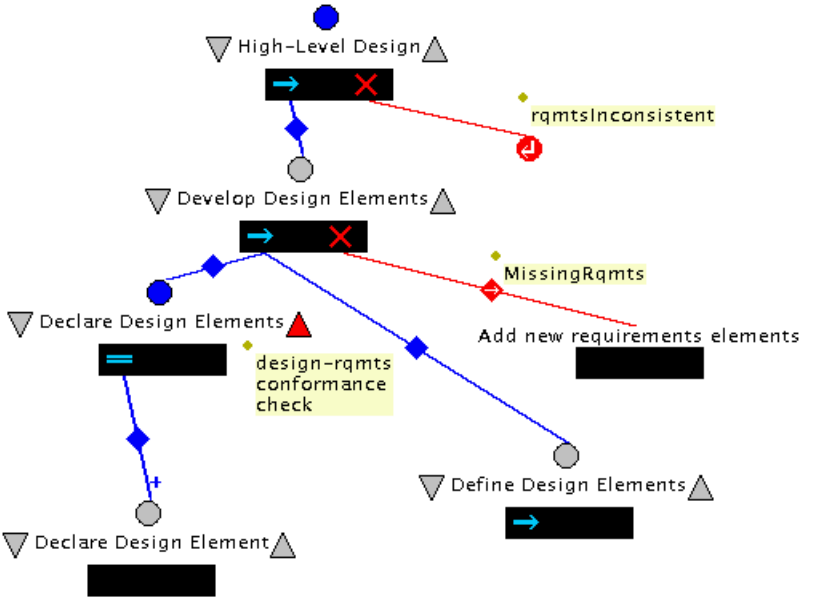


Fig. 3. High-Level Design activities

The right-arrow in the root step in Figure 1 indicates that the substeps are to be executed in sequence from left-to-right, starting with Requirements and continuing with High-Level Design. Figures 2 and 3 show elaborations of the Requirements and High-Level Design steps, respectively.[1]

The triangle to the right of Declare and Define Rqmt in Figure 2 indicates a *post-requisite*, a step that is executed when Declare and Define Rqmt completes. In this case, the post-requisite is a Requirements Review. If the post-requisite completes without any errors, then Declare and Define Rqmt completes successfully. However, if errors are found in the Requirements Review, a rqmtReviewFailed exception is thrown. In Little-JIL, exception handling is scoped by the step hierarchy. So, in this case, the rqmtReviewFailed exception will propagate to the Develop Rqmt Element step. The handler attached here (beneath the red "X" in the black bar) indicates that we should *restart* the Develop Rqmt Element step and recreate that requirement element.

Once requirements elements have been declared and defined, we proceed to High-Level Design. As can be seen in Figure 3, after all design elements have been declared (by Declare Design Elements), a design-rqmts conformance check post-requisite is executed. During this review, we could check that all design elements have associated

[1] In these and other Little-JIL figures, we show some information in comments (shaded boxes) because the editor we use for creating Little-JIL programs does not show all information pertinent to a step in one view. The use of comments here to reflect information that is more formally captured in other views does not reflect incompleteness in the overall process model.

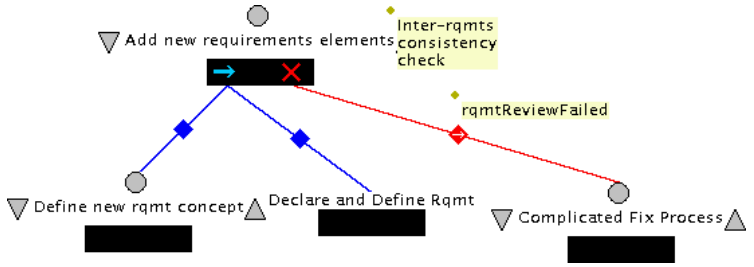


Fig. 4. Rework in the context of design

requirements elements. If we discover that there are design elements that lack adequate support in requirements, we throw a MissingRqmts exception. In this context, this exception is handled by the exception handler called Add new requirements elements, which is elaborated in Figure 4. Add new requirements elements first defines a new requirements concept for use as input to the Declare and Define Rqmt step, which was defined earlier in the Requirements phase in Figure 2.

In addition to the features illustrated in the figures above, Little-JIL also allows the specification of resource requirements [29]. Managers of processes are very concerned in general with resource allocation and utilization, so resource specification has first-class status in Little-JIL. Resource-related cost issues are also of concern specifically in relation to rework, and the accurate definition of processes has been considered a prerequisite to first assessing the true costs of rework and then minimizing them [19].

Figure 5 shows a step Declare and Define Rqmt with a resource specification which indicates that the agent for the step must be a Rqmt Engr. In this example, we specify only the resource type Rqmt Engr. However, the specification can be any legal query that our externally defined resource manager can execute. For example, we could specify that Declare and Define Rqmt needs a Rqmt Engr with attributes indicating that he or she knows UML use cases. At run-time, the resource specification is interpreted by a management component of the Little-JIL runtime environment (Juliette) as a query for a resource from the resource model. The resource manager will choose a resource from those available in the resource model that satisfy the specification (or throw an exception if the request cannot be satisfied).

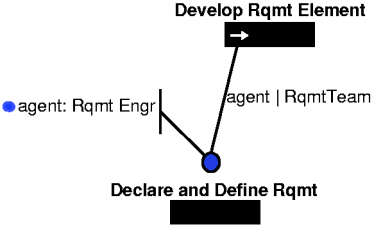


Fig. 5. Resource Specification

This example incorporates two occurrences of rework. The first is part of the requirements phase, when the failure of Declare and Define Rqmt triggers (via an exception) the reexecution of Develop Rqmt Element. In this case the rework is accomplished by repeating the same activity in the same context. Of course, the process will have progressed from the state in which the original work was done. The second occurrence of rework is when Declare and Define Rqmt is referenced from the design phase. In this case an activity that is first invoked in one context is later invoked in a different context, but for essentially the same purpose. That is, to complete a specific requirement that contributes toward the completion of the whole requirements specification.

2.3 Analysis

Based on analysis of the example-scenario program, what can we understand about rework in the process? What is distinctive about the rework context and what is retained from the context of the original work?

We can plainly see the contexts in which rework occurs, at specific points within the requirements phase (the original context) in one case and within the design phase (a different context) in the other case. The rework activity is explicitly represented by a step in the process. This step has a specific interface and (in this example) a particular postrequisite (entailing a requirements review when a new requirement is defined). The rework activity is the same one used in initial development.

During initial development the activity is invoked as part of normal control flow but both occurrences of rework are triggered by exceptions. Input parameters are passed from different sources in the different contexts. In the requirements phase the principal input is the result ultimately of requirements elicitation (not shown in the program), whereas in the design phase the input is the output of the step Define new rqmt concept.

In its invocations for both initial development and rework, the activity carries the same postrequisite, that is, it is subject to review of the requirements specification. This reflects the continuity of purpose of the activity. However, if the review is not passed, the exception that will be thrown will lead to very different exception-handling processes. In the requirements phase, the exception simply triggers a reexecution of the activity. In contrast, during the design phase, the exception may trigger a much more elaborate repair process. Details are not elaborated here but, for example, repair might involve changing the design elements, or even revisiting the requirements with the customer, activities that would not be necessary, or even sensible, during the requirements phase. The continuation after this handling is also different in this context: instead of repeating Declare and Define Rqmt, the process continues with the Add new requirements elements step (as indicated by the right arrow on the edge).

When we use step Declare and Define Rqmt in the design phase, compared to the invocation in the requirements phase, different resources can be acquired, for two reasons. First, the state of the resource model might be different – different resources might be available or new resources might have been added or removed, for example, due to staffing changes. Second, Little-JIL allows an invoking step to narrow the specification of the space of resources to be considered for an invoked step. The specification on the edge in Figure 5 indicates that the agent must come from the RqmtTeam, which is a resource collection passed from the parent step. In the reinvocation in the design phase,

however, we can constrain the agent specification with a different annotation on the edge (or, in fact, not constrain it at all to indicate that any Rqmt Engr will do).[2]

This example has shown that rework can be described with a variety of process language constructs and semantics, some derived from conventional programming languages, some derived from other sources. Furthermore, because Little-JIL is executable, rework specified this way can actually be executed according to a rigorous semantics, thus providing effective, automated support. Little-JIL programs can be executed using Juliette [12], our interpretation environment for Little-JIL. Juliette faithfully executes the programs according to the Little-JIL semantics by assigning work to execution agents at appropriate times. The agents choose among those steps assigned to them which steps to execute, and when. In this way, we can allow the flexibility that is needed by opportunistic development, while still providing an overall framework for the process, including rework. Thus, by using rigorous semantics, both custom and borrowed from general programming languages, we can provide automated support for realistic software development processes.

3 Related Work

3.1 Rework Models

There is a common belief in the importance and inevitability of rework in real software development. Surprisingly, though, the term is absent from the indices of many software engineering textbooks (for example, [18,30,32]) and from descriptions of the Capability Maturity Model [28].

Still, practitioners argue that software development does not proceed without rework and that rework should thus be modeled explicitly in software processes [19,9]. Rework is assumed in many of the popular software development life cycles. These often describe the life cycle as a nominal flow with rework activities that cause cycles in the nominal flow. Other life cycle models are explicitly based on iteration, for example the Spiral Model [8] and the Unified Process [22]. However, in such life cycles the repeated iterations do not generally represent rework but new, incremental development. Even in incremental development, some rework is bound to occur, and how rework (as opposed to forward iteration) should be handled or represented is not well described. Additionally, life-cycle models are typically presented at a relatively abstract level that is independent of important, practical details of a rework context and activities that must be captured for definitions of rework processes to be useful.

An opportunistic view of software development is seen in design tools such as Argo [31] and Poseidon [4]. These provide automated support for design activities, including automated "critics" that can point out design flaws that may entail rework. However, they do not support modeling of the design process in general or of design

[2] Actually, different resources can be acquired for different invocations of the activity in the same context, as between the initial work and rework in the requirements phase. That is because the resource pool may change from one invocation to the next and the resource specification will be reevaluated upon each invocation. If the same resources should be used for each invocation in one (or more) contexts, this can be specified explicitly in the resource query.

rework in particular. Specific kinds of rework have also been studied (for example, refactoring [17]). While this work is useful, automated support is generally lacking because triggers, and the resultant rework, are not formally defined and not integrated into the overall development process. The lack of automated support deprives practioners of those advantages cited in Section 1.

3.2 Modeling Languages

Workflow and process researchers have studied the software development process as a formal object. This research has produced many process languages. These languages have various constructs, in various combinations, that make them more or less well suited for defining rework processes.

A feature of Little-JIL that seems to offer much benefit is control abstraction and the ability to invoke a step from multiple contexts and with varying parameters. Some languages use procedural or functional abstractions based on conventional programming languages (for example, HFSP [25] and APPL/A [33]). Comparable step or activity abstractions are offered by other languages (such as Oikos [27], EPOS [15], ALF [10], and JIL [34]). Another common approach is based on Petri nets and comparable sorts of flow graphs, some of which also offer control abstraction (for example, SLANG [6], FUNSOFT Nets [16]). Process languages based on finite state models also sometimes offer control abstraction in the form of state or subchart abstraction (e.g., WIDE [11] and STATEMATE [20]).

The main advantage of control abstraction and invocation semantics for definition of rework processes is in allowing an activity that is initially invoked at one point to be reinvoked at another if the work it achieved initially must be redone. Capturing rework in this natural way in languages that lack these semantics is problematic.

Another relatively common notion is the work context. A work context is a scope that groups tools, artifacts, tasks, and roles. Work contexts are often isolated scopes, sometimes with transactional properties. Little-JIL lacks a notion of work context, but supports the specification of artifacts, resources, agents, and substeps from which a work context can be constructed by an external system. Some languages that support work contexts directly are Merlin [23], ALF [10], and Adele-2 [7]. As defined in these languages, work contexts can usually be invoked from (and return to) multiple contexts, thereby supporting rework by reinvocation. Additionally, the various transactional properties of work contexts help to shield rework efforts from conflicts with ongoing activities. Several other process languages have transactional notions, including AP5 [14], Marvel [24], and APPL/A [33]. HFSP [35] supports rework directly with a redo clause which can be used to indicate reinstantiation of a step with different parameters.

Scoping is particularly important for rework. One reason is for the introduction and binding of control and data elements that constitute a particular rework context.[3] With Little-JIL, for example, a step introduces pre- and postrequisites, substeps and their control flow, reactions and exception handlers, and resource and data parameters. Scoping

[3] Strictly speaking it is not the scope that does this directly, but a construct, such as a step in Little-JIL, with which a scope is associated. Nevertheless, it is natural to think of such constructs as scopes, since we typically introduce them in a program for purposes of creating a scope containing certain elements.

is also important for rework as a means to control visibility. Control of visibility is especially important for rework because of the potential for interference between rework and ongoing activities and the need to manage impacts to shared and dependent artifacts. In Little-JIL, steps define a strictly local scope for data and a hierarchical scope for exception handlers and execution agents. Languages based directly on general-purpose programming languages or paradigms can adopt scoping rules from those approaches. Many flow-graph and Petri-net based languages allow for hierarchical scoping of contexts by nesting [6,16,11]. Languages with step, activity, or work context abstractions also use these as a means of introducing and binding process elements and controlling visibility.

A number of process and workflow languages represent data definition or flow. This is important in representing the artifacts that are subject to rework, dependent artifacts, dependency relationships, and the flow of artifacts to (and from) rework activities. Little-JIL represents the flow of artifacts between steps, as do practically all software process and workflow languages. Little-JIL does not support data modeling but is intended to be used with external data definition and management services. Some process languages, such as Marvel [24] and Merlin [23], do not support full data definition but do enable attributes to be associated to artifacts. These can be used to reflect the "state" of the artifact, including its state with respect to rework (for example, whether it is "complete", "outdated", "under revision", and so on). A few languages offer full data modeling capabilities (for example, APPL/A [33] and FUNSOFT Nets [16]), thus allowing the definition of rework processes based on details of the data affected.

Organization and resource modeling are somewhat analogous to product modeling and, like product modeling, provide the basis for an important form of parameterization of rework contexts. Many workflow languages especially support some form of organization, role, or resource modeling(for example, [1,2,3]). Little-JIL allows resources, including execution agents as a special case, to be specified for process steps, although resource modeling itself is intended to be supported by an external service. Some other process languages, such as StateMate [20] and FUNSOFT Nets [16], also incorporate organization modeling as a first-class element. Other software-process languages, for example, Merlin [23], EPOS [15], and ALF [10], allow the specification of particular kinds of software-oriented resources, such as user roles and tools, that may be involved in particular activities.

Exception handling is also an important part of rework processes. Rework is generally triggered by some form of exception, and rework processes can be viewed as a form of exception handling. Exceptions and exception handling are common notions in programming languages. APPL/A [33] adopted Ada-style exception handling, and its successors JIL [34] and Little-JIL continue to represent exception handlers as first-class elements. With Little-JIL we have found them important for specifying rework processes (as discussed in Section 2). Surprisingly, few other software process languages include general exception handling. There are many languages, that provide support for the handling of exceptions that can be specified as consistency conditions (for example, AP5 [14], Marvel [24], Merlin [23], EPOS [15], ALF [10], and others). Exception handling in the programming-language style is uncommon in workflow languages, although there are examples of it [2,3,1].

4 Recommendations and Applications

Rework processes exhibit a variety of characteristics, both within themselves and in relation to the overall software process. Many different kinds of language construct are thus potentially relevant to the formalization of rework processes. In formalizing rework processes in Little-JIL we have taken the approach of representing rework processes as steps that can be invoked from multiple contexts (parent steps). These contexts can be specialized in various ways to reflect the particular circumstances and requirements under which initial development and rework occur. These contexts may differ with respect to pre- and postrequisites, associated activities and reactions, and exception handlers. The contexts in general and the rework activities in particular can be parameterized with appropriate data, resources, and execution agents, according to their place and purpose in the overall process. The assignment of resources and agents can be made statically or dynamically, and control flow within the rework context and the rework process may be more or less strictly determined, as appropriate for the process. In particular, control flow may be "hard coded", left entirely open to the choice of the execution agent, or controlled or not to various intermediate degrees.

From a programming-language perspective, the sorts of constructs and semantics used in Little-JIL to support formalization of rework include control abstraction, interfaces, parameters, control and data flow, scoping, messages (events) and message handlers, and exceptions and exception handlers. We find that the context of rework, which abstractly is a rich notion, is best specified using a flexible combination of these constructs. We find that the rework activity, which can be highly parameterized, is well represented by a procedure-like construct, particularly as that can allow the rework activity to be invoked from and tailored to multiple points in a process at which rework may be found necessary.

The particular collection of mechanisms used in Little-JIL was chosen because the purpose of the language is to support specification of the coordination aspects of software development and similar processes. That is, Little-JIL is intended to support the specification of processes with particular emphasis on the orchestration of steps including especially the binding of agents, resources, and artifacts to those steps. Additionally, Little-JIL is intended to allow these aspects of processes to be specified and understood by people who might lack extensive training or experience in programming but who nevertheless require precise, rigorous process definitions. We believe that Little-JIL addresses these goals for rework processes.

As discussed in Section 3.2, there are a number of additional types of constructs that other sorts of process languages include that can also be usefully applied to the formalization of rework. These include (among others) transactional constructs, data modeling, and consistency specification and enforcement. Data modeling, for example, is appropriate for processes in which control depends closely on the details of data, whereas transactions can be important where data integrity and consistency need to be assured in the face of potentially conflicting activities. In general, constructs for the formalization of rework processes should be chosen according to the kind of information to be captured, the purposes for which it is to be captured, and the circumstances under which the process specification will be developed and used, including the availability

of supporting technologies and the qualifications of modelers, analysts, and others who work on or with the process specifications.

We believe that the formalization of rework processes using process-programming languages affords benefits in three main areas. The first is process understanding. The act of defining processes itself often brings a new understanding, and process definitions, once formulated, can be shared with process engineers, managers, planners, customers, and collaborators. Because the process definitions are formal, they have a precise and rigorous semantics that promotes clarity. This formality leads to the second area of benefit, which is analyzability. Process programs are amenable to automated analysis, which can be used to assess the correctness and other properties of rework processes. This can be useful in verifying and evaluating large process programs that may be beyond the scope of an individual to manually analyze in detail. Analysis of Little-JIL programs is described in [13]. Formality also enables benefits of the third kind, namely, those based on support for process execution. The executability of process programs can support process automation and guidance, thereby promoting process efficiency, control, and fidelity. Additionally, as a process executes, it can be automatically monitored, and information about the process can be gathered for purposes of measurement, analysis, evaluation, planning, and accounting.

Benefits in these areas are particularly applicable to rework. Formalization of a process in a process-programming language can clarify where various kinds of work and rework occur in a process and suggest opportunities to minimize and optimize rework efforts. Analysis can help to compare alternative process models and to verify properties relating to the sequencing, timing, impact, and other properties of rework tasks. Execution support can help to assure that rework activities are carried out as specified, supported appropriately, coordinated with other activities, and accomplished correctly and efficiently with minimal impact. Monitoring mechanisms can track initial development and rework costs, help to show the benefits of rework mitigation efforts, and facilitate planning and management for ongoing activities. For all of these reasons, we believe that using process-programming languages to formalize rework processes not only teaches us something about process languages but also has the potential for significant practical benefit in an area of software development that continues to be problematic and costly.

5 Conclusions and Future Work

Software-process programming holds the clear prospect of providing a number of benefits to software engineers. Chief among these is the prospect that, through the use of sufficiently powerful and precise languages, software-process programs can effectively elucidate the nature of key software processes. In this paper, we have demonstrated the potential of an effective process programming language to elucidate the concept of rework. Rework is a common software development concept, referred to often by practitioners, but seldom addressed or carefully defined in software engineering textbooks. One explanation for this, we believe, is that the notion of rework requires specific semantic notions in order to be adequately explained and understood. Such notions include procedure invocation, scoping, and exception management, among others. Thus, we have

demonstrated that effective process languages can materially add to our understanding of commonly used software engineering terms and practices.

We have also demonstrated the potential for an executable software-process programming language to aid practice by being the basis for more powerful, precise, and effective guidance of practitioners. While the example given in this paper is relatively simple, it nevertheless suggests that rework situations can become rather complex and intricate. In such situations, the availability of an executable process definition to help guide practitioners through the rework process, and then back to mainstream development, would seem to have considerable value. In cases where there are multiple practitioners engaged in multiple rework instances driven by multiple contingencies, a clear, disciplined process, supported by effective process interpretation, would seem to offer the potential for important savings in confusion, errors, effort, and cost.

While we are convinced of the potential for software-process programming to offer advantages in clarifying the nature of key processes and effectively aiding practitioners in performing these processes, the work described here is hardly a definitive demonstration of these benefits. Thus, we propose to continue these investigations by using process-programming languages to program increasingly complex and realistic rework processes. In this work, we expect that our processes will entail more complexity, greater use of exception management, more involved examples of process variation, and more realistic specifications of resources. Having developed such processes, we expect to use our process interpretation facilities to support their execution, in order to gauge their value in guiding real practitioners engaged in real software development.

Through these continuing experiments we expect to gain greater understandings of the sorts of linguistic semantic features of most value in supporting the representation of rework processes, and indeed wider classes of realistic software processes. In addition we expect that this research will lead to rework processes of known value and utility. As these processes will have been captured in a rigorous language, they would then be reproducible and usable by others, including real practitioners. In this way, we expect to demonstrate how effective process languages can improve the state of software practice through improved understandings of the nature of software processes.

Acknowledgements. This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032, by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231, and by the National Science Foundation under Grant No. CCR-0204321. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U.S. Dept. of Defense, the U. S. Army, The National Science Foundation, or the U.S. Government.

References

1. FileNet Business Process Manager; URL: http://www.filenet.com.
2. PAVONE Process Modeler; URL: http://www.pavone.com/web/uk/pages.nsf/goto/home.
3. MultiDESK Workflow; URL: http://www.multidesk.com/.
4. Poseidon for UML. http://www.gentleware.com.
5. *First CeBASE eWorkshop: Focusing on the cost and effort due to software defects*. NSF Center for Empirically Based Software Engineering, Mar. 2001. http://www.cebase.org/www/researchActivities/defectReduction/eworkshop1/.
6. S. Bandinelli, A. Fuggetta, and S. Grigolli. Process modeling in-the-large with SLANG. In *Proc. of the Second Int. Conf. on the Soft. Process*, pages 75–83. IEEE Computer Society Press, 1993.
7. N. Belkhatir, J. Estublier, and M. L. Walcelio. ADELE-TEMPO: An environment to support process modeling and enaction. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 187–222. John Wiley & Sons Inc., 1994.
8. B. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.
9. K. Butler and W. Lipke. Software process achievement at Tinker Air Force Base. Technical Report CMU/SEI-2000-TR-014, Carnegie-Mellon Software Engineering Institute, Sept. 2000.
10. G. Canals, N. Boudjlida, J.-C. Derniame, C. Godart, and J. Lonchamp. ALF: A framework for building process-centred software engineering environments. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 153–185. John Wiley & Sons Inc., 1994.
11. F. Casati, S. Ceri, S. Paraboschi, and G. Pozzi. Specification and implementation of exceptions in workflow management systems. *ACM Trans. on Database Systems*, 24(3):405–451, Sept. 1999.
12. A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and A. Wise. Logically central, physically distributed control in a process runtime environment. Technical Report 99-65, U. of Massachusetts, Dept. of Comp. Sci., Nov. 1999.
13. J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. Verifying properties of process definitions. In *Proc. of the 2000 Int. Symp. on Soft. Testing and Analysis (ISSTA)*, pages 96–101, Aug. 2000.
14. D. Cohen. *AP5 Manual*. Univ. of Southern California, Information Sciences Institute, March 1988.
15. R. Conradi, M. Hagaseth, J.-O. Larsen, M. N. Nguyên, B. P. Munch, P. H. Westby, W. Zhu, M. L. Jaccheri, and C. Liu. EPOS: Object-oriented cooperative process modelling. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33–70. John Wiley & Sons Inc., 1994.
16. W. Deiters and V. Gruhn. Managing software processes in the environment melmac. In *Proc. of the Fourth ACM SIGSOFT/SIGPLAN Symp. on Practical Soft. Development Environments*, pages 193–205. ACM Press, 1990. Irvine, California.
17. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
18. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Englewood Cliffs, NJ, 1991.
19. T. Haley, B. Ireland, E. Wojtaszek, D. Nash, and R. Dion. Raytheon Electronic Systems experience in software process improvement. Technical Report CMU/SEI-95-TR-017, Carnegie-Mellon Software Engineering Institute, Nov. 1995.
20. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Soft. Eng.*, 16(4):403–414, Apr. 1990.

21. J. Herbsleb, A. Carleton, J. Rozum, J. Siegel, and D. Zubrow. Benefits of cmm-based software process improvement: Initial results. Technical Report CMU/SEI-94-TR-013, Carnegie-Mellon Software Engineering Institute, Aug. 1994.
22. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Object Technology Series. Addison-Wesley, 1999.
23. G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 103–129. John Wiley & Sons Inc., 1994.
24. G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In B. Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131–140, Kona HI, Jan. 1990.
25. T. Katayama. A hierarchical and functional software process description and its enaction. In *Proc. of the Eleventh Int. Conf. on Soft. Eng.*, pages 343–353. IEEE Comp. Society Press, 1989.
26. J. King and M. Diaz. How CMM impacts quality, productivity, rework, and the bottom line. *CrossTalk - The Journal of Defense Software Engineering*, pages 3–14, Mar. 2002.
27. C. Montangero and V. Ambriola. OIKOS: Constructing Process-Centered SDEs. In A. Finkelstein, J. Kramer, and B. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 33–70. John Wiley & Sons Inc., 1994.
28. M. C. Paulk, C. V. Weber, B. Curtis, and M. B. Chrissis. *Guidelines for Improving the Software Process*. Addison-Wesley, New York, 1995.
29. R. M. Podorozhny, B. S. Lerner, and L. J. Osterweil. Modeling resources for activity coordination and scheduling. In *Proceedings of Coordination 1999*, pages 307–322. Springer-Verlag, Apr. 1999. Amsterdam, The Netherlands.
30. R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, fourth edition, 1997.
31. J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Argo: A design environment for evolving software architectures. In *Proc. of the Nineteenth Int. Conf. on Soft. Eng.*, pages 600–601. Assoc. of Computing Machinery Press, May 1997.
32. I. Sommerville. *Software Engineering*. Addison-Wesley, fifth edition, 1996.
33. S. M. Sutton, Jr., D. Heimbigner, and L. J. Osterweil. APPL/A: A language for software-process programming. *ACM Trans. on Soft. Eng. and Methodology*, 4(3):221–286, July 1995.
34. S. M. Sutton, Jr. and L. J. Osterweil. The design of a next-generation process language. In *Proc. of the Sixth European Soft. Eng. Conf. held jointly with the Fifth ACM SIGSOFT Symp. on the Foundations of Soft. Eng.*, pages 142–158. Springer-Verlag, 1997. Zurich, Switzerland.
35. M. Suzuki, A. Iwai, and T. Katayama. A formal model of re-execution in software process. In *Proc. of the Second Int. Conf. on the Soft. Process*, pages 84–99. IEEE-PRESS, Feb. 1993. Berlin, Germany.
36. A. Wise. Little-JIL 1.0 Language Report. Technical Report 98-24, U. of Massachusetts, Dept. of Comp. Sci., Apr. 1998.
37. A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton, Jr. Using Little-JIL to coordinate agents in software engineering. In *Proc. of the Automated Software Engineering Conf.*, Sept. 2000. Grenoble, France.

Lessons Learned and Recommendations from Two Large Norwegian SPI Programmes

Reidar Conradi[1], Tore Dybå[2], Dag I.K. Sjøberg[3], and Tor Ulsund[4]

[1] NTNU, conradi@idi.ntnu.no
[2] SINTEF, tore.dyba@sintef.no
[3] Simula Research Laboratory/University of Oslo, dag.sjoberg@simula.no
[4] Bravida Geomatikk, tor.ulsund@bravida.no

Abstract: Software development is an experimental discipline, i.e. somewhat unpredictable. This suggests that software processes improvement should be based on the continuous iteration of characterization, goal setting, selection of improved technology, monitoring and analysis of its effects. This paper describes experiences from the empirical studies in two large SPI programmes in Norway. Five main lessons were learned: 1) It is a challenge for the industrial partners to invest enough resources in SPI activities. 2) The research partners must learn to know the companies, and 3) they must work as a multi-competent and coherent unit towards them. 4) Any SPI initiative must show visible, short-term payoff. 5) Establishing a solid baseline from which to improve is unrealistic. Based on these lessons, a set of operational recommendations for other researchers in the area are proposed.

Keywords: software process improvement, empirical studies, industrial collaboration.

1 Introduction

Improving the software process, or the way and with what means we develop software, is recognized as a key challenge in our society – cf. the American PITAC report [1] and the European Union's framework programmes [2].

The first three authors of this paper were responsible for the software process improvement (SPI) work jointly conducted by three research institutions in two successive, cooperative, industrial Norwegian SPI programmes, called SPIQ and PROFIT. The fourth author was the overall manager of both programmes. A dozen software-intensive companies, mostly small- and medium-sized enterprises (SMEs) participated in the programmes.

This paper describes the main lessons learned from the seven years of experience in these two programmes from *the point of view of the authors*. We describe potential motivations for why companies and individuals take part in such a programme, and that many of these motivations may make the SPI work very hard. We also focus on requirements to the involved researchers for successful conduct of an SPI programme. E.g. they must familiarize themselves with each company, and work as a multi-competent and coherent unit towards the company. It is important to show visible, short-term payoffs, while complying with long term business strategies and research

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 32–45, 2003.

objectives. Finally, we describe the importance of generating new understandings and new actions, in whatever order they evolve.

Based on these lessons learned, we propose a set of operative recommendations for other researchers in the area. We will also apply these recommendations ourselves in a new, successor SPI programme, SPIKE, that we have just started.

To make our general position on SPI more explicit, we will start by characterizing some existing SPI approaches and their assumptions. In our view, SPI covers both *process assessment*, *process refinement*, and *process innovation*. Typical improvement approaches have involved SPI frameworks such as CMM [3], BOOTSTRAP [4], SPICE (Software Process Improvement and Capability dEtermination, ISO/IEC 15504) [5], and the Quality Improvement Paradigm (QIP) [6]. CMM has later been supplemented with the IDEAL improvement model and with Personal Software Process and Team CMM.

Most of these frameworks have become well-known among practitioners and researchers. However, such frameworks implicitly assume rather stable and large organizations ("dinosaurs"), and that software can be systematically developed in a more "rational" way – cf. the legacy of Total Quality Management (TQM) [7]. Frameworks in this category may indeed work well, if the focus is process refinement, as reported in success stories of using CMM in large companies like Hughes and Raytheon in the early 1990s and in more recent studies [8].

However, the opposite situation applies for small companies ("upstarts"), relying on knowledge-based improvisation [9] to rapidly innovate new products for new markets. Even in the US, 80% of all software developers work in companies with less than 500 employees [10]; that is, in SMEs. Rifkin [11] blatantly claims that we have offered the "wrong medicine" to such companies, which have consequently failed in their SPI programs. In contrast to the mentioned CMM studies in the US, several European SPI studies have been performed about SMEs [12, 13]. These studies conclude that short-term priorities – combined with business and market turbulence – may severely prevent, hamper and even abort well-planned and pragmatic SPI efforts. Papers by [14] and [15] elaborate on some of these issues.

Our general position has therefore has been to downscale and make applicable approaches from several SPI frameworks, in particular QIP and TQM, and to apply these in concrete development projects called pilots. Organizational learning has been facilitated through various follow-up actions inside each company, as well as across companies through joint experience groups, shared programme meetings and seminars, technical reports, and two pragmatic method books.

QIP assumes that software development is experimental (not quite predictable) and therefore needs to be conducted accordingly. QIP suggests that projects within an organization are based on a *continuous iteration* of characterization, goal setting, selection of improved technologies, project execution with changed technologies, monitoring of the effects, and post-mortem analysis and packaging of lessons learned for adoption in future projects. Furthermore, measurement is regarded essential to capture and to effectively reuse software experience. An effective and long-lasting cooperation between academia and industry is also necessary to achieve significant improvements. See [16] for a reflection on 25 years of SPI work at NASA.

In our SPI programmes, individual pilots were implemented according to the model of the European System and Software Initiative (ESSI) [17]. Here, an improve-

ment project and a development project (pilot) are managed as two separate, but strongly connected parts □in a so-called Process Improvement Experiment (PIE).

The remainder of this paper is organized as follows. Section 2 describes the mentioned Norwegian SPI programmes and the approaches taken. Section 3 describes the lessons learned. Section 4 presents some operational recommendations based on the lessons learned. Section 5 concludes and contains ideas for further work.

2 The Norwegian SPI Programmes

This section describes the SPIQ, PROFIT and SPIKE programmes for industrial SPI in Norway. SPIQ and PROFIT are finished, while SPIKE is upstarting.

2.1 General about the Norwegian SPI Programmes

The first programme, *SPIQ* or *SPI for better Quality* [18], was run for three years in 1997-99, after a half-year pre-study in 1996. The successor programme, *PROFIT* or *PROcess improvement For IT Industry*, was run in 2000-02. Both programmes were funded 45% by The Research Council of Norway and involved three research institutions (NTNU, SINTEF, and University of Oslo) and ca. 15-20 IT companies, both SMEs and large companies. More than 40 SPI pilot projects have been run in these companies. A follow-up programme, *SPIKE* or *SPI based on Knowledge and Experience*, is carried out in 2003-05 with 40% public funding.

All three programmes were and are being coordinated and lead by one of the industrial partners, Bravida Geomatikk (previous part of Norwegian Telecom), which acts on behalf of one of the major Norwegian IT organizations, Abelia. The public support of ca. 1 mill. Euro per year is mostly used to pay 10-12 part-time researchers, travel expenses, administration and deliveries from the companies. The main annual contribution to a company is a fixed amount (now 12,500 □), plus 300-400 free researcher hours to help carry out a pilot project and thus improve each company. The companies may recently also be compensated extra for direct participation in concrete experiments (see section 4). A total of six PhD students will be funded by these three programmes (three of which have already received their PhD), and over 30 MSc students have so far done their thesis related to these programmes.

The programmes provide networks of cooperation both between researchers and the IT industry, between researchers on a national and international level, and among the participating companies. Figure 1 below shows the principle cooperation and work mode:

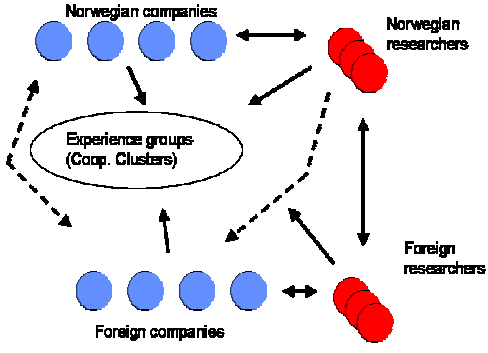


Fig. 1. Cooperation mode in the SPI programmes

Typical work in the industrial companies includes pilot projects to test out a given improvement technology, like novel inspection techniques, agile development methods, electronic process guides, or use of software experience bases for estimation (see later). There has been local planning and follow-up activities around each pilot project, involving one or two researchers and an industrial partner. There have also been monthly coordination meetings between the researchers and the programme manager, and two common technical meetings (seminars) per year. In addition comes researcher-lead experience groups (clusters) on subjects like experience bases and testing. A bi-monthly newsletter has been issued (on paper and web), and various technical reports written with industry as a targeted audience. International collaboration has in some cases also involved direct, inter-company exchanges.

Education has effectively been used to spread results. For instance, several revised and new courses have been extended with SPI-related topics and formal experiments around SPI. MSc and PhD students participate regularly in industrial case studies, and this has been highly appreciated by all parties.

This type of *shared collaboration programmes* over a sufficient number of years have proved advantageous in counteracting the volatility problems of SMEs and their priorities. *Learning* in our context therefore assumes *cross-company* activities, and thus willingness to share own experience, bad and good. Luckily, most companies have similar problems. That is, effective long-term SPI learning can take place as a joint endeavour by and for the IT industry. In this task, academia (the three research partners and their foreign contacts), the Abelia industrial association, and The Research Council of Norway act as initiating and coordinating bodies. Thus the *whole* IT industry can learn and improve, not only single and spurious companies (SMEs).

2.2 The SPIQ Programme in 1997–99

We refer to the previous description of general work mode. The main result of SPIQ (www.geomatikk.no/spiq) was a first version of a pragmatic *method handbook* [19]. This handbook combined aspects of *TQM* and *QIP* including the latter's *Goal-*

Question-Metric method [20] and *Experience Factory* concept [21]. SPIQ has furthermore, as mentioned, adopted the *ESSI* PIE model. An *incremental* approach was generally advocated, e.g., relying on action research [23]. Five *case studies* from the SPIQ companies were also documented.

2.3 The PROFIT Programme in 2000–02

In PROFIT (www.geomatikk.no/profit), we applied the same overall work mode. We concentrated on improvement under change and uncertainty, learning organizations, and novel technologies like XP and component-based development. We also wrote a second, down-scaled and more accessible *method handbook* [24], issued by a Norwegian publisher.

Mostly in parallel with PROFIT, we are running an INTER-PROFIT add-on programme in 2001-04 to promote international experience exchange and cooperation between Norwegian and European companies and researchers. Cooperation agreements are in effect between INTER-PROFIT and the CeBASE (www.cebase.org) project in 2000-02 lead by University of Maryland, the ESERNET thematic network (www.esernet.org) in 2001-03 lead by the Fraunhofer IESE in Kaiserslautern, and VTT in Finland. Several joint seminars and workshops have been arranged. All three research partners participate in the International Software Engineering Research Network, ISERN (www.iese.fhg.de/ISERN).

2.4 The Upstarting SPIKE Programme in 2003–05

We have just started a successor programme in 2003–2005, SPIKE (www.abelia.no/spike). The technical focus is adaptation and trials of "context-dependent" methods, i.e. finding out the "correct" application of e.g. incremental methods, object-oriented analysis and design, electronic process guides, knowledge management, and estimation technologies. As in SPIQ and PROFIT, we support empirical studies, shared activities like experience groups and seminars, international cooperation, and PhD studies. See section 4 on Operative Recommendations.

2.5 Some Industrial and Research Goals

In these SPI programmes, some overall *industrial goals* have been (examples):

- at least half of the companies shall run an overall improvement program.
- all the participating companies shall continually run an improvement project linked to a pilot development project.
- all companies shall participate in at least one experience group, e.g. within:
  - SPI in changing environments (technical, organizational, and market)
  - knowledge management and experience data bases
  - model-based development (often based on UML)
  - project management and estimation
- all companies shall make presentations in national or international conferences.

Some *research goals* have similarly been (examples):

- develop methods and guidelines for process improvement with focus on experience and knowledge reuse techniques for environments with a high degree of change.
- document the results in articles and case studies.
- disseminate the results through a close cooperation with universities and colleges.
- contribute to PhD and MSc theses, and an updated curriculum in software engineering and SPI.

2.6 Some Typical SPI Pilot Projects in SPIQ and PROFIT

Figure 2 shows nine typical SPI pilots with their companies and improvement themes:

- Fjellanger-Widerøe, Trondheim, 1998: Improving the inspection process.
- TietoEnator, Oslo, 1998-99: Establishing an experience database for consulting.
- Genera, Oslo, 1999-2000: Improving development methods for web-applications.
- Mogul.com, Oslo/Trondheim, 2000-01: Improving estimation methods for use cases.
- Kongsberg Spacetec, Tromsø, 2001: Establishing an experience database, using PMA.
- NERA, Oslo, 2001-02: SPI by incremental and component-based development.
- Ericsson, Grimstad, 2002: Improving design inspections by OO reading techniques.
- ObjectNet, Oslo, 2002: Experimenting with XP to improve the development process.
- Kongsberg-Ericsson, 2002: Comparing UML with textual requirement formalisms.

Fig. 2. Examples of companies and work themes

As mentioned, a total of 40 SPI pilots have been performed in SPIQ and PROFIT. This corresponds to about 10 pilots per year, as some went over two years.

2.7 Main Results from the SPIQ and PROFIT Programmes

The two programmes have achieved and reported results for many actors and in many directions:

1. for each participating company, e.g. improved methods and processes, and better process understanding,
2. for the Norwegian IT industry, e.g. as experience summaries with revised process guidelines and lessons learned, and by establishing meeting places such as experience groups, seminars, and a web site,
3. for the broader software community, e.g. as collections of published papers and international research cooperation,
4. for the research partners, e.g. as revised educational programs and through MSc and PhD candidates.

5. on a national level: creation of an SPI community between industry and academia, and internal in academia.

The main external results come from points 2 and 3, and comprise revised methods, lessons learned and recommendations in how to carry out SPI in small and medium sized IT-organizations [25, 26]. In addition, almost 100 refereed publications on related topics have been published, and 3 PhD theses defended as part of the SPIQ and PROFIT programmes.

3 Lessons Learned

Many empirical studies have been conducted in the Norwegian industry as part of these SPI programmes. This section describes the main lessons learned from the point of view of the research managers and the programme manager, that is, the authors.

Lesson 1: It is a challenge for the industrial partners to invest enough resources in SPI activities

The authors were research and programme managers of the two SPI programmes. In our opinion, the perhaps largest practical problem was to keep up the SPI activities within the companies. The performance of a company is closely related to the effort (person-hours) it is willing to spend on the SPI activities. There were several reasons for why many of the companies did not spend as much effort as they initially had planned. Understanding these reasons is important in order to achieve success in an SPI programme. This issue is closely related to the underlying *motives* of the companies and the individual contact persons for taking part in partially externally funded SPI programmes such as SPIQ and PROFIT. A company will ask itself "*what's in it for the company?*" Similarly, an individual will ask: "*what's in it for me?*" We have observed the following motives of a company:

- *Improving its software processes.* As seen from research councils that fund SPI programmes, this should be the main motivation of a company.
- *Increasing the level of competence and knowledge in certain areas.* Although a company is not particularly interested in SPI, it may be interested in using an SPI programme to increase the competence in certain technology areas.
- *Publicity*. Several companies use the collaboration with SPI researchers as a selling point.
- *Be present where things happen*: Even if a company is not genuinely interested in SPI, it may wish to participate in such SPI programmes, just to be present where interesting things possibly happen. The company will not risk to miss information or activities that might be considered important by their customers or competitors.
- *Networking and recruitment*: Connecting to companies, research institutes, and universities has proved a real motivation. For example, since many SPI researchers also teach at universities, they have indirectly been recruitment channels for the companies. The companies are also interested in collaborating with the other companies in the programmes on business issues (in addition to SPI activities).

- *Money*. The Research Council of Norway partially supports the companies' SPI activities.
- *Inexpensive consultancy work*: Some companies have work duties, that are not SPI activities, but which they believe they can use researchers in an SPI program to carry out. That is, they may consider researchers as cheap consultants.

The industrial people in the SPI programmes may also individually have various motives for taking part, amongst others:

- Supporting the motives of the company such as those described above. In particular, supporting the SPI activities of the company.
- Personal interest in SPI activities. This may be the case, even if SPI is not considered important within the company.
- Personal interest in research. This may be the case, even if research is not considered important within the company.
- Increased level of competence in certain technology areas. This may be the case, even if those areas are not considered important within the company.
- Internal status in the company. Holding a high profile and demonstrating a good personal network may increase the status and respect of one's colleagues.

The first two motives of a company are the ones that comply with the research councils' intention of the SPI programmes. Understanding the other companies and the motives of the participating individuals, make it easier for the research partners in the programmes to understand the behaviours a participating company and respond accordingly. Note, that even though the main motivations for taking part in an SPI programme are not the 'ideal' ones, the company may still contribute positively to the programme. It is when a company does not contribute to a programme, the underlying motives should be revealed and appropriate actions taken.

Related to the problem of spending sufficient resources on SPI activities within a company, is not only the time of the people involved, but the ability and internal position of the contact persons. To have an impact on the SPI within the company, the peoples should have relevant competence, experience, respect, and position in the company [27]. We have experienced that several companies do not wish to allocate key technical persons to SPI work, because they are considered too important in ordinary business projects (this phenomenon is also seen in collaborative projects, in quality assurance work, and in standardization activities). Therefore, to save costs, the companies let junior or 'less important' persons take part in the programmes. As a consequence, the SPI impact within the company is reduced.

Although tailored involvement is required by an organization for successful SPI, it is no precondition for successful *research*. We experienced in SPIQ and PROFIT several cases where data from a company gave interesting research results, although the SPI internal in the company was neglected.

Lesson 2: The research partners must put much effort in getting to know the companies

The impact of an SPI programme does, of course, depend on the resources spent by the researchers, and their competence and enthusiasm. Another success factor is a

humble attitude of the researchers towards the situation of the practitioners, and the interest and ability to learn to know the actual company. The researcher should help to solve concrete problems of an organization, cf. action research. For studies where a deep understanding of an organization and its processes is important, the researcher will gain much goodwill if he or she takes part in some of the organization's regular activities. That is, to give courses, to participate in social activities etc. The presentation of the research results to a company should be tailored towards different roles and needs, and answer "what is important in this study for me".

Another reason to learn in detail about a company, is that such information may be required to identify the kind of companies and contexts to which the results can be generalized. That is, where they are relevant and valid.

Lesson 3: The research partners must work as a multi-competent and coherent unit towards the companies

After seven years of experience, we have learned that a research team for a successful SPI programme primarily must be *competent*. That is, it must cover the relevant technical areas in a convincing manner, e.g. testing, incremental development, component-based development etc. In addition comes knowledge of SPI-related methods, often coupled with insight in organizational science and data analysis. Likewise, the researchers should have adequate industrial experience and insight. E.g. putting fresh university candidates to "improve" seasoned developers is ill-advised, although junior researchers can grow into able assistants, often having ample time to work and familiarize with the practitioners.

The second main issue is that the team must cooperate well internally, i.e. be externally in line. There can and should be tough scientific and other discussions among the researchers themselves, but towards industry, the team should appear as a *coherent* unit. Within a company there are many stakeholders with different interests and agendas in the local SPI effort. The companies are openly reluctant to collaborate with a research team that disagrees on the approach and activities towards the company, simply because it complicates the already complex, local SPI effort even further.

To begin with, the SPIQ and PROFIT programmes were run by four research institutions. Due to internal problems of collaboration, which were partly visible to the collaborating companies, one of the research institutions had to leave the programme after two years. After seven years, the working environment among the remaining researcher institutions is very good. In other words, do not underestimate the time it takes to create a well-functioning SPI research team of a certain size (we have been from 10-15 researchers at any time).

Lesson 4: Any SPI initiative must show visible, short-term payoff

The research community is used to think that improvement actions will have long-term pay off in terms of increased competitiveness, improved product quality, increased competence etc. However, it is often the task of the SPI responsible within the individual company to visualize the payoffs of their investments in SPI. If they

fail to do so, it may cause lack of confidence and support from management, which again is an important prerequisite for succeeding in SPI.

For instance, top management in a cooperating company said suddenly that they wanted to release the internal SPI responsible from his tasks, because they could not see that he had delivered the expected SPI results. The management had expected documentation of a new process model in the form of a printed handbook. The SPI responsible, on the other hand, had concentrated his effort on establishing company-wide motivation for the new SPI initiative and was planning extensive experience harvesting sessions to build the new process description. This had not been visible to the management; instead they saw a large consumption of human resources without any concrete results.

This story shows the importance of being open about the overall process from the very beginning, and to explain how the SPI activities eventually will lead to a worthwhile benefit. Even large companies will not embark upon 5-year SPI plans.

Lesson 5: Establishing a solid baseline from which to improve is unrealistic

The conventional way of thinking about SPI puts generation of new understandings and the associated actions in a *sequential* order, i.e., first understanding, then action. For example, in the early Japanese software factories, a strong emphasis was put on gathering data on existing processes, before changing or improving them [28]. QIP similarly advocates that we should first understand what we do, before we attempt to change it – in line with the Plan-Do-Check-Act loop in TQM.

SMEs face two main challenges with this approach: 1) an ever-changing environment, and 2) few projects running at any given point in time. As a consequence, they have few data, which they can analyze and use to build an understanding. In addition, collected data will soon be outdated and left irrelevant or – in the best case – uncertain. Taken together, this implies that SMEs need to utilize their data as soon as it is available, extract the important information for learning, and engage in expedite improvement actions. There is simply no opportunity to collect long time series or amass large amounts of data, needed for traditional improvement approaches such as TQM's Statistical Process Control.

A specific challenge involves balancing the refinement of the existing skill base with the experimentation of new ideas to find alternatives that improve on old ideas, see again [9]. Since the most powerful learning comes from direct experience, actions and understandings often need to be reversed. Therefore, the generation of new understandings and new actions, in whatever order they evolve, is fundamental to successful SPI. This matches research results from organizational science on combined working and learning, cf. the "storytelling" metaphor in Brown and Duguid's seminal study of the (poor) use of experience bases at Xerox [29].

4 Operational Recommendations

Based on our experiences, we have in the new SPIKE programme introduced the following pragmatic guidelines:

- *Agree on expectations*. In addition to a formal contract between the companies and the programme manager, the mutual expectations among the company and researchers should be clarified in a project plan.
- *Teams instead of individuals.* To ensure continuity, we will for each company co-operate with a team instead of a single contact person. The same applies on the research side.
- *Rapidly identify high risk companies.* If a company tends to show little interest (defers or cancels meetings, does not reply to emails, etc.), we will quickly confront the company with this behaviour. Since our experience is that people do not change behaviour in this area, we will try to replace passive contact persons with more enthusiastic ones.
- *Be flexible upon problems*. Flexibility and responsiveness are particularly important when things seem to be going badly, e.g. when people say they have insufficient time or resources for a task. In such cases we will do things differently, extend deadlines, compromise on the task, offer an extra pair of hands, and so on.
- *Provide timely and accurate feedback.* The real payoff from using data analysis in the organizational learning cycle comes when the data is fed back to the local teams, from which it was collected, and problem solving begins.
- *Picking a good topic ("killer application").* The ideal topic for an SPI programme is locally relevant, based on sound evidence, and able to demonstrate tangible benefits in a short time. Such a focus on small and immediately useful result may, however, not always go hand in hand with the researchers needs and interests. Our solution is to offer direct payment for extra effort spent on long-term SPI work internally or on results that mostly are relevant to the research community or industry at large. This suggests that SPI programmes should focus both on short-term and long-term alignment of SPI goals with business goals and research objectives. An important challenge is thus to achieve a high degree of mutual understanding of current business objectives, as well as to align long-term SPI goals with business and research strategies.
- *Tailor payment for concrete activities.* The most active companies will naturally get the lion s share of the programme resources (i.e. free researcher support and direct payment). In SPIQ we bureaucratically requested that each company got a fixed sum for each finished report, such as an initial project plan, intermediate status reports, and a final report. The experience was that the company felt that writing formal reports did not contribute to the SPI effort of the company. The quality of such reports was also too poor for research purposes. In the following PROFIT, we therefore introduced a flat payment model, where each company was given 100,000 NOK (12,500) as long as they contributed with a minimum of activity. Since this model did not stimulate effort over a minimum, SPIKE allows to pay for extra, focused activities. For example, if a company wants to run a controlled experiment on a certain technology, we will also pay the company for the marginal extra effort involved, e.g. 10 employees each working 5 hours 70 per person-hour.
- *Tailor payment to those involved*. It is important that the payment is directed to those who actually perform the SPI work. Particularly in large companies, the payment may easily end up a sink with no gain for those who are involved. The money should benefit those involved at least at the departmental level. We have

experienced that even small incitements can be effective. For example, in one company, we were successful in organizing a lottery where each interviewed person was given a ticket in a lottery where the prize was a hotel weekend in the mountains. In another case, we simply paid each individual 1,000 NOK (125 □) to fill in a questionnaire.

5 Conclusion and Future Work

In the previous section we have outlined some critical factors and lessons learned from the SPIQ and PROFIT programmes.

Some **national-level results** are:

- Many profitable and concrete process improvements in Norwegian IT companies, as well as a better understanding and learning of underlying needs and principles.
- An effective and consolidated cooperation network between researchers and IT industry. This expands a Norwegian tradition for cooperative R&D programmes.
- A fast, national build-up of competence in SPI and empirical methods, resulting in two method books (in Norwegian).
- Upgraded education in software engineering, and many MSc and PhD graduates.
- An extended cooperative network of international contacts.

Some **overall lessons for SPI methods and -programmes** are:

- Textbook methods and approaches to SPI must be fundamentally rethought and down-scaled to be effective in an IT industry with mostly SMEs and a generally high change rate.
- (Inter-)company and long-term learning for volatile SMEs can successfully be organized as a joint effort between companies and research institutions, with proper links to industrial associations and a founding research council. Such efforts must span at least 5 years □in our case 3 times 3 years.
- There should be a close coupling between working and learning. E.g., experience bases and quality systems should be created, introduced, and maintained in a truly collaborative and incremental way [30, 31].

Future work. This paper has described the lessons learned from the point of view of the research managers and the programme manager. We conducted a small survey among the companies to identify their views. They generally seemed happy with the support from the programme, but in several areas the competence and behaviour of the researchers could obviously be improved. One reason for the sometimes low activity in the companies (cf. lessons 1, Section 3), *might* be that the effort of the researchers was not felt sufficiently good. Future work could include interviews with the participants from the industry to identify their real opinions.

Acknowledgements. We thank our colleagues Letizia Jaccheri and Tor St□lhane from NTNU, Torgeir Dings□yr, Geir Kjetil Hanssen, Nils Brede Moe, Kari Juul Wedde and Hans Westerheim from SINTEF, and Bente Anda, Erik Arisholm, Hans Gallis, Magne J□rgensen and Espen Koren from the Simula Research Laboratory/University

of Oslo for their effort in the programmes. We would also like to thank all the companies that took part: Bergen Data Consulting, Bravida Geomatikk, Computas, EDB 4tel, Ergo Solutions, Ericsson, Firm, Fjellanger Widerøe, Genera (Software Innovation), Icon Medialab, Kongsberg Defence Communication, Kongsberg Ericsson Communication, Kongsberg Spacetec, MaXware, Mogul, Navia Aviation, Nera, Numerica Taskon, OM Technology, Siemens, Storebrand, Tandberg Data, Telenor Geomatikk, TietoEnator and TV2 Interaktiv. All programmes were funded by the Research Council of Norway.

References

[1] The President's Information Technology Advisory Committee, "Advisory Committee Interim Report to the President", Aug. 1998, 66 p. See http://www.itrd.gov/ac/interim/.

[2] European Commission, "Information Society Technologies: A Thematic Priority for Research and Development" -- 2003–2004 Work Programme", 90 p. See http://fp6.cordis.lu/fp6.

[3] Marc C. Paulk, Charles V. Weber, Bill Curtis, and Mary B. Chrissis, *The Capability Maturity Model for Software: Guidelines for Improving the Software Process*, SEI Series in Software Engineering, Addison-Wesley, 1995, 640 p.

[4] Volkmar Haase, Richard Messnarz, Günther Koch, Hans J. Kugler, and Paul Decrinis, "BOOTSTRAP: Fine-Tuning Process Assessment", *IEEE Software*, 11(4):25–35, July 1994.

[5] *SPICE, Software Process Improvement and Capability dEtermination*, 1998. See on-line version on http://www.sqi.gu.edu.au/spice/.

[6] Victor R. Basili and Gianluigi Caldiera: "Improving Software Quality by Reusing Knowledge and Experience", *Sloan Management Review*, 37(1):55–64, Fall 1995.

[7] W. Edwards Deming, *Out of the crisis*, MIT Center for Advanced Engineering Study, MIT Press, Cambridge, MA, 1986.

[8] Bill Curtis, "The Global Pursuit of Process Maturity", *IEEE Software*, 17(4):76–78, July/Aug. 2000 (introduction to special issue on SPI results).

[9] Tore Dybå, "Improvisation in Small Software Organizations: Implications for Software Process Improvement", *IEEE Software*, 17(5):82–87, Sept./Oct. 2000.

[10] Robert P. Ward, Mohamed E. Fayad, and Mauri Laitinen, "Thinking objectively: Software Improvement in the Small", *Comm. of ACM*, 44(4):105–107, April 2001.

[11] Stan Rifkin, "Discipline of Market Leaders and Other Accelerators to Measurement", Proc. 24th Annual NASA-SEL Software Engineering Workshop (on CD-ROM), NASA Goddard Space Flight Center, Greenbelt, MD, USA, 1–2 Dec. 1999, 6 p.

[12] Tor Stålhane and Kari Juul Wedde, "SPI – Why isn't it more used?", Proc. EuroSPI'99, Pori, Finland, 26–27 October, 1999, 13 p.

[13] Fabiano Cattaneo, Alfonso Fuggetta, and Donatella Sciuto, "Pursuing Coherence in Software Process Assessment and Improvement", *Software Process: Improvement and Practice*, 6(1):3–22, March 2001.

[14] Erik Arisholm, Bente Anda, Magne Jørgensen, and Dag Sjøberg, "Guidelines on Conducting Software Process Improvement Studies in Industry", Proc. 22nd IRIS Conference (Information Systems Research Seminar In Scandinavia), Keuruu, Finland, 7-10 August 1999, pp. 87–102.

[15] Reidar Conradi and Alfonso Fuggetta, "Improving Software Process Improvement", *IEEE Software*, 19(4):92–99, July/Aug. 2002.

[16] Victor R. Basili, Frank E. McGarry, Rose Pajerski, and Marvin V. Zelkowitz, "Lessons Learned from 25 Years of Process Improvement: The Rise and Fall of the NASA Software Engineering Laboratory", Proc. 24th Int'l Conference on Software Engineering, Orlando, Florida, USA, May 19–25, 2002, pp. 69–79. ACM/IEEE-CS Press.

[17] Luisa Consolini and G. Fonade, "The European Systems and Software Initiative – ESSI: A review of Current Results", Final Version, The European Commission's Directorate General III, Industry, 1997. See http://www.cordis.lu/esprit/src/stessi.htm.

[18] Reidar Conradi, "SPIQ: A Revised Agenda for Software Process Support", In Carlo Montangero, editor, Proc. 4th European Workshop on Software Process Technology (EWSPT'96), pp. 36–41, Nancy, France, 9–11 Oct. 1996. Springer Verlag LNCS 1149.

[19] Tore Dybå et al., SPIQ metodebok for prosessforbedring (V3, in Norwegian), UiO/SINTEF/NTNU, 14. Jan. 2000, ISSN 0802-6394, 250 p. See also http://www.geomatikk.no/spiq.

[20] Victor R. Basili, Gianluigi Caldiera, and Hans-Dieter Rombach, "The Goal Question Metric Paradigm", In [22], pp. 528–532, 1994.

[21] Victor R. Basili, Gianluigi Caldiera, and Hans-Dieter Rombach, "The Experience Factory", In [22], pp. 469–476, 1994.

[22] John J. Marciniak, editor, *Encyclopedia of Software Engineering – 2 Volume Set*, John Wiley and Sons, 1994.

[23] Davydd J. Greenwood and Morten Levin, *Introduction to Action Research: Social Research for Social Change*, Thousand Oaks, California, Sage, 1998.

[24] Tore Dybå, Torgeir Dingsøyr, and Nils Brede Moe: "Praktisk prosessforbedring", Fagbokforlaget, ISBN 82-7674-914-3, 116 p. (in Norwegian, the PROFIT method book). See also http://www.geomatikk.no/profit.

[25] Tore Dybå, "An Instrument for Measuring the Key Factors of Success in Software Process Improvement", *Journal of Empirical Software Engineering*, 5(4):357–390, Dec. 2000.

[26] Tore Dybå, "Enabling Software Process Improvement: An Investigation of the Importance of Organizational Issues", PhD Thesis, NTNU 2001:101, ISBN 82-471-5371-8, 5 Nov. 2001, 332 p. See http://www.idi.ntnu.no/grupper/su/publ/pdf/dybaa-dring-thesis-2001.pdf.

[27] Khaled El-Emam, Dennis Goldenson, James McCurley, and James Herbsleb, "Modelling the Likelihood of Software Process Improvement: An Exploratory Study", *Journal of Empirical Software Engineering*, 6(3):207–229, Sept. 2001.

[28] M. A. Cusumano, *Japan's Software Factories*, Oxford University Press, 1991.

[29] John S. Brown and Paul Duguid, "Organizational Learning and Communities of Practice: Toward a Unified View of Working, Learning, and Innovation", *Organization Science*, Vol. 2, No. 1 (Feb. 1991), pp. 40–57.

[30] Reidar Conradi, Mikael Lindvall, and Carolyn Seaman, "Success Factors for Software Experience Bases: What We Need to Learn from Other Disciplines", In Janice Singer et al., editors, "Proc. ICSE'2000 Workshop on Beg, Borrow or Steal: Using Multi-disciplinary Approaches in Empirical Software Engineering Research", Limerick, Ireland, 5 June 2000, pp. 113–119.

[31] Reidar Conradi and Tore Dybå, "An Empirical study on the utility of formal routines to transfer knowledge and experience", In Volker Gruhn (Ed.): "Proc. European Software Engineering Conference 2001 (ESEC'2001)", Vienna, 10–14 Sept. 2001, ACM/IEEE CS Press, ACM Order no. 594010, ISBN 1-58113-390-1, pp. 268–276.

An Approach and Framework for Extensible Process Support System

Jacky Estublier, Jorge Villalobos, Anh-Tuyet LE, Sonia Sanlaville, and German Vega

LSR-IMAG
220, rue de la Chimie BP53
38041 Grenoble Cedex 9
France
{jacky.Estublier, Jorge.Villalobos, anh_Tuyet.le,
sonia.jamal, german.vega} @imag.fr

Abstract. The issue of building a Process Support System Environment (PSSE), or a family of PSEE, to make them interoperate or to use them to pilot applications or services requires new solutions; there is almost no hope for a single system to address correctly all the issues.
This paper presents how we have addressed, realized and experimented a unified framework and approach. The idea is not to build the perfect PSEE but to provide a systematic way to compose specialized process services in two dimensions: horizontally, extending the functionalities of the system; and vertically providing different possible implementations of the selected functionalities.
The paper presents the framework and its principles, and gives practical examples and experimentations on how the framework was used to create a family of process support systems, the Apel family; how the family is managed and extended; how different process engine were coordinated, and how process-driven applications are handled in our system.

Keywords: MDA, AOP, EAI, BPM, interoperability, Model driven Software Engineering, COTS, Process driven application, workflow, federations.

1 Introduction

Our work on process started in the late 80s. The issue addressed at that time was change control support, as part of our work on Adele Software Configuration Manager [8]. The experience showed us that a trigger-based system is powerful (product based approach) but does not provide a high level view of the process. Therefore, in the early 90s, we tried to integrate Adele with Process Weaver, an activity based process support system. But we realized this was very hard to do and the result was unstable [21]. Then we tried to build a full fledge process support system, APEL, as a combination of parts.
Even if successful, this work led us to a complex system; the Apel kernel was still large, and the interoperability mechanisms, based on message passing were somehow ad-hoc and not general enough [2][10][11]..

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 46–61, 2003.

Therefore the next step was to address inter operability in general and to define general concepts and a framework allowing to solve the many issues raised by interoperability between heterogeneous systems sharing concepts and knowledge. Here systems are not necessarily process support systems [9][1].
Interestingly, it was the experimentation in real industrial applications that forced us to generalize and extend the work; the need was to better evolution control and extensibility features.
Horizontal extension was identified as a way to extend the process functionalities by additional and specialized features. We realized that handling extensibility at a pretty high level of abstraction was possible, and was bringing in many very interesting possibilities.
Vertical extension, in contrasts, is in charge of bridging the gap between the abstract level and the real tools and services used to implement the high level functions. The experience gained showed us that these two very different mechanisms are complementary and both needed to build complex systems.
The approach was used to realize experiences addressing other issues related with process technology. These experiments proved to be surprisingly successful. This paper reports on the principle we used and on the experiences when using our framework for:

- Building a process system as a federation of specialized tools and services around a process kernel.
- Building a system that makes cooperate different process engines.
- Building a process-driven application.

This paper addresses these different points. Chapter 2 presents the general approach and chapters 3, 4 and 5 show how the approach is used for our PSEE system i.e. point 1 above. Chapters 6 and 7 address successively the two last points.

2 Extensible Architecture

Applications undergo two types of evolution: functional evolution, when the problem domain changes, and adaptation, when the characteristics of the solution change. The latter is also called non-functional evolution, and it is often related to the technological changes in the application's environment.
The object-oriented approach was originally developed to simplify software evolution. Unfortunately, objects are only concerned with functional evolution; they have serious problems coping with the majority of non-functional concerns, which are usually scattered in many classes, in obscure ways. Experience shows that extensibility is not a directly addressed by object-orientation: using objects does not guarantee that the software will be easily modifiable. Objects are not, therefore, the composition units we are seeking for an extensible architecture.
Currently, new paradigms have emerged to deal with the intrinsic problems of objects. In particular, we have aspect-oriented programming (AOP) [4][5]and component models [7].
In aspect oriented programming, an application is built as the integration of "aspects" which are different solutions to different concerns. Each concern represents a problem

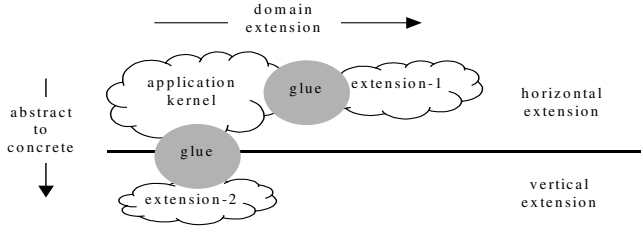
facet. The basic idea is to define, separately, on the one hand the application logic and on the other hand the different concerns, and to weave them all later on.
This approach has several advantages but it poorly supports evolution because the weaving is performed directly on the language structures that can evolve. The application and its aspects are too closely related. The underlying problem is that in the AOP architecture there are no composition elements, but only a mechanism for code weaving. For this reason, we do not consider that AOP proposes an extensible architecture.
In component-oriented programming, the basic structure of the architecture is a graph where nodes are (black box) components described by means of their functional capacities, and the links represent the provide/require relationship. The only missing decision is the concrete implementation of each one of those boxes. Thanks to this decoupling, local evolution inside one component is well supported, but global evolution (a change of the graph structure) or adaptation are hardly handled.
The only composition mechanism is the functional connection, which permits to substitute different implementations of the same functionality, but is not sufficient to support unexpected evolution of the problem domain. Therefore, using components as evolution units is not completely satisfactory.
We want to build applications by composition of high-level elements. Those elements are neither objects nor components, and the extension mechanism is not the simply the connection of well-defined interfaces.
In itself, this goal is not new, and in the recent years interesting work has been performed to reach this objective, through different means. The following presents, in a general way, how we have reached that extensibility goal.

2.1 Building Applications by Extension

An application is defined within a domain, called the problem domain. A domain can be thought as a conceptual structure with precise semantics. This structure defines the concepts pertaining to the domain, their restrictions, and the relationships among them, i.e. a meta-model.
The base of our architecture is a software application that implements a solution in a domain. This application contains and manages a sub-set of the domain concepts, and its semantics reflects the operational model of the domain. We will make the hypothesis, in the following, that the conceptual structure of the application reflects, at least partially, the domain meta-model.
We have defined an extension mechanism for that application, in such a way that specializations of the application domain concepts, and new relationships among them, can be easily added. The extensions should respect the semantics of this application (there must be no interference), but on the other hand, they can take full advantage of the existing kernel to define their own semantics.
There are two types of extension: horizontal and vertical. Horizontal extensions extend the kernel, specializing the application, while vertical extensions consider that the concepts managed by the application are abstract concepts and therefore make a translation from abstract concepts and abstract actions, into concrete execution.
This concrete execution can be performed at least partially by external tools and applications.

In short, we have an architecture based on the following elements:

- An application that implements a solution in a problem domain.
- A set of horizontal extensions of the application that either extends or refines the problem domain and the solution.
- A set of vertical extensions of the application seen as an abstract machine, which represents mapping between abstract action to concrete lower level executions.
- An extension mechanism that connects the application with the extensions.



2.2 Extension Mechanism

To perform horizontal extension, we need to relate the kernel and the extension concepts and to modify the kernel methods to include the new concepts semantics. We have to weave the application method with the extension code.
For the vertical extensions, we need to include in the application methods the code that maps the abstract actions, occurring in the application, to concrete executions, including tools execution.
In both cases, we are weaving the application code with code intended to perform or to call the extensions. In many respect we share AOP (Aspect Oriented Programming) concerns.
Our extension mechanism is the *Extension Object Machine* (EOM). The EOM provides mechanisms to add new attributes (structural extension) and new methods to compiled Java classes and to catch the calls to existing methods and to give the control to an external *controller*, which is charged of making semantic extension.

2.3 The Framework

Our goal is to provide the concepts, mechanisms and tools allowing programmers and designers to extend a kernel application. In contrast with AOP, we do not provide a generic but semantic free mechanism, but dedicated tools and formalisms for vertical and horizontal extension, with a number of predefined non-functional properties. Roughly, the framework contains:

- Editor to describe the external applications and tools
- Editor to add conceptual associations between the kernel application and the extensions. This editor allows also one to describe the extensions semantic in terms of methods extensions.
- An extension weaver, which modifies the kernel to consider the extensions as a part of the application.

3 Abstract Process Machine: The Kernel

To take vantage of the above approach, we have to find out what is the application kernel; in our case what is a process kernel. The issue we had to face was to find out the minimal set of concepts sufficient for supporting process execution, even if there is probably not a single solution to this problem.

There are many definitions of what a process is. Among these definitions we can mention "a systematic way resources are used to produces products", or "a partially set of ordered activities with aim of reaching a goal" [12]. Processes always revolve around three fundamental concepts: activities, products and resources, as illustrated by many process meta-models [20].

Traditionally, process systems can be split into activity based and product based approaches; resource based approaches being better handled by other kind of tools like project managers, planners and others. A traditional difficulty in process support is to find a balance between the product and the activity views. Activity based approaches are recognized to provide a high and global view of the process, at the cost of a less detailed account of how artifacts are managed while, conversely, product oriented approaches can be very fine grained but the process is scattered into many product types which hamper to get a global view and state of the process.

Provided our goal, which is to provide a process kernel around which all other aspects will be "plugged", we selected a high level and activity based approach.

Therefore the basic concept selected is the activity. An activity is a step in a process that contributes to the realization of an objective. It aims usually at generating or modifying a set of products. An activity has a *desktop* in which are found the products required for that activity to be performed. The *ports* define and control the communication between an activity desktop and the external world; they are the only externally visible part of an activity. An activity can have any number of input and output, synchronous or asynchronous ports. An activity has a type, which defines its structure.

A product is a piece of information produced, transformed and consumed by activities. The system assumes products can be copied, duplicated, transferred between activities. Products, in our kernel are typed and named, however the type information is not interpreted; it is simply a name (string). The products managed by the kernel are abstractions of real world entities, but the kernel does not try to know which they are. The product name is essentially used to define data flows. For simplicity reasons, a product name is global in the system.

The *data flow* shows how products circulate between activities that consume, transform or produce them. The control flow expresses the relative (and partial) ordering of activities. A data-flow is a directional "connector" between two (and exactly two) ports, one end of the data-flow being the output data-flow of a port, and the other end being the input data-flow of another port. A data flow is associated with 0 or more products (name and type). A data flow with 0 product is a control flow.

A single port can be the origin or destination of an arbitrary number of data flows. A process model is simply a graph where nodes are activities and arcs are data flow.

A port manages a stack of instances for each incoming or outgoing data flow. A port is said to be *full* when one and exactly one instance of each product entering that port

for each data flow is present. When an input port is full, the port products are transferred to the activity desktop, and activity enters state *ready* (see Fig. 2).
In contrast with products, for the kernel, a resource is something expensive, available in limited amount. It is not possible to create easily, to duplicate or to send resources through the network. Resources also have a limited impact on the kernel; we only recognize that an activity has a responsible, and optionally other resources that at model level are referenced by a symbolic name and a type (role). Names and roles are only strings for modeling purpose; the role being defined as a placeholder for a real resource during execution. Resources can be seen as annotations to the kernel.

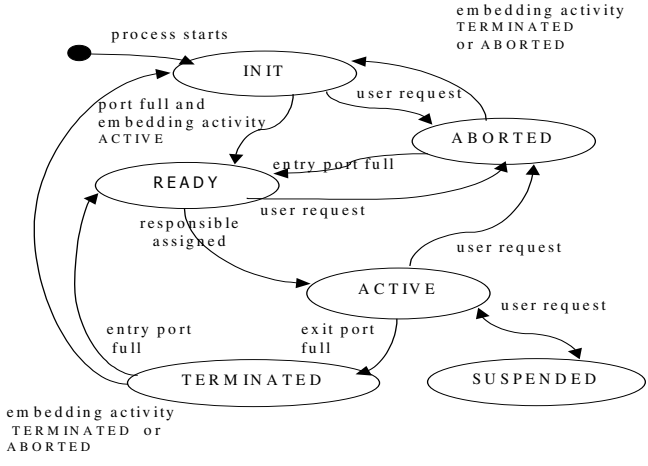


Fig. 1. Activity state diagram

The kernel knows that resources must be assigned to activities before they can really perform. To that end, the activity state diagram introduces a special state *ready* which semantic is: "the activity has the required products to start, but do not have yet the required resources". It is only when the resources are assigned that the activity enters the *active* state (see Fig. 1).
The *ready* state is a hook toward and external resource manager in charge of assigning resources. A state change is performed calling method *changeState (newState)*; the extension machine catches this call and calls a resource manager. The resource manager (or any other program) is supposed to realize the resource assignment, based on the information available in the process (like the responsible attribute) and/or elsewhere, and then must execute method *assigned* on the *ready* activity to make it *active*. From the kernel point of view assigning resources has no meaning. Resources are an interesting case since it defeats the principle that the kernel application should be unaware of its possible extensions; here we had to define an explicit hook to the system, in the form of the *ready* state, and we rely explicitly on an external intervention (a call to method *assigned*) to continue the process.

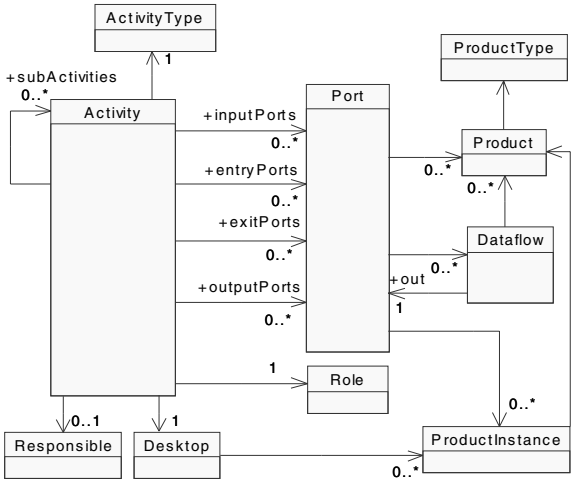


Fig. 2. Process meta-model

Provided that we are looking for a kernel system as simple as possible, it seems the system described so far is sufficient. Indeed we realized soon that it is not the case. Some features are not easy to add as an extension, and therefore we decided to add them to the kernel. Multiple ports, in input as well as in output, asynchronous ports, multiple instance activities and external activities are such features.

Fig. 3 shows a model fragment where some of the above characteristics are displayed. Product *Doc* is sent by activity *A* to both the activities *B* and *C*. *B* and *C* are concurrent activities, therefore product *Doc* is cloned such that two identical copies are provided to *B* and *C*. It means the product model provided by APEL is a versioned product model. Indeed a product is defined as having common attributes (including name and reference) and versioned attributes. Each copy may have different values for the versioned attributes, but they all share the common attributes values. APEL clones products but does not try to merge them; for example, in *D*, two copies of the same *Doc* product are provided in input.

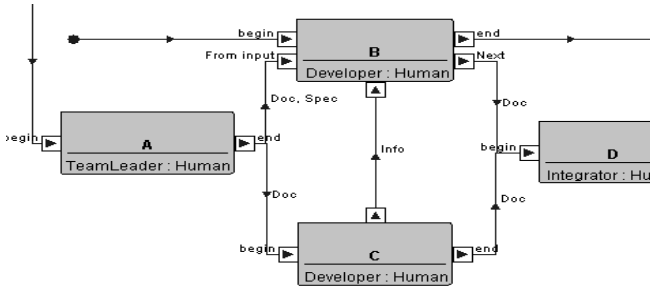


Fig. 3. Process model

Our experience shows that the last implementation of APEL, based on the ideas expressed in this paper, is more stable, extensible and powerful that the initial versions of the tool, thanks to the clear separation of domains and responsibilities.

4 Horizontal Extensions

An extension is said to be horizontal if it is possible to identify relationships between the concepts of the kernel application and those of the extension, that is relationships between their meta-models. It means that the extension is at the same level of abstraction as the application. In the following sections we illustrate the kernel extension mechanism by some of the horizontal extensions we have developed for our own process systems.

4.1 Product Management

For APEL, a product is an abstract entity that can be mapped to different concepts, in actual extended applications. In our case, we were concerned with *products* representing *documents* generated and changed through the software development cycle.
We defined a document as a hierarchy of files and directories. Documents are versioned and can be annotated with attributes, at the global or version level.
As depicted in the figure, there are some direct relationships between the concepts in the kernel and those in the extension. Once integrated, we need to extend the classes in the kernel to keep track of the different mappings introduced by this extension. For instance, the association between a product type, defined in the process model, and a document type, defined in the product model, must be added. Similarly, we have to add new attributes to the product instance to keep track of the reference to the actual document and version it represents.
Despite the structural similarities between the two meta-models, it is important to emphasize that they represent different concepts that are linked at runtime by the federation engine. Products represent ongoing work in a process instance, while documents are persistent entities whose lifecycle extends beyond the process duration.
For example, creating a product in the process is understood as making a relationship between the new product and a document. At this point, we have several alternatives: we can create a new document, or we can create a new version of an existing document, or we can associate the latest revision of an existing document. Conversely, when the product is deleted in the process, it just means that the association is broken, but the document version continue to exist, and can be reintroduced later on in the same or another process as a new product.
To synchronize the product and document lifecycles we use the EOM, which intercepts the appropriate method, calls in the kernel (create and delete, for example) and invoke the code in charge of doing the binding between a product and a particular document.
By decoupling the process model from the product model we can achieve a great deal of flexibility. It is possible, for example, to define a generic process model that can be

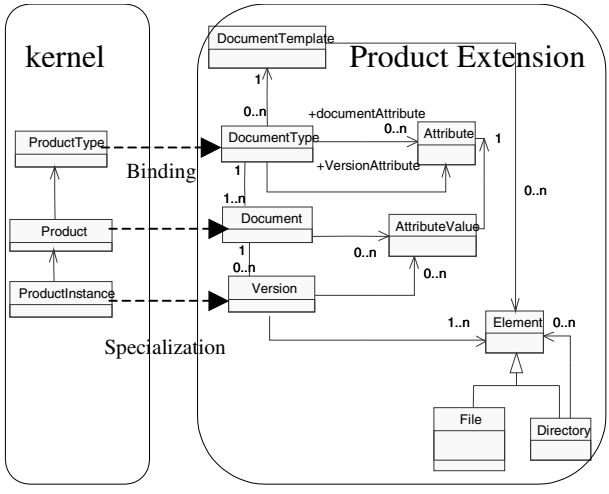


Fig. 4. Product Extension

reused with different document types, chosen at execution time by the extension logic. It is also possible to associate to the same process model different product managers, handling the different product types; for example, some products can be associated with data base entities, others with program entities, and still others managed by specialized tools like a PDM (product data management), a CAD or any tool managing entities.

In the same way, it is also possible to automate document versioning, by associating a version policy to the state of activities in the process.

The same approach has also been used to associate a State Transition Diagram (STD) to some product types. To do so, using the EOM, we add a *state* attribute to these products and the product manager is in charge of synchronizing it to the state of the real product. It is possible to do the same for activities, refining their active state, and defining an STD for some activity types, allowing therefore a closer activity control.

Resource management follows the same schema.

4.2 Concurrent Engineering

As mentioned previously, the core application does not try to reconcile two or more versions of the same product. This task is delegated to the concurrent engineering extension.

The issue addressed is how to control the work performed by a team working on the same document with the goal of producing, collectively, a new version of this document. This is a pretty hard issue, and one of our ongoing research interests.

Our current proposition is an approach based on the definition of high-level concurrent engineering policies and a hierarchical structure of teams, following roughly the work decomposition. Every team has an integrator who is in charge of enforcing the policy. Each team members can play the role of integrator for the team

in the next level of the tree. The hierarchy of teams is mapped to the hierarchy of embedded activities, and the cooperative group to multiple instances activities, see Fig. 5.

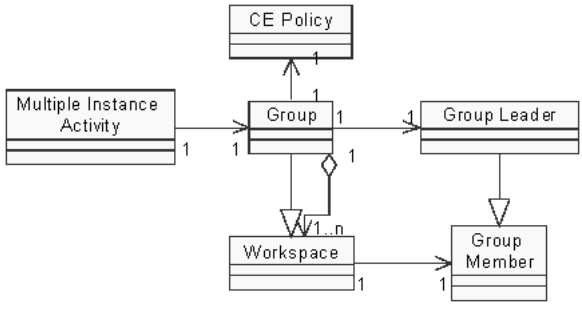


Fig. 5. Concurrent Engineering Extension

However, to identify a group, and associate it to a collaborative tree, it is necessary to identify a pattern in the process model. As shown in Fig. 6, the pattern includes an activity, representing the group, and a multiple instance activity, representing the members of the group, communicating asynchronously with the group desktop; each desktop being associated with a workspace

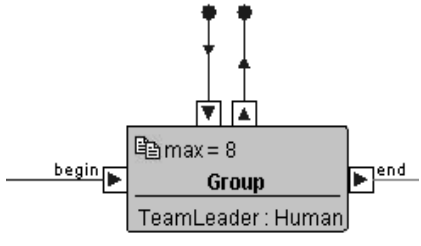


Fig. 6. Concurrent Engineering Pattern

can produce changes in the core process, for example when users create cooperative groups. The extension is then bi-directional, as it must also react the other way around, when for example, the process activates a collaborative activity a corresponding team must be created.

The integration of the CE system was easy because the required concepts were already present in the core application: asynchronous communications, multiple instance activities, and typed activities. This demonstrates that the choice of the core concepts is not an easy task and must be performed with future extensions in mind; which, to some extend, contradicts the extension approach.

It is now clear for us that the major strong point of the approach is the capability to easily complement the core application with complex, dedicated and advanced extensions. A similar result, but related to cooperative work, is commented in [19].

5 Vertical Extension: The Performance

Difference between enaction and performance has been a hot topic for long. Indeed, enacting (a buzzword for executing) a process model does not, per se, act on the real world (performance world). Performance, in our system, is seen as the link between the kernel application (which executes the process model), and the real world. The real world can be made of software tools and application, most often interactive, or sensors or anything else.

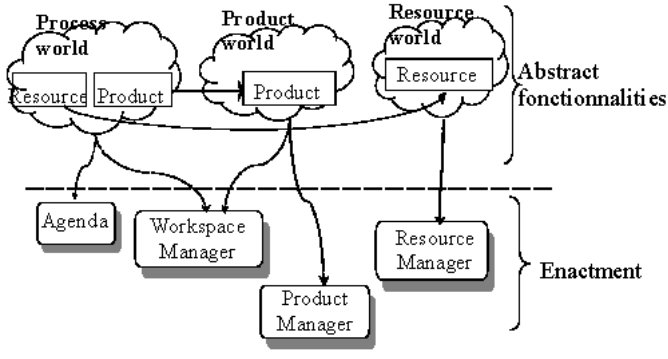


Fig. 7. Vertical extension: using concrete tools

This is called vertical extension since the process engine activities are to be mapped on low-level tools API and other applications. There is no meta-level relationship, but "simply" a translation from a process evolution (e.g. activity A becomes active) and functions to be required from some tools.

It is interesting to note that, for a given process model, the set of concrete tools that will be used to make it perform, is not necessarily fixed. It means that we are free to choose tools providing similar capabilities, as expected in these abstract worlds, depending on our context and our preferences.

In our case, we have an agenda tool used to allow users to interact with the process at run time, making observable the current process state and making possible, for users, to start, terminate activity and so on.

The Resource Manager, in charge of the resource world, is usually linked to a real tool already existing in the company.

Together, the Workspace Manager and the Product Manager tool implement the product world. The product manager deals really with physical files and directories management. It handles also physical versions of all products in a repository. In contrast, the workspace manager deals with strategies of product management in users workspaces. Concretely, it deals with when and how a version of products should be created. Our Workspace Manager supports two kinds of historical versioning: (1) local and (2) global historical versioning.

Local historical versioning is about the management of versions inside a workspace (so, it belong to the execution of a single activity). In contrast, global historical versioning handles versions that are to be kept even after the process that created them disappeared. They are persistent product versions (a single logical version).

6 Process-Driven Applications

Recently, workflow technology has been widely used to compose heterogeneous and existing applications in order to build a new one. Let us call this kind of application a global application. The BPM (Business Process Management) approach is a typical example of this orientation [5].
In BPM, the workflow model is an abstraction view of the interactions graph between concrete applications; each activity in the workflow model represents application activation while link between two activities specifies the execution order.
This approach extends EAI (Enterprise Application Integration), making explicit and external the control flow [14][15][16][17][18]. However, it has drawbacks we consider very important:

- **Strong coupling** between the workflow model and the set of concrete applications. An activity represents just an abstract view of a concrete application.
- **Shared knowledge** is not addressed by BPM. Sharing knowledge reveals the needs to guarantee state consistency of the shared knowledge and therefore dependency constraints between applications.
- **Evolution** when applications evolve. The workflow model must be reconstructed, recompiled, redeployed each time that a constituent application evolves or is replaced by another one.
- **The problem to solve is not explicit**, that is, the problem that the set of concrete applications solve collectively is expressed nowhere, which make difficult the understanding as well as the maintenance of that "solution".

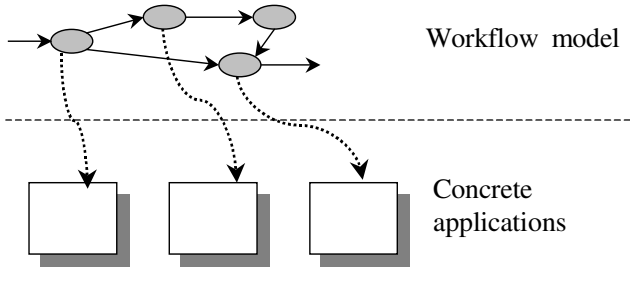


Fig. 8. The BPM approach

Our extensible architecture solves the issues presented above. Instead of linking directly the workflow model with concrete applications, we recommend to:

- **Model explicitly the problem** to solve by making explicit the important concepts of the problem world through the definition of its meta-model. The solution in the problem world will be an application handling the meta-model

concepts, and will be considered the kernel application that will be completed and extended using our extension mechanisms.

- **Use horizontal extension** mechanism to link the process world with the problem world. In practice, identify which entity types in the application will be associated with product and resource types in the process. Thought this link, the application kernel can be evolved according to the progress of the process system. Thus, the process world controls and drives indirectly the application kernel.
- **Extend vertically** the application kernel by using the set of concrete applications. Therefore, these applications will be driven by the evolution of the application kernel, itself controlled indirectly by the process.

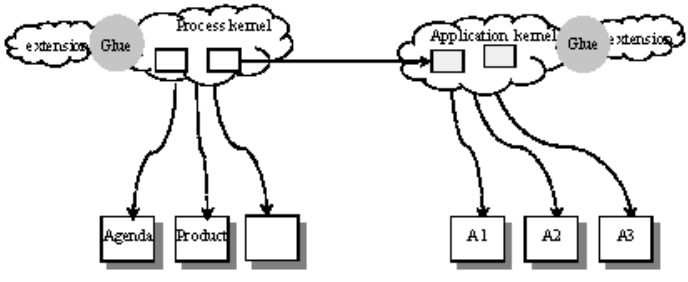


Fig. 9. Process driven in extensible architecture

This approach also extend recent proposition like [13] since we propose a clear separation of functions, a systematic approach, tools to support the approach and a framework for execution control.

7 Experimentations and Validation

We believe very important to stress that our claims are supported by the fact most of the solutions presented in this paper, and many other, have been realised, some of them are currently in industrial exploitation.
One basic claim is that starting from a simple kernel and using extensions it is possible to get a complete and full fledge system. The fact all these systems run, some of them at the industrial level validate the feasibility claim.
We claim that our horizontal extensibility allows for almost unlimited functional extension, with very little changes in the existing kernel and extensions. The experiments we conducted with different product managers, different versioning, different data models, and some advanced feature like our concurrent engineering system, prove this claim is true. Nevertheless, more experiment will be needed to see what are the limits and drawbacks of our approach.
We claim that our vertical extension allows for very much flexibility in implementing the global application, in handling the process performance issues, and in isolating the high level application from technical, platform and other non-functional changes.

These claims have been largely demonstrated; we have used different version managers; we have used different technologies (sockets, RMI, EJB, Web service and SOAP) and different platform (Windows, Unix), and different commercial applications, without having to change the process and its extensions.
Finally we have claimed that, using our approach, it is possible to build systems simultaneously lighter weight, more efficient and more powerful in a given application domain, since only the required features are present. Our experience with Apel largely confirms that claim. Indeed, Apel, as presented in [2] was 98734 LOCS in 473 classes; the new Apel kernel is 6459 LOC in 33 classes. The new kernel is between 2 and 3 orders of magnitude faster, not talking about robustness, reliability and maintainability.
Also very interesting, so far, we never had to use the Apel advanced features not present in the new kernel. Conversely we use product manager, concurrent engineering and workspace extensions that it was not possible to add into Apel, at least without major rewriting. We really got a much more powerful, efficient and reliable system with our technology than with the previous approaches.
These numbers can be argued since the new system needs the federation to run. But the federation engine is only 18K LOC, and all the tools provided for the design and modelling are altogether 20K LOCS; they are generic, and have been used for completely different applications.
Therefore we believe that our claims are largely validated by the experience.

8 Conclusion

After a very slow and timid progression, process (and workflow) support is becoming pervasive. It can be found in business and office automation, but also now in the many process-driven approaches.
The relationships between process models, one or more process engine(s) and the many tools, services, applications that constitute today applications are becoming extremely complex. It is time to rethink what is a PSEE, what is its role, how it is structured, how it can be extended, and what are its relationships with the other parts of a complex system.
This paper presents a new way to build PSEEs based on a simple kernel and on independent and specialized pieces. We have developed a framework and a method allowing composing these pieces in two dimensions: horizontal and vertical. The horizontal extensions are meant for extending the functional capabilities of the resulting PSEE, while the vertical dimension is responsible for controlling the performance i.e. the relationship with the existing tools, actors and resources of the company.
This composition approach allows building “a la carte” PSEEs containing the features needed by the client, including very advanced ones, and only the needed features. In this way the resulting PSEE is lightweight, efficient and targets precisely its application domain. The horizontal composition mechanism works at a high abstraction level, and pieces really ignore each other, allowing for true composition.

The vertical composition allows for an independent performance control. For a given process model, many different implementations, tools, products or resources may be involved. Therefore, the evolution of the execution context, of the tools (versions or new tools) and resource availability do not interfere with the process definition. Much better reuse and process performance control is achieved that way.
The approach and technology we propose also solves many problems found when process technology is used building process-driven applications and services, or when heterogeneous PSEEs have to collaborate in the same application.
We believe the approach we present is a significant step forward in process technology itself as well as in the domains that rely on process technology. It should contribute to the widespread dissemination of process technology as a central element for complex software design and development.

References

1. J. Estublier, T. Le-Anh, J. Villalobos. "Using Federations for Flexible SCM Systems". Proceedings of the 11th International Workshop on Software Configuration Management (SCM-11), USA, May 2003.
2. J. Estublier, S. Dami, and M. Amiour. “APEL: A Graphical yet Executable Formalism for Process Modeling”. Automated Software Engineering, ASE journal. Vol. 5, Issue 1, 1998
3. F. Leymann, D. Roller, "Workflow-Based Applications". IBM Systems Journal, Vol. 36, No. 1, 1997.
4. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J.-M. Loingtier, J. Irwin. “Aspect-Oriented Programming”. Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP ‘97). Lecture Notes in Computer Science vol.1241, Springer-Verlag, June 1997.
5. K. Lieberherr, D. Orleans, J. Ovlinger. “Aspect-Oriented Programming with Adaptive Methods”. Communications of the ACM, Vol. 44, No. 10, pp. 39–41, October 2001.
6. H. Smith, P. Fingar "Business Process Management: The Third Wave". Meghan-Kiffer Press, ISBN: 0929652339, December 2002.
7. F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda. "Volume II: Technical Concepts of Component-Based Software Engineering". Technical Report CMU/SEI-2000-TR-008, Carnegie Mellon University, May 2000.
8. J. Estublier and S. Dami and M. Amiour. High Level Process Modeling for SCM Systems. SCM 7, LNCS 1235. pages 81–98, May, Boston, USA, 1997
9. J. Estublier, H. Verjus, P.Y. Cunin. "Designing and Building Software Federations". 1st Conference on Component Based Software Engineering. (CBSE), Warsaw, Poland, September 2001.
10. J. Estublier, M. Amiour, S. Dami. "Building a Federation of Process Support System". Conference on Work Activity Coordination and Cooperation (WACC), ACM SIGSOFT, Vol. 24, No. 2, San Francisco, USA, February 1999.
11. J. Estublier, P.Y. Cunin, N. Belkhatir. "An Architecture for Process Support Interoperability". ICSP 5, pp. 137–147, Chicago, Illinois, USA, June 1998
12. B. Curtis, M.. I. Kellner, J. Over. "Process Modeling", Communications of the ACM, Volume 35, No. 9, September 1992.
13. Valetto G., Kaiser G. “Using Process Technology to Control and coordinate Software Adaptation”. ICSE, Portland May 2003.

14. J. Sutherland, W. van den Heuvel. "Enterprise Application Integration and Complex Adaptive Systems". Communications of the ACM, Vol. 45, No. 10, October 2002.
15. J. T. Pollock, “The Big Issue: Interoperability vs. Integration”, EAI Journal, pp. 48–52, October 2001.
16. J. C. Lutz. "EAI Architecture Pattern". EAI Journal, pp. 64–73, March 2000.
17. S. Van den Enden, E. Van Hoeymissen et al. "A Case Study in Application Integration". Proceedings of the OOPSLA Business Object and Component Workshop, 15th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, USA, 2001.
18. J. Lee, K. Siau , S. Hong. "Enterprise Integration with ERP and EAI". Communications of the ACM, Volume 46. Issue 2, February 2003.
19. Alf Inge Wang: "An Evaluation of a Cooperative Process Support Environment", In Proceedings "IASTED International Conference on Software Engineering and Applications" SEA 2002, Cambridge, MA, USA 4–6 November 2002, 10 pages.
20. Workflow Management Coalition "Workflow Standard - Interoperability - Abstract Specification", Document Number WFMC-TC-1012, Version 2.0b, 30 November 1999.
21. J. Estublier and S. Dami. *Process engine interoperability: An experiment*. In C. Montangero, editor, *European Workshop on Software Process Technology (EWSPT5)*, LNCS, Nancy, France, October 9–11 1996. Springer Verlag, pages 43–61.

Quality Ensuring Development of Software Processes

Alexander Förster and Gregor Engels

Department of Computer Science
University of Paderborn
D-33095 Paderborn, Germany
{alfo,engels}@upb.de

Abstract. Software development is a complex process where many organizational units, persons, systems and artifacts are involved. In companies that exceed a certain size the business processes become difficult to handle and the quality of the product can decrease. A process-oriented view on software development is increasingly popular witnessed by current publications on the software development process and the application of modern quality management systems. The ISO 9000 norm describes a quality management system that provides a process oriented view on general production processes and is widely used in many industries. In this paper we suggest a systematic way of describing and developing software processes that contain certain desirable properties and fulfill quality management demands. Therefore, the design pattern approach known from object-oriented software engineering will be adapted to the modeling of business processes. Using this approach the requirements of the ISO 9000 norm can be translated to software development process patterns. These patterns can be used in modeling or reengineering the business processes of a software company in a way that the processes fulfill important properties of quality management systems and improve the overall quality of the resulting software product. Additionally, finding quality management patterns in software development processes can indicate the existence of a functioning quality management system to certification authorities and customers.

1 Introduction

The development of software is a business process comparable to business processes in other industries. The general wish to improve the quality of business processes can not only be found in the "old economy" service and producing industries, also software companies follow this trend by spending reasonable effort in advances in the software development process. A possible combination of this effort together with the advances in modern object-oriented software design and modeling can be found in [1].

Many software development companies use different languages to describe their business processes and workflows in order to improve their efficiency and quality. These languages have in common that they describe a set of activities connected by a control flow and information about artifacts and responsibilities. Yet they are lacking

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 62–73, 2003.

means to model higher-level structures and abstractions of concepts like, for example, quality management properties. A small example can illustrate the problem.

A business unit produces individually customized software units according to the customer's specification. As these software units will be delivered directly to a key account customer reaching high quality requirements is essential for the maintaining the sales. This makes it necessary to include quality control into the business process of producing these software units. This quality control usually consists of regular audits (verifying, validating, monitoring or testing activities according to predefined quality objectives [ISO 9001 p. 23]) combined with reporting to the quality management department to be able to review and develop the software development process as a whole. How can a software development process be designed to include all necessary properties and activities for this kind of quality control? How can these properties and activities be abstracted from the production process itself?

In object-oriented software systems could in contrast to that properties be inherited from a class or an interface `QualityControl` which defines all necessary methods. With the languages available to model business processes this is not possible. In this work we want to present a way to describe higher-level structures of processes that can be communicated and used in modeling or re-engineering business processes and will help to ensure that certain requirements are met by the business process. Furthermore it will give an idea how it can be validated that necessary properties for a functional quality management system based on ISO 9000 are present in the business processes.

Section 2 discusses business processes in general and ways to describe them. Section 3 gives a brief overview of quality management. In section 4 it will be shown how patterns can be applied to business processes. Section 5 formulates a template for the description of business process patterns. Finally, section 6 shows different ways of applying a pattern to a concrete business process and how important aspects of quality management are implemented into a business process by applying a quality management pattern using a short example. The paper ends with a conclusion and an outlook on future work.

2 Business Processes

Businesses create their value to the stakeholder in the value chain, which describes the transformation of goods or information in physical state, place or time performed by different organizational entities of the company [2]. The basic outline of all the processes involved in the value creation is called *business logic*. So business logic denotes the way the fundamental transformation of goods or service common to a certain industry is performed but it is also individual in the sense that every company is distinct from its competitors, which gives it a competitive advantage or disadvantage in the market [2].

Going more into detail the business logic and the value chain is formed by a number of different business processes for different fields of operation. A business process could describe operations like the production of a certain good, the procurement of raw materials, the acceptance of a customer's call to the hotline etc. It

consists of *activities* performed by human or automated actors. These activities are the basic building blocks of the process. In the business process and workflow management literature there exists an established way of handling the description of business processes and workflows by dividing it into a number of *aspects*. Aspects mean in this context different sets of properties that do not overlap (in other words: are orthogonal to each other) [3, 4].

The *functional aspect* describes what has to be done to perform an activity.

The *organizational aspect* describes who or what automated system performs an activity and gives information on the relationships between organizational entities.

The *informational aspect* describes what information is needed and produced by performing an activity denotes the structure and flow of data.

The *behavioral aspect* describes the order, in which activities are performed and the control structures that connect them to form a whole process.

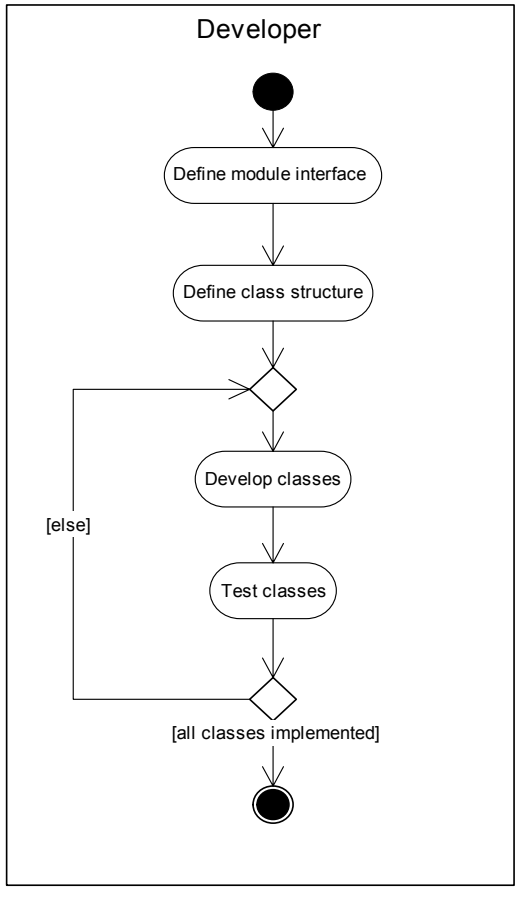


Fig. 1. An example of a software development business process

The understanding of a business process in this context is rather that of a business workflow with the difference that automated execution (e. g. on a workflow management system) is neither mandatory nor intended. Therefore a process consists of activities that are interconnected by a control structure. Activities are performed by organizational units including persons and machines in defined roles. Artifacts are objects that can flow between activities and be exchanged between organizational units. In this work we want to present business processes as UML activity diagrams [8, 9]. Figure 1 shows an example of a software development process. The next section gives a brief overview of quality management as a source for business process patterns.

3 Quality Management

The complexity of modern products like software systems increases continuously as well as the complexity of modern organizations and businesses. That makes it necessary to design business processes that contain structures and properties which allow the handling of complexity and possible errors resulting from it. Total Quality Management (TQM) systems provide a systematic and process oriented way to design business processes that reduce complexity and ensure quality throughout the whole production process.

The quality management movement emerged in the Japanese car manufacturer industry in the 1950ies. In order to gain market shares on the highly competitive American car market Just-In-Time production was introduced to reduce costs of extensive stock capacities and enable faster production cycles, therefore increasing the efficiency of the car production and allowing cheaper market prices. Zero-Error production was a prerequisite for Just-In-Time production, because suppliers and producers are tied very closely together and low stock capacities make them dependent on each other. Minor errors can therefore lead to halt the whole production process [Oess89].

In the following decades quality management has gained momentum and spread out into American and European companies, at first mainly in the producing industries. The American military and space technology issued in 1959 a set of rules for quality assessments called "Quality program requirements" as MIL-Q-9858. At the end of the 70ies the Technical Committee TC 176 "Quality Management and Quality Assurance" of the "International Organization for Standardization" was found on behalf of a German initiative. Many parts of the MIL-Q-9858 together with other developments lead to the creation of the ISO 9000 norm for quality management, which is now the most popular TQM system world-wide.

4 Applying Patterns to Business Processes

The requirements on business processes described in the quality management norm ISO 9000 are used in this work as the source for business process patterns. Generally patterns can be used for two different main purposes: forward and reverse engineering. The forward engineering approach means that a pattern will be used to construct a business process from the beginning whereas the reverse engineering approach wants to analyze an existing process in order to find or match patterns and properties already existing in a business workflow.

4.1 Patterns in Business Process Construction

Patterns describe solutions for problems in a context. Early works on design patterns have been made in different contexts like for example architecture [5]. The methodology was adopted in the software engineering world because it happens frequently in software design projects that similar design problems occur over and over again in different fashions. Design Patterns provide proven solutions to common problems. The book of Gamma, Helm, Johnson and Vlissides [6] is probably the best known collection of design patterns in software engineering.

The application of design patterns does not only provide solutions to frequent design problems but has further advantages. Most important is the fact, that applying patterns gives a well-defined structure to the constructed design. Patterns can be identified by their individual names and their functionality can be looked up in the pattern catalogue. If one developer wants to explain how a solution works he could say to the other something like "it works by the XXX pattern" and therefore complex structure become tractable. By that way design knowledge can easily be communicated.

Yet the application of patterns allows a great deal of flexibility: another main advantage of the pattern approach is that the structures of the solution provided by a number of patterns can usually be intermixed and combined. This makes it possible to handle different demands on a complex system simultaneously. Using patterns, a number of business logic and technical aspects can be integrated into the design at the same time without losing overall perspective. System independent design can be performed by restricting the design solely on business logic patterns and apply technical structures or patterns later in the design process.

Finally a pattern catalogue is a concise way of representing design knowledge and good practice so that building up pattern catalogues for business processes is a good way to formulate and communicate knowledge about business process design.

4.2 Patterns in Business Process Analysis

The quality management example in this work is also an example for the value of pattern recognition in existing processes. A company that desires to be certified ISO

9000 compliant needs to verify to the certification authority that the business processes fulfill certain properties defined in the norm. With respect to the aspect of quality control such properties can be for example that quality objectives are defined before the production process takes place, that the quality of the product has to be assessed according to the defined objectives afterwards and that a quality management department exists which is provided with the necessary information to develop the quality management system itself. All these aspects can be brought together in a pattern called "Quality control". Analyzing an existing business process and finding a quality management pattern in this process means that certain quality management properties are fulfilled and therefore pattern matching can be used to prove to a certification authority or customers that important parts of a quality management system are implemented. The next section describes a systematic way of writing down business process patterns.

5 A Pattern Description Language for Business Processes

As soon as the number of patterns used increases, a catalogue of patterns has to be formulated in a thorough and concise way to gain all advantages of the pattern approach. Applicable patterns should be easy to find in a pattern catalogue, to compare and to cope with. This is usually achieved by defining a pattern description template that contains the three basic elements problem, context and solution together with a number of other sections according to the application domain. Each pattern in the catalogue is described using that template and each section of the template contains different properties of the pattern.

Templates such as those used be Gamma et al. [6] and Meszaros [7] are specifically designed for the description of patterns in software engineering and cannot be directly applied to business processes. This makes it necessary as a first step to define a template that meets the special needs of the application domain. Concerning business processes and workflows as application domain, the above mentioned behavioral, organizational and informational aspects need to find their equivalent in the template. Figure 2 shows a proposal for a business process template.

The section "Name" is important to be able to communicate about a pattern and therefore making it part of the design knowledge. A short "Description" of the pattern gives a brief overview to be able to find an appropriate pattern quickly. The section "Application Context" has to be consulted in order to see if a chosen pattern is really applicable in the given situation. A specialty concerning patterns for business processes is the section "Organizational Context". A business process pattern usually requires the existence of certain organizational units that are responsible for performing activities. In this section all information about such organizational units is put together.

Name
The pattern name describes the most important property of the pattern in a few words.

Description
A short description of the pattern and the problem that is solved. This section should contain only a few words.

Application Context
describes the requirements for the application of the pattern.

Organizational Context
This section shows the assumptions about the organizational structure in which a pattern should be applied.

Problem
A detailed description of the problem including the forces that lead to the creation of the pattern and to its application. Optionally containing an example scenario.

Solution
A full description of the structure of the solution. The section could contain an activity diagram, a Petri net etc.

Result
describes the resulting context after the application of the pattern. This could be the fulfillment of requirements or invariants or the creation of certain data.

Reasons
for choosing the given solution.

Related Patterns
with similar content or that work together with the actual one.

Fig. 2. A business process pattern template

The sections □Problem□ and □Solution□ are self-explaining. In connection with business processes the section □Result□usually contains a list of documents or other data units that were created by the execution of the process and therefore pays attention to the informational aspect. The sections □Reasons□and □Related Patterns□ provide information that is useful for choosing a pattern for a given problem.

6 Example

To illustrate the description of patterns using the template defined in the preceding section and the concepts of modeling business processes with patterns we want to give an example for a business process pattern derived from the requirements of the ISO 9000 norm. A central part of a quality management system is quality control. This means that the result of a production process has to be compared to predefined

quality objectives by a supervisor. The next subsection is the description of the pattern "Quality Control" derived from the ISO 9000 norm.

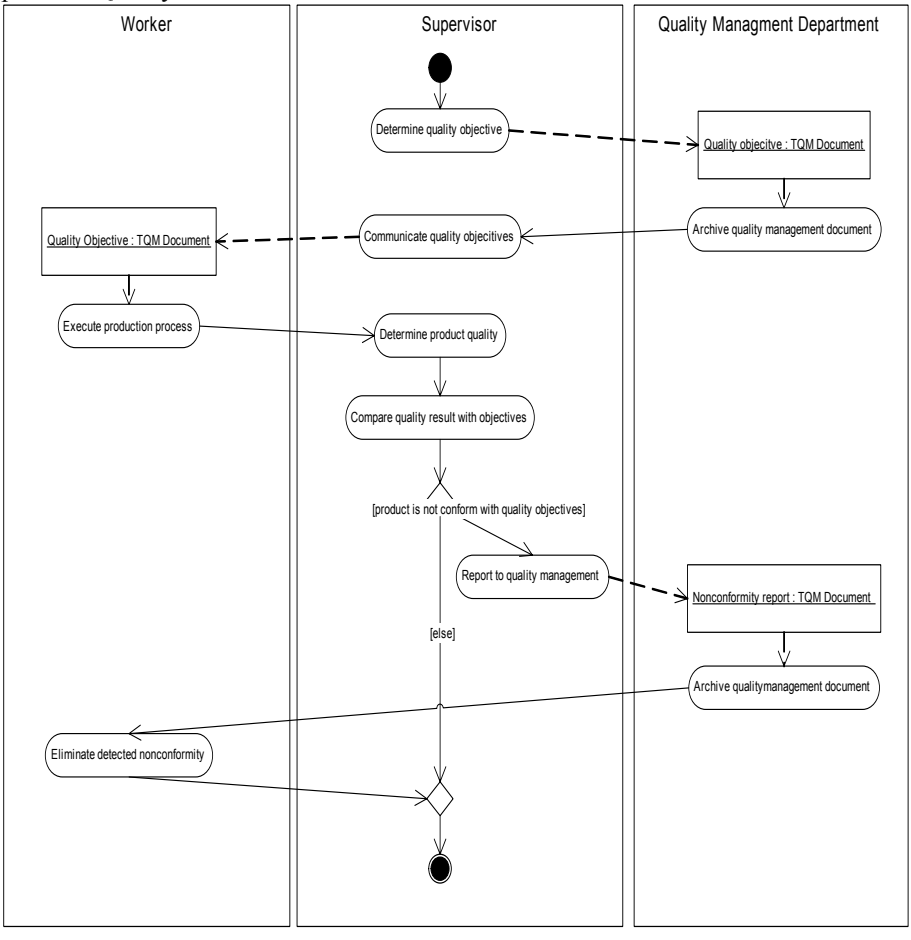


Fig. 3. Activity diagram as part of the section "Solution" of the example pattern: "Quality control"

6.1 Example Pattern

Name: Quality Control
Description: How can a constant high level of quality be reached?
Application Context: Products and services are the result of the execution of business processes.
Organizational Context: Products are produced by production workers. A supervisor controls the products and is responsible for the production workers. The organization

has a quality management department that is responsible for the quality management system as a whole.

Problem: The quality of the products and services shall be adequate to fulfill the needs of the customer. No product shall reach the customer that does not fulfill the quality requirements. Information shall be collected that allows the identification of repeated errors and their cause. Documents are created and archived that prove to the customer that the product quality has been tested and confirmed.

Solution: Quality tests are highly necessary for processes in which products for external customers are produced. The quality objectives have to be determined before the production process starts. The quality objectives have to be communicated to the production workers. Quality nonconformities have to be eliminated. All quality information has to be collected and sent to the quality management department for statistical analysis, systemic improvements etc.

Result: Quality is defined according to the customers needs. Products reaching the customer are quality tested. Information about quality objectives and test results are available at the quality management department.

Reasons: Regular defined quality test are the basic building block of a quality management system. Early-stage quality objective definition is often omitted or forgotten but necessary for a functional quality management system. The application of the pattern reminds the business process modeler of including all necessary activities for quality control. Further treatment of the quality management documents shall not be part of this pattern.

Figure 3 shows the solution provided by the pattern □Quality Control□ in an activity diagram.

6.2 Application of the Pattern

Now this pattern can be applied to construct a new process out of an existing process by adding quality control. Figure 1 shows an example business process describing an incremental software development process. Two different ways of applying the pattern to the business process shall be presented.

Figure 4 shows that the software production process from figure 1 is included into the pattern process replacing and therefore refining the activity □Execute production process□. The activities in the pattern are defined as neutrally as possible to make it applicable in different contexts. In this refinement process some activities are concretized according to the application domain as for example □Debug code and eliminate errors□.

Figure 5 shows a more sophisticated way of implementing the pattern into the production process. It exploits the fact that the production process contains some kind of quality control itself by the repeated testing of the implemented classes.

The process in figure 4 relates to the software production process as a black box and is therefore an example of simple refinement. In contrast to that the process in figure 5 shows the flexibility of the pattern approach by the fact that far more complex applications of a pattern are possible. This process shows a rather tight

connection between quality management and the developing process. Here the pattern is intermixed with the production process.

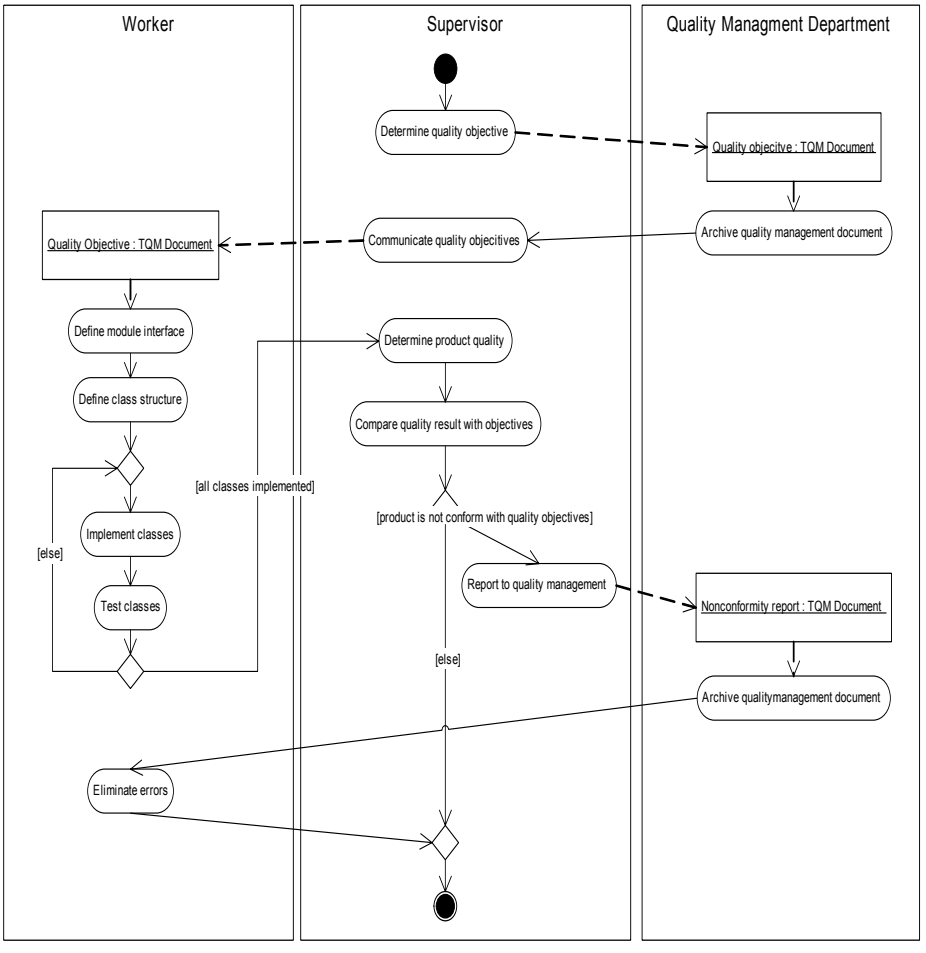


Fig. 4. Simple implementation of the pattern "Quality Control"

Both results of the business process modeling process as shown in figure 4 and figure 5 are possible ways of refinement and the application of a pattern in a technical sense. Both fulfill the quality management requirements as stated in the "Problem" section of the pattern. It is a question of managerial decision if a company prefers a rather loose or tight coupling of the quality management system into the development respectively production process. These two examples show how differently and flexibly process patterns can describe high-level structures and properties and how they can be included into concrete production processes in order to fulfill certain requirements, e. g. those of quality management.

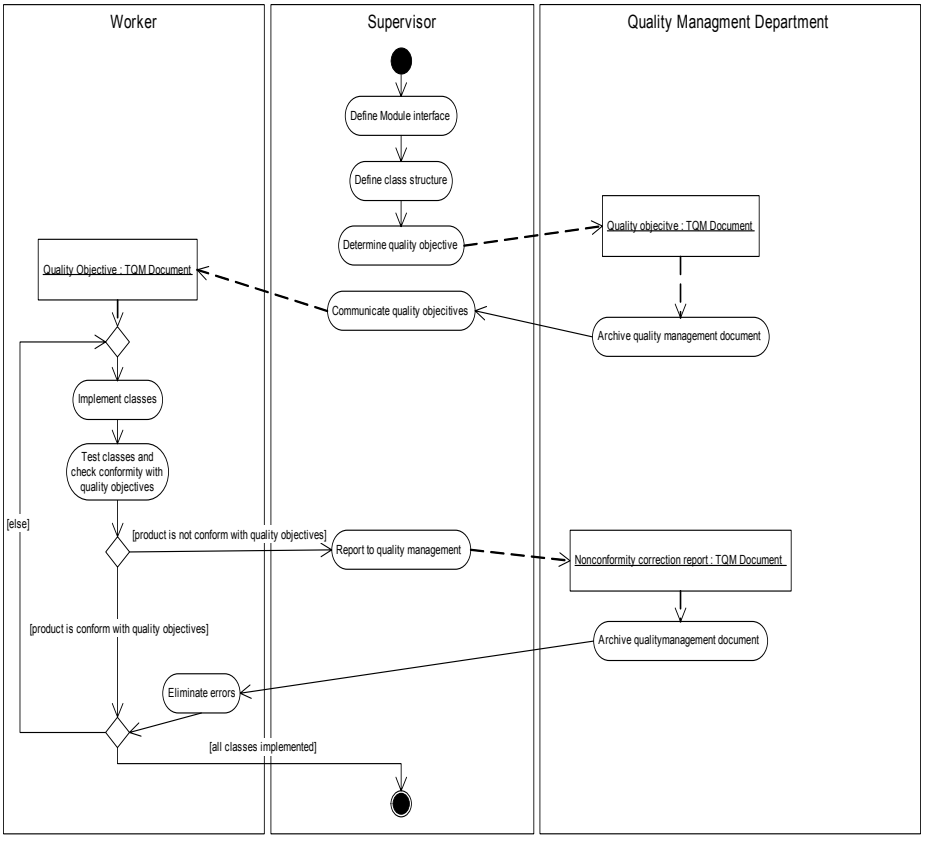


Fig. 5. Sophisticated implementation of the pattern □Quality control□

7 Conclusion

In this paper we could show that the pattern approach can be beneficial not only in object-oriented software design but also in business process design. The example of quality management patterns indicates that these benefits are not restricted to forward engineering of business processes but that the pattern approach also be helpful in analyzing properties of given processes. Future work in this area is intended to describe how pattern matching in a business process can be defined and how the existence of a pattern in a process can be validated on a more formal basis.

References

[1] I. Jacobson, G. Booch, J. Rumbaugh. *The Unified Process*. Addison-Wesley, Reading, Mass., 2001.
[2] M. E. Porter. Competitive Advantage creating and sustaining superior performance. Free Press, New York, 1998
[3] B. Curtis, M. I. Kellner, and J. Over. Process modeling. *Communications of the ACM*, 35(9), 1992.
[4] S. Jablonski, C. Bussler. *Workflow Management: Modeling Concepts, Architecture and Implementation*. International Thomson Computer Press, London, 1996.
[5] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel. *A Pattern Language*. Oxford University Press, New York, 1977.
[6] E. Gamma, R., R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1995.
[7] G. Meszaros. *A Pattern Language for Pattern Writing*, http://www.hillside.net/patterns/writing/patternwritingpaper.htm. November 2002
[8] M. Fowler, K. Scott. UML distilled a brief guide to the standard object modeling language. Addison-Wesley, Reading, Mass., 2000
[9] Object Management Group. *OMG Unified Modeling Language Specification*, Version 1.4, ftp://ftp.omg.org/pub/docs/formal/01-09.67.pdf. September 2001

A UML-Based Approach to Enhance Reuse within Process Technology*

Xavier Franch[1] and Josep M. Rib[2]

[1] Universitat Politècnica de Catalunya (UPC),
c/ Jordi Girona 1-3 (Campus Nord, C6) E-08034 Barcelona (Catalunya, Spain)
franch@lsi.upc.es

[2] Universitat de Lleida
C. Jaume II, 69 E-25001 Lleida (Catalunya, Spain)
josepma@eup.udl.es

Abstract. Process reuse (the ability to construct new processes by assembling already built ones) and process harvesting (the ability to build generic processes that may be further reused, from existing ones) are two crucial issues in process technology. Both of them involve the definition of a set of mechanisms, like abstraction, adaptation, composition, etc., which are appropriate to achieve their goals. In this article, we define a general framework to process reuse and harvesting that proposes a complete set of mechanisms to deal with both activities. This general framework is particularized to the context of the PROMENAE software process modelling language. A process reuse case study which composes various reuse mechanisms is presented in the context of PROMENADE.

1 Introduction

A crucial challenge in software process modelling is the ability to construct new software process models (SPMs) in a modular way by reusing already constructed ones. This requires, on the one hand, to identify the mechanisms that may be used in order to reuse existing SPMs and, on the other hand, to provide a notation whose expressiveness, flexibility and standardization make the model reuse task feasible.

The importance of modularity-reuse capabilities within a process model language (PML), both in the fields of workflow management and software process modelling has been broadly recognized in the literature [Kal96, AC96, Jacc96, Per96, Car97, Kru97, PTV97, JC00, RRN01, etc.]. These capabilities are intended mainly to provide mechanisms to *harvest* the relevant aspects of the constructed models and to *reuse* them in the construction of new ones. Some process reuse frameworks have been proposed which address this particular issue [JC00, RRN01]. Furthermore, there are various PMLs that cover this issue [INCO96, Bog95, Car97, Jacc96, AC96, ABC96, EHT97, etc.]. However, at least two lacks may be found in such proposals:

* This work has been partially supported by the spanish project TIC2001-2165, from the CICYT program.

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 74–93, 2003.

- The process reuse frameworks we have gone through do not propose a wide range of reuse mechanisms which cover most of the situations that may be encountered in the task of harvesting/reuse SPMs.
- Moreover, we are not aware of any PML that defines rigorously a wide range of reuse mechanisms (based on those suggested in some framework), along with a standard notation to represent them.

Both issues have a negative impact on the usability of current PMLs for modelling real cases. For example, most PMLs do not provide a rigorous definition for the combination of some SPMs to construct a new one in a modular way. Hence, it is not clear how a SPM could be described in such PMLs as a combination of smaller component models. Moreover, it is not clear either in which way the ambiguities and redundancies that would come up during the combination procedure would be resolved and simplified.

In this respect, the objectives of this article are the following:

- To provide a powerful and expressive reuse framework focused on the language mechanisms that are necessary in the context of process technology to achieve reuse. This leads to the identification of a set of reuse mechanisms. Notice that we only focus on modelling issues. Therefore, we do not consider other aspects (like model retrieval and configuration management) that are also relevant to reuse.
- To propose a particular definition of the identified mechanisms in the context of a specific PML. This definition should include the features of *expressiveness* (ability to model many situations) and *standardization* (the reuse mechanisms will be mapped into UML) and should support the identification and application of *process patterns* (i.e., common solutions to common problems that may be adapted and reused).

The context of the work we present in this article is PROMENADE, a PML in the field of SPM that has been defined with the aim of improving the features of *expressiveness, flexibility, standardization* and *reusability*. Details on the language may be found on [FR99, RF01, RF02, Rib02].

The rest of the paper is structured as follows: section 2 presents a framework for reuse which is oriented to provide a list of mechanisms that guarantee appropriate model reuse capabilities. Afterwards, section 3 defines some of such mechanisms in the context of PROMENADE and section 4 presents an example which illustrates how these mechanisms can be used and composed. Finally, section 5 presents the conclusions and compares our approach with other related work.

2 The Process Reuse Framework

We devote this section to the presentation of a framework which focuses specifically on the mechanisms that a PML should provide in order to enhance process model reuse.

2.1 A Mechanism-Centered Framework for Process Reuse

Traditionally, three levels of abstraction have been established in the process modelling literature for process models (PMs) (see, for instance, [DKW99]) yielding to three different types of models: the *template model* (an abstract model that offers a template solution to a problem), the *enactable model* (a process model whose enactment may start) and the *enacting model* (an instantiation of an enactable model). In the context of process reuse, we keep this notation just changing the term *template model* for *process pattern.* The term *process model* may refer to a model at any of the three levels. We remark that an enactable model may contain some non-implemented tasks to be refined during enactment, using some delayed binding mechanism, as introduced below.

More precisely, the notion of *process reuse* refers to a pair of general and complementary activities which are involved in it:

- *Harvesting*: the process of transforming one or more PMs (enactable models or process patterns) into one process pattern so that it can be reused afterwards.
- *Reuse*: the process of transforming one or more PMs (enactable models or process patterns previously harvested or constructed ad-hoc) into one enactable model.

The activities of *harvesting* and *reuse* may be carried out using the following types of mechanisms:

- *Abstraction mechanisms*. To get rid of some specific or undesired details of a PM. Abstraction mechanisms come up in harvesting activities.
- *Adaptation mechanisms*. To make a PM (usually a process pattern) either more specific or suitable to be reused in a particular process.
- *Composition mechanisms*. To combine a set of PMs in order to create a new one. Composition mechanisms are used both in reuse and harvesting activities.
- *Delayed binding mechanisms*. To delay until enactment time the selection of a specific part of a SPM.

These mechanisms may be grouped along two different harvesting and reuse strategies:

- *Bottom-up*. Construction of PMs using either composition or generalization mechanisms. Using a bottom-up strategy we may obtain a more general model from another specific one (useful for harvesting) or a more complex model from its parts (useful for harvesting, if applied to process patterns and for reuse, if applied to enactable models).

- *Top-down*. Construction of PMs using adaptation mechanisms. Using a top-down strategy we may obtain a more specific model from another general one (useful for reuse). We do not use the top-down strategy for harvesting.

In both cases, the resulting PM is in the template or the enactable level. Both strategies may be combined in a reuse activity. In the same way, the two forms of the bottom-up strategy may be combined in a harvesting activity. A whole reuse process combining the activities of reuse and harvesting may itself be modelled as a sequence of reuse mechanisms application, following any valid path in the framework shown in fig. 1. Section 4 provides an example of this idea. Notice that we have defined types of mechanisms to move along all the feasible directions of the diagram, therefore, this is a complete reuse framework with respect to the reuse mechanisms.

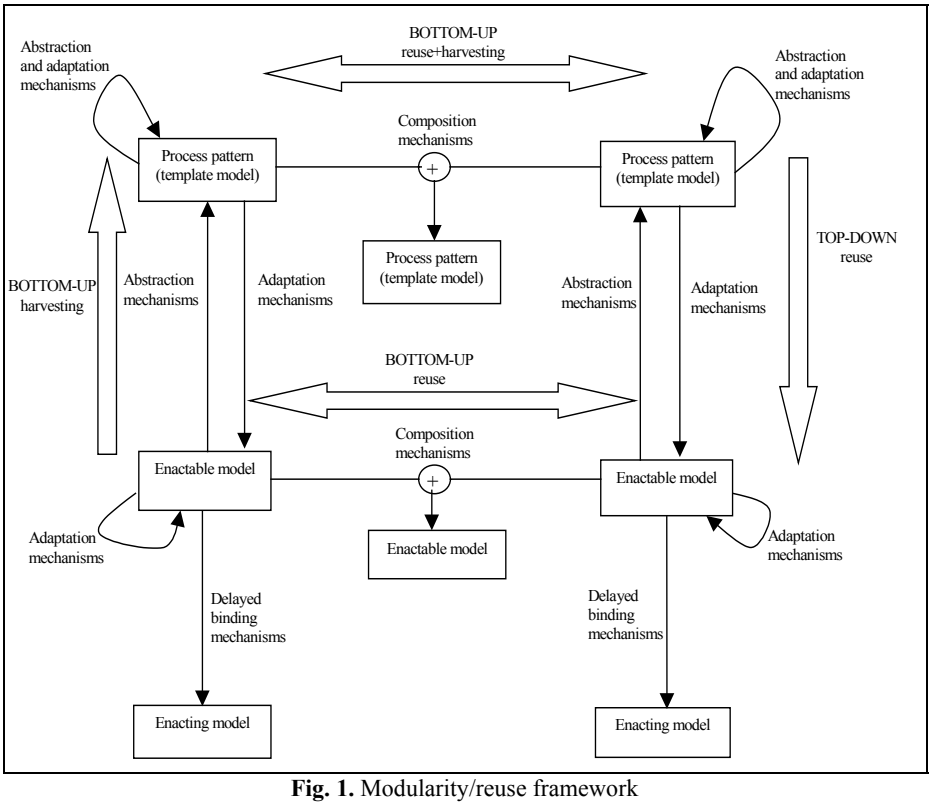


Fig. 1. Modularity/reuse framework

2.2 Harvesting and Reuse Mechanisms

In this section we detail and decompose the reuse types of mechanisms presented above. Notice that these types are not designed *ad-hoc* for a specific environment; thus, they may be applied in different approaches to process reuse. The application of

these mechanisms in the context of a specific PML (PROMENADE) is shown in sections 3 and 4. See Table 1 for a summary.

Abstraction mechanisms. Two abstraction mechanisms are considered, which correspond to the two ways that are normally used in order to abstract details from a complex model:

1. *Generalization*. By means of this mechanism, some elements of a SPM are abstracted in such a way that the components, properties and structure of the resulting SPM (*generalized* model) are true for the former one (*specialized* model). We also say that the generalized model may be substituted for the specialized one and also that the specialized model conforms with the generalized one. This generalization is usually achieved by the consistent removal of some elements from the initial SPM.

Table 1. Harvesting and reuse mechanisms

Category	Subcategory	Mechanism
Abstraction		Generalization
		Parameterization
Adaptation	Addition	Specialization
		Instantiation
	Removal	Projection
	Substitution	Renaming
		Semantic substitution
Composition	Shallow composition	Grouping
	Deep composition	Combination
	Inclusion	Inclusion
Delayed binding		Refinement
		Inclusion

2. *Parameterization.* By means of this mechanism, the abstracted elements of a SPM are encapsulated by means of parameters**.**

Adaptation mechanisms. Adaptation mechanisms involve some sort of model modification. The following modifications can be applied to a SPM:

1. *Addition of new elements to the SPM*. This makes the SPM more specific and leads to two different mechanisms:
   - *Specialization*: Consistent addition of some elements to the SPM in such a way that the resulting SPM (the *specialized* model) may substitute the former one and add more details to it.
   - *Instantiation: Tailoring* a parameterized SPM to a specific context by determining its parameters).

   Notice that both mechanisms (specialization and instantiation) are the opposite to the ones presented before.

2. *Removal of unnecessary elements from the SPM*. We use the *projection* mechanism for this purpose. This mechanism restricts the model to the selected elements and their context (i.e., it obtains a *view* of it). Notice that a generalization may be considered a removal of elements with abstraction purposes while a projection is a removal of elements with adaptation purposes.
3. *Substitution of elements of the SPM*. This substitution may be performed at two different levels (lexical and semantic) leading to two mechanisms: *renaming mechanism* (lexical substitution) and *semantic* substitution *mechanism* (substitution of some model elements for other different ones which model similar concepts).

Composition mechanisms. We distinguish three ways to carry out a composition of a set of PMs:

1. *Shallow composition.* This is a grouping of a set of PMs without adding any additional specific semantics (i.e., the behaviour of the resulting model is the sum of the behaviours of the components). Furthermore, the namespaces of the component PMs are kept separated in the resulting one (i.e., no merging of namespaces). This composition is performed by the *grouping* mechanism.

2. *Deep composition.* This is a composition of a set of PMs by adding some additional behaviour to the resulting model (in PROMENADE, this is done by defining several precedence relationships between the main tasks of the components). The namespaces of the component PMs are merged into a single one. This is performed by the *combination* mechanism.

3. *Inclusion.* In this kind of composition, the functionality of an entire PM is incorporated as a subtask into the behaviour of a composite task (which belongs to another PM), possibly with some additional behaviour. This is performed by the *inclusion* mechanism.

Delayed binding mechanisms. By delayed binding we mean the possibility to enact models that are not completely described (i.e., the implementation of some activities will be decided during enactment; [Per96] uses the terms *primitivation* and *stratification* to refer to this). These models would be completed at enactment time using the *refinement* mechanism (i.e., a non refined task is substituted for a specific refinement of that task that has been included in the model at modelling time) and/or the *inclusion* one (i.e., a non-refined task *t* is substituted for an already constructed model which is included into the enacting one in the place of *t*).

3 Mechanism Definition in PROMENADE

PROMENADE [FR99, RF01, RF02, Rib02] is a second generation PML that promotes standardization by providing a two-tiered extension of the UML metamodel (both supplying an explicit UML metamodel extension and defining a UML-profile

[Rib02]). Expressiveness in model construction is assured by providing both proactive and reactive control: proactive control is defined by means of different kinds of (predefined and user-defined) precedence relationships, while reactive control is based on ECA-rules. In this paper we focus on the PROMENADE constructs that address process model reuse.

Specifically, an expressive definition of the parameterization/instantiation mechanisms has been provided in PROMENADE in the context of *process patterns* (in fact, process patterns have been defined as parameterized SPMs). The rest of mechanisms presented above have been defined as PROMENADE operators. Thus, the term *operator* in this article refers to the specific definition of a reuse mechanism within the context of PROMENADE. These operators may be applied both to enactable models or to process patterns, which enhances the PROMENADE reuse approach expressiveness and power.

We do not provide in this article the specific PROMENADE definition for all the above-mentioned mechanisms. This would require a thorough presentation of the language, which is beyond the scope of this paper (see [Rib02]). For the sake of an example, we outline the most relevant aspects concerning the *combination* operator (section 3.1) and also *process patterns* (section 3.2). Afterwards, we develop in detail an example in which we try to provide a general view of the whole reuse framework, namely the modelling of a Catalysis pattern for software development.

All the PROMENADE behavioural and reuse constructs have been mapped into UML (hence, all the diagrams used to describe the behaviour or reuse of a PROMENADE model are standard UML). Wherever an extension to the UML metamodel has been necessary, we have used the UML built-in extension mechanisms (i.e., stereotypes, tagged values and constraints).

3.1 The Combination Operator

This operator combines some SPMs by adding several precedence relationships among their main task classes. The namespaces of the component SPMs are merged into a single one.

More specifically, a SPM m is obtained from the combination of a set of models $\{m_1,...,m_n\}$ and a set of precedences $\{a_1,...,a_m\}$ in the following way:

- The static part of m is the superposition of the generalisation hierarchies of $m_1,...,m_n$, together with the union of their association, aggregation and dependency relationships.
- The dynamic part of m is built by combining the maintasks of each model $m_1,...,m_n$ with the precedences $\{a_1,...,a_m\}$. In this way, the main task of m is constructed by using as its parameters the parameters given to the combination operator; as its subtask classes, the main task classes of the component models; and as its proactive behaviour (i.e., precedence relationships) the precedence relationships stated in the combination operator.

The application of the combination operator may lead to conflicts with the names unicity property (i.e., name inconsistencies), redundancies and residues.

- *Consistency problems:* Two elements that belong to different models have the same identifier but they are not identical (i.e., their definitions differ).
- *Redundancy problems:* Two elements in the same model that represent the same concept but have different identifiers.
- *Residue problems:* An element that takes part in a composite model since it belonged to one of its component models but which is not required in the composite one.

Coping with inconsistencies

Inconsistencies are clearly the most important problem that we face in the application of the combination operator. If two elements of the same category that belong to two SPMs that are to be combined have the same identifier and different definition an inconsistency in the combined SPM will be generated. Two models without inconsistencies are said to be *compatible.*

The combination operator may be applied modulo a renaming in order to obtain compatibility for the component models. For this reason, a list of lexical substitutions is given as a parameter of the combination operator. These substitutions are carried out previously to the process of combination itself (i.e., previosuly to the merging of namespaces).

However, this lexical substitution list could be empty or incomplete (i.e., some inconsistencies could remain in component models). If the combination operator detects some inconsistencies, the combination cannot be completed.

The process of getting rid of the inconsistencies achieved by means of a renaming is called *compatibilization* and is performed by the combination operator before the combination itself.

Coping with redundancies

The compatibilization process enforces that the set of models that are to be combined are compatible but does not ensure the absence of other *combination* problems in the composite model (the model resulting from the combination). These problems are *redundancies* and *residuals.* We deal with both issues when the combination has been performed and a combined resulting model has been obtained. We remove redundant and residual elements in that model by means of a *simplification* process.

As we have said, an element of a combined model is redundant if it models the same concept as another element already in the model (we call them *corresponding elements* or we say that one is the *counterpart* of the other). Both elements may be equivalent or not (they may be just similar). In any case, since they belong to the same model, they will have distinct names. The first part of the simplification process consists in removing all redundant elements.

Consider a pair of corresponding elements. Two different cases should be distinguished:

- Both corresponding elements are equivalent. This case may be addressed by a lexical substitution. Therefore, the list of renamings that are necessary to make both elements identical can be included into the lexical substitution list that we have presented in the previous section.
- Both corresponding elements are not equivalent. In this case, we will carry out a simplification process which consists in the suppresion of one of them. This suppression is not trivial since there may exist some other model elements which are not redundant but that refer to the redundant elements. Those references must be substituted for references to the counterpart of the redundant one (i.e., to the one that will remain in the model). Therefore, prior to the elimination of the redundant element, a substitution process must be carried out.

  The combination operator has a parameter that provides a *semantic substitution list*. This is a list of pairs. Each pair contains a redundant element and its counterpart. The combination operator substitutes all references to the redundant elements throughout the model for their counterparts and finally, eliminates the redundant elements.

Coping with residuals

Residuals are elements that come up in the combined model (since they belonged to some component model) and that are not wanted anymore. The second part of the simplification process consists in removing those elements and any reference to them in the combined model. Similarly to the previous case, the combination operator will have a list of residual elements.

Summing up, the application of the combination operator is depicted in figure 2 and consists in:

- Making the component models compatible by means of a lexical substitution process (compatibilization).
- Combining the component models to get a composite one with a specific behaviour for its main task class, which is given by a set of precedence relationships established between the component SPMs.
- Simplifying the resulting composite model by getting rid of redundant and residual elements.

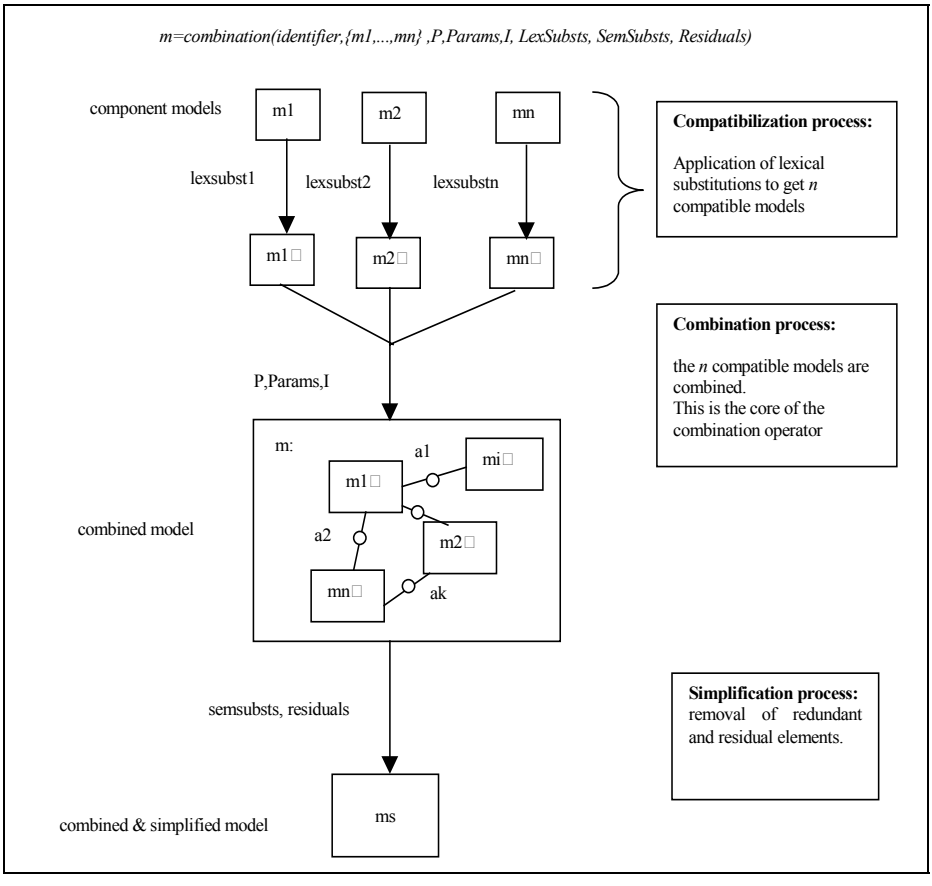


Fig. 2. The whole combination operator

In the PROMENADE metamodel, the relationship between a model *m* that has been obtained from a combination of models $m_1,...,m_n$ and its component models $m_1,..., m_n$ is established by the association *model-combination*. This association induces, by definition, a dependency stereotyped <<*combination*>> from the composite SPM to the component ones. Section 4 contains some examples of graphical representation of this operator. We remark that in this section we have just outlined the meaning of the combination operator, for brevity purposes. [Rib02] presents a formal definition of this and the rest of operators in PROMENADE.

3.2 Process Patterns in PROMENADE

Parameterization/instantiation reuse mechanisms are defined in PROMENADE by means of process patterns. These process patterns may be combined with other SPM

or process patterns to construct a bigger SPM or process pattern, respectively. Process patterns are integrated into the PROMENADE metamodel, which has been defined as an extension of the UML metamodel.

The PROMENADE approach to process patterns can be characterized by the following key features (see [Rib02] for a complete description):

- A definition of process patterns integrated into the PROMENADE metamodel is given (see fig. 3). In its turn, the PROMENADE metamodel has been defined as an extension of the UML metamodel.

  In this respect, a process pattern is defined as a subclass of a SPM. In particular, a process pattern in PROMENADE is defined as a parameterized SPM.

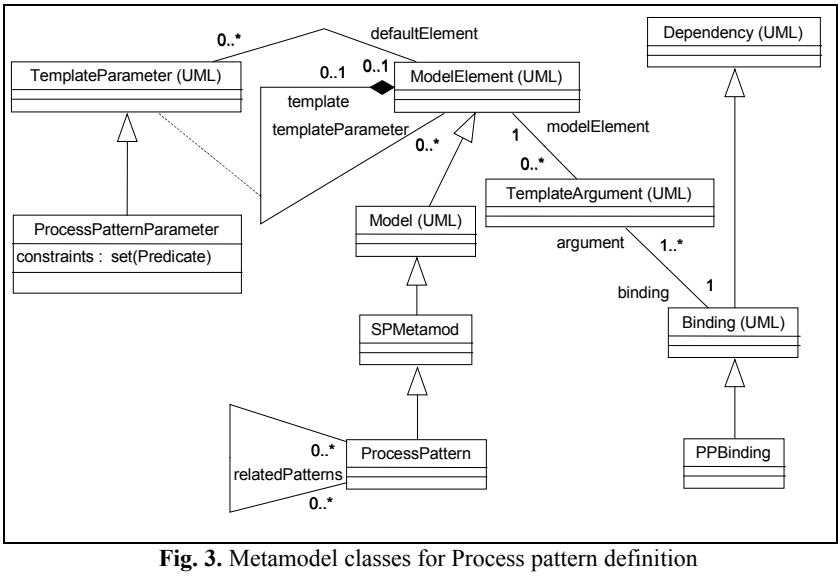


Fig. 3. Metamodel classes for Process pattern definition

- Expressive and flexible parameterization.

  Some parameters may encapsulate some structural or behavioural aspects of a process pattern. Any *Task, Document, Tool, Role, Agent, SPM* or *Class* may be a process pattern parameter. Therefore, process patterns allow the definition of generic SPMs which may be reused in different situations by means of a specific instantiation of its parameters (through the adaptation mechanisms). PROMENADE allows the definition of OCL constraints on process patterns parameters and provides a specific definition of the correctness of a parameter binding. Two metaclasses (*ProcessPatternParameter* and *ProcessPatternBinding*) have been incorporated to the PROMENADE metamodel in order to deal with both issues.

  An example of such constraints on the parameters may be the following: □the document class that instantiates the parameter *P* of the pattern must have as subdocuments the document classes *A* and *B"*.

- Since in the PROMENADE metamodel a process pattern is, in particular, a SPM (see fig. 3), any PROMENADE modularity/reuse operator may be applied to process patterns.
- Specific process patterns features (*name, author, intent, initial context, result context, applicability*, etc.) have been defined for PROMENADE process patterns in its metamodel.
- The SPM that constitutes a process pattern may be described with rigour and precision using the features of PROMENADE (i.e., precedence relationships, communications and ECA-rules may be used to describe the behaviour of a process pattern.).

Notice that the rigour of a pattern description is concerned with the language elements used to generate that description, not with the granularity of the description. In particular, a loosely defined process pattern may be rigorously described by means of precedences between abstract tasks, that may be pattern parameters which need not be precisely stated.

4 A Reuse Example

Catalysis [DC99] is a collection of software development processes which are described in a modular way by the application of several patterns. These patterns share a common structure (i.e., the static elements used by them, like *Spec, Collaboration, Design*, are supplied by Catalysis). They are described informally in natural language and may be used in the description of one or more Catalysis processes.

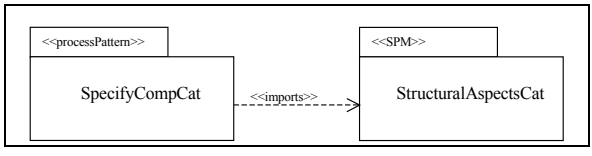
One of the Catalysis patterns which is used more recurrently in different processes is the one intended to specify a component (pattern 15.7 in [DC99]; let us call it *SpecifyCompCat*). This pattern relies on the notions of *Collaboration* and *SpecType* in order to construct the specification for a component referred to a specific role. The strategy followed by this pattern is the following: (1) definition of the static model, which provides the vocabulary to specify the operations; (2) specification of the operations; (3) assignment of responsibilities.

Figure 4 shows a definition of this pattern in PROMENADE. It contains no parameters since it is not intended to be used universally but only within the Catalysis processes (i.e., the structural elements, including *Collaboration* and *SpecType* are defined in the Catalysis context and imported by all the Catalysis patterns). Notice that PROMENADE allows a rigurous definition of the pattern. In addition to the usual textual features included in most pattern languages (*author, keywords, intent, etc.* [Amb98]), other aspects described with a more precise (and standard) formalism are included.

In particular, the pattern structural elements are presented by means of a UML class diagram (this diagram is encapsulated in *StructuralAspectsCat* and it is shared by all the Catalysis processes); its behaviour is described by means of a set of precedence relationships between the component activities (which may be decomposed in other subactivities). These relationships establish some temporal precedences between activities; include some parameter binding to link the documents involved in them;

and are depicted by means of stereotyped UML dependencies. The details of the semantics are not necessary in the rest of the paper (see [Rib02] for a complete description). The behavioural description of this pattern is skipped for the sake of brevity; however, an example of process behaviour description is shown in figure 5.

Catalysis does not support the statement of software quality attributes (i.e., non-functional requirements [ISO99], in short NFR) in a component specification. As other approaches do [CNM99, CL99], it can be considered useful to incorporate such NFRs in that specification. We have done this in [RF01].



Name: SpecifyCompCat
Author: JMR & XFG
Keywords: ***software component, specification, Catalysis.***
Intent: To incorporate into a specification document (SpecType) the functional specification of a software component according to a specific role.
Pattern parameters: No parameters defined.
Pattern structure: Imported from *StructuralAspectsCat*
Initial context: A component has to be designed and implemented. The participants should be available and ready to start the specification task.
Input parameters:
- A collaboration document correspond-
- The specification document (partially constructed) (*spec: SpecType*).

Result context: The specification document of the component has been built.
Output parameters:
- A specification document that contains the component specification corresponding to a specific role. (*spec:SpecType*).

Applicability: No additional constraints have been defined.
Participants: One or more members of the specifier team
Structure: The structure shared by all Catalysis processes (imported from *StructuralAspectsCat*). Not shown
Process: Not shown.
Related patterns: ComponentSpecification

Fig. 4. Definition of the *SpecifyCompCat* pattern

SpecifyCompWithNF is a specific SPM which states that a component specification should consist of a functional and a non-functional specifications and also a validation of the specification. This SPM proposes the strategy consisting in performing the functional specification first. It also applies a concrete strategy for specifying the functional and the non functional requirements (whose details are encapsulated in the subtasks *FSpecifyComp* and *NFSpecifyComp*). The behavioural description of *SpecifyCompWithNF* SPM is depicted in figure 5.

In the current situation, it is interesting to incorporate NFRs into the *SpecifyCompCat* Catalysis process pattern by reusing *SpecifyCompWithNF*. Not all the steps that we will follow are strictly necessary but, in this way, we will illustrate how various reuse operators may be composed along the reuse framework (see figure 1). The proposed operator composition process is shown in figure 6. Notice that the diagram depicted in this figure is standard UML. The relationships between the various models are represented by means of UML dependencies. The sense of the arrows has been selected according to the UML semantics. The UML metamodel has been extended

with several stereotypes and tagged values using the standard UML extension mechanisms (see [UML01]). The steps followed in this reuse process are detailed next.

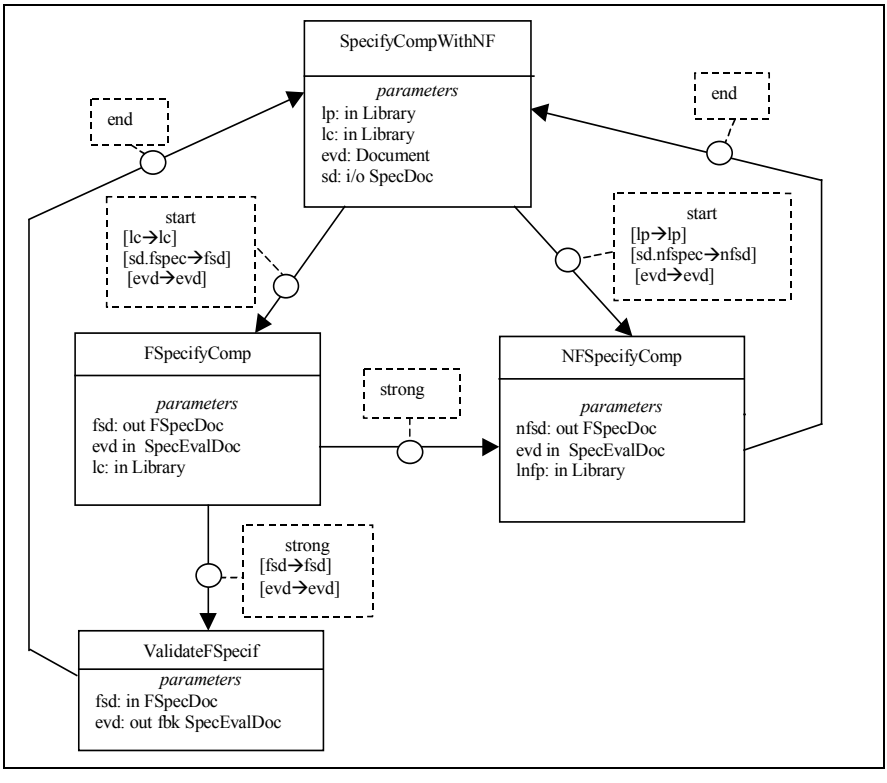


Fig. 5. Behaviour description of the model *SpecifyCompWithNF*.

Step 1. Abstraction of *SpecifyCompWithNF→SpecifyCompWithNFPattern*

First of all, we will build the process pattern *SpecifyCompWithNFPattern* which abstracts the particular specification strategy so that different functional and non-functional methodologies can be used in each particular situation. This abstraction process is carried out by means of a *parameterization*. The result is shown in figure 7. By the parameterization of the documents and tasks that are necessary to perform this pattern, many different specific strategies to carry out a specification may be used to instantiate it (e.g., by diagrams, by templates, purely textual, formal □model-oriented, equational, etc..). To preserve consistency, some requirements are established on these parameters (e.g., the document type that instantiates the parameter specification document *SpecDoc* must have as subdocuments, a functional specification and a non-functional specification documents). These requirements have been expressed in OCL. Notice again that the model for the process pattern is not described just textually but in a rigorous manner using the PROMENADE PML constructs, which have been mapped into UML. In particular, notice that PROMENADE allows the rigorous

definition of parameters by means of their types and of constraints for the parameter instantiation. Notice also that other patterns which propose a different component specification strategy or which do not perform a non-functional specification may coexist with the proposed one.

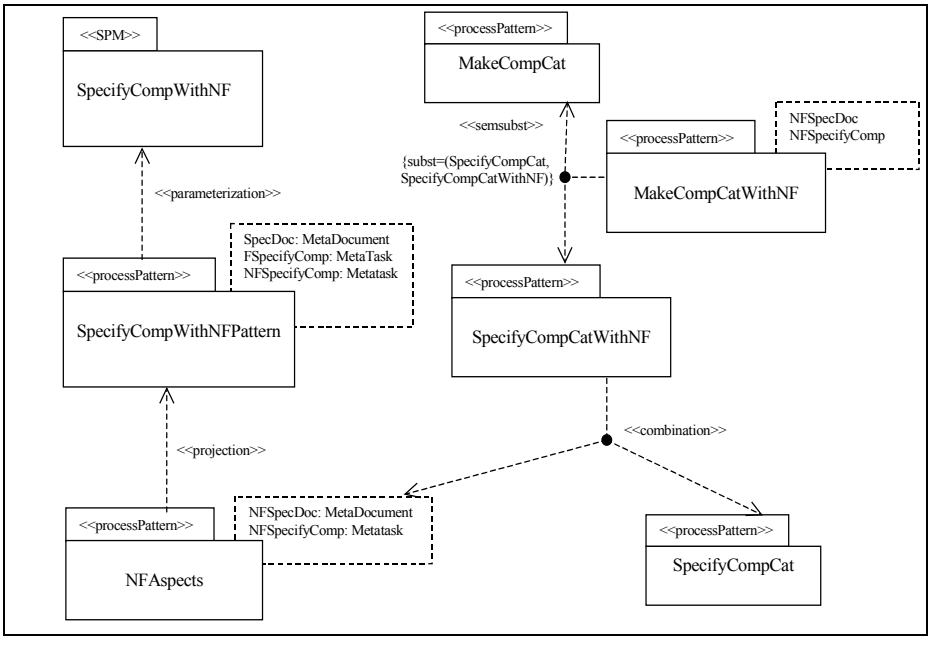


Fig. 6. Reuse example. Operator composition process

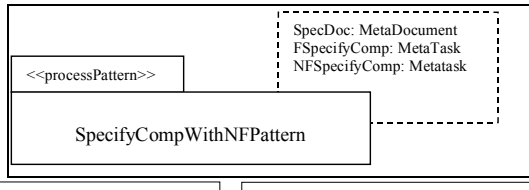
Step 2. Adaptation of *SpecifyCompWithNFPattern→NFAspects*

We do not want to incorporate the whole functionality of *SpecifyCompWithNFPattern* into *SpecifyCompCat*; only its non-functional aspects. For this reason we will adapt this pattern. The idea is to perform a projection of *SpecifyCompWithNFPattern* onto its non-functional aspects. Therefore, we apply the *projection* operator, once the elements on which the projection is carried out have been identified.

The projection is carried out on the elements that provide the non-functional component specification, namely: *NFSpecDoc*, as document class; *NFSpecifyComp*, as task class; and *Library*, as other cathegory of classes. Therefore, the following operator usage must be performed: *NFAspects* = projection(*SpecifyCompWithNFPattern*, {*NFSpecDoc, NFSpecifyComp, Library*}).

The resulting model (*NFAspects*) will keep (by definition of *projection* in PROMENADE) the context of the classes on which the projection has been performed. As a result, other classes like *FSpecDoc* and *SpecEvalDoc* will belong to *NFAspects* (since they are parameters of the task class *NFSpecifyComp*, hence, part of its context). Something similar will happen with *ValidateFSpec* (since this task class have *FSpecDoc* as parameter).

Step 3. Combination of *SpecifyCompCat* with *NFAspects* →*SpecifyCompCatWithNF*
This is the central step of the reuse process we are following. The specification of the NF aspects will be combined with the Catalysis pattern that models the specification of a software component. The new pattern (*SpecifyCompCatWithNF*) will establish some additional behaviour: the functional specification should be made before the non-functional one.



Name: SpecifyCompWithNFPattern
Author: JMR & XFG
Keywords: ***software component, specification, software attributes, component, non funcionality.***
Intent: To provide a document (SpecDoc) which contains the functional and non-funcitonal specification of a software component; or to review a previous specification.
Pattern parameters:
The specification document type (*SpecDoc*).
A task to perform the functional specification (*FspecifyComp*)
A task to perform the non functional specification (*NFspecifyComp*)
Initial context: A component has to be designed and implemented. The participants should be available and ready to start the specification activity.
Input parameters:
A library of components (*lc: Library*)
A library of non-functional properties (*lp: Library*) (optional)

An evaluation document (if it is a specification revision) (*evd: Document*) (optional)
The specification document to be reviewed (if it is a specification revision) (*sd: SpecDoc*).
Result context: The specification document of the component has been built.
Output parameters:
•The specification document completed (*sd: SpecDoc*).
Applicability: Some constraints have been defined on the pattern parameters:
(1) SpecDoc.allSupertypes->includes(Document)
(2) Specdoc.subdocs->includes(FSpecDoc)
(3) Specdoc.subdocs->includes(NFSpecDoc)
(4) FSpecifyComp.allSupertypes->includes(Task)
(5)FspecifyCom.params->
exists(p|p.type=FSpecDoc)
(6) NFSpecifyComp.allSupertypes->includes(Task)
(7)NFspecifyCom.params->
exists(p|p.type=NFSpecDoc)
Participants: One or more members of the specifier team
Structure: (not shown)
Process: (not shown)

Fig. 7. The *SpecifyCompWithNFPattern* process pattern

Some considerations must be made prior to the application of the combination operator:

(1) Study of compatibility. Since there are no name conflicts between models, it is possible to put them together. If this were not the case, some lexical substitutions should be made.

(2) Study of redundancies. We look for semantical equivalences in the involved models. We find that two types of documents with different names are equivalent (from the new model point of view), because they both refer to the functional specification of a component. We build thus a set of pairs (in this case, just one) that will be passed as a parameter of the combination operator. Therefore, we have *semsubsts*={(*FSpecDoc, SpecType*)}. Instances of (our) *FSpecDoc* class will be substituted by equivalent ones from the (Catalysis) *SpecType*

(3) Identification of residual classes. We look for elements that become unnecessary in the new model, in order to remove them. We identify as residual classes all those coming from *NFAspects* that are not related to functional capabilities

and that are not redundant classes either. In this way we have *residuals*={*SpecEvalDoc, ValidateFSpec, Library*}.

(4) Statement of precedence relationships between the main tasks of the component models. Both models become coordinated from the behaviourial point of view. We choose to apply the non-functional specification right after the functional one. Therefore, we provide the following precedence list: *precs*={(*strong*, [*SpecifyCompCat*], [*NFAspects*])}. This list contains just one precedence relationship of type *strong* which ensures that the non-functional specification of a component will start only after the successful end of the functional one.

(5) We will choose the parameters of the composite model to be the same as those of the *SpecifyCompCat* with the addition of the non-functional specification document. *params*={(*col, in, Collaboration*), (*sp, out, SpecType*), (*nfsp, out, NFSpecDoc*)}

Suming up, the operator is applied as: *combination(SpecifyCompCatWithNF, {SpecifyCompCat, NFAspects}, precs, params, λ, semsubsts, residuals)*.

Step 4. Composition of *SpecifyCompCatWithNF* within *MakeCompCat*

The new pattern *SpecifyCompCatWithNF* may be used within other Catalysis processes. If these processes have not been constructed yet, *SpecifyCompCatWithNF* will be incorporated in the moment of their construction by means of the *inclusion* operator. Otherwise, the *combination* operator may be used. For instance, the main task of *MakeCompCat* (an already constructed model to build a component using the Catalysis methodology) includes the model *SpecifyCompCat* as a part of its functionality. This model may be substituted for the pattern *SpecifyCompCatWithNF* by the application of the combination operator, which would include the pair (*SpecifyCompCat, SpecifyCompCatWithNF*) in the *semsubst* list.

5 Conclusions and Related Work

We have presented a general and expressive process reuse framework focused on the mechanisms that are necessary in order to achieve harvesting and reuse of processes. In this respect, we have defined a wide range of reuse mechanisms and we have adapted these mechanisms to a specific PML called PROMENADE. This PML particularizes virtually all the mechanisms defined in the framework and provides a standard UML representation for them. Finally, we have outlined a reuse example consisting in the incorporation of non-functional requirements into the specification of a software component (a specification obtained using the Catalysis methodology). The example has been presented in PROMENADE. Some aspects of our approach have been issued in it: the joint application of reuse operators following a path along the framework depicted in fig. 1; the expressiveness of those operators and the standard UML representation of the relationships between the models involved in the reuse process. Our reuse methodology has been applied to the modelling of the Catalysis software development process [DC99] (see [Rib02], for a complete description).

[JC00, RRN01, Per96] present other reuse frameworks. Essentially, these frameworks identify a reuse life-cycle together with the requirements for a PML in order to supply reuse capabilities (therefore they are not restricted to mechanisms). [JC00] and [RRN01], the most complete ones, identify *generalization*, *inheritance*, *composition*, *projection* and *parameterization* to be the required reuse and harvesting mechanisms that a PML should provide. However, they just present them as general mechanisms. They do not provide particular definitions of them in the context of a PML. Notice that we introduce a systematic enumeration of mechanisms and we identify some new ones (i.e., *inclusion, renaming, semantic substitution,* different types of *composition*, etc.). We also propose an expressive parameterization mechanism (including the definition of constraints concerning parameters) and we propose specific definition of each mechanism in the context of PROMENADE (this is done in detail in [Rib02]). Furthermore, we provide a more general definition of *harvesting* and *reuse*: we do not restrict the harvesting (reuse) notion to the generation of a more abstract (specific) model from a more specific (abstract) one; a composition of several models at the same abstraction level may also be a harvesting/reuse activity.

In SPM, our field of study, modularity-reuse abilities provided by PMLs are scarce. E^3 [Jacc96], OPSIS [AC96], PYNODE [ABC96] and [EHT97] provide limited reuse capabilities (mostly based on views and generalization/inheritance in the case of E^3). They do not offer expressive ways to combine already constructed models (e.g., building the behaviour of the composite model as a customizable combination of the behaviours of the components). They do not offer process pattern support either.

In the related area of workflow management, many approaches use just the *cut-and-paste* strategy to deal with the topic of reuse [Kru97]. However, there exist some PMLs, like In-Concert [INCO96], OBLIGATIONS [Bog95], APM [Car97, Kru97], MOBILE [HHJ99] that address it in a more sofisticated, although limited way, even in the case of APM, which is the most powerful one. It provides a top-down approach to reuse-based model construction. A model is a pattern with some indefined activities. These activities may be substituted by adaptable template fragments. However, it does not support either a systematic pattern definition or a bottom-up reuse strategy (e.g., model composition). Overall, these languages offer a poor approach to the modelling of process patterns [JC00].

In summary, although some approaches that support reuse do exist in both the fields of SPM and workflow management, they do not provide all the mechanisms required in the frameworks by [JC00, RRN01]. On the other hand, some mechanisms like *composition* are only supplied by few PMLs and using not very expressive approaches (e.g., superposition). Other constructs, like parameterization and support for process patterns are scarce and poorly achieved [JC00]. We are not aware of any PML, both in the fields of SPM and workflow management, which defines a reuse framework endowed with all the mechanisms we have outlined in section 2. Furthermore, to our knowledge, no PML uses a standard notation to express reuse abilities.

References

[ABC96] Avrilionis, D.; Belkhatir, N; Cunin, P-Y. Improving Software Process Modelling and Enacting Techniques. In C. Montagnero (Ed.) Proc. of the 5th. European Workshop on Software Process Technology (LNCS-1149). Nancy, France. October, 1996.

[AC96] Avrilionis, D.; Cunin, P-Y.; Fernstr□m, C. OPSIS: A View-Mechanism for Software Processes whcih Supports their Evolution and Reuse. in Proc. of the 18th. Intl. Conf. on Software Engineering (ICSE-18). Berlin, Germany. March, 1996.

[Amb98] Ambler, S.W.: Process Patterns: Building Large-Scale Systems Using Object Technology. New York: SIGS Books/Cambridge University Press.

[Bog95] Bogia, D.P.: Supporting Flexible, Extensible Task Descriptions In and Among Tasks. Ph. D. thesis from Dept. of Computer Science, University of Illinois at Urbain Champaign, 1995.

[Car97] Carlsen, S. "Conceptual Modelling and Composition of Flexible Workflow Models", PhD-thesis, NTNU - Norwegian University of Science and Technology, Trondheim, Norway, 1997.

[CL99] Cysneiros, L.M.; Leite, J. "Integrating Non-Functional Requirements into Data Modeling". Procs. 4th ISRE, June 99, Limerick (Ireland).

[CNM99] L. Chung, B.A. Nixon, E. Yu, J. Mylopoulos: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers, ISBN 0-7923-8666-3. October 1999.

[DC99] D□Souza D.F.; Cameron, A: Objects, Components and Frameworks with UML. The Catalysis Approach. Addison Wesley, 1999.

[DKW99] Derniame, J.-C.; Kaba, B.A.; Wastell, D. (eds.): Software Process: Principles, Methodology and Technology. Lecture Notes in Computer Science, Vol. 1500. Springer-Verlag, Berlin Heidelberg New York (1999).

[EHT97] Engels, G.; Heckel, R.; Taentzer, G.; Ehrig, H. A View-Oriented Approach to System Modelling Based on Graph Transformation. In Proc. of the European Software Engineering Conference (ESEC□97). LNCS-1301. Springer-Verlag. 1997.

[FR99] Franch, X.; Rib□, J.M. Using UML for Modelling the Static Part of a Software Process. In Proceedings of UML □99, Forth Collins CO (USA). Lecture Notes in Computer Science (LNCS), Vol. 1723, pp. 292□307. Springer-Verlag (1999).

[HHJ99] Heinl, P.; Horn, S.; Jablonski, S. et alt.: A Comprehensive Approach to Flexibility in Workflow Management Systems. In proceedings of the International Joint Conference on Work Activities Coordination and Collaboration (WACC□99), San Francisco, USA, 1999.

[INCO96] InConcert 3.0 product information. http://www.xsoft.com/XSoft/products/ict/ic30.html

[ISO99] ISO/IEC Standards 9126 (Information Technology □Software Product Evaluation □Quality Characteristics and Guidelines for their use, 1991) and 14598 (Information Technology □Software Product Evaluation: Part 1, General Overview; Part 4, Process for Acquirers), 1999.

[Jacc96] Jaccheri, M.L. "Reusing Software Process Models in E3", IEEE International Software Process Workshop 10, Dijon France, June, 1996.

[JC00] J□rgensen H.D.; Carlsen, S, Writings in Process Knowledge Management: Management of Knowledge Captured by Process Models, SINTEF Telecom and Informatics, Oslo STF40 A00011, ISBN 82-14-01928-1, 2000-01-27.

[Kal96] Kalinichenko, L. Leonid A. Kalinichenko: Type Associations Identified to Support Information Resource Reuse in Megaprogramming. In Proceedings of the Third International Workshop on Advances in Databases and Information Systems, ADBIS 1996, Moscow, Russia, September 10□13, 1996.

[Kru97] Kruke, V: Reuse in Workflow Modeling Diploma Thesis. Department of Computer Systems. Norwegian University of Science and Technology. 1997

[Per96] Perry, D.E.: Practical Issues in Process Reuse. In proceedings of the International Software Process Workshop, 10 (ISPW□10). June, 1996

[PTV97] Puutsj□rvi, J.; Tirry, H.; Veijalainen, J. Reusability and Modularity in Transactional Workflows Information Systems. Vol. 22 N. 2/3 pp. 101□120, 1997.

[RF01] Rib□ J.M; Franch X.: Building Expressive and Flexible Process Models using an UML-based approach. Proceedings of the 8th. European Workshop in Software Process Technology. Witten (Germany). Lecture Notes in Computer Science (LNCS), Vol. 2077, pp. 152□172. Springer-Verlag (2001).

[RF02] Rib□ J.M; Franch X.: A Precedence-based Approach for Proactive Control in Software Process Modelling. In Proceedings of the SEKE-2002 conference (Software Engineering and Knowledge Engineering). ACM Press. Ischia (Italy). September, 2002.

[Rib02] PROMENADE: a UML-based Apprach to Software Process Modelling. PhD thesis. Dept. LSI, Politechnical University of Catalonia (2002).

[RRN01] Reis, R; Reis, C; Nunes, D.: Automated Support for Software Process Reuse: Requiremetns and Early Experiences□ in Proceedings of the 7th International Workshop on Groupware (CRIGW-01) Darmstadt (Germany) September, 2001.

[UML01] Unified Modelling Language (UML) 1.4 specification. OMG document formal/ (formal/2001-09-67). September, 2001.

Integrated Measurement for the Evaluation and Improvement of Software Processes

Félix García, Francisco Ruiz, José Antonio Cruz, and Mario Piattini

Alarcos Research Group, University of Castilla-La Mancha, Paseo de la Universidad, 4
13071 Ciudad Real, Spain
{Felix.Garcia, Francisco.RuizG, Mario.Piattini}@uclm.es,
jacruz@proyectos.inf-cr.uclm.es
http://alarcos.inf-cr.uclm.es/english/

Abstract. Software processes have an important influence on the quality of the final software product, and it has motivated companies to be more and more concerned about software process improvement when they are promoting the improvement of the final products. The management of software processes is a complex activity due to the great number of different aspects to be considered and, for this reason it is useful to establish a conceptual architecture which includes all the aspects necessary for the management of this complexity. In this paper we present a conceptual framework in which the software process modeling and measurement are treated in an integrated way for their improvement. As a support to the improvement a collection of software process model metrics is proposed. For the management of the measurement process, GenMETRIC, an extensible tool for the definition, calculation and presentation of software metrics, has been developed.

1 Introduction

The research about the software process has acquired a great importance in the last few years due to the growing interest of software companies in the improvement of their quality. Software processes have an important influence on the quality of the final software product, and for this reason companies are becoming more and more concerned about software process improvement, when they are promoting the improvement of the final products. Software applications are very complex products, and this fact is directly related to their development and maintenance. The software process is therefore a process, yet with special characteristics stemming from the particular complexity of the software products obtained. To support the software process evaluation and improvement, a great variety of initiatives have arisen establishing reference frameworks. Among these initiatives, of special note are CMM [22], CMMI [23], the ISO 15504 [10] standard, and, given its importance, the improvement has been incorporated into the new family of ISO 9000:2000 [12], [13] standards that promote the adoption of a focus based on processes when developing, implementing or improving a quality management system. Among the above-mentioned improvement initiatives, CMMI (Capability Maturity Model Integration) stands out as being especially impor-

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 94–111, 2003.

tant. Within the context of CMMI, the company should continuously **understand**, **control** and **improve** its processes, in order to reach the aims of each level of maturity. As a result of effective and efficient processes, a company will, in return, receive high quality products that satisfy both the needs of the client and of the company itself.

The successful management of the software process is necessary in order to satisfy the final quality, cost, and time of the marketing of the software products. In order to carry out said management, four key responsibilities need to be assumed [6]: **Definition**, **Measurement**, **Control** and **Improvement** of the process. Taking these responsibilities into account, it is very important to consider the integrated management of the following aspects to be able to promote process improvement:

- **Process Modeling**. Given the particular complexity of software processes, deriving from the high diversity of elements that have to be considered when managing them, it is necessary to effectively carry out a definition process of the software process. From the process modeling point of view, it is necessary to know which elements are involved before processing them. A software process can be defined as the coherent set of policies, structures, organisation, technology, procedures and artifacts needed to conceive, develop, package and maintain a software product [4]. Software process modeling has become a very acceptable solution for treating the inherent complexity of software processes, and a great variety of modeling languages and formalities can be found in the literature. They are known as "Process Modeling Languages" (PML) and their objective is to precisely represent, without ambiguity, the different elements related to a software process. In general, the following elements (general concepts, although, with different notations and terms) can be identified in a software process in the different PMLs [4]: **Activity**, **Product**, **Resource** and **Organisations and Roles**. Faced with the diversity of existing process modeling proposals, a process metamodel becomes necessary. This metamodel can serve as a common reference, and should include all of the aspects needed to define, as semantically as possible, the way in which the software is developed and maintained. With this goal, the Object Management Group recently proposed the SPEM (Software Process Engineering Metamodel Specification) [17] metamodel, that constitutes a language for the creation of concrete process models in a company.
- **Process Evaluation**. In order to promote software process improvement, it is very important to previously establish a framework for analysis (with the aim of determining its strong and weak points). An effective framework for the measurement of the software processes and products of a company, must be provided, in order to carry this out. The other key aspect to be considered, is the importance of defining and validating software process metrics, in order to evaluate their quality. The previous step of the software processes improvement, is their evaluation, and this goal requires the definition of metrics related to the different elements involved in software processes. Due to the great diversity of elements involved in software processes, the establishment of a common terminology for the definition, calculation and exploitation of

metrics is fundamental for the integrated and effective management of the measurement process.

The integration of the modeling and evaluation of software processes is a fundamental factor for a company to reach a high degree of maturity in its processes, as identified by CMMI. Therefore, it is vital that processes be well understood and improved. This means that it is necessary for them to be well defined, and that an effective measurement process should be carried out previously.

In this article we propose a conceptual framework which integrates the modeling and measurement of the software processes to promote their improvement. This framework incorporates the elements necessary to facilitate the definition and evaluation of software processes. Besides, in order to support the evaluation of the process from a conceptual point of view, a set of metrics have been defined.

Firstly, we present a general view of the conceptual framework. In Section 3, a generic metamodel for the integration of the measurement, is described. It has been defined and incorporated into the conceptual architecture, aiming to establish the needed reference for integrated measurement in an organisation. In the following section, a set of representative metrics for the evaluation of software process models, are presented. In Section 5 the *GenMetric* tool, an extensible tool developed to support integrated measurement in a company, is described. Finally, some conclusions and further works are outlined.

2 Conceptual Framework for the Modeling and Measurement of Software Processes

In order for a company to carry out integrated management of its software processes, it is very important for it to establish a rigorous base for:

- the definition of its process models, using singular terminology and precise and well-defined semantics.

- the integrated management of measurement in the company, using a measurement metamodel that is the framework of reference for the creation of concrete measurement models (database measurement, design, analysis result or work product measurements, process model measurements, etc…).

A conceptual architecture with four levels of abstraction has been defined in order to integrate these two very important aspects in a software process. This architecture is based on the MOF (Meta Object Facility) standard for metamodeling, based on object technology [16] proposed by the Object Management Group (OMG).

The aim of MOF is to specify and manage metadata on different levels of abstraction. MOF describes an abstract modeling language (based on the nucleus of UML). In Table 1 the MOF standard conceptual architecture and its application to the framework of work proposed for the improvement of the software process is shown:

Table 1. MOF conceptual levels and their application for integrated improvement.

Level	MOF	Environment Application
M3	MOF-model (meta-meta-model)	MOF-model
M2	Meta-model	Software Process Engineering Metamodel (SPEM) Generic Measurement Metamodel (ISO 15939)
M1	Model	Concrete Process Models Concrete Measurement Models
M0	Data	Instances of Process Models (concrete projects in the real world) Instances of Measurement Models (results of the application of the measurement model)

The lower level of the conceptual architecture, M0, includes the results for:

- The application of a process model, for example, a model for evaluation and improvement [7], or a maintenance model [20] to a concrete software project. At this level of architecture, the results of the execution of a concrete process model will be registered.

- The application of a measurement process. At this level the values obtained following the application of a concrete measurement model will be registered. For example, the values from the measurement of a relational database or the values from the measurement of UML class diagrams.

The data managed at level M0 are instances of the data represented in the next level up, M1. At this level, according to the conceptual architecture proposed, concrete models for the definition of the software process and concrete models for their measurement will be included. From the definition point of view, at this level the company will include its process models, for example, the model for development, maintenance, evaluation and improvement process, etc. From the measurement point of view, this level will include the concrete measurement models used by the company. For example, this could include concrete measurement models for the measurement of relational [3], object-relational [19], active [5] databases, etc. and concrete models for measuring software artifacts such as UML class diagrams [8], state transition diagrams [9], etc. Moreover, at this level the company could also dispose of measurement models for the defined process models themselves. A collection of metrics of software process models is described in Section 4.

All of the models defined in level M1 are instances of the concepts represented in M2. Therefore, in the M2 level of abstraction of the conceptual architecture, generic metamodels for the creation of concrete models should be included. In our framework the generic metamodels required are:

- **Software Process Metamodel**, with which concrete process models can be defined. SPEM [17] has been chosen as a software process metamodel due to

its wide acceptation in the industry. This metamodel contains the constructors needed to define any concrete software process model. The conceptual model of SPEM is based on the idea that a software development process consists of the collaboration between abstract and active entities, referred to as process roles, that carry out operations, called activities, on tangible entities, called work products. SPEM is basically structured in 5 packages that are: **Basic Elements**, which includes the basic elements needed to describe processes; **Dependences**, that contains the dependences necessary in order to define the relationships between the different process modeling elements, like for example, the ”precedes” dependence, which is a relationship between activities, or between work definitions, and indicates “beginning-beginning”, “end-beginning” or “end-end” dependences; **Process Structure**, which includes the structural elements through which a process description is constructed.; **Process Components**, which contains the elements needed to divide one or more process descriptions into self-contained parts, upon which configuration management processes or version controls can be applied; and **Process Life Cycle**, that includes the process definition elements that help to define how the processes will be executed. In short, SPEM makes the software process integrated management, within the proposed conceptual architecture easier, since the concepts of the different models are grouped under a common terminology.

- **Measurement Metamodel**, with which it is possible to define concrete measurement models. This metamodel is described in detail in Section 3.

In the final conceptual level of the architecture, M3, all of the concepts of the process metamodel and measurement metamodel are represented. This is done using the MOF abstract language, which is basically composed of two structures: MOF class and MOF association (these are the main elements for us, although others do exist such as: package, type of data, etc…). In this way, all of the concepts in level M2 are instances of MOF class or MOF association, for example, the SPEM concepts like, “Activity”, “Work Product” and concepts of the measurement metamodel like “Metric”, “Indicator”, “Measurement Unit” are instances of MOF class, and the relationships “Activity precedes Activity”, “Work Product precedes Work Product” or “Metric has a Measurement Unit”, are instances of MOF association.

With this architecture, it is possible to perform integrated management of the software process improvement, since the process definition and its measurement are systematically integrated. The MANTIS-Metamod [7] tool, which allows for the definition of metamodels (based on the MOF language constructors) and of models (based on the constructors of their metamodels), has been developed as a means of support to this conceptual architecture. A repository manager [21] that uses the XMI standard (XML Meta-data Interchange) [18] to promote portability of the defined models and metamodels is used for management of the storage and exchange of the metadata from the conceptual architecture.

3 Measurement Metamodel for the Integrated Process Evaluation and Improvement

A fundamental element to take into consideration when establishing a framework for process improvement, is the possibility of defining objective indicators of the processes that allow a software company to efficiently evaluate and improve its processes at any given moment. Evaluation standards like CMM, CMMI, ISO 15504, ISO 9000:2000 have assigned an important role to measurements in order to determinate the status of the software processes. A measurement process framework must be established in order to do so.

A good base for developing a measurement process is the one provided by CMMI [23]. In CMMI a new key process area called "Measurement and Analysis" is included. The aim of this area is to develop and establish a measurement capacity that can be used to support the company's information needs, and this implies broadening the concepts included in the CMM model. According to CMMI, the first step in the measurement process is to identify the measurement objectives, so that, in a second step, a measurement and analysis process can be implemented. This requires the measurement to be integrated in the different work processes of a company. It is very important for a company wishing to implant an effective measurement process to be able to precisely define concrete measurement models that, being supported by an integrated measurement tool, allow the appropriate and necessary automation for process evaluation.

Most of the problems in collecting data on a measurement process are mainly due to a poor definition of the software measures being applied. Therefore, it is important not only to gather the values pertaining to the measurement process, but also to appropriately represent the metadata associated to this data. In [14] a method for the specification of measurement models is defined with the aim of capturing the definitions and relationships between software measurements. The proposed framework is made up of three levels of abstraction for measurement, starting from a generic measurement model and moving up to automation of the gathering of metric values on a project level. This idea of abstraction is fundamental in order to be able to effectively integrate the measurement process into the organisation.

Therefore, it is very convenient to introduce a generic metamodel for measurement, making it possible to derive concrete measurement models that make up the base for assessment and improvement processes in an organisation. In Figure 1 our proposal for a measurement metamodel based on the ISO 15939 [11] standard is represented in UML.

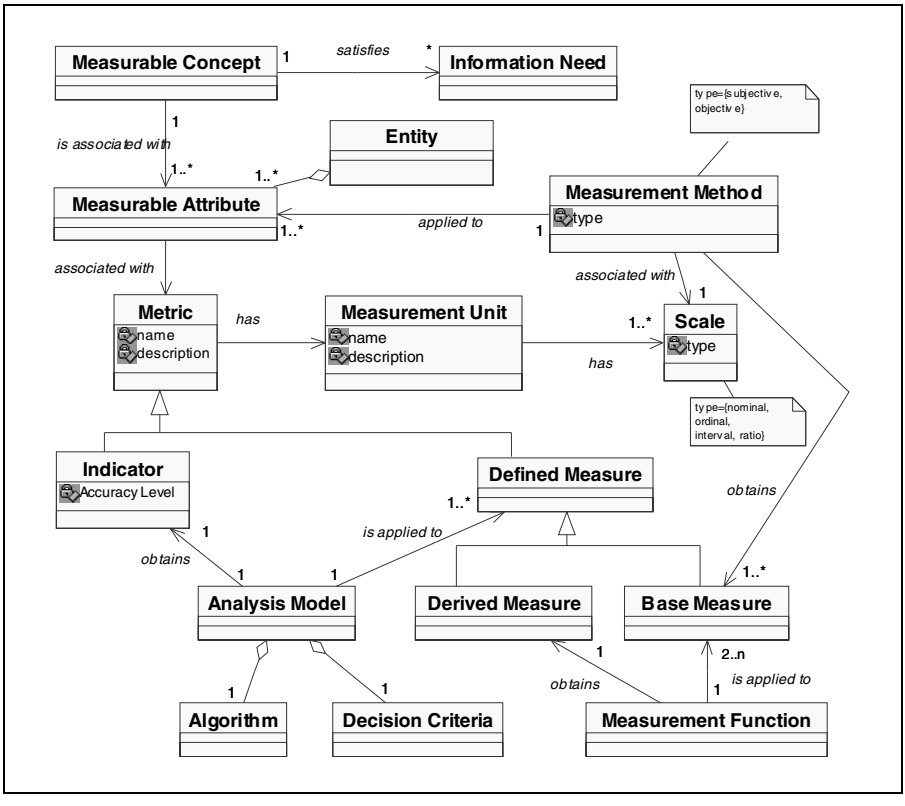


Fig. 1. Generic Metamodel for the Measurement Process

As can be observed in Figure 1, under the measurement point of view, the elements on which properties can be measured are "**Entities"**. An entity is an object (for example, a process, product, project or resource), that can be characterised through the measurement of its **"Measurable Attributes"** which describe properties or characteristics of entities, which can be distinguished quantitatively or qualitatively by human or automatic means. The aim of attributes is to satisfy specific information needs such as, "*the need to compare software development productivity with respect to a determined value*". This abstract relation between attributes and information needs is represented by the element called **"Measurable Concept"**, that, in this case, would be *"productivity ratio of software development"*. As measurable attributes, attributes of the developed product size or of development effort could be used.

All measurable attributes are associated to a metric, which is an abstraction of the different types of measurements used to quantify, and to make decisions concerning the entities. All metrics are associated to a unit of measure (for example, code lines), which at the same time belong to a determined scale. In accordance with the standard, the 4 scales distinguished are: nominal, ordinal, interval and ratio, although other classifications can be established like in [14]. The three types of metrics are:

- **Base Measurement**, defined in the function of an attribute, and the method needed to quantify it (a measurement is a variable to which a value is assigned).

- **Derived Measurement**, a defined measurement in function of two or more values of base measurements.

- **Indicator**, a measurement that provides an estimate or assessment of specific attributes, derived from a model with respect to information needs. The indicators are the base for analysis and decision-making. These measurements are the ones that are presented to the users in charge of the measurement process.

The procedures for calculating each of the metric types are:

- The values of the base measurements are reached with "***Measurement Methods***" that consist of a logical sequence of operations, generically described, used to quantify an attribute with respect to a specific scale. These operations can imply activities such as, counting occurrences or observing the passing of time. The same measurement method can be applied to multiple attributes.

- The derived measurements are obtained by applying a "***Measurement Function***", which is an algorithm or calculation carried out to combine two or more base measurements. The scale and unit of the derived measurement depends on the scales and units of the base measurements.

- The indicators are obtained with an "***Analysis Model***". An analysis model produces estimates and assessments relevant to the defined information needs. It consists of an algorithm or calculation that combines one or more base measurements and/or derivates with determined decision-making criteria. All decision-making criteria is composed of a series of limit values, or used objects for determining the need to research, or to describe the confidence level with regard to a determined result. These criteria help to interpret the measurement results.

Using this reference metamodel it is possible to measure any element of a process or data model. Taking into account that our main objective is the software process improvement, and therefore, the evaluation, itis necessary to establish the relationship between the main elements of the software process metamodel and the main elements of the software measurement metamodel. This relationship is represented in Figure 2.

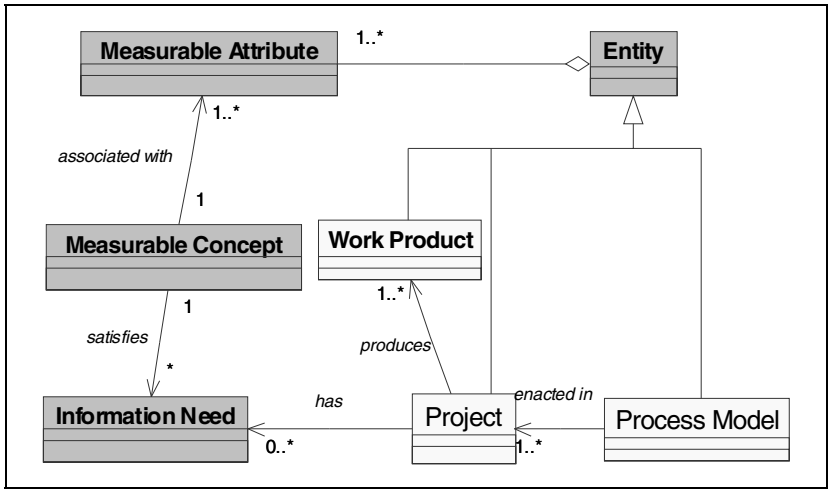


Fig. 2. Relation between the software process metamodel and the measurement metamodel

As we can observe in Figure 2, any software process model is enacted in concrete software projects. As a result of carrying out a software project, certain work products are produced and all software projects are required to satisfy some information needs. The main candidate elements to be measured in order to evaluate the software process are:

- **The Software Process Model**. It could be very convenient to research if the model of software processes has an influence on its final quality. For this reason, with the framework proposed, it is possible to define metrics related with the constructors of the software process metamodel. For example, if we apply the measurement metamodel to the elements of the SPEM model, we could measure important elements like the class *Activity,* and the classes *Work Product and Process Role.* These elements of the model have a set of measurable attributes, such as for an activity: *"the number of activities with which there is a precede type dependence"*. This attribute would be calculated with a metric to satisfy an information necessity like, *"Evaluate the software process coupling"* and, in this case, the unit of measure would be of *"Ratio"* type. This issue will be treated in the following section.

- **The Work Product**. This is a fundamental factor in the quality of the software process. Work Products are the result of the process (final or intermediate), and their measurement is fundamental in order to evaluate the software processes. With the framework proposed, it's possible to measure the quality attributes related with the artifacts or work products, by defining the metamodels related with the different work products. For example, if we have to evaluate the quality of a UML class diagram we have to incorporate into the framework, the UML metamodel and metrics necessary.

In this way, with the framework proposed, the work of an assessment and improvement process is eased, since the fulfillment of the software processes carried out in a determined organisation are quantitatively registered.

4 Proposal of Software Metrics for the Evaluation of Software Processes Models

The study of the possible influence of the software process model complexity in its execution could be very useful. For this reason the first step is the definition of a collection of useful metrics in order to characterise the software process models. In this section a collection of metrics of software process models are going to be defined in order to evaluate their complexity. These metrics have been defined according to the SPEM terminology, but they can be directly applied to other process modeling languages. The metrics proposed could be classified like model level metrics, if they evaluate the characteristics of a software process model, or like fundamental element (activity, process role and work product) metrics, if they describe the characteristics of a model element. For this reason they will be described separately.

4.1 Model Level Metrics

The process model level metrics (PM) proposed are:

- **NA(PM)**. Number of **Activities** of the process model.
- **NSTP(PM)**. Number of steps (tasks) of the process model.
- **NDRA(PM)**. Number of dependence relationships between activities of the process model.
- **RSTPA(PM).** Ratio of steps and activities. Average of the steps and the activities of the process model.

$$RSTPA(PM) = \frac{NSTP(PM)}{NA(PM)}$$

- **AC (PM)**: Activity Coupling in the process model. This metric is defined as:

$$AC(PM) = \frac{NA(PM)}{NDRA(PM)}$$

- **NWP(PM):** Number of **Work Products** of the process model.
- **RWPA(PM):** Ratio of work products and activities. Average of the work products consumed (input), modified (input/output) or produced (output) by the activities.

$$RWPA(PM) = \frac{NWP(PM)}{NA(PM)}$$

- **NPR(PM):** Number of **Process Roles** of the process model.
- **RPRA (PM):** Ratio of process roles and activities. Average of the process roles and the activities of the process model.

$$RPRA(PM) = \frac{NPR(PM)}{NA(PM)}$$

4.2 Fundamental Element Level Metrics

- **Activity Metrics**:
  - **NSTP(A)**. Number of **Steps** (tasks) of an Activity.
  - **NWPIn(A)**. Number of Input Work Products of the Activity.
  - **NWPOut(A).** Number of Output Work Products of the Activity.
  - **NWPInOut(A).** Number of Input-Output Work Products of the Activity.
  - **NWP(A).** Total Number of **Work Products** related to an Activity.

$$NWP(A) = NWPIn(A) + NWPOut(A) - NWPInOut(A)$$

  - **RWPIn(A)**. Average of the Input Work Products in activity A.

$$RWPIn(A) = \frac{NWPIn(A)}{NWP(A)}$$

  - **RWPOut(A)**. Average of the Output Work Products in activity A.

$$RWPIn(A) = \frac{NWPOut(A)}{NWP(A)}$$

  - **RWPInOut(A)**. Average of the Output Work Products respect to the total number of Work Products in activity A.

$$RWPInOut(A) = \frac{NWPInOut(A)}{NWP(A)}$$

  - **NR(A).** Number of responsible **Roles** of an Activity.
  - **NPD(A)**. Number of **Activities** which are **predecessors** (activity dependences of input) of Activity A.

- **NSD(A)**. Number of **Activities** which are **successors** (activity dependences of output) of Activity A.
- **ND(A)**. Total number of dependences of activity A.

$$ND(A) = NPD(A) + NSD(A)$$

- **PR(A)**. Average of the predecessors activities with respect to the total number of dependences in activity A.

$$PR(A) = \frac{NPD(A)}{ND(A)}$$

- **PS(A)**. Average of the successors activities with respect to the total of dependences in activity A.

$$PS(A) = \frac{NSD(A)}{ND(A)}$$

- **Process Role Metrics**:
  - **NARP(R)**. Number of **Activities** who's responsibility is role R.
  - **RRPR(R)**. Ratio of responsibility of the process role. Ratio between the activities in which role R is responsible and the total number of activities in the model.

$$RRPR(A) = \frac{NARP(R)}{NA(PM)}$$

- **Work Product Metrics**:
  - **NAWPIn(WP)**. Number of **Activities** in which the work product is of input.
  - **NAWPOut(WP).** Number of **Activities** in which the work product is of output.
  - **NAWPInOut(WP).** Number of **Activities** in which the work product is of input/output.

  - **NAWP(WP)**. Number of Activities related with the Work Product.

$$NAWP(WP) = NAWPIn(WP) + NAWPOut(WP) - NAWPInOut$$

  - **RDWPA(WP)**. Ratio of dependence of the work product. Ratio between the activities related with the work product and the total number of activities in the model.

$$RDWPA(WP) = \frac{NAWP(WP)}{NA(PM)}$$

4.3 Example

Figure 3 shows an example of a simplified software process model which belongs to the Rational Unified Process [2]. For the graphical representation of the model the SPEM notation [17] has been used. The values of the metrics proposed are shown in the tables 2, 3 and 4.

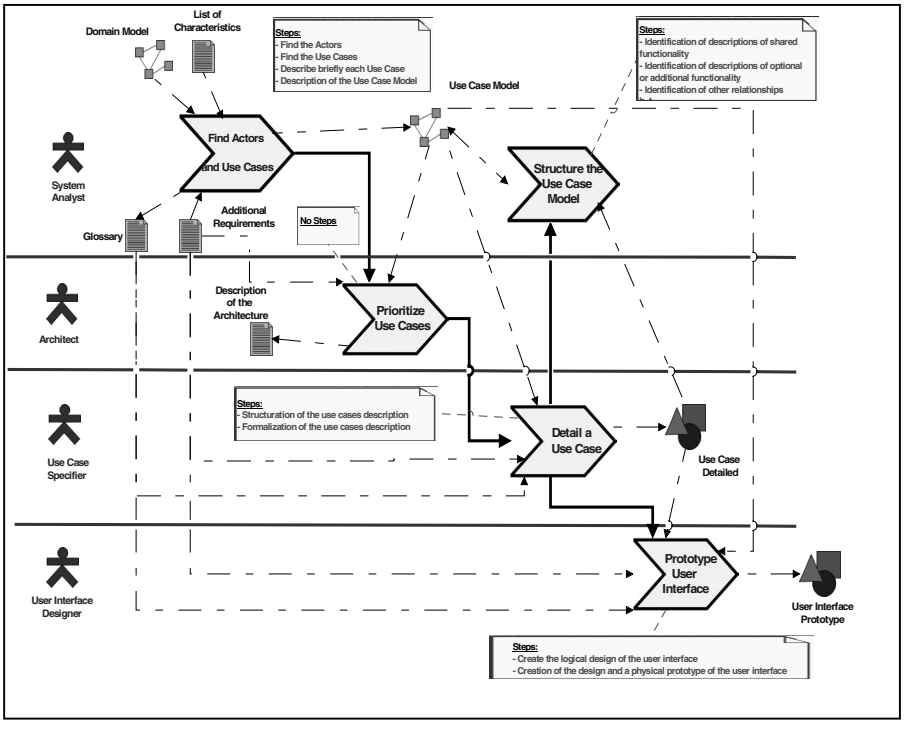


Fig. 3. Example of a Software Process Model represented with SPEM.

Table 2. Model Level Metrics

Metric	Value	Metric	Value
NA(PM)	5	NWP(PM)	8
NSTP(PM)	11	RWPA(PM)	8/5=1,6
NDRA(PM)	4	NPR(PM)	4
RSTPA(PM)	11/5=2,2	RPRA(PM)	4/5= 0,8
AC(PM)	5/4= 1,25		

Table 3. Metrics of the Activity "Detail a Use Case".

Metric	Value	Metric	Value
NSTP(A)	2	RWPInOut(A)	0
NWPIn(A)	3	NR(A)	1
WPOut(A)	1	NPD(A)	1
NWPInOut(A)	0	NSD(A)	2
NWP(A)	4	ND(A)	3
RWPIn(A)	3/4= 0,75	PR(A)	1/3=0,33..
RWPOut(A)	1/4= 0,25	PS(A)	2/3=0,66..

Table 4. Metrics of Work Product and Process Role examples.

Process Role "System Analyst" Metrics	Value	Work Product "Use Case Model" Metrics	Value
NARP(R)	2	NAWPIn(WP)	4
RRPR(R)	2/5=0,4	NAWPOut(WP)	2
		NAWPInOut(WP)	1
		NAWP(WP)	4+2-1=5
		RDWPA(WP)	5/5=1

This is only the first step in the overall metrics definition process [3]. The following step is the formal validation of the metrics, and then, it is fundamental to run empirical studies in order to prove the practical utility of the metrics defined. As a result of this step (and of the complete method) we will be able to accept, discard or redefine the metrics presented in this paper. An important number of metrics have been proposed, and the validation process is fundamental for the selection of the adequate metrics which fulfill our objective.

5 GenMETRIC. Extensible Tool for the Integrated Management of the Measurement Process

Aiming to offer automatic support to the integrated measurement process commented on in the previous sections, we have developed the GenMETRIC tool. GenMETRIC is an extensible tool for the definition, calculation and visualisation of software metrics. This tool for the integrated management of the measurement process supports the definition and management of software metrics. Moreover, the tool supports the measurement metamodel based on ISO 15939 that has been proposed for better support and management of the integrated measurement process.

For the management of the measurement process, the tool can import information on the following elements, represented in XMI document form [18]:

- **Domain Metamodels** on which metrics are defined. The elements of each metamodel are stored to be able to carry out a measurement process on them. For example, if a relational database is to be measured, it will be necessary to previously define the elements of the relational metamodel, like: Table, Attribute, Interrelation, etc...
- **Domain Models**. The models are instances of the metamodels, and it is of interest to carry out a measurement process on them. For example, the schema (model) of the database of a bank would be an instance of the relational metamodel on which we could carry out a measurement process.
- **Metric Models**. Metric models allow the defined metrics to be consistently stored. In order to do so, the metric models used by the tool are instances of the measurement metamodel proposed in the previous section.

The information imported by the tool on the different domain metamodels, and on the metrics is persistently stored in a **XMI** based **Repository**. The calculation of the metrics defined is performed by using the information in the *Repository.* The different models and metamodels needed are defined and represented in XMI with the MANTIS-Metamod [7] tool. The relationship between GenMetric and MANTIS-Metamnod is represented in the following figure:

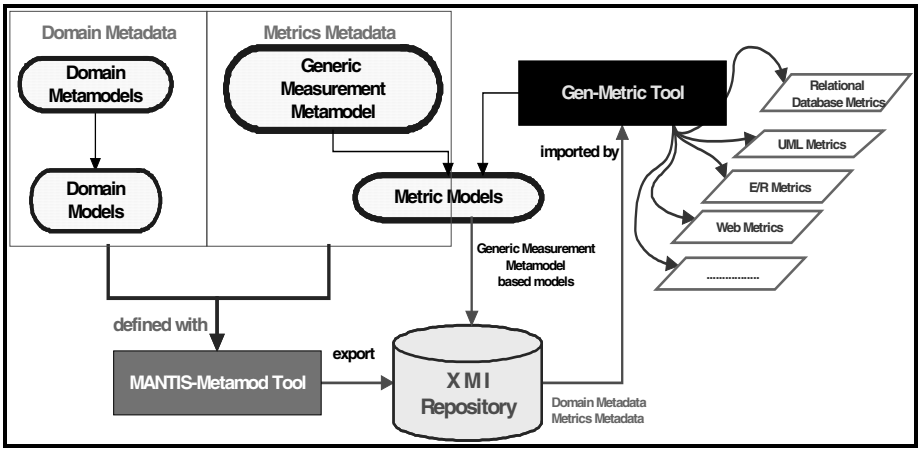


Fig. 4. Relationship between MANTIS-Metamod and Gen-Metric

As we can observe in Figure 4, the repository is the key element for the integrated management of the measurement process. The metadata are defined and exported in XMI with MANTIS-Metamod tool. The information of the repository is imported by GenMetric for the management of the metrics needed, and with GenMetric, the user can build metrics models (based on the generic metamodel). These models are exported to the repository.

GenMetric provides the user with a powerful interface for the definition, calculation and visualisation of metrics. From the perspective of the use of the tool, two roles

have been defined. They are: *Administrator,* that completely controls the functionality of the tool, allowing it to define, calculate and visualise any metric, and *User,* that has access to the calculation and visualisation of the metrics that have already been defined. In Figure 5 the interface for the definition of metrics is represented:

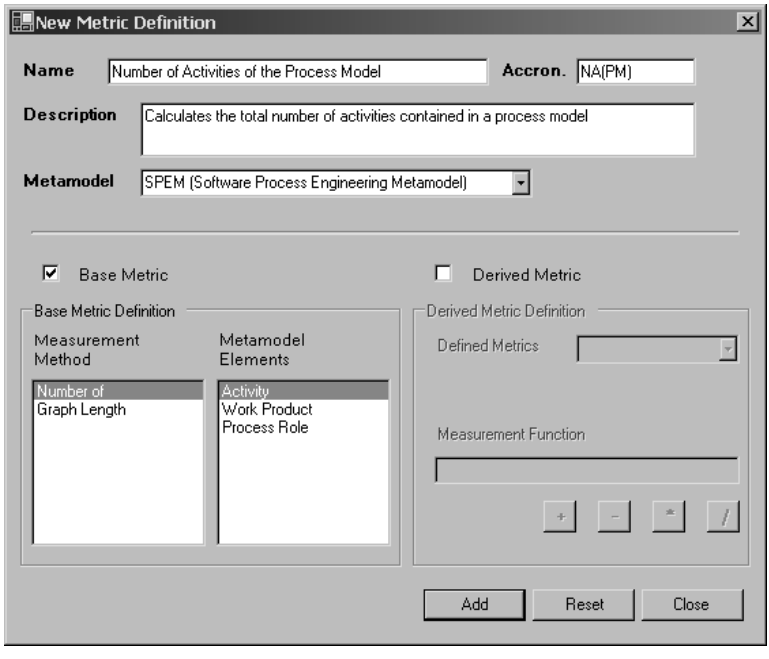


Fig. 5. Definition of a new metric with GenMetric.

An integrated and automatic environment for measurement is provided with the proposed tool. Being a generic tool, the definition of any new metric on the existing domain metamodels is possible, without having to code new modules. Furthermore, the tool is extensible, which eases the incorporation of new domain metamodels, for example a metamodel for defining web elements (formed by web pages, links between pages, etc.) and in this way it is possible for concrete domain models, web sites for example, to be measured. Moreover, as it works with XMI documents, it eases communication and the possibility of openly importing new domain metamodels, or domain models and metrics stored in other repositories based on MOF.

6 Conclusions and Future Work

In this work, a conceptual framework to promote the improvement based on integrated modeling and measurement of software processes has been presented. The SPEM metamodel is used for the modeling of processes under a common terminology, and a metamodel based on the ISO 15939 standard, easing the management of an integrated

measurement process to promote improvement in a company's processes, has been defined as an integrated framework for measurement.

In order to evaluate the influence of the complexity in the software process models in their enactment, some metrics have been proposed. These metrics are focused on the main elements included in a model of software processes, and may provide the quantitative base necessary to evaluate the changes in the software processes in companies with high maturity levels, which are applying continuous improvement actions [1].

As a means of support for the integrated measurement, the *GenMetric* tool has been developed to define, calculate and visualise software metrics. With the tool it is possible to incorporate new types of metrics and new types of elements to measure, since its architecture has a generic and extensible design.

With the proposed framework, any company dedicated to the development and/or maintenance of software can effectively define and evaluate its processes as a step prior to promoting their improvement. Furthermore, as the framework is based on the MOF standard, the simple extension and modification of its elements is possible, with the incorporation and modification of the necessary metamodels, since all of them are represented following the common terminology provided by the MOF model. Along the lines for improvement for future studies, we can point out the following:

- Description of concrete measurement models, using the generic metamodel, to effectively support the evaluation and improvement of software processes.
- Formal and empirical validation of the metrics proposed to study the relationship between the influences of the software process models complexity in their enactment.

Acknowledgements. This work has been partially funded by the TAMANSI project financed by "Consejería de Ciencia y Tecnología, Junta de Comunidades de Castilla-La Mancha" of Spain (project reference PBC-02-001) and by the DOLMEN project (Languages, Methods and Environments), which is partially supported by FEDER with number TIC2000-1676-C06-06.

References

1. Pfleeger, S.L.: Integrating Process and Measurement. In Software Measurement. A. Melton (ed). London. International Thomson Computer Press (1996) 53–74
2. Jacobson, I, G. Booch and J. Rumbaugh,. The Unified Software Development Process. Addison Wesley (1999)
3. Calero, C., Piattini, M. and Genero, M.: Empirical Validation of referential metrics. Information Software and Technology". Special Issue on Controlled Experiments in Software Technology. Vol.43, Nº 15 (2001)
4. Derniame, J.C., Kaba, B.A. and Wastell, D.: Software Process: Principles, methodology and technology. Lecture Notes in Computer Science 1500 . Springer (1999)
5. Díaz, O., Piattini, M. and Calero, C.: Measuring triggering-interaction complexity on active databases. Information Systems Journal. Elsevier Science. Vol. 26, Nº 1 (2001)

6. Florac, W. A. and Carleton, A.D.: Measuring the Software Process. Statistical Process Control for Software Process Improvement. SEI Series in Software Engineering. Addison Wesley (1999).
7. García, F., Ruiz, F., Piattini, M. and Polo, M.: Conceptual Architecture for the Assessment and Improvement of Software Maintenance. 4th International Conference on Enterprise Information Systems (ICEIS'02). Ciudad Real, Spain, April (2002), 610–617
8. Genero, M., Olivas, J., Piattini, M. and Romero, F.: Using metrics to predict OO information systems maintainability. 13th International Conference Advanced Information Systems Engineering (CAiSE'01), (2001), 388–401
9. Genero, M., Miranda, D. and Piattini, M.: Defining and Validating Metrics for UML Statechart Diagrams. 6th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE), (2002), 120–136
10. ISO/IEC: ISO IEC 15504 TR2:1998, part 2: A reference model for processes and process capability, (1998)
11. ISO IEC 15939, Information Technology – *Software Measurement Process*, Committee Draft, December (2000)
12. International Organization for Standardization (ISO). *Quality management systems - Fundamentals and vocabulary*. ISO 9000:2000, (2000). See http://www.iso.ch/iso/en/iso9000-14000/iso9000/selection_use/iso9000family.html
13. International Organization for Standardization (ISO). 2000. *Quality management systems - Requirements* ISO 9001:2000, (2000)
14. Kitchenham, B. A., Hughes, R.T. and Linkman, S.G.: Modeling Software Measurement Data. IEEE Transactions on Software Engineering. 27(9), (2001), 788–804
15. OMG Unified Modeling Language Specification; version 1.4, Object Management Group. September (2001). Available in http://www.omg.org/technology/documents/formal/uml.htm
16. Meta Object Facility (MOF) Specification; version 1.4. Object Management Group. April (2002). In http://www.omg.org/technology/ documents/formal/mof.htm
17. Software Process Engineering Metamodel Specification; adopted specification, version 1.0. Object Management Group. November (2002). Available in http://cgi.omg.org/cgi-bin/doc?ptc/02-05-03.
18. OMG XML Metadata Interchange (XMI) Specification; version 1.2. Object Management Group. January (2002). In http://www.omg.org/technology/documents/formal/xmi.htm
19. Piattini, M., Calero, C., Sahraoui, H. and Lonis, H.: Object-Relational Database Metrics. L' object. ISSN 1262–1137. HERMES Science Publications, Paris. Vol.7, Nº 4, (2001)
20. Ruiz, F., Piattini, M. and Polo, M. An Conceptual Architecture Proposal for Software Maintenance. International Symposium on Systems Integration (ISSI, Intersymp'2001). Baden-Baden, Germany (2001), VIII:1–8
21. Ruiz, F., Piattini, M., García, F. and Polo, M. An XMI-based Repository for Software Process Metamodeling. Proceedings of 4th International Conference on Product Focused Software Process Improvement (PROFES'2002). Lecture Notes in Computer Science (LNCS 2559), Markku Oivo, Seija Komi-Sirviö (Eds.). Springer. Rovaniemi (Finland). December (2002), 546–558
22. Software Engineering Institute (SEI). The Capability Maturity Model: Guidelines for Improving the Software Process, (1995). In http://www.sei.cmu.edu/cmm/cmm.html
23. Software Engineering Institute (SEI). Capability Maturity Model Integration (CMMI^{SM}), version 1.1. March (2002). In http://www.sei.cmu/cmmi/cmmi.html

Process Support for Evolving Active Architectures

R. Mark Greenwood[1], Dharini Balasubramaniam[2], Sorana Cîmpan[3], Graham N.C. Kirby[2], Kath Mickan[2], Ron Morrison[2], Flavio Oquendo[3], Ian Robertson[1], Wykeen Seet[1], Bob Snowdon[1], Brian C. Warboys[1], and Evangelos Zirintsis[2]

[1] Department of Computer Science, The University of Manchester, Manchester, M13 9PL, UK.
{markg,robertsi,seetw,rsnowdon,brian}@cs.man.ac.uk
[2] School of Computer Science, The University of St. Andrews, St. Andrews, Fife, KY16 9SS, UK.
{dharini,graham,ron,kath, vangelis}@dcs.st-and.ac.uk
[1] ESIA, Universite de Savoie, 5 Chemin de Bellevue, 74940 – Annecy-le-Vieux, France.
{Sorana.Cimpan,Flavio.Oquendo}@esia.univ-savoie.fr

Abstract. Long-lived, architecture-based software systems are increasingly important. Effective process support for these systems depends upon recognising their compositional nature and the active role of their architecture in guiding evolutionary development. Current process approaches have difficulty with run-time architecture changes that are not known a priori, and dealing with extant data during system evolution. This paper describes an approach that deals with these issues. It is based on a process-aware architecture description language (ADL), with explicit compose and decompose constructs, and with a hyper-code representation for dealing with extant data and code. An example is given to illustrate the ease-of-use benefits of this approach.

1 Introduction

There is now a substantial proportion of software development that is based on assembling a set of appropriate components into the required system. This typically involves configuring generic components to the particular situation, and writing the necessary glue code to bind the components into a coherent system. This component-oriented approach clearly has impact on the software process used to develop and evolve such systems. The process needs to represent the essential compositional nature of the system. When the system is long-lived, its development process is evolutionary. It is based around the system's decomposition into components, the replacing or modifying of those components, and the recomposition of the evolved system.

A popular way of designing long-lasting compositional systems is to adopt an architecture-based approach. The system architecture describes the assembly of the system in terms of functional components, and the connectors that link them together. This architecture provides a high-level view that is used to understand the system and

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 112–127, 2003.

guide its evolutionary development. In most cases it is essential that this architecture is not static. As the system evolves the architecture itself must evolve. These are the systems that we consider to have *active architectures*.

Software developers evolving a long-lived system have to deal with both the operational software and the process for evolving (managing) the operational software. This evolutionary process might include software for re-installing the system, or specific components. In addition, it might include utilities that migrate essential state information from the current to the evolved system.

In this paper we present an approach to dealing with this key software process problem. It is based on three features:

- Use of the software architecture to structure the evolutionary development process as well as the software itself.
- The architecture-based software process explicitly represents the composition, decomposition and re-composition that are at the heart of the evolution process.
- The use of a hyper-code representation [5,28] so that the current state or context can be effectively managed during the evolution.

The approach is illustrated through an example. This example is deliberately kept small so that it can be explained within the confines of this paper. The aim is to show the ease with which the approach deals with aspects that are often ignored in software processes. In section 2 we place our approach in the context of related work. In particular, we contrast the problem of supporting the evolution of active architectures with the more conventional project-based software processes. In section 3, we examine composition and decomposition in more detail. In section 4, we introduce the concept of hyper-code that is an essential novel feature of our approach. In section 5, we describe the architecture description language (ADL) used in the small example in section 6. Section 6 steps through an example evolution of the example active architecture system, illustrating the three features of the approach. Section 7 describes further work and Section 8 concludes.

2 Related Work

The relationship between architecture and evolution is widely acknowledged. The Unified Process describes software as having both form (architecture) and function, and stresses the relationship between architecture and evolution. “It is this form, the architecture, that must be designed so as to allow the system to evolve.”[15] From a software maintenance background, [24] describes the importance of a flexible architecture in enabling software systems to evolve and continue to meet the changing requirements of their users.

The use of an explicit run-time representation of a system’s architecture to aid evolution of the system at run-time is described in [22]. Archstudio [21] is a tool suite that supports architecture-based development. Changes are made to an architectural model and then reified into implementation by a runtime architecture infrastructure.

We use the term active architecture to indicate that the architecture evolves “in synch” with the system. Run-time linguistic reflection is used to ensure that the architecture models the current state of the system. Any evolution is either an explicit change of both the architecture and the system, or directly reflected in the active architecture. Other work on so called self-organising systems is progressing but they are generally 'less active'. For example in [9], specific component managers identify external architecture changes by listening to events, and then react in order to preserve architecture constraints. The constraints themselves cannot be evolved.

While the relationship between architecture and evolution is recognised, the full potential of combining software process modelling techniques, to explicitly represent system evolution, and the system architecture, to structure that process representation, has not been fully realised. Traditionally, the process modelling research community has adopted a simplistic view of the aims driving business, or software, process modelling [6, 7, 8]:

1. Model a process using a textual or diagrammatic representation. (This model is often called a process model definition.)
2. Translate (or compile) the model into a process support system (PSS).
3. Instantiate the library model to give a process model enactment. (This is often called a process model instance.)
4. The PSS runs the model enactment to support the process performance of a specific process instance.

This is based on the context of a software development organisation that undertakes a number of distinct software projects over a period of time. They create a process model enactment for each project. The assumption is that the textual or diagrammatic model represents current best practice, and one of the main benefits of modelling is to disseminate this to all projects. There may be some additional customisation of the model to a specific project when it is instantiated, but the assumption is that the general form of the model is project independent. The focus on process model evolution is on the textual or diagrammatic representation so that future projects can learn from the experience of earlier ones.

The simplistic view of process modelling is closely aligned with a corresponding view of the core of software engineering:

1. Create a program using a textual or diagrammatic representation. (The program source code)
2. Compile the program in a specific run time environment. (This creates the executable program or object code.)
3. Start (or deploy) the executable program. (This creates a process, an instance of the executable program, in the run time environment.)
4. The run time environment runs the process to have the desired effect.

In this view the assumption is that the program is going to be run many times. The emphasis for evolution of the program is at the representation (source code) level. This means that only future program runs will benefit from improvements as the program evolves. The development process is often supported by tools, such as source code control systems, that help maintain control over the evolution of the representation.

One feature that both the simplistic views mentioned above share is the one-off, one-way translation from representation into values. In [13] we discussed how this is a particular special case of a more general relationship, and that an ongoing, two-way translations between representations and values are needed for evolutionary processes.

Clearly the above software engineering view is not appropriate for long-lived systems that can not be rebuilt from scratch when a change is required. There are two issues: scale, and extant data and code. The simplistic view above is a monolithic approach making it inappropriate for large problems, and provides no help for the re-use of existing components. The notion of composing a system out of smaller parts, which may themselves be composed from smaller parts and so on, is an essential tool in the management of complexity.

However an architecture-based approach addresses the issue of scale. The architecture provides a structure. This allows the representation (source code) of different parts (components) to be developed independently, often by separate teams. Some parts can be re-used source, or object code, so long as we have a representation of its interaction with the rest of the system. The architecture structure is also used to provide a build structure that manages the dependencies between parts, and is a key influence on deployment. A common deployment approach is to have several parts, or sub-systems, that can be started independently, so that individual sub-systems can be evolved without requiring changes to the other sub-systems.

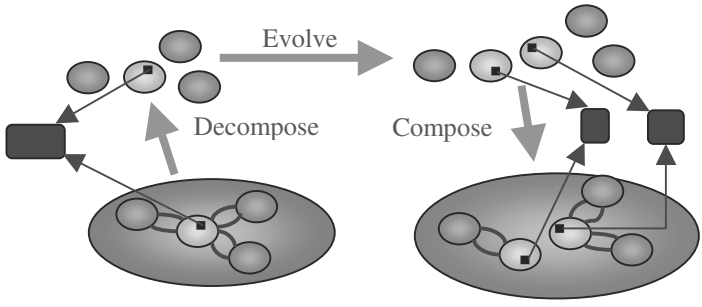
An architecture-based approach does not address the issue of extant data and code. The core issue here is that the current running version of a sub-system, which requires changing, may have valuable data that must be passed on to its evolved version, as well as running code that must be evolved. There is typically no way of referring to current existing data and code values in a source code representation. A typical work around is for the current running version to have a mechanism for writing out its data to a file or database, and the new evolved version includes initialisation code that reads this saved data. This requires some a priori knowledge so that the required data can be saved at the right time, and can be complex for data structures that can include shared or circular references [1].

The problems of extant data and code are typically tackled by ad-hoc solutions that are specific to a system. This is another example of how the one-way translation from representations to values places artificial limits on the potential of process support systems. Hyper-code technology [5, 28] promotes a more general two-way “round trip” translation, which can be performed many times throughout the lifetime of the process.

3 Composition and Decomposition

An essential property of evolutionary systems is the ability to decompose a running system into its constituent components, and recompose evolved or new components to form a new system, while preserving any state or shared data.

This scenario is modelled in the diagram below. The original system can be thought of as a composition of three client components communicating with a server



component. The server component refers to some data outwith these four components. This system can then be decomposed into its components with no communication. Note that the server component still maintains its link to the data.

We may then choose to evolve the components so that the clients stay the same while the server is divided into two different components. The new server components still maintain links to the data referred to by the original server. These five components can then be composed together to form a new system with one client communicating with one server and the other two clients communicating with the second server.

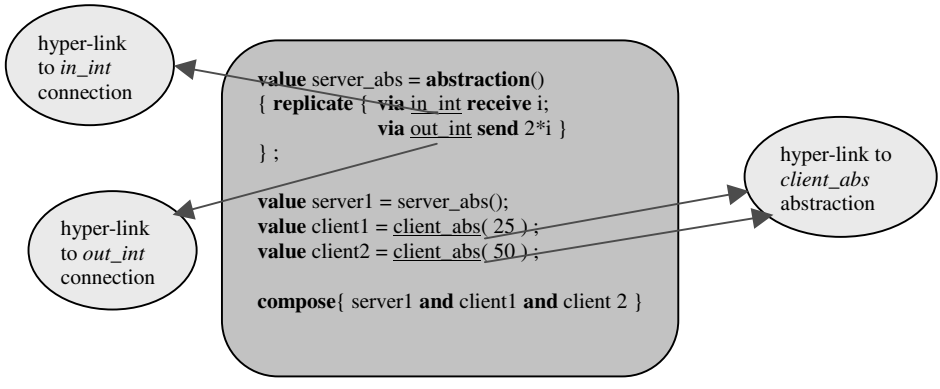
Note that we can interpret this diagram from both an architecture and a process perspective. From the architecture perspective the diagram captures the structure of the current and evolved systems, and the relationships between them in terms of which components are unchanged (the clients), modified (the server) or replaced. From a process perspective the diagram captures how to evolve from the current to the evolved system: decompose into parts, keep the clients, modify the server (splitting it into two), and recombine in the new configuration.

4 Hyper-Code

The *hyper-code* abstraction was introduced in [5] as a means of unifying the concepts of source code, executable code and data in a programming system. The motivation is that this may ease the task of the programmer, who is presented with a simpler environment in which the conceptually unnecessary distinction between these forms is removed. In terms of Brooks' *essences* and *accidents* [29], this distinction is an accident resulting from inadequacies in existing programming tools; it is not essential to the construction and understanding of software systems. In a hyper-code system the user composes hyper-code and the system executes it. When evolving the system, for example because an error has occurred, the user only ever sees a hyper-code representation of the program, which may now be partially executed. The hyper-code source representation of the program is structured and contains text and links to extant values.

The figure below shows an example of hyper-code representation of the ArchWare ADL, which is described in section 5 below. The links embedded in it are represented by underlined tokens to allow them to be distinguished from the surrounding text. The

first two links are to connection values *in_int* and *out_int* which are used by the *server_abs* abstraction to communicate with other abstractions. The program also has two links to a previously defined abstraction *client_abs*. Hyper-code models sharing by permitting a number of links to the same object. Instances of server and client abstractions are then composed to create a client-server system. Note that code objects (*client_abs*) are denoted using exactly the same mechanism as data objects (*in_int* and *out_int*). Note also that the object names used in this description have been associated with the objects for clarity only, and are not part of the semantics of the hyper-code.



The ability of hyper-code to capture closures allows us to represent parts of a system after decomposition without losing their context. It provides representations which can be used for both evolving the components and recomposing them into the new system.

The potential benefits of modelling and supporting evolving processes have been well recognised in the software process modelling community. Many process modelling research systems have included facilities for process evolution [3,8,27]. The most common mechanism for providing these facilities is through reflective programming techniques.

A significant problem has been that although such facilities make it possible to write evolving processes, they are frequently hard to write and understand. It is not the basic facility for introducing new code that causes the complication, but the problem of dealing with extant data. This can be particularly severe if the required evolution depends on the current state of the process being evolved. For example, Process*Web* [23, 26] has distinct reflective facilities for dealing with code and data [13]. For code the reflective facilities can provide a copy of the current source, edit this to produce new source, and apply this new code. For data the process evolver needs to write a meta-process model which, when it is run, will use the reflective facilities to obtain the current data values, manipulate them as required, and then transfer the required values from the meta-process model to the process model being evolved (for an example see [4]). In short, while in Process*Web* it is possible to write a universal meta-process for dealing with code changes, the meta-process model for data changes is specific to the model being evolved. For data changes the process evolver has to think at the meta-meta-process level in order to create the specific

meta-process required. This is a result of the fact that there is a source representation for any code value that needs to be evolved, but there is no source representation for every data value.

Hyper-code provides a unification of source and data, and hides the underlying reflective facilities. The benefit here is not that hyper-code enables us to do anything that was not possible with the reflective facilities of Process*Web*; it is that hyper-code offers the ease-of-use that is needed for these facilities to be fully exploited.

5 A Process-aware Architecture Description Language

A software architecture can be seen as a set of typed nodes connected by relations. When describing architectures, the nodes are termed components and the relations termed connectors. These components and connectors and their compositions have specified behaviours, and are annotated with quality attributes. The ArchWare ADL [20] takes the form of a core description language based on the concept of formal composable components and a set of operations for manipulating these components—a component algebra. The key innovation in the ArchWare ADL is the use of mobility to model systems where the topology of components and connectors is dynamic rather than static; new components and connectors can be incorporated and existing ones removed, governed by explicit policies. This focus on architectural evolution at design time and run time distinguishes the ArchWare ADL from others that only deal with static architectures or where the state space of possible changes is known *a priori*. It is this focus on architectural evolution that makes the ArchWare ADL a suitable language for expressing both the architecture of an evolving system and the process of evolving the system. It is for this reason that we refer to it as a process-aware architecture description language (ADL).

The ArchWare ADL is the simplest of a family of languages designed for software architecture modelling. It is based on the concepts of the π-calculus [19], persistent programming and dynamic system composition and decomposition. Indeed, the language design draws heavily on previous work carried out by the Persistent Programming Group at St Andrews on persistent programming languages [17,16,1,18], by the Informatics Process Group at Manchester on process modelling [11,26,27,12] and by the Software Engineering Group at Annecy on formal description languages [4].

The ArchWare ADL is a strongly typed persistent language. The ADL system consists of the language and its populated persistent environment and uses the persistent store to support itself. To model component algebra, the ADL supports the concepts of behaviours, abstractions of behaviours and connections between behaviours. Communication between components, represented by behaviours, is via channels, represented by connections. The language also supports all the basic π-calculus operations as well as composition and decomposition.

6 Example: A Client-Server System

The concepts discussed earlier are illustrated in this section using an example written in the ArchWare ADL. Consider a server that disseminates data about time, date and the position of a satellite. A number of clients may request data from this server.

The functionality of the server and the client can be modelled as abstractions in the ArchWare ADL. When applied, these abstractions yield executing behaviours. Such behaviours are the components that make up the client-server system. The repetitive nature of both client and server functionalities is captured using recursion.

Components interact by communicating via connections. Each component may specify the connections it uses to communicate with others. At the time of composition, these connections may be renamed to make communication possible.

```
! client
recursive value client_abs = abstraction()
{    value c_put = free connection () ;          !
request connection
     value c_get = free connection( string ) ;  ! reply
connection
     via c_put send ;
     ! send request
     via c_get receive s : string ;        ! receive
reply
     via c_display send s ;
     ! display reply
     client_abs()
     ! client calls itself
};
```

In the simple client code above, a client sends a signal via connection *c_put*, then receives a reply via connection *c_get*, and then sends the reply value via connection *c_display*.

In the example server below, the connection used determines the nature of the request. For example, a request received via connection *c_put_s_get_time* will be for time. The server will choose to receive a request from one of the three connections and respond to it.

```
! Global data items to keep count of server activities
value time_count, date_count, pos_count = loc( integer
) ;

! server
recursive value server_abs = abstraction(){
     value  c_put_s_get_time, c_put_s_get_date, !
connections to receive requests
          c_put_s_get_pos = free connection() ;
     value  s_put_c_get_time, s_put_c_get_date, !
connections to send data
          s_put_c_get_pos = free connection(string) ;
     choose{           ! server makes a choice of which request
to service
        { via c_put_s_get_time receive ;   ! request for
time
          via s_put_c_get_time send time ; ! send time
          time_count := 'time_count + 1 } ! increment time
count
     or
        { via c_put_s_get_date receive ;   ! request for
date
          via s_put_c_get_date send date ;       ! send
date
          date_count := 'date_count + 1     }     !
increment date count
     or
        { via c_put_s_get_pos receive ;    ! request for
satellite position
          via s_put_c_get_pos send satellite_position ;
     ! send position
          pos_count := 'pos_count + 1       }} ; !
increment position count
     server_abs()
```

Having defined server and client abstractions, we will now create a client-server system by composing one server and three client instances with appropriate renaming. Note that other topologies are also possible, for example two servers and five clients. Renaming ensures that corresponding client and server connections are matched for communication. Defining the composition as a value gives us a handle (*CS_system1*) to the resulting behaviour.

```
! build client-server system
value CS_system1 = {
     compose{                                ! compose components
        client_abs()
     ! client for time
          where { CS_1_time renames c_put,
                  CS_2_time renames c_get}
     and  client_abs()
     ! client for date
          where{ CS_1_date renames c_put,
                   CS_2_date renames c_get }
     and  client_abs()
     ! client for position
          where{ CS_1_pos renames c_put,
                   CS_2_pos renames c_get }
     and  server_abs()
     ! server
        where{ CS_1_time renames c_put_s_get_time,
               CS_2_time renames s_put_c_get_time,
               CS_1_date renames c_put_s_get_date,
               CS_2_date renames s_put_c_get_date,
               CS_1_pos renames s_put_c_get_pos,
               CS_2_pos renames s_put_c_get_pos }
} ;
```

Once the system starts executing, we may wish to change its structure. Feedback from the system, efficiency concerns and changing requirements can contribute to such a decision. We begin this process by decomposing the system into its component parts. The *with* construct gives us handles to the components.

```
! decompose system
decompose CS_system1 with c1, c2, c3, s1 ;
```

Necessary changes can then be made by evolving or redefining some components. In this case we wish to split the functionality of the server into two by creating two new servers, one serving time alone and the other serving date and satellite position. Therefore we create two new abstractions to replace the old *server_abs*.

Using hyper-code representations of the abstractions will enable us to define the new abstractions to use the current values of the count variables without them having to be stored and explicitly reinitialised.

```
! time server
recursive value time_server_abs = abstraction()
{    value  s_get_time = free connection() ;
     value  s_put_time = free connection( string ) ;
     via s_get_time receive ;
     via s_put_time send time ;
     time_count := 'time_count + 1 ; ! reference to
extant data
     time_server_abs()
};! date and satellite position server
recursive value date_sat_server_abs = abstraction()
{    value  s_get_date, s_get_sat_pos = free
connection();
     value  s_put_date, s_put_sat_pos = free connection(
string ) ;
     choose {
          {  via s_get_date receive ;
             via s_put_date send date ;
             date_count := 'date_count + 1 } ! reference
to extant data
     or
          {  via s_get_sat_pos receive ;
             via s_put_sat_pos send satellite_position ;
             pos_count := 'pos_count + 1 }} ; ! reference
to extant data
     date_sat_server_abs() ;
} ;
```

A new client-server system can then be formed by composing the two new servers with the decomposed clients appropriately.

```
! make new client-server system
value CS_system2 = {
     compose{
          c1 where {CS_1_time renames c_put,
                    CS_2_time renames c_get}
     and  c2 where{ CS_1_date renames c_put,
                    CS_2_date renames c_get }
     and  c3 where{ CS_1_sat_pos renames c_put,
                    CS_2_sat_pos renames c_get }
     and  time_server_abs()
        where{ CS_1_time renames c_put_s_get_time,
               CS_2_time renames s_put_c_get_time }
     and  date_sat_server_abs()
        where{ CS_1_date renames c_put_s_get_date,
               CS_2_date renames s_put_c_get_date,
               CS_1_sat_pos renames c_put_s_get_pos,
               CS_2_sat_pos renames s_put_c_get_pos}
};
```

Now client *c1* will communicate with *time_server* and clients *c2* and *c3* will communicate with *date_sat_server*.

7 Further Work

The example described in section 6 illustrates the core idea of using a process-aware ADL to support the evolutionary development of a system. The ADL has been developed as part of the ArchWare project, which is delivering customisable architecture-based environments for evolving systems. To build an effective environment, several additions are required to the core idea described above. The core ADL is relatively low-level and does not have the domain abstractions to provide the higher-level view required by people evolving the system. The ArchWare project is using the notion of styles to allow users to customise the ADL through the development of abstractions that are appropriate to the specific domain and system. The ArchWare project is also tackling evolution consistency, through the notion of the annotation of an architecture with properties, and providing architecture analysis tools. The essential idea is to enable users to set constraints in terms of architecture properties, and prove that the evolution of a system does not violate any of these constraints.

In providing scalable support for evolving systems, a cellular approach is promising. Each component can have its own evolutionary development process that is responsible for local changes to that component [11]. This may have to make requests to the corresponding development processes of sub-components to achieve its desired change, and to signal to its super-component's process when a required change is not within its control. The ArchWare project builds upon previous work in process instance evolution [12], which provides a process for evolving an engineering process through a network of meta-processes that matches the product breakdown structure (the architecture of the engineering product).

The architecture analysis approach is essentially a pre-emptive approach to managing the evolution of a system. An alternative is a healing approach, typified by autonomic systems, where the focus is on automatically recognising what is wrong and initiating the appropriate actions to resolve the situation [1,2]. To achieve this ongoing management and tuning the ability to control decomposition, evolution and re-composition explicitly is essential. Monitor components receive meta-data arising from probes and gauges, which dynamically measure various aspects of the running system. When deemed necessary, the monitors initiate the evolution of selected components, by decomposing the component assemblies in which they are embedded (producing hyper-code), updating the decomposed hyper-code representation to use alternative components, and recomposing the result. Various policy rules for when, what and how to evolve may be designed to pursue different goals. Indeed these policies may themselves be components that are subject to evolution.

The approach that we have described is not specific to the software development process. In particular the approach can be applied to evolving the evolutionary development process itself. In an ArchWare environment a generic meta-process, such as the Process for Process Evolution P2E [27], can be used to specialise the

evolutionary development process for a component within the system, for example to use specific software development methods or tools.

Another area where we plan to evaluate this approach is in evolving *in silico* experiments in the bioinformatics domain [10]. Much biological experimentation now takes place *in silico*, combining, comparing and collating biological data and analysis tools that are available on the web. An *in silico* experiment is a process, describing the various data retrieval, transformation and analysis steps that yield the required outputs. As more resources become available, and as experimental best practice changes, these experiments evolve. In addition, scientists want to be able to evolve *in silico* experiments as they are running. If an experiment takes hours, or even days, to complete then the scientists involved want the ability to examine the intermediate results, and if necessary make changes to the later stages of the experiment. The use of process technology to provide an explicit representation of *in silico* experiments provides a way of effectively sharing best current practice within a community. An architecture-based approach offers a promising way of composing and evolving larger experiments from smaller experimental fragments. The potential benefits of a hyper-code representation that can be used through the experimental lifecycle, and enable experiments to directly refer to extant data, are very similar to those in the software engineering domain. The bioinformatics domain is also of interest because the coordination of resources into experiments shares many characteristics with the electronic coordination of businesses into business networks [14].

8 Conclusion

In this paper we have identified a class of software systems: long-lived architecture-based systems. Such systems are becoming more common and supporting their evolutionary development is a current key challenge for software engineering. This is a challenge where process technology can make a contribution. The compositional nature of these systems means that the architecture plays a dual role. It structures the (operational) software system, and it structures the evolutionary development of that operational software system. However, it is important to recognise that the architecture is active; it evolves as the structure of the system evolves. Furthermore the evolutionary development of these systems involves transferring important state information, extant data, between the current and the evolved system.

The approach that we have described and illustrated addresses these issues through the novel combination of three features:

- Exploiting the software architecture to structure the evolutionary development process as well as the software itself. This configures the process to its context (the software being evolved).
- The architecture-based software process explicitly represents the composition, decomposition and re-composition that are at the heart of the evolution process. The process is expressed in a process-aware Architecture Description Language (ADL) that has explicit compose and decompose constructs.
- The use of a hyper-code representation so that the current state or context can be effectively managed during the evolution. The unification of source code and

data in hyper-code eliminates different ways of managing code and data evolution, and the hyper-code representation is used throughout the lifetime of the system.

The benefits of this approach are ease of use and ease of understanding. It provides a simpler abstraction over the basic reflective programming facilities for those evolving the system. They are able to concentrate on the essential problems of describing the required evolution. The approach also directly models the compositional nature of the system through the use of compose and decompose primitives in a process-aware ADL. This gives the benefits of a run-time architecture description that is synchronised with the evolving system. Those evolving the system always have an up to date, high-level view of the system, and state of the system within its evolutionary development process.

There are other potential benefits of this approach. The compositional nature encourages the creation of libraries of re-usable software components, and software process fragments. The hyper-code representation means that such extant values in the environment can be easily referenced and re-used. However, the approach is only an essential foundational building block, and further facilities are needed to give an effective environment that supports the evolutionary development process of long-lived, architecture-based systems.

Acknowledgements. This work is supported by the EC Framework V project ArchWare (IST-2001-32360), and the UK Engineering and Physical Sciences Research Council (EPSRC) under grants GR/R51872 (Reflective Application Framework for Distributed Architectures) and GR/R67743 (^{my}Grid: Directly Supporting the E-Scientist). It builds on earlier EPSRC-funded work in compliant systems architectures (GR/M88938 & GR/M88945).

References

[1] Atkinson, M.P. and Morrison, R.: Orthogonally Persistent Object Systems. VLDB Journal 4, 3 (1995) 319–401

[2] Autonomic Computing: IBM's Perspective on the State of Information Technology. IBM, http://www.research.ibm.com/autonomic/ (2002)

[3] Bolcer, G.A. and Taylor, R.N.: Endeavors: A Process System Integration Infrastructure. In Proc. ICSP'4, Brighton, UK, IEEE Comp. Soc. Press, (1996) 76–85

[4] Chaudet, C., Greenwood, R.M., Oquendo, F. and Warboys, B.C.: Architecture-driven software engineering: specifying, generating, and evolving component-based software systems. IEE Proc.–Software 147, 6 (2000) 203–214

[5] Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C., Moore, V.S. and Morrison, R.: Unifying Interaction with Persistent Data and Program. In: Sawyer, P. (ed): Interfaces to Database Systems. Springer-Verlag, In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed) (1994) 197–212

[6] Dowson, M., and Fernström B.C.: Towards Requirements for Enactment Mechanisms. In: Proceedings of the Third European Workshop on Software Process Technology, LNCS 775, Springer-Verlag, (1994) 90–106

[7] Feiler, P.H., and Humphrey, W.S.: Software Process Development and Enactment: Concepts and Definitions. In: Proceedings of the 2nd International Conference on Software Process, Berlin, (1993) 28–40
[8] Finkelstein, A., Kramer, J., and Nuseibeh, B. (eds): Software Process Modelling and Technology. Research Studies Press, (1944)
[9] Georgiadis I., Magee J. and Kramer J: Self-Organising Software Architectures for Distributed Systems. In: Proceedings of the ACM SIGSOFT Workshop on Self-healing Systems, Charleston, South Carolina, (2002)
[10] Goble, C., Pettifer, S., and Stevens, R.: Knowledge Integration: In silico Experiments in Bioinformatics in The Grid: Blueprint for a New Computing Infrastructure Second Edition eds. Ian Foster and Carl Kesselman, 2003 in press.
[11] Greenwood, R.M., Warboys, B.C., and Sa, J.: Co-operating Evolving Components – a Formal Approach to Evolve Large Software Systems. In: Proceedings of the 18th International Conference on Software Engineering, Berlin, (1996) 428–437
[12] Greenwood, M., Robertson, I. and Warboys, B.: A Support Framework for Dynamic Organisations. In the Proceedings of the 7th European Workshop on Software Process Technologies, LNCS 1780, Springer-Verlag, (2000) 6–21
[13] Greenwood, R. M., Balasubramaniam, D., Kirby, G.N.C., Mayes, K., Morrison, R., Seet, W., Warboys, B.C., and Zirintsis, E.: Reflection and Reification in Process System Evolution: Experience and Opportunity. In the Proceedings of the 8th European Workshop on Software Process Technologies, LNCS 2077, Springer-Verlag, (2001) 27–38
[14] Greenwood, M., Wroe, C., Stevens, R., Goble, C., and Addis, M.: Are bioinformaticians doing e-Business? In Matthews, B., Hopgood, B., and Wilson, M. (Eds) In "The Web and the GRID: from e-science to e-business", proceedings of Euroweb 2002, Oxford, UK, Dec (2002), Electronic Workshops in Computer Science, British Computer Society http://www.bcs.org/ewic
[15] Jacobson, I., Booch, G., and Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, 1999.
[16] Kirby, G.N.C., Connor, R.C.H., Cutts, Q.I., Dearle, A., Farkas, A.M. and Morrison, R.: Persistent Hyper-Programs. In Albano, A. and Morrison, R. (eds): Persistent Object Systems. Springer-Verlag, In Series: Workshops in Computing, van Rijsbergen, C.J. (series ed) (1992) 86–106.
[17] Kirby, G.N.C.: Persistent Programming with Strongly Typed Linguistic Reflection. In: Proceedings 25th International Conference on Systems Sciences, Hawaii (1992) 820–831
[18] Morrison, R., Connor, R.C.H., Cutts, Q.I., Dustan, V.S., Kirby, G.N.C.: Exploiting Persistent Linkage in Software Engineering Environments. Computer Journal, 38, 1 (1995) 1–16
[19] Milner, R.: Communicating and mobile systems: the π-calculus. Cambridge University Press (1999)
[20] Oquendo, F., Alloui, I., Cimpan, S., Verjus, V.: The ArchWare ADL: Definition of the Abstract Syntax and Formal Semantics. Deliverable D1.1b, ArchWare IST Project, http://www.arch-ware.org, Dec (2002)
[21] Oreizy, P., Medvidovic, N. and Taylor, R.N.: Architecture-Based Runtime Software Evolution. Proc. ICSE'20, Kyoto, Japan, IEEE Computer Society Press (1998) 177–186
[22] Oreizy, P. and Taylor, R.N.: On the role of software architectures in runtime system reconfiguration. In Proc. of the International Conference on Configurable Distributed Systems (ICCDS 4), Annapolis, MD. (1998)
[23] Process*Web*: service and documentation http://processweb.cs.man.ac.uk/ (accessed on 10 Apr 2003)

[24] Rank, S., Bennett, K., and Glover, S.: FLEXX: Designing Software for Change through Evolvable Architectures. In Henderson, P. (Ed), Systems Engineering for Business Process Change: collected papers from the ERSRC research programme, Springer (2000) 38–50
[25] Sutton, Jr., S.M., Tarr, P.L., and Osterweil, L.: An Analysis of Process Languages. CMPSCI Technical Report 95-78, University of Massachusetts, (1995)
[26] Warboys B.C., Kawalek P., Robertson T., and Greenwood R.M.: Business Information Systems: a Process Approach. McGraw-Hill, Information Systems Series, (1999)
[27] Warboys, B. (ed.): Meta-Process. In Derniame, J.-C., Kaba, B.A., and Wastell, D. (eds.): Software Process: Principles, Methodology, and Technology, LNCS 1500, Springer-Verlag (1999) 53–93
[28] Zirintsis, E., Kirby, G.N.C., and Morrison, R.: Hyper-Code Revisited: Unifying Program Source, Executable and Data. In Proc. 9th International Workshop on Persistent Object Systems, Lillehammer, Norway, (2000)
[29] Zirintsis, E.: Towards Simplification of the Software Development Process: The Hyper-Code Abstraction. Ph.D. Thesis, University of St Andrews, (2000)

Providing Highly Automated and Generic Means for Software Deployment Process

Vincent Lestideau and Noureddine Belkhatir

Adele Team
LSR-IMAG , 220 rue de la chimie
Domaine Universitaire, BP 53
38041 Grenoble Cedex 9 France
{Vincent.Lestideau, Noureddine.Belkhatir}@imag.fr

Abstract. We present a new approach for the management and enactment of deployment process by a deployment processor ORYA (Open enviRonment to deploY Applications). ORYA aims to integrate technology relevant to process support in deployment systems. The supposed context is a large scale networked organization. The deployment processor called ORYA provides deployment functionalities to distributed, autonomous, reactive processing entities representing workstations and servers. Based on these functionalities deployment processes can be enacted and monitored in an efficient manner. In this paper we focus on the distributed, multi-server, multi-client sites architecture of ORYA and the evaluation of its application to a real industrial case study dealing with a large transportation information system.

1 Motivation

Software deployment is a an ordered set of activities (software process) (Fig. 1), that will be carried out over time to deploy a software. It is generally described as a life cycle with encompasses several sub-processes as : Release, Install, Activate, Deactivate, De-install, De-release, Update and Adapt [3]. Deploying a software to a large scale networked user environment involves many activities like sites introspection and test for needed components, modification of various configuration files to establish the conditions needed by the new software, transfer…

During the last few years, software deployment has been the focus of intense activity in terms of products, standards and research work [16] for networked environments. The networked organizational structure of the companies implies new requirements on deployments systems. The current generation of deployment systems placed more emphasis on automation of process deployment in decent rally and distributed large installations of servers and stand-alone workstations.

In this paper, we propose a novel approach to build flexible, customizable and enactable software deployment systems based on the concept of executable descriptions. These objectives reduce considerably the cost of large scale deployment setting. The key point here is that the deployment domain can benefit from process technology. The separation of process description from the monolithic tools would

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 128–142, 2003.

result in greater visibility of the process and greater flexibility. In this paper we consider this new approach in the light of a real industrial experience we carried out in software deployment for a large and complex transportation application. This paper gives an overview of this experimentation and goes on describing ORYA.

Section 2 presents the overview of the deployment processor ORYA. Section 3 gives more insight of the deployment process model. In section 4 we present the experimentation with an industrial case study in a transportation information system. Before the conclusion, the Section 5 presents a first preliminary evaluation of our experience and some related works.

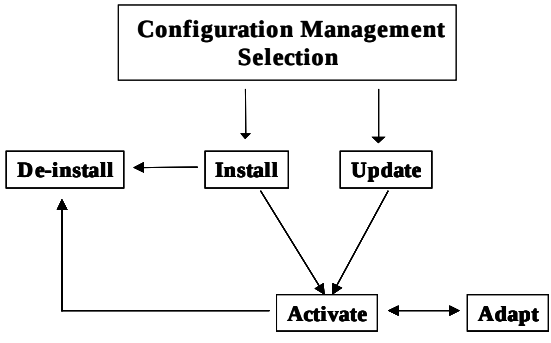


Fig. 1. The deployment life cycle

2 ORYA Overview

Installing an application on one site can be seen as a simplistic manual activity, however if it is necessary to add some configuration requirements (like installing the more compatible version of the application) the activity becomes more complex. The installation remains always realizable manually from the target installation site or from a distant one. If moreover, we want to install the same application on many sites at the same time (but under different versions according to the installation sites), the installations become impossible to be performed manually. For this reason we have created a deployment PSEE [2] to completely automate the deployment process. In the following sections we describe our deployment environment and followed by the models used to allow the automation.

2.1 Architecture and Design of ORYA

2.1.1 Topology

For automation purposes, we have introduced three entities allowing to deploy applications on one or more sites; an entity to store the applications, another to represent the sites and a last one to manage the deployment:

- **Application Server (AS)** : It is a repository of packaged applications. A package is composed of a set of typed resources (data, scripts…) representing the application,

a set of executable files allowing to install the application and a manifest describing the application in terms of dependencies, constraints and features (name, version…)

- **Target (T)**: It is the deployment target, where the applications have to be installed and executed. It can be a computer or a device (a stamping ticket for instance). Each target is described by a site model in terms of already deployed applications and physical description (disk space, memory…). This description permits to deploy and to install the application in the more consistent way according to the site configuration. Many user profiles can be taken into account.
- **Deployment Server (DS)**: It is the core of our deployment environment and is responsible to manage deployment. The DS searches the appropriate package to be deployed on the target, performs the package transfer (even through out firewalls) and guaranties that the new installed application works correctly. The DS has to deal with several problems such as dependencies or shared components, and must ensure that other existing applications continue to work.

2.1.2 The Environment

Figure 2 presents the environment in which we have experimented our deployment process.

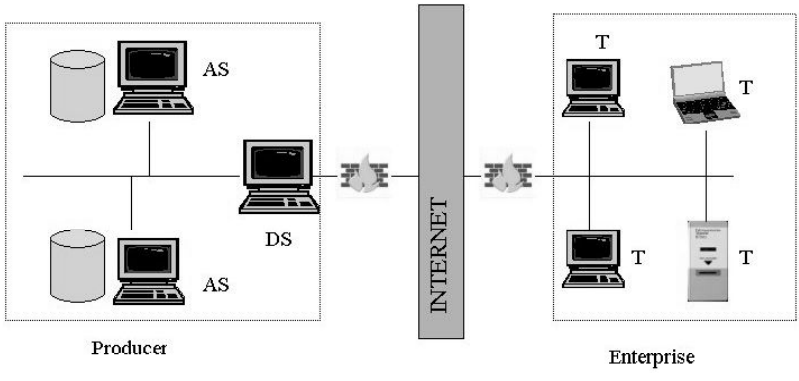


Fig. 2. Deployment environment

The producer hosts a set of application servers and one deployment server. It is connected to Internet via a potential firewall. The deployment targets belong to an enterprise. The enterprise can be also behind a firewall. The goal of this experiment is to automate (via Internet) the deployment towards the enterprise.

2.2 Deployment Process: Models and Concepts

To automate the deployment, it is necessary to define clearly many models for the application to be deployed, the deployment process, the targets, the application servers, etc. In this paper only the two first models are considered: the deployment process model (based on a process model) and the application model.

2.2.1 Deployment Process Model

Figure 3 describes an implementation of a process model [13] related to deployment. An activity can be either a basic activity or a complete deployment process. The transfer or the unpackage activities are examples of basic activities. A deployment process is a composite activity, made by some ordered basic activities and deployment processes. Thanks to this generic model it is possible to describe in particular the activities (install, update, uninstall…) of the deployment life cycle (see the Fig. 1).

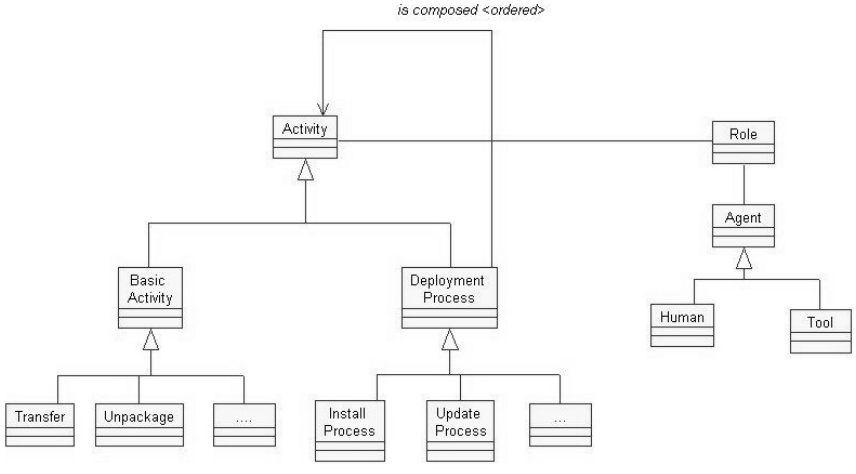


Fig. 3. Our deployment process Model

Each activity is played by a role executed by an agent. An agent can be either a human-user or a tool. In our case, we try to use often tools to automate as maximal as possible the deployment process. However in some situations a human-user is needed; for instance to allow an administrator to validate the deployment.

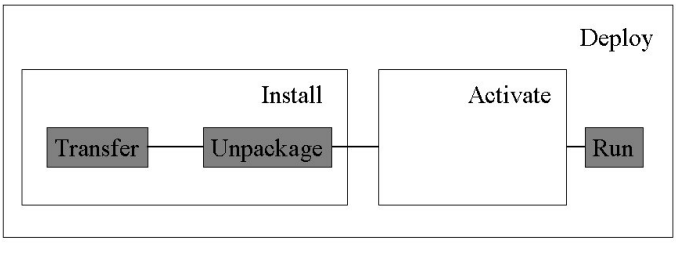


Fig. 4. A simple example of our deployment process

Figure 4 is a simple example of a such deployment process model. A basic activity is represented by a gray box and a deployment process by a white box. In this example, we describe an activity (Deploy) composed of two deployment processes (Install and Activate) and a basic activity (Run). The Install activity is also composed of two basic activities (Transfer and Unpackage).

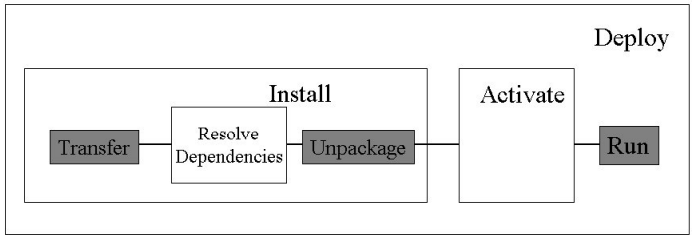


Fig. 5. A more complex example of our deployment process

The same activity can be more complex (Fig. 5). For instance, the Install activity can be composed of two basic activities and a deployment process(Resolve Dependencies), whose function is to resolve the dependencies of the application. Section 3 shows a practical example we have tested in an industrial environment.

2.2.2 Application Model

This application model (Fig. 6) describes an application from the deployment point of view and more particularly in the case of installation. Consequently our model does not contain some information (like connection between components) generally present in other application models like CCM [11].

Many information are necessary:

- Information on the application itself (specified in a manifest) like the application name, the producer name, the version number, the license number…
- Information on resources composing the application (code, components…) like the component version, its location…
- Deployment constraints. Two types of constraint can be distinguished: software and hardware constraints. An application "A" that can not be deployed if another application "B" is already installed is an example of a software constraint. A hardware constraint implies a particular hardware configuration (memory, space disk, etc.) needed by the application.
- Software dependencies. A software dependency is another necessary application for the execution of the application to be deployed. A dependency can be light or strong. A strong dependency must be resolved during the deployment (if not the deployment is stopped) and a light dependency can be resolved after the deployment. This model is also used to create the application package

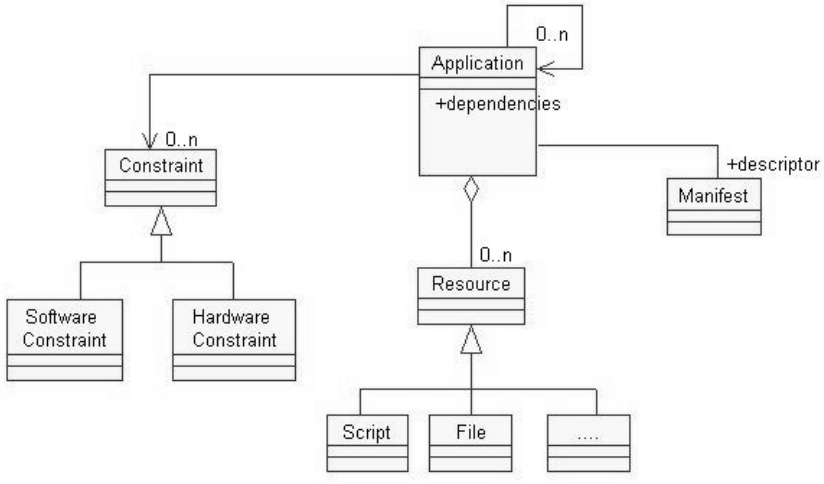


Fig. 6. Our application Model

3 Process Modeling

In this section we decompose the deployment into two parts. The first one represents all the activities performed before the physical installation. The second part describes the physical installation. It is important to precise that all the process is managed by the deployment server and can be executed remotely. All the processes presented in this paper has been specified using the Apel tool [4]

3.1 The Deployment Process

The goal of the process presented in Fig. 7 is to install an application on only one target. In the entry point of the process we precise the application name and the target name. In our environment we have several application servers hosting several packages. The process consists of finding an application package among many ones. The selected package must answer the deployment requirements: assures that application will work correctly and that other existing applications will not be affected. The process must then resolve the dependencies, transfers and finally installs the application.

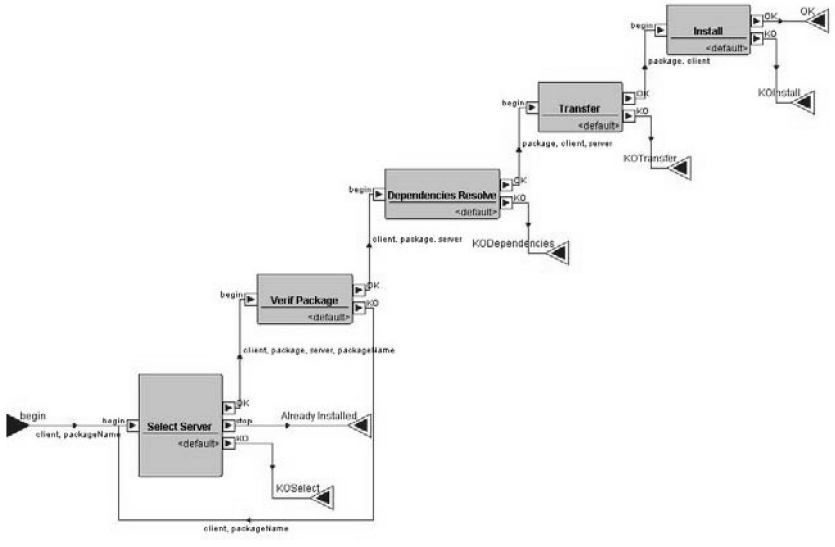


Fig. 7. The deployment process

The presented process is composed of many deployment processes (install…) and many basic activities (transfer…). These activities will be described in more details in the following sections.

3.1.1 Select Server Activity

This basic activity () receives in entry the target and the application names. Its goal is to prepare the deployment. First it verifies that the application is not already installed (otherwise the deployment process is stopped (stop port in the figure)). Then the deployment server looks for an application server containing one of packages representing the application. Once an application server is found it must verify that the package can be deployed on the target (the activity is terminated by the OK port). If the package cannot be deployed another package will be searched. If no package is found (or if no package can be deployed) the activity is stopped (terminated by the KO port) and the deployment process is finished.

3.1.2 Verif Package Activity

This basic activity (Fig. 7) receives in entry the package, the target, the application and the application server names. Its goal is to verify that the package can be deployed on the target. These verifications concern more particularly hardware constraints like memory or disk space. We distinguish software constraints and software dependencies. Unlike a software dependency resolved during the next activity, a software constraint must be verified before the installation. All software and hardware constraints are specified in a special file of the package: the application descriptor. If at least one of these verifications is not satisfied, the activity is interrupted (terminated by the KO port). In this case, the process go back to the previous activity to look for another package. In the other case the activity is

successfully finished (terminated by the OK port) and the process continues to resolve the dependencies of the package thanks to the next activity.

3.1.3 Dependencies Resolve Activity

It receives in entry the result of the previous activity: the target and the application server names and the package and checks the software dependencies. A dependency must be resolved during the deployment (in case of strong constraint) or after the deployment (in case of light constraint). For each dependency a checking is done to verify if it is not already installed on the target. If the dependency is not installed the deployment server executes a new deployment process to deploy the dependency on the target. If problems occur during the process related to the dependency, many strategies tell if the initial process must be stopped or not. If all dependencies are resolved, the activity is successfully finished and the transfer activity can be started.

3.1.4 Transfer Activity

It realizes the transfer of the package from the application server to the target. In reality the transfer can be achieved in two steps: first the deployment server retrieves the package from the application server and then it transfers it to the target. If this activity fails (network breakdown for instance), the deployment can be stopped or delayed. The install activity can started if the transfer activity is successfully terminated.

3.1.5 Install Activity

It is a complex process and is responsible of the physical installation. In our vision an installation is composed of an extract activity of the package, a set of installation steps (“Install-Step”) and a set of uninstallation steps (“Uninstall-Step”). A step is composed of a script to be executed and some associated verifications. If a problem occur during the installation the process must be able to return to the initial state. Therefore the install activity is also composed by some uninstall steps. The uninstall is not a basic activity, because potential problems may occur at different moments of the deployment and the uninstall process can be different according to this moment. The next paragraph is dedicated to the install process.

3.2 Focusing on the Install Process

As seen before, the install process (Fig. 8) is one activity of the general deployment process (Fig. 7). To our point of view an installation can be decomposed in several steps. This decomposition responds to the need of carrying out checks during the deployment. Each checking (or checking set) implies the creation of a step in the installation. It is supposed that a step corresponds to the execution of a script followed by a checking associated to the result of this script. All these information are describing in an XML Schema file (named “manifest”) containing in the package. A complete description of this XML schema [18] is beyond the scope of this paper. The package contains also data corresponding to the application. Now, we describe the different activities belonging the install activity.

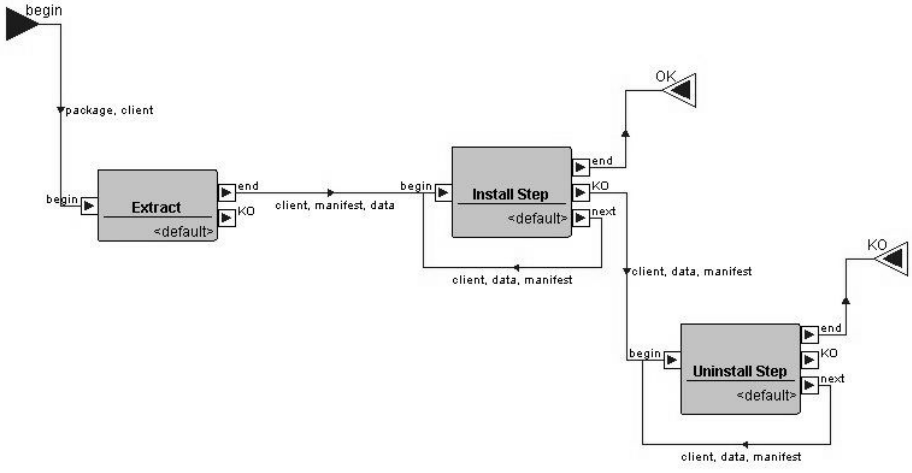


Fig. 8. The install process

3.2.1 Extract Activity

It extracts the package on the target in a temporary folder. Once extracted the manifest informs the process how many install/uninstall steps composed the install process and how these steps can be performed. The package contains also some resources, scripts and checking (designated as "data" in the process).

3.2.2 Install Step Activity

Each install step activity is composed of two basic activities (Fig. 9): execution and checking. The former executes the script related to the install step and the latter performs the necessary verifications.

If at least one step does not succeed, the install process must be interrupted. In this case the process must undo all the modifications already done; this is the responsibility of the uninstallation process. In case of success of all install steps, the installation is successfully finished as well as the deployment process.

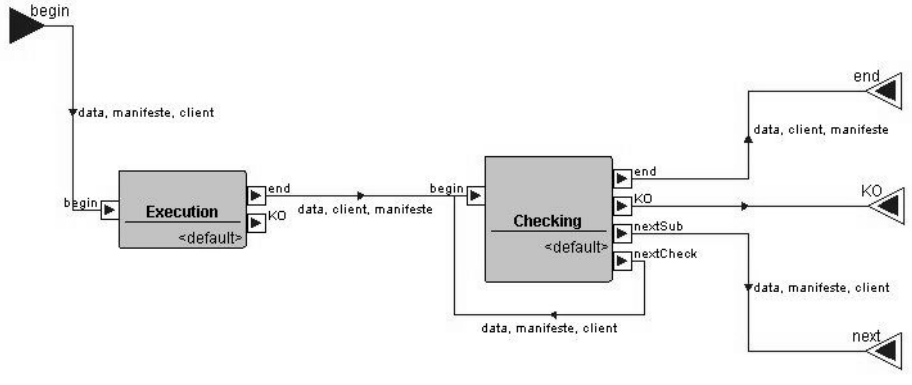


Fig. 9. Sub-installation process

3.2.3 Uninstall Activity

If foreseen problems occur during the installation the uninstall process must return the target to a consistent state. In others words it must undo all already modifications. The XML Schema associates to each install step an uninstall one. The uninstall process is composed of symmetric activities to those of the install process.

4 Experience with a Real Industrial Case Study

As part of our deployment works, we collaborate with the Actoll Company [14]; a software company specialized in the development of transportation information systems. The goal of this collaboration is to study and implement the deployment process of a large scale federation application in transportation area. Transportation companies use this application to solve problems with federation of transportation systems (railway, highway, airline…). First, we present the application (in terms of structure), the requirements for the deployment and our deployment architecture.

4.1 Centr'actoll Application

This application is composed of a set of seven servers (Fig. 10). Each server has some software dependencies and to install them, some hardware constraints (like memory or disk space) must be checked. Some of these servers are built on database technology.

Before our collaboration managers deploy their applications in a manual way. The deployment is a hard task with many verifications during the deployment itself (for instance, to check service NT creation) and for the majority of them, they are realized manually by a user behind each physical site. Actoll company wishes to automate the deployment of all servers, via Internet and without manual interventions.

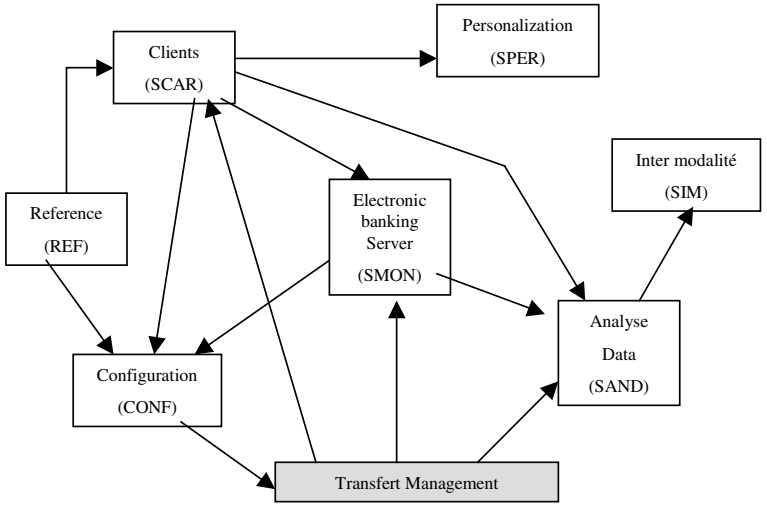


Fig. 10. Architecture of Centr'Actoll Application

Figure 10 shows the general application of Centr'Actoll. This application exists in different versions (according to the number of servers used). These servers may have some dependencies between them. These dependencies are represented in Fig. 10 by the arrows. It exists other dependencies for a server than the other servers. For instance the SCAR depends of ORACLE and TOMCAT software. And each server has some constraints like the space-disk size (of 1Go to 20Go), the OS (Windows), X25 network cards, etc.

From this description of the architecture, we have created the package of each server with its description and we have tested our deployment environment for this application in the real context of the Actoll enterprise. In addition to deploy the application, we have identified some deployment requirements (see the next section) and realized an implementation of our environment trying to respect all of these requirements (distribution, monitoring…). We have used two technologies or tools developed in our team:

- A process engine with its graphic process creation tool named Apel.
- The federation technology allowing to execute and to control in a distant way a tool even over firewalls [5] [10].

4.2 Requirements

We have identified several requirements such as:

- **Configuration**: The application can be installed according to different configurations; for instance one machine by sub-application or several sub-applications by machine.
- **Distribution**: Deployment must be possible either via Internet (even in case of firewalls) or via a local network (in particular for some test and developing reasons).
- **Automation level**: The deployment system must be able to offer different (and customizable) levels of automation. For instance, deployment can be completely automated or contrarily each deployment step must be validated.
- **Security level**: Generally, client company doesn't wish that other companies install some applications on their network. In this case, our system must be able to interact with administrators to validate the wished (by producer company, i.e. Actoll). This requirement is strongly associated with automation requirements (see above).
- **Scheduling & Performance**: In case of large scale deployment, a deployment system must be able to handle different strategies like big-bang (application is deployed on all the target machines), or incremental (application is deployed on one (or on group of) machines). Network characteristics (like bandwidth…) must be also taken into account during the deployment.
- **Monitoring**: We must be able to monitor the deployment.
- **Roll Back**: If the deployment doesn't work for a machine, we must inform the system and/or administrators and we must undo all deployment activities already realized to come back to the old configuration.

- **Tests and Verifications**: In case of the deployment of an application, some verifications must be realized at the end but also during the deployment. This verifications affect the next deployment steps.

5 Evaluation of the Experience

Many ad-hoc solutions and tools for deployment have been proposed providing monolithic and hard coded solutions. Current deployment systems do not support the explicit definition of the deployment process independently from the tools and do not let to customize features like adding specific deployment tools. Deployment facilities are either not powerful enough and cannot be sufficiently tailored to application needs or they provide too much functionality which in most cases remains unexploited because unsuitable but increases the system cost and affect the system performance.

In several industrial organizations, the complexity of deployment is becoming such that more systematic approaches are required as the limitations of ad-hoc approaches experienced daily. More advanced requirements arise and request the customization of the system to the evolving and specific application needs.

For example, in our case study, an application implementing the Actoll process requires the basic PSEE functionality and the ability to tailor the process and integrate specific deployment tools. Administrator can customize each of the specific tools and integrate them in the core. For example they can introduce a new introspection tool specific to a new site configuration. Likewise, they can have the process uses a specific packaging tool. This requires process functionality which integrates with deployment tools executed in a networked environment.

This characteristic is identified as one of the requirements for a new generation of deployments system architecture.

These reasons have led to the ORYA approach. ORYA is a highly customizable deployment processor which can be enhanced with specific tools and process. ORYA has the ability to be changed and configured for use in a particular context.

We have presented a solution outline for making large scale deployment. This approach leads processor make it possible to control the way tools interact with the engine, as such making it possible to ensure consistency.

Finally the economic of our approach has been studied. Particularly for complex applications, the advantages that can be gained using a deployment processor are high. The overhead for deploying applications has been reduced considerably from one week (manually) to one hour to deploy a server using the automation deployment processor ORYA. The productivity and quality of service of the system manager was increased because more applications are available with a better level of consistency. On this way, the system manager can devote more time to conceptual activities.

In summary the case study discussed in this paper has proven that ORYA can implement deployment process with different requirements. Additionally, the quantitative evaluation in terms of design and run time costs shows that the automation provides a viable alternative for application administrators who need customizable deployment functionality.

5.1 Related Works

Deployment has been an issue for years. Many solutions have been proposed and implemented. Many existing industrial deployment tools exist like Tivoli [17] or InstallShield [15]. Often they are bound to one deployment process and they offer a unique deployment solution (to one target) or a multi-site deployment but with ad-hoc solutions. However these existing tools are often very efficient in some case (install process, configuration process…) but they do not cover all the deployment life cycle. ORYA allows to integrate these tools and to benefit of their capacities. There are research efforts to provide new technologies [1] [9] to integrate support for the whole deployment life cycle and addressing complex issues as dynamic adaptation for example [7] [8]. Some standardization works like [12] deal with some specific application models (component based models). None of these approaches provide a suitable software installation and common system for several companies. Software Dock [6] is an example that deploys generic applications to one target with a hard coded deployment process. Software dock has been developed in the University of Colorado. The aim of software dock is to provide a general solution for the deployment life cycle. It allows describing, using the DSD (Deployable Software Description) formalism, a number of valid configurations.

Software Dock
ORYA shares several characteristics with Software Dock. They both propose new deployment architecture covering the entire deployment life cycle. However many features set ORYA apart from Software Dock.

- A key feature of ORYA design stems from the fact that our approach is more generic. For instance we offer the possibility to the administrator to describe the install process in terms of executions, strategies and verifications. However in software dock the process and deployment strategies are hard coded and cannot be modified.
- Another main issue is the integration into the deployment process of tools (legacy) which encapsulate different concerns . In this context, we provide a framework for the development of interfaces between the process system and applications (legacy tools). On this way; we separate process logic from activity logic which is embedded in user tools. This feature is not present in S/W Dock.
- A federated process component in ORYA that provides the support required to transparently distribute process execution across the enterprise. This feature is not present in S/W Dock.
- The coarse granularity level of the deployment process in software dock is the activity (install, update, adapt…) of deployment life cycle whereas in our approach the level is more fine.
- ORYA relies on a PSEE that provides process functionalities whereas software dock relies on a agent based platform.

Finally in contrast with software dock, ORYA aims at providing a generic and flexible architecture.

6 Conclusion

The paper presents a new deployment platform based process deployment engine which monitors the global architecture of a system and manages and executes the provided process model. Our approach is characterized by some important features :

- An open environment offering mechanisms to integrate legacy deployment tools and to choice dynamically the most adapted tools during the deployment.
- A process sensitive based system offering support to define, execute and monitor deployment process.
- A general purpose application model shared across the network. Could be instantiated for specific application models (for instance a component based model).
- ORYA puts an emphasis on large scale deployment (deploying applications on many targets sites).
- The design of ORYA takes multi-platform capability into account, and it is successfully being used in a mixed environment for instance Windows / Linux / MacOS.

In this paper we have described our generic platform and we have applied and implemented it on a real industrial case study: a transportation information system. We have shown by some scenarios the different facets of the deployment process. This research extends applicability of process systems since they haven't been applied to the deployment. We propose a generic deployment process support kernel to provide support for process management and execution. The first application of ORYA generates a positive feedback from the software administrator, responsible of the deployment tasks.

The contribution of this paper can be summarized as follows:

- It describes a novel architecture for a distributed environment providing the functionality for deployment process execution and monitoring.
- It presents and evaluates the results of a real case study

Further work remains necessary to cover the whole life cycle of deployment.

References

1. N.Belkhatir, P.Y. Cunin, V. Lestideau and H. Sali. An OO framework for configuration of deployable large component based Software Products. OOPSLA 2001. Tempa, Florida, USA. 14-18 October 2001.
2. N. Belkhatir, J. Estublier, and W. Melo. ADELE-TEMPO: An Environment to Support Process Modelling and Enaction. book Software Process Modelling and Technology, pages 187–217. John Willey and Son inc, Research Study Press, Tauton Somerset, England, 1994.
3. A.Carzaniga, A. Fuggetta, R.S. Hall, A. van der Hoek, D. Heimbigner, A.L. Wolf. A Characterization Framework for Software Deployment Technologies. Technical Report CU-CS-857-98, Dept. of Computer Science, University of Colorado, April 1998.

4. S. Dami, J. Estublier and M. Amiour. APEL: A Graphical yet Executable Formalism for Process Modeling. Kluwer Academic Publishers, Boston. Process Technology. Edited by E. Di Nitto and A. Fuggetta..Pages 61–97. January 1998.
5. J. Estublier, A. T. Le and J. Villalobos. Tool Adoption Issues in a Very Large Software Company-ACSE 2003, Portland, Oregon, USA, May 2003.
6. R.S. Hall, D.M. Heimbigner, A. van der Hoek, and A.L. Wolf. An Architecture for Post Development Configuration Management in a wide-area network. In Proceedings of the 1997 International Conference on Distributed Computing Systems, pages 269–278. IEEE Computer Society, May 1997
7. M. Ketfi, N. Belkhatir and P.Y. Cunin. Automatic Adaptation of Component-based Software Issues and Experiences. PDPTA'02, Las Vegas, Nevada, USA, June 2002
8. M. Ketfi, N. Belkhatir and P.Y. Cunin. Dynamic updating of component-based applications. SERP'02, Las Vegas, Nevada, USA,June 2002
9. V. Lestideau, N. Belkhatir and P.Y. Cunin. Towards automated software component configuration and deployment, PDTSD'02, Orlando, Florida, USA, July 2002.
10. A.T. Le, J. Estublier and J. Villalobos. Multi-Level Composition for Software Federations. SC'2003, Warsaw, Poland, April 2003
11. J. Mischkinsky - *CORBA 3.0 New Components Chapters* – CCMFTF Draft ptc/99-10-04 -- OMG TC Document ptc/99-10-04 October 29, 1999
12. Specification for Deployment and Configuration of Component-based Distributed Applications. Proposal to the OMG MARS RFP: Deployment and Configuration of Component-based Distributed Applications. http://www.omg.org/cgi-bin/doc?mars/2003-03-04
13. The Workflow Management Coalition Specification- The Workflow Reference Model, january 1995, http://www.wfmc.org/standards/standards.htm
14. Actoll Enterprise Web Site: http://www.actoll.com/en/presentation.htm
15. Web Site InstallShield : http://www.installshield.com/default.asp
16. Object Management Group. Web Site : http://www.omg.org
17. Web site Tivoli, http://www.tivoli.com
18. XML Schema Web Site : http://www.w3.org/XML/Schema

Flexible Static Semantic Checking Using First-Order Logic

Shimon Rura and Barbara Lerner

Williams College, Computer Science Department,
Williamstown, MA 01267 USA
{03sr, lerner}@cs.williams.edu

Abstract. Static analysis of software processes is important in assessing the correctness of processes, particularly since their long duration and distributed execution make them difficult to test. We describe a novel approach to building a static analyzer that can detect programming errors and anomalies in processes written in Little-JIL. We describe semantic rules declaratively in first-order logic and use xlinkit, a constraint checker, to check the processes. We have used this approach to develop a checker that can find simple syntactic errors as well as more complex control and data flow anomalies.

1 Introduction

Formal descriptions of software processes can help us understand, analyze, and execute complex processes. But the value of process descriptions depends on their correctness: an invalid process description is unusable, and an improperly specified process description can be misleading. To help process developers verify the validity of their processes and catch some common mistakes, we developed a static semantic checker which checks a process against a set of rules expressed in first-order logic. Our approach quickly yielded a tool that catches simple language errors. More surprising, however, is that this simple approach has scaled to more challenging static analysis problems and has the potential to provide a lightweight yet effective checking framework for many phases of process development and analysis.

The semantic analysis that we describe was built for Little-JIL [9], a process language being designed and developed by researchers in the Laboratory for Advanced Software Engineering Research at the University of Massachusetts, Amherst and at Williams College. It has a simple graphical syntax, which allows for an intuitive visual representation of process structure, and well-defined semantics designed to be rich enough to allow analysis and execution of complex processes. Static analysis is particularly valuable for process developers, because the prolonged and distributed nature of most processes can make testing, as well as more formal dynamic analyses, prohibitively expensive. To this end, we are pursuing several projects that enable static analysis of Little-JIL processes. In this paper, we describe a novel approach to static analysis. Using xlinkit [6], a COTS constraint checker, we have developed a checker that examines Little-JIL processes and detects a variety of anomalies, including language violations, race conditions, lost data, and infinite recursion. The semantics to be checked are expressed as a set of rules written in first-order logic. This approach has made it easy to turn assertions about program

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 143–153, 2003.

structure into the parts of a working checker. The checker's architecture, which keeps semantic rule declarations separate from verification and reporting procedures, makes it easy to extend the checker with new anomaly detection rules and to adjust existing rules if language features change.

2 Overview

A Little-JIL process is represented as a hierarchy of steps, expressed in a simple graphical syntax. The Little-JIL interpreter uses this description to coordinate the flow of tasks and information between agents, which may be human or automated and interact with the interpreter via an API or GUI. Steps are connected with a number of different edges: substep edges, prerequisite and postrequisite edges, reaction and exception handling edges. The characteristics of steps and edges determine how data and control flow between agents. (The Little-JIL Language Report [8] describes the language in detail.)

A developer creates a process using a specialized Little-JIL editor by creating and manipulating steps, represented graphically, and links between steps, represented as lines. Each step has an icon (the *sequencing badge*) that describes the control flow for its substeps. For example, a step with the sequential badge completes when all of its substeps have completed, while a choice step completes when exactly one of its substeps is completed. (The choice of which substep to perform is deferred until runtime). A leaf step, which cannot have substeps, is dispatched to an external agent which notifies the Little-JIL interpreter via its API when the step is completed.

The process programmer provides the details of a step's interface and the data passed between steps by selecting a step or edge and editing its properties in a separate form. The visual representation of a process displays step names, sequencing badges, and the edges that connect steps to their substeps and exception handling steps, but suppresses additional information about step interfaces, prerequisites and postrequisites, and data flow between steps. Thus while the process diagram makes it easy to check that substeps are connected as intended, it is more cumbersome to verify other important properties, such as that all input parameters to a step are bound.

The editor enforces proper syntax and performs some static semantic checks. The editor generally prevents the introduction of constructs that would violate language semantics. For example, it will not allow the process programmer to create a parameter binding that results in a typing error. The editor's incremental checking is best suited for preventing the creation of inconsistent processes; it is not well suited to ensuring that a process description is complete. For example, while it guarantees that all parameter bindings are well-typed (a consistency check), it does not check whether all the input parameters of a step are bound to some value (a completeness check). Since Little-JIL is an interpreted language, there is no compiler to detect these errors. As a result, without an additional semantic checker, many violations of language semantics can go undetected until runtime.

In addition to linguistic errors such as these, a programmer might use the language in a way that is technically correct but suggests that the process is incomplete. For example, sequential and choice steps are expected to have substeps, but the language defines the behavior of these steps when they have no substeps. If a sequential step

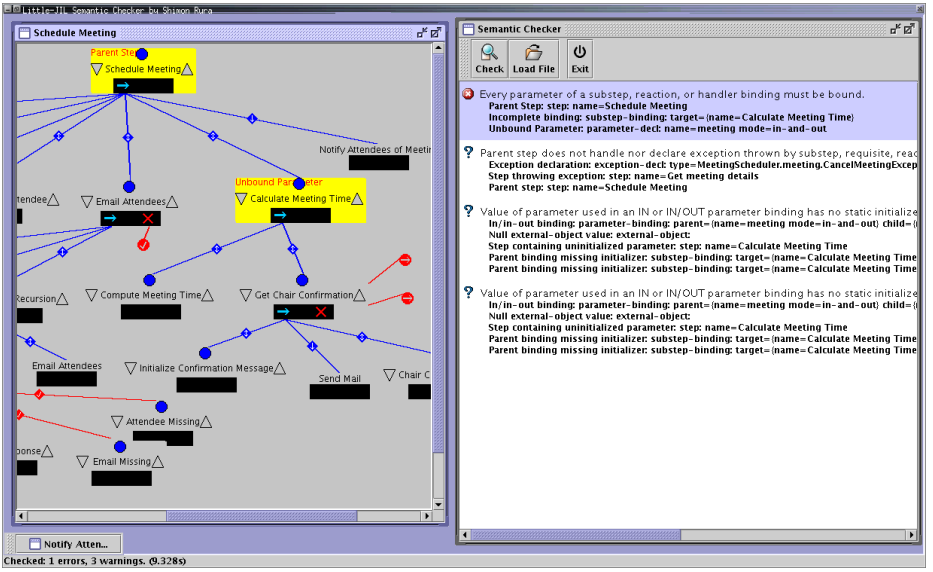


Fig. 1. Reporting errors in the Little-JIL Semantic Checker

with no substeps executes, it will complete successfully since all of its (zero) children complete successfully. If a choice step with no substeps executes, it will always fail because a choice step requires exactly one substep to complete successfully. The use of non-leaf sequencing badges when a step has no substeps suggests a programming error. This type of anomaly is not detected by the editor, but merits a warning before execution.

In considering how to address these issues of checking static semantics, we decided to use a novel approach that would allow us to express the language semantics declaratively rather than use Java to extend the editor with a semantic checker written in a more traditional style. The benefits of this approach stem from the more concise and encapsulated representation of the semantic rules. Rather than needing to extend multiple parts of the editor to support each semantic check, we can express each semantic rule purely in terms of program structure. This makes semantic checks easier to develop, understand, and maintain.

Figure 1 shows an example of the semantic checker in operation. The top panel displays the Little-JIL process as it appears in the editor, while the bottom panel displays the error and warning messages. When the user clicks on an error message, the steps of the process that are related to the error message are highlighted as shown.

In the remainder of this paper, we present more details on this mechanism of describing the semantic rules declaratively and our experiences with the semantic checker.

3 Encoding Rules in First-Order Logic

The Little-JIL static semantic checker makes use of xlinkit [6], a commercial constraint checking tool. Given one or more documents and a set of consistency rules expressed

```
<consistencyrule id="warnNonLeafNoChildren">
   <forall var="nonLeaf" in="//step[@kind != 'leaf']">
      <exists var="sub" in="$nonLeaf/substeps/substep-binding"/>
   </forall>
</consistencyrule>
```

Fig. 2. xlinkit Consistency Rule to Find Non-Leaf Steps with No Substeps

as first-order logic predicates relating document elements, xlinkit attempts to satisfy the rules across the documents. If a rule cannot be satisfied, xlinkit reports an inconsistency, identifying the document elements for which the rule cannot be satisfied.

To use xlinkit, we generate an XML representation of the Little-JIL process and define our semantic rules using the consistency rule schema provided by xlinkit. The consistency rule schema allows us to express first-order logic expressions using XML syntax. Within the rules, XPath [11] expressions are used to select sets of elements or values from the document. In our case, the document elements correspond to Little-JIL language elements, and the XML document structure mirrors the process structure. Thus the encoding of many conditions is quite straightforward. To report errors to the user, we map the XML document elements involved in an inconsistency back into the steps of the process so that they can be highlighted as we display the corresponding textual error message as shown in Figure 1.

For example, the logical rule that asserts that each step with a non-leaf sequencing badge should have one or more substeps can be expressed in first-order logic as:

$$\text{let } NonLeaves = \{s | s \in Steps \wedge kind(s) \neq Leaf\} \text{ in}$$
$$\forall\, nonLeaf \in NonLeaves, |substeps\,(nonLeaf)| > 0$$

In this rule $Steps$ denotes a set containing all the steps in the process, the function $kind$ reports the kind of sequencing badge the step has, and the function $substeps$ reports the set of substeps that a step has.

To encode this rule in xlinkit's consistency rule schema, we must translate the first-order logic syntax into XML[10]. Within the rules, we use XPath[11] expressions to denote the set of non-leaf steps and the sets of substeps of these non-leaves. The XML encoding of this rule is shown in Figure 2. `//step[@kind != 'leaf']` is an XPath expression that denotes the set of all steps not designated as leaves and `$nonLeaf/substeps/substep-binding` is the set of all substep edges coming from a step $nonLeaf$. To verify this rule, xlinkit will attempt to associate each non-leaf step with a substep binding. These associations are called *links* and when a link cannot be completed, xlinkit reports it as inconsistent. In this case, xlinkit will report an inconsistency for each non-leaf step that has no substep bindings. For each inconsistency, it will identify the non-leaf that violates the consistency rule.

Figure 3 shows the checker output when run on a process where one non-leaf step (the root) has substeps, while another (named `Bad NonLeaf`) does not. The error panel provides a textual message and identifies the step violating the rule by name. The left

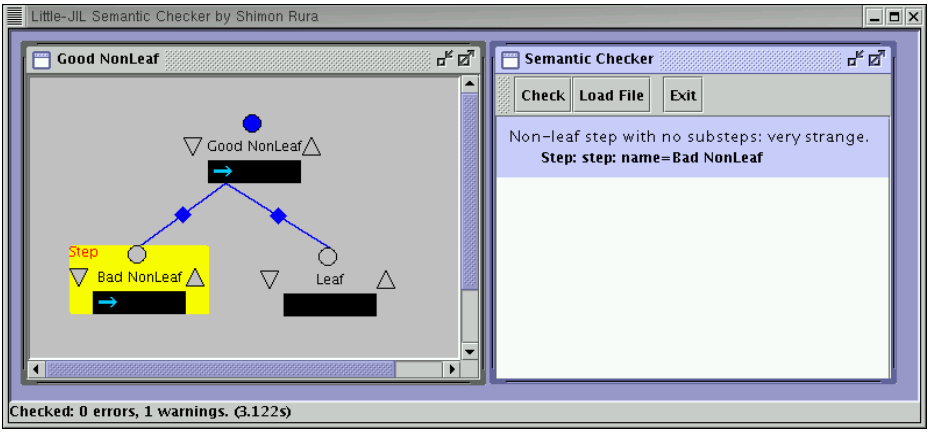


Fig. 3. Violation of the Rule that Non-Leaves must have substeps

```
<?xml version="1.0"?>
<!DOCTYPE program
  SYSTEM "http://www.cs.williams.edu/~lerner/littlejil.dtd">
<program root="Diagram1">
  <diagram root="Step1" id="Diagram1">

    <step name="Good NonLeaf" kind="sequential" id="Step1">
      <substeps>
        <substep-binding target="Step2"></substep-binding>
        <substep-binding target="Step3"></substep-binding>
      </substeps>
    </step>

    <step name="Bad NonLeaf" kind="sequential" id="Step2">
    </step>

    <step name="Leaf" kind="leaf" id="Step3">
    </step>

  </diagram>
</program>
```

Fig. 4. XML Representation of a Simple Little-JIL Process

panel highlights the step visually. The XML that corresponds to this process is shown in Figure 4.

When the rule is applied to the XML representation of the process, there are two steps that satisfy the forall clause: the root and the step named `Bad NonLeaf`. The step named `Leaf` does not satisfy the forall clause because its $kind$ attribute has the value $leaf$. When evaluating the root, the XML definition shows that it contains two substep-bindings, thereby satisfying the rule. Looking at the XML for `Bad NonLeaf`, however, we see that it fails to satisfy the rule and is therefore flagged as being inconsistent.

The rule shown above is among the simplest rules. The semantic checker currently supports the following rules:

Language semantic rules:

- Prerequisites and postrequisites should not have out parameters. (They should behave as pure boolean functions.)
- All input parameters to a step should be initialized.
- A step should handle or explicitly throw all exceptions its substeps can throw.
- The root step should have an agent.
- The agent of a non-root step should either be acquired locally or passed in from the parent.
- Deadlines should only be attached to leaves.

Programming anomalies:

- All steps with non-leaf sequencing badges should have substeps.
- All steps should be connected to the process graph.
- All steps should be reachable.
- A non-leaf step should only have handlers for exceptions that its substeps throw.
- A non-leaf step should only declare that it throws exceptions that its substeps throw.
- Multiple substeps of a parallel step should not modify the same parameter. (This is a possible race condition.)
- All parameters to a step should be bound.
- The values of all output parameters should be propagated to other steps.
- Recursion should terminate.
- All items in an interface should have unique names.

While it is not surprising that first-order logic can be used to check local properties on a step, it is more surprising that it can also be used to check for fairly complicated data flow and control flow anomalies. As an example of a more complicated rule, we present details on the rule that checks whether an output parameter is propagated to other steps in the process. First, we offer an explanation for why we have this rule. All real work is performed by the agents when they are assigned a leaf step. Part of what they may do is to produce a data result that can then be passed up to its parent and from there distributed to other parts of the process. Since a leaf step can appear in multiple contexts in a process, it is possible that it will have output parameters that are only needed in some contexts and can be ignored in other contexts. The proper thing to do when ignoring output data is to not provide a parameter binding to its parent for this output data. If a parameter binding is provided, then we would expect to see this data being propagated from the

parent to another step, such as another child, an exception handler, a postrequisite, or up to its own parent. The consistency rule that we describe next checks whether this propagation occurs.

$$
\begin{aligned}
&\text{let } NonLeaves = \{s | s \in Steps \wedge kind(s) \neq Leaf\} \text{ in} \\
&\quad \forall\, nonLeaf \in NonLeaves, \\
&\quad \forall\, substep \in substeps(nonLeaf), \\
&\quad \forall\, outParamBinding \in outParams(substep) \\
&\quad \text{let } paramName = nameInParent(outParamBinding) \text{ in} \\
&\qquad ((kind(nonLeaf) == Sequential \vee kind(nonLeaf) == Try) \wedge \\
&\qquad\quad \exists\, laterSubstep \in laterSubsteps(subStep), \\
&\qquad\quad \exists\, inParamBinding \in inParams(laterSubstep), \\
&\qquad\qquad paramName == nameInParent(inParamBinding)) \\
&\qquad \vee \exists\, parentOutParamBinding \in outParams(nonLeaf), \\
&\qquad\quad nameInChild(parentOutParamBinding) == paramName))
\end{aligned}
$$

In this rule, $outParams$ identifies the set of parameter bindings where a step outputs a value, while $inParams$ identifies the set of parameter bindings where a step inputs a value. $nameInParent$ is a function that returns the name of a parameter in a parent step involved in a parameter binding to a child, while $nameInChild$ returns the name of the parameter in the child step involved in a parameter binding to a parent. $laterSubsteps$ returns the set of substeps that are the sibling of a given step and follow that step. The rule first finds all parameter bindings where a child outputs a parameter value to a parent. It then checks that the parameter is used by checking that the parameter is input to a subsequent child or is itself output to its parent's parent.[1] Because of the flexibility of XPath and the structure of the XML representations of Little-JIL processes, each of these functions is easy to express. The XML representation of this rule, shown in Figure 5, is more lengthy but essentially expresses the same condition.

4 Checker Architecture

The checker is implemented in Java, as is xlinkit. The architecture of the checker is shown in Figure 6. The checker consists of three major components: the user interface, the LJILChecker component, and the LJILFetcher component. The user provides a filename containing a Little-JIL process to be checked via the user interface. This is passed through the LJILChecker to xlinkit which uses a custom xlinkit *fetcher* which automatically invokes a Little-JIL to XML translator when xlinkit is invoked on a Little-JIL file. Xlinkit performs the consistency checks using the rules stored in the rule file. Xlinkit returns a set of links to the LJILChecker identifying the document elements that violate consistency rules along with the rules violated. LJILChecker uses metadata from the violated rules to translate the links into LJILError objects. Each `LJILError` object contains a message and a set of `LJILErrorOperand` objects which carry titles and identifying information

[1] The complete rule also checks whether the parameter is passed to an exception handler, reaction, or postrequisite. These tests are similar but have been omitted for brevity.

```
<1-- For all non-leaves -->
<forall var="nonLeaf" in="//step[@kind != 'leaf']">

 <!-- For all out parameters of children of a non-leaf -->
  <forall var="outParamBinding"
    in="$nonLeaf/substeps/substep-binding/parameter-binding
      [@mode = 'copy-out' or @mode = 'copy-in-and-out']">

    <!-- Bind substep to the child involved in the parameter
         binding -->
    <forall var="substep" in="id($outParamBinding/../@target)">

    <or>
      <or>
        <and>
          <!-- The step has sequential or try control flow -->
          <or>
            <equal op1="$nonLeaf/@kind" op2="'sequential'"/>
            <equal op1="$nonLeaf/@kind" op2="'try'"/>
          </or>

          <!-- And the same parameter is passed as an in-parameter
               to a later substep -->
          <exists var="inParamBinding"
           in="$outParamBinding/../following-sibling::substep-binding
            /parameter-binding[
            @in-parent = $outParamBinding/@in-parent and
            @mode != 'copy-out']"/>
        </and>

        <!-- Or, no matter what step kind, the parameter is passed
             up to the parent of the non-leaf -->
        <exists var="parentOutParamBinding"
          in="//substep-binding[@target = $nonLeaf/@id]
          /parameter-binding[@in-child = $outParamBinding/@in-parent
          and @mode != 'copy-in']"/>
      </or>

      <!-- Conditions to check for passing to exception handlers,
           reactions, and postrequisite omitted -->
     </or>
  </forall>
  </forall>
</forall>
```

Fig. 5. Rule to Check for Propagation of Output Values

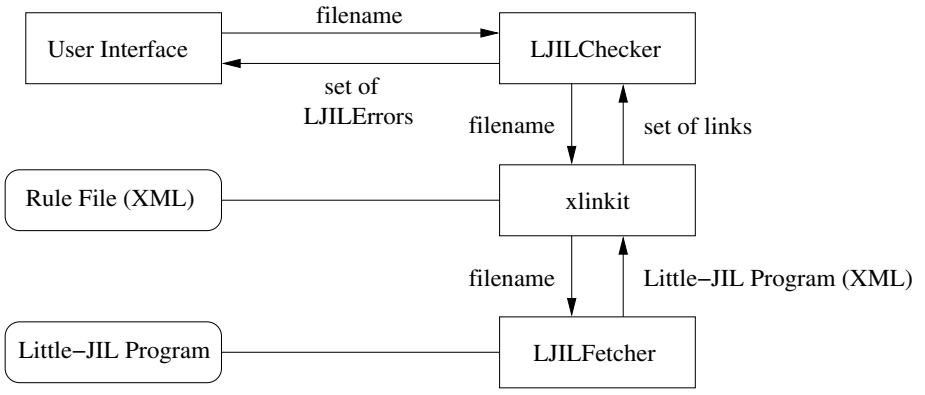


Fig. 6. Architecture of the Little-JIL Semantic Checker

for program elements referenced in the error message. The checking can be invoked either from a command-line printing interface, or from a GUI which uses `LJILErrorOperand` information to locate and visually highlight components of error messages (Figure 1).

The checker is designed to be easy to extend whenever new rules are formalized, whether due to language changes or improving knowledge of programming practice. So that rules can be added or changed by editing only the rule file, the error messages are specified with metadata contained in each rule. For example, in the rule in Figure 2 for detecting non-leaf steps with no substeps, we use the following `header` element:

```
<header>
  <description>
    Non-leaf step with no substeps: very strange.
  </description>

  <meta:msg mode="warning"/>
  <meta:operand seq="1" title="Step"/>
</header>
```

The error or warning message itself is provided in the header's description element. For further information, we take advantage of xlinkit's metadata facility, which allows free-form content within elements in the `meta` namespace. Thus the `meta:msg` tag describes a `mode`, either *warning* or *error*, indicating whether violations of the rule should be flagged as warnings or errors.

For each rule violation, xlinkit returns a *link*, a list of what document elements were bound to quantifiers in the rule before it was violated. In Rule 2, a consistent link would match a `step` element with a `substep-binding` element; an inconsistent link would contain only a `step` and result in a warning message. For useful reporting of these error elements, the `meta:operand` tag associates a title with an operand sequence number. Here, a warning message will give the title "Step" to its first operand as shown in the right panel of Figure 3. Though a title seems superfluous in this case, titles can be crucial to interpreting messages that refer to multiple language elements.

5 Future Work

Xlinkit was created to check the consistency of information encoded in several related XML documents rather than just within a single document as we use it here. A complete Little-JIL process consists of the coordination process discussed in this paper as well as several external entities: a resource model describing the resources available in the environment in which the process is executed and agents that carry out the activities at the leaf steps. Xlinkit offers a good foundation for building a tool to check whether a resource model and a collection of agents can provide the services required by the process. A tool like this is critical in determining whether a process will be able to succeed prior to its execution.

Another avenue to explore is whether xlinkit could be used to verify process-specific properties in addition to language semantics and general coding guidelines. Ideally, we would want to allow people writing these specifications to use a more convenient syntax than XML and to be able to write the specifications without needing to know the XML structure of Little-JIL processes. This therefore requires development of a specification language that can be translated automatically to the syntax required by xlinkit.

We are also exploring other avenues of static analysis of Little-JIL processes. Jamieson Cobleigh did some initial evaluation of applying the FLAVERS data flow analysis tool [3] to Little-JIL processes [1]. His approach required manual translation of Little-JIL into a model that FLAVERS could analyze. We are currently investigating automating this translation. We are also pursuing work toward the use of LTSA [5] to perform model checking of Little-JIL processes, and have had some encouraging initial results in this effort.

6 Related Work

The novelty of this work lies in its use of first-order logic to define the semantic analyzer. Formal notations are commonly used to define the semantics of programming languages. In particular, operational semantics and denotational semantics allow for fairly natural mappings to interpreters, typically written in functional programming languages. It is less clear how to derive static semantic checkers or coding style conformance tools from semantics written in these notations.

Verification tools, such as model checkers, data flow analyzers and theorem provers, often make use of formal notations to specify desired properties to be proven. Unlike our checker, however, these tools usually operate on a specialized semantic model derived from a program rather than the program's syntax itself. The development of these models may be very difficult, depending on the language to be checked.

LOOP [7] is a tool that translates Java code, annotated with formal specifications, into semantics in higher-order logic for analysis with a theorem prover. ESC/Java [4], which also uses theorem proving technology, is based on axiomatic semantics. Bandera [2] is a front-end for static analysis tools that builds models from Java source code with guidance from a software analyst. These tools focus on proving desirable runtime properties, rather than enforcing language semantics and coding guidelines.

Acknowledgements. Systemwire, the makers of xlinkit, provided us with excellent software that made this project possible. Christian Nentwich provided prompt and helpful technical support without which we might still be wading through the XML jungle.

The Little-JIL team at the University of Massachusetts, Amherst, particularly Aaron Cass and Sandy Wise, were helpful in discussing and suggesting rules to check. Sandy Wise also provided the code to produce XML representations of Little-JIL processes and to display Little-JIL processes in a form suitable for annotation with error messages.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9988254. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. Verifying properties of process definitions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 96–101, Portland, Oregon, August 2000.
2. James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
3. Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, pages 62–75, December 1994.
4. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, Berlin, June 2002.
5. Jeff Magee and Jeff Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 1999.
6. C. Nentwich, W. Emmerich, and A. Finkelstein. Static consistency checking for distributed specifications. In *International Conference on Automated Software Engineering (ASE)*, Coronado Bay, CA, 2001.
7. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proceedings of the 7th International. Conference On Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 299–312. Springer, 2001.
8. Alexander Wise. Little-JIL 1.0 language report. Technical Report TR 98-24, University of Massachusetts, Department of Computer Science, 1998. Available at ftp://ftp.cs.umass.edu/pub/techrept/techreport/1998/UM-CS-1998-024.ps.
9. Alexander Wise, Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, and Stanley M. Sutton Jr. Using Little-JIL to coordinate agents in software engineering. In *Proceedings of the Automated Software Engineering Conference (ASE 2000)*, pages 155–164, Grenoble, France, September 2000.
10. World Wide Web Consortium. Extensible markup language (XML). http://www.w3.org/XML/.
11. World Wide Web Consortium. XML path language (XPath). W3C Recommendation, November 16, 1999. Version 1.0. http://www.w3.org/TR/xpath.

A Compliant Environment for Enacting Evolvable Process Models

Wykeen Seet and Brian Warboys

Informatics Process Group
Department of Computer Science
University of Manchester
United Kingdom
{seetw, brian}@cs.man.ac.uk

Abstract. Process models have an inherent need to rapidly evolve in order to maintain synchronization with the human domain that they are supporting. To support the required evolution, it is not sufficient that the support environment is built by making the entire application as flexible as possible to support all known future evolution processes. This is not only impossible to achieve in practice but will also result in a complex environment that will itself be hard to evolve. A better approach will be to build the support environment on top of a systems architecture that continuously provides the best-fit support for the environment. This assumes that the systems architecture is tuned and retuned to the application as the layers of the architecture are built in compliance to the support environment and subsequently the process models.
A compliant systems architecture provides this best-fit architecture by essentially providing an approach to identify a systems architecture that is composed by separating the mechanisms and policies of the application. This compliant architecture is coupled with a flexible up-call/down-call infrastructure for evolving the mechanisms and policies that allow the base configuration to be changed through structural reflection capabilities.
In this paper an experiment to construct a compliant environment for enacting process models is described. Concrete examples of how this compliant environment provided better support for the evolution required by a process model framework are then given.

1 Introduction

Process models are designed to support human systems by providing a mixture of prescription, enactment support and facilities for making the processes explicit. As these process models are functioning within the context of a human domain which is constantly changing, the process models are also subject to constant change in order to remain in synchronization with the human processes. This phenomena has been interpreted as the Law of Requisite Variety(Ashby's Law)[1,2], where 'only variety can absorb variety'. Most process modelling environments have a pre-set approach to provide support for evolution by incorporating some mechanisms within the system to provide reflective compilation and execution facilities. The flexibility of these environments can be limited as the support for evolution is only provided by the immediate layer. This layer is

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 154–163, 2003.

often implemented as a Virtual Machine(VM) to enact the process models with some support for invoking external tools as a way to extend the environment. An alternative approach is to provide a compliant environment where the underlying layers of the systems architecture are continuously adapting to the needs of the application, which in this case, are that of an evolving process model.

2 Compliant Systems Architecture

A compliant systems architecture (CSA)[12,13] can be described as a 'best-fit' architecture for the supported application. This is described as a systems architecture that is *compliant* to the needs of an application as opposed to the more conventional approach where a generic architecture is constructed in order to support all possible applications. A non-compliant systems architecture thus has the following potential shortcomings:-

1. Tuned only for a common generic set of applications. Although the bottom-up focus is suitable for a common set of applications, an application which is not within this 'generic' set will require that most of the missing or unsuitable mechanisms be re-written somewhere within the application. This usually results in the original mechanisms being lost to the application as their utility is replaced.
2. Provide redundant underlying mechanisms which might never be used and may even interfere with other required mechanisms which the application currently utilises.
3. Usually built to be as simple as possible and thus the assumption is that it cannot be flexibly reconfigured.

To achieve compliance for the needs of an application, the architecture is layered by separating the mechanisms and policies of the support environment where their separation provides a best fit for the mechanism and policy needs of the application. A *policy* can be regarded as a strategy for achieving a goal and a *mechanism* is the method by which the objective is achieved. An example of a policy is a thread scheduling algorithm and the mechanism can thus be the priority measure and the subsequent instantiation or suspension of each thread. This separation of mechanism and policy is extended throughout the underlying architecture. The general rule of structuring the mechanisms and policies is thus:-

$$policy_n + mechanism_{n-1} = mechanism_n$$

where the mechanisms from the layer below, *n-1*, coupled with the policies implemented at the current layer, *n*, will together form the mechanism for the higher layers, $\geq n$.

We define communication between the layers as *down-calls* in the case of level $\geq n$ calling level $< n$, and *up-calls* in the reverse case.

It should be clear that the duplication and repetition of the mechanisms and policies across the application and its underlying architecture will result in a non-compliant system.

[13] prescribed a set of criteria for structuring a generic compliant systems architecture. They are listed as :-

1. the number of layers in the architecture
2. the system functions required, eg, recovery, scheduling, clock ticks, etc
3. the method used for specifying policy information;
4. the method used for passing information between layers and system functions(up-calls, down-calls, horizontal calls)

2.1 The Context for the Experiment

The above set of criteria were used to construct a specific compliant Virtual Machine environment for enacting evolvable process models.

The experimental machine was constructed using the CSA toolset developed by the CSA project[1] consisting of a persistent programming language, ProcessBase, an interface to the ProcessBase Environment, Hyper-Code, the abstract machine for executing the language, ProcessBase Abstract Machine(PBAM) and the underlying operating system, Arena.

ProcessBase [10] is a strongly typed persistent programming language designed to act as a target language to support a wide variety of possible process modelling languages. The language provides library functions to extend its feature set. Down-calls to underlying mechanisms are provided by function calls and up-calls are provided as exceptions in ProcessBase. ProcessBase can be configured for a compliant architecture by customising the down-calls to the opcode provided by the underlying abstract machine, PBAM.

The interface to this ProcessBase environment is provided by a Hyper-Code System(HCS) [14,5]. Hyper-Code provides a single representation of the executing program together with its equivalent textual form. This provides a powerful programming paradigm allowing for the browsing, editing and execution (including reflection) of enacting programs using a single unified tool. This is clearly a very useful tool to include in a process modelling system which supports dynamic evolution of models. A HCS has the following domain operations, *reflect*, *reify*, *execute*, *transform* which provides the underlying mechanisms of a HCS. An example set of policy that makes use of these mechanisms, as illustrated in [14], are the **Explode**, **Implode**, **Evaluate**, **Edit** and **GetRoot**. For example, the **Evaluate** policy utilises the *execute* and *reflect* domain operations with the following rule where R is the representation domain of a HCS:-

evaluate(r) is *execute(reflect(r))* where $r \in R$.

PBAM[11] is the abstract machine written in C++ for executing compiled ProcessBase code. It provides a basic set of core instructions for executing ProcessBase core code and can be extended by introducing new opcodes. At its current state, the PBAM interpreter has to be recompiled in order to support new opcodes.

The Arena Operating System(OS)[8] was built to be highly reconfigurable by implementing a very simple nano-kernel which can be reconfigured by Hardware Objects

[1] The "Compliant Systems Architecture CSA Phase 2" project was an EPSRC funded project at Manchester (GR/M88945) and St Andrews (GR/M88938), which was a continuation of the CSA Phase 1 work. More information is available from http://www.cs.man.ac.uk/ipg/csa.html and http://www-ppg.dcs.st-and.ac.uk/Projects/CSA2/.

(HWO) that provides the access to mechanisms provided by the hardware. The policies for accessing the mechanisms offered by the HWO are provided by a user-level library resource called Arena Resource Managers(ARM). Interfaces provided by the user-level ARMs have three components:-

1. the operations available to the application code,
2. the operations used by other managers, and
3. the operations used by the HWO.

Significantly, each of the above mentioned components provides the interface needs for a compliant systems architecture by supporting the down-calls, horizontal calls and up-calls to Arena respectively.

3 A Compliant Process Environment

3.1 π-SPACE

The π-SPACE language was selected as the process model specification language to be supported by the experimental VM. π-SPACE[4] is an Architecture Description Language(ADL) that is based on the π-calculus [9]. It has the following core ADL constructs such as *Components*, *Connectors*, *Ports*, *Behaviours*, *Operations* and *Channels*. There are also operations for:-

- Putting components together and taking them apart, *Compose*, *Decompose*,
- Attaching and reattaching communication channels, *attach*, *reattach*,
- Sending and receiving messages on communication channels, *channel<msg>*, *channel(msg)*,
- Supporting dynamic change using the dynamic operator, π,
- Specifying conditions for process enactment, *when*, *whenever*,
- User defined Operations which are written in the base language, in this case ProcessBase.

In summary, π-SPACE provides an ADL that supports dynamic composition of components and connectors that are specified using a formal process algebra. It was chosen for two reasons:

- being based on the π-calculus it produces a highly flexible and generic form of process model but one which can be formally reasoned about
- it is notoriously difficult to provide a convenient environment for the execution and evolution of such high level specification languages

These properties allowed a single experiment to be constructed which could adequately demonstrate a possible general approach applicable to a large number of process modelling cases.

3.2 Constructing a Compliant π-SPACE Virtual Machine(VM)

The approach taken to structure the VM was by considering the above mentioned four criteria for structuring a compliant architecture:-

1. the number of layers in the architecture. Only one layer was added to the architecture. This layer provided the necessary abstractions for π-SPACE core constructs and their operations. Each π-SPACE core construct was mapped directly onto a ProcessBase view type and each operation was mapped onto ProcessBase functions within a ProcessBase library. A view type in ProcessBase is an n-ary aggregate of all the possible types in the ProcessBase language
2. the system functions required, eg, recovery, scheduling, clock ticks, etc. π-SPACE constructs required two additional system functions. These are the scheduling functions to support the process enactment of the components and the communication channel functions.
3. the method used for specifying policy information. Policy information is specified in π-SPACE which will be compiled into ProcessBase and enacted on top of the PBAM VM.
4. the method used for passing information between layers and system functions(up-calls, down-calls, horizontal calls). As each π-SPACE core construct and operation are mapped onto a corresponding ProcessBase type and function, the down-calls and up-calls are ProcessBase function calls and their return values. Horizontal calls are provided by operations on communications channels.

Code generation rules were generated for each π-SPACE construct that was then mapped into its equivalent enactable form in ProcessBase. The following table shows an example code mapping of a component type definition in π-SPACE to its ProcessBase equivalent.

Defining a π-SPACE Component	**ProcessBase equivalent**

define component type Writer[...]
{
port request_check **:** Request[...] ||
behaviour write **:** Write[...]
}
	!! Type definition
	type Writer **is view[**
	typeTag **: int;**
	request_check**:**Request**;**
	write**:**Write**;**
	start_behaviour**:loc[fun()]]**
	!! Instance Generator
	let gen_Writer $<-$ **fun(...)**
	{
	let Writer_start_behaviour $<-$ **fun()**
	{ ...
	}
	view(typeTag $<-$ componentTag**,**
	write $<-$ port_id1**,**
	write $<-$ write**,**
	start_behaviour $<-$ Writer_start_behaviour)
	}

A π-SPACE VM that is compliant for enacting Process Models has been constructed with the following three components:-

1. Libraries written in ProcessBase that implements a one to one mapping of the π-SPACE core abstractions.
2. A π-SPACE Reflective Compiler that provides support for run time modification to enacting models.
3. A Hyper-Code[14] front-end tool for supporting π-SPACE hyper-code programming since this is the essential paradigm underpinning the run time modification of enacting models.

The π-SPACE VM was constructed using the ProcessBase interpreter that runs on top of Linux. Linux proved to be a rather convenient example of a non-compliant system and hence further experiments were also conducted on Arena as an investigation into issues of compliance.

Figure 1 shows the generic compliant architecture as described in [13] and its realisation into a specific compliant architecture for a process environment. The Components, Up-call/Down-call view provides a more detailed view on the VM.

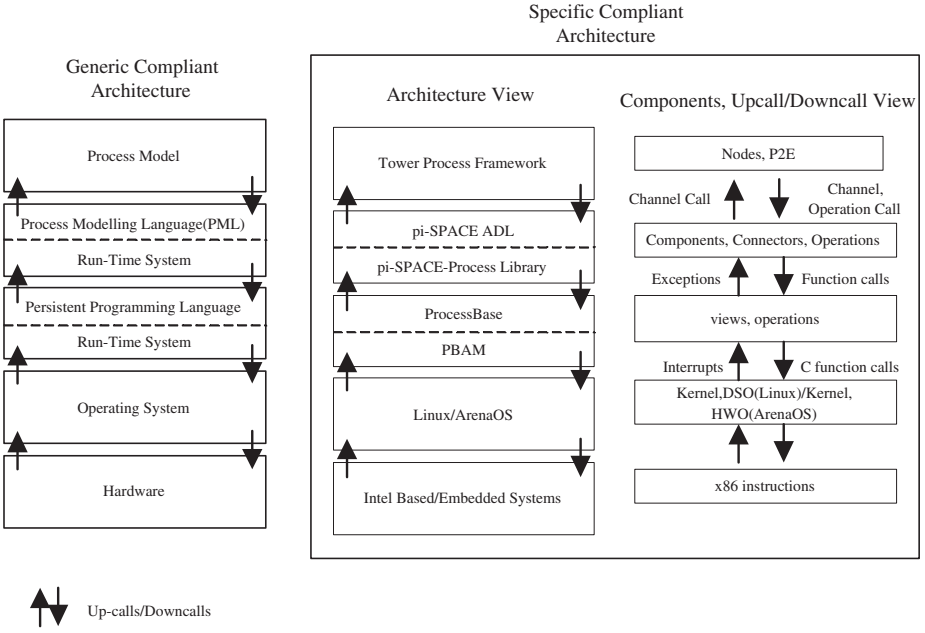


Fig. 1. A Complaint π-SPACE VM Architecture

4 Illustrations of Compliance

A process modelling framework, called Towers [6], was implemented to run on the π-SPACE VM. This prototype implementation was designed as an experiment to expose some issues of process evolution support. A Towers framework consists of Nodes with process fragments within them and a meta process called the Process for Process Evolution (P2E) for evolving the process fragments. Nodes are ordered into a hierarchy which usually reflects how a software product is generated. The hierarchy of nodes could be multidimensional when orthogonal nodes of concurrent processes that produce products which are required by the current hierarchy are modelled.

Through the experience gained from implementing the Towers process framework, a number of specific illustrations of the relationship between compliance and the support for evolution can be observed. Two specific examples of non-compliance and the approach to achieve compliance for the application are now described.

4.1 Thread Scheduling

π-SPACE components in particular *Behaviours* and *Operations* have their own individual threads of execution. As ProcessBase itself does not have native thread support, the support of threads has been implemented by extending the opcodes supported by the PBAM interpreter. The implementation of ProcessBase threads support on the Linux OS are mapped directly to the POSIX thread[7] implemented on Linux. Some process models that were written required some form of control or at least some feedback from the scheduler on the execution of some threads. As such these process models required the underlying OS to provide different scheduling schemes.

An issue that was discovered during the implementation of the Tower Process Model framework was that during run time evolution, each Node has its own reference to a reflective compiler for compiling and installing the change. As there is no way of knowing when the compilation phase completed and subsequently when the change was installed and enacted, a meta process, which was coded into the process model by suspending some threads and waiting on them, had to be implemented.

Essentially, this required a rewrite of the thread scheduling mechanisms at the ProcessBase layer as there are no mechanisms at the OS layer that meet the needs of the above mentioned evolution process. The mechanisms for thread scheduling provided by the OS layer are thus used in a limited manner. Linux threads are still utilised but certain mechanisms to support thread-like scheduling had to be implemented in ProcessBase. Duplicating similar mechanisms on another layer resulted in a non-compliant complex system.

The approach is different when we utilise the Arena OS as then we can select and even specify different thread scheduling schemes. This is more compliant to the application as the underlying mechanisms offered by the operating system meets the policy needs of the supported application. What is key here is that the mechanisms are provided and implemented at the system layer which also makes the best use of all the underlying mechanisms and policies of the system.

4.2 Communications Channel

The π-SPACE channel has been implemented in ProcessBase using view structures to encapsulate the π-SPACE types that can be sent down the channels. The operations on these channels were then implemented as ProcessBase functions which provide buffering and semaphore locking mechanisms for ensuring that only one thread can access the global structure which stores the information for managing operations on channels.

This resulted in complicated data structures and algorithms which should be better provided by the underlying operating system.

Each π-SPACE communication channel is mapped directly onto a communication channel type in ProcessBase. In this manner, the policy for channel communication is the same. However, at the OS layer, the mechanisms used are different. On Arena, the mechanisms are provided by the Networking HWO. The down-calls are thus C calls down to the particular Networking HWO and the up-calls from the HWO are Arena interrupts. On Linux, a decision was made to completely bypass the underlying non-compliant sockets implementation. This meant that the mechanisms for channels such as concurrency control and buffering had to be implemented at the ProcessBase Layer.

This is an example where a non-compliant operating system might have only some of the mechanisms to support the application policies and so cause it to be bypassed altogether by re-implementing a more compliant version. This is usually achieved by either reusing parts of the available mechanism or by completely re-implementing a more suitable version while leaving the underlying mechanism intact.

To maintain compliance, a systems architecture is reconfigured by allowing the underlying mechanisms, which in this case are embedded at the OS level, to be exposed to the upper levels. A compliant architecture allows a layer to be bypassed to allow the architecture to maintain its compliance with the application. From a physical viewpoint, the mechanisms and policies are still in their respective layers, but from a logical view, the mechanisms of the underlying layers are now exposed upwards. This ensures that the logical layers are compliant, in that the mechanisms are made available to the layer where the mechanisms are best suited. This also encourages the reuse of mechanisms.

Figure 2 shows the logical views of a compliant and a non-compliant architecture.

5 Conclusions and Future Work

An environment was constructed following the guidelines, from the CSA project, suggested for structuring a compliant system architecture. This supported the notion that 'a design to structure the separation of mechanisms and policies' allows an architecture to be changed easily to accommodate possible changes required by the application. Furthermore, for other application evolution that required more changes at the lower layers, a compliant architecture facilitates these changes while maintaining the compliance of the system architecture to the application. An example was used to illustrate how compliance reduced the complexity and redundancy of underlying mechanisms whilst increasing reuse. Another illustration was provided that shows how evolving process models can be better supported by a compliant architecture that provides more control over the underlying mechanisms as and when the process models requires it.

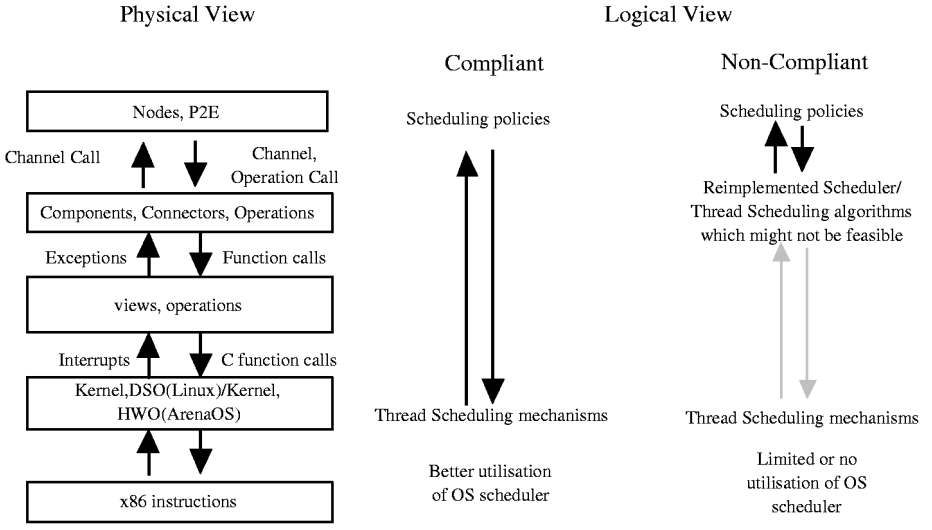


Fig. 2. Compliant and non-compliant views of the thread scheduling issue

The evolution process enacted to maintain architectural compliance has been achieved by explicitly coding the changes, as and when required, as a proof of the feasibility of the approach. However, it is not hard to see how it can be extended to allow for automatic change management once the process can be formalised. It will be especially interesting to formalise the evolution process so that it can be enacted by the environment itself. This will require the use of a dynamic loader at the operating system layer and this has become recently available [3] in the Arena test system.

There is much work required in order to more rigorously describe the measure of compliance of an architecture to that of the supported application. A better understanding of architectural metrics will allow us to measure the degree of compliance of one architecture to another depending on which application is to be supported. This approach would provide benefits that are threefold, firstly it will allow us to have a common semantic of what compliance actually means for different applications. Secondly, it will allow comparisons between compliant architectures of interest. Thirdly, which is a direct result from the second benefit, is that it will allow us to formalise the comparison function and thus allow the selection of compliant architectures for supporting the necessary evolution which needs to be enacted within process models.

Though the benefits of flexibility can shown by the example of exposing the scheduler mechanisms from the underlying OS to the application, it is as yet unclear what the flexibility accorded by exposing the underlying mechanisms will benefit for example, the automated evolution of process models since there are no formal semantics to describe the evolution in terms of the changes required at the architectural layers. A possible solution is to provide a reasoning engine that will provide a feedback mechanism that allows the process model to decide if the underlying changes are safe. This will allow an

evolution process to proceed without compromising system stability and other system constraints in the name of compliance.

Acknowledgements. We gratefully acknowledge the very large contribution made to this experiment by Ron Morrison's research team at St Andrews University especially Dharini Balasubramaniam and Vangelis Zirintsis. Thanks are also due to Flavio Oquendo and Christelle Chaudet at the University of Savoie at Annecy for both providing the definition of the π-SPACE language and always responding quickly when any problems of definition arose.

References

1. W. R. Ashby. *An introduction to cybernetics*. Chapman, 1956.
2. Stafford Beer. *Designing Freedom*. John Wiley and Sons, 1974.
3. S Beyer, K Mayes, and B.C Warboys. Application-compliant networking on embedded systems. *5th IEE International Workshop on Networked Appliances*, 2002.
4. Christelle Chaudet, R.M. Greenwood, Flavio Oquendo, and Brian Warboys. Architecture-driven software engineering: Specifying, generating and evolving component-based software systems. *IEE Proceedings: Software*, 2000.
5. R. Mark Greenwood, Dharini Balasubramaniam, Sorana Cmpan, Graham N.C. Kirby, Kath Mickan, Ron Morrison, Flavio Oquendo, Ian Robertson, Wykeen Seet, Bob Snowdon, Brian C. Warboys, and Evangelos Zirintsis. Process Support for Evolving Active Architectures. In *Proceedings of the 9th European Workshop on Software Process Technology*, 2003.
6. R.M. Greenwood, Ian Robertson, and B.C. Warboys. A Support Framework for Dynamic Organizations. *Software Process Technology, 7th European Workshop, EWSPT 2000, volume 1780 of Lecture Notes in Computer Science*, 2000.
7. IEEE. *(ISO/IEC) [IEEE/ANSI Std 1003.1, 1996 Edition] Information Technology - Portable Operating System Interface (POSIX) - Part 1: System Application: Program Interface (API) [C Language]*. IEEE, 1996.
8. K. R. Mayes and J. Bridgland. Arena-a run-time operating system for parallel applications. In *Proceedings of 5th EuroMicro Workshop on Parallel and Distributed Processing (PDP'97)*, 1997.
9. Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.
10. R Morrison, D Balasubramaniam, M Greenwood, Kirby GNC, K Mayes, DS Munro, and BC Warboys. Processbase reference manual (version 1.0.6). Technical report, Universities of St Andrews and Manchester, 1999.
11. R Morrison, D Balasubramaniam, M Greenwood, GNC Kirby, K Mayes, DS Munro, and BC Warboys. Processbase abstract machine manual (version 2.0.6). Technical report, Universities of St Andrews and Manchester, 1999.
12. R. Morrison, D. Balasubramaniam, R.M. Greenwood, G.N.C Kirby, K. Mayes, D.S Munro, and B. Warboys. An approach to compliance in software architectures. *IEE Computing and Control Engineering Journal, Special Issue on Informatics*, 11(4):195–200, 2000.
13. R. Morrison, D. Balasubramaniam, R.M. Greenwood, G.N.C Kirby, K. Mayes, D.S Munro, and B. Warboys. A compliant persistent architecture. *Software-Practice and Experience*, pages 363–386, 2000.
14. E Zirintsis. *Towards Simplification of the Software Development Process: The Hyper-Code Abstraction*. Phd, University of St Andrews, 2000.

Decentralised Coordination for Software Process Enactment

Jun Yan[1], Yun Yang[1], and Gitesh K. Raikundalia[2,1]

[1] CICEC - Centre for Internet Computing and E-Commerce
School of Information Technology
Swinburne University of Technology
PO Box 218, Hawthorn, Melbourne, Australia 3122
Email:{jyan,yyang}@it.swin.edu.au

[2] School of Computer Science and Mathematics
Victoria University
P.O. Box 14428, Melbourne City, MC 8001, Australia
Email: Gitesh.Raikundalia@vu.edu.au

Abstract. Software process enactment mainly involves coordination of relevant team members to enact various tasks, which is supported by a centralised client/server architecture traditionally. This paper adopts concepts from peer-to-peer computing and presents an innovative approach to support decentralised process enactment. With this approach, team members are supported with coordination by direct communication among peers, which reflects applications' increasingly distributed nature better. In addition, as a software process may be modelled incompletely at modelling time, a mechanism supporting run-time refinement of incomplete processes is also described in this paper.

1 Introduction

Software process research is based on a philosophy that the entire exercise of software development can be modelled as a complex process. Consequently, software process research deals with methods and technologies used to support modelling, analysis, improvement and enactment of software development activities [6]. For the past two decades, software process research has experienced tremendous growth and reached a reasonable degree of maturity. Regarding software process enactment, at least four assumptions underlying software process research have been identified:

1. Software processes are processes too [6]. Software process technology shares many commonalities with and addresses similar problems to other process support technology such as workflow, in terms of scope, methods and techniques [3]. Generally speaking, a process is normally composed of tasks which are partially-ordered [4]. Thus, the enactment of a software process is the procedure of enacting various partially-ordered tasks to achieve the process objectives.

F. Oquendo (Ed.): EWSPT 2003, LNCS 2786, pp. 164–172, 2003.

2. A software process is normally too complex to be defined completely at modelling time. Alternatively, a software process is always modelled at different levels of abstraction. Major tasks, artifacts and roles are elicited and the structure of the model is documented at first while further elaboration of the model is completed during process enactment. Thus, an automated run-time facility should be introduced, which enables the refinement of the software process on-the-fly without bringing the system down [8].
3. In most cases, a non-trivial software process involves team members [14]. Various tasks in a software process may be executed by various team members with cooperation. Therefore, the software process support environment schedules these partially-ordered tasks by coordinating team members. This coordination is often facilitated by process support software, which manages the interaction among team members automatically and thus enables the "automation" of the process.
4. Given the nature of the application environment and technology involved, software process enactment, and process-centred applications in general, are inherently distributed [7,11]. For example, team members involved in a software process and external tools (e.g., CASE tools) used in a software process may be physically dispersed. This could allow for the 24/7 working mode (24 hours a day and 7 days a week) while resulting in intense communication and cooperation [10].

In summary, enactment of complex software processes is normally teamwork-based and carried out in a distributed environment. Thus, coordination plays a key role in software process enactment. Research on software process enactment support is normally based on the dominating centralised client/server system model, which seems to "hit the brick wall" with many problems unsolved. Therefore, it is argued by the authors that the distributed nature of software process enactment needs to be reflected better by decentralised coordination [12]. At the same time, this decentralised coordination should also support the process refinement and process modification on-the-fly.

This paper introduces SwinDeW, which is an ongoing project aiming at providing decentralised coordination for software process enactment based on the peer-to-peer (p2p) distributed system model. The next section describes a p2p-based decentralised framework for teamwork support. Then Section 3 presents decentralised software process enactment support. After that, a mechanism supporting run-time incomplete process refinement is discussed in Section 4, followed by a further discussion in Section 5. Finally, Section 6 concludes this paper and outlines authors' future work.

2 A p2p-Based Framework for Teamwork

Centralised process support has encountered many problems due to the following reasons: (1) centralised coordination is often based on rules without negotiation enabled, which makes it inapplicable for applications requiring certain flexibility such as software process support; (2) centralised coordination is heavy-weighted and can

be overwhelmed by communication; and (3) centralised coordination relies on centralised information storage, which is vulnerable to failures. With respect to these difficulties, a decentralised process-centred environment needs to be explored to incorporate more flexible and light-weighted process support, which does not imply the presence of: (1) a centralised data repository; nor (2) centralised coordination. In the last decade, quite a few efforts contributed to distributed and decentralised process support in various communities such as software engineering, groupware and workflow [2,5,9].

On the other hand, p2p, which can be defined simply as the sharing of computer resources and services by direct exchange, is driving a major shift in the area of genuinely distributed computing [1]. While each peer behaves as a client in the client/server model, a peer is also capable of performing server functions in a p2p system. Technically, p2p enables better scalability and flexibility, and eliminates the risk of single–source bottleneck, by distributing data and control. Thus, p2p reflects applications' distributed nature better and allows for more negotiation and more flexible cooperation.

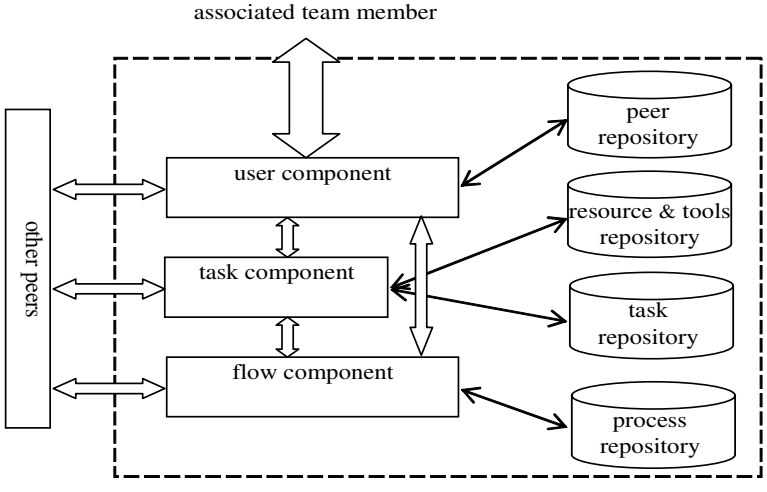


Fig. 1. Structure of a peer

SwinDeW combines concepts from process support technology and p2p technology and builds a p2p-based decentralised process support environment [12]. In SwinDeW, each peer is denoted as a software component residing on a physical machine to enable direct communication with other peers for enactment of a software process while, in most cases, a peer is associated with a team member. With support of peers, each team member is a self-managing and independent entity accountable for a specified contribution to the completion of software development. As depicted in Figure 1, a peer consists of a *user component*, a *task component* and a *flow component*, which interact with one another internally and externally to support process enactment. At the same time, these three components represent a number of software process features and facets in a precise and comprehensive way:

- A user component of a peer is the "bridge" between the associated team member and the software process environment. On one hand, a user component provides development procedures and operations to the associated team member by delivering essential information of the software process. On the other hand, a user component represents the roles of the associated team member in the process in terms of capabilities (e.g., software analyst and Java programmer). A user component manages a peer repository, which stores information about other relevant peers it detected.
- A task component of a peer is in charge of the execution of tasks conducted by the associated team member. In most cases, a task requires some artifacts as inputs, which are normally created by preceding tasks, and generates some artefacts as outputs, which are normally transmitted to succeeding tasks. For example, a coding task in a software process may read some software design documents from preceding tasks and create software code as output, which are inputs of succeeding tasks such as software testing. A task component can also administrate resources and invoke tools in the procedure of task execution as modelled. Thus, a task component manages both a task repository and a resource & tool repository.
- A flow component of a peer helps to fit an individual task into the software process. The main purpose of a flow component is to deal with data dependency and control dependency among tasks by handling incoming and outgoing messages. In this way, the software process enactment can be coordinated step-by-step as pre-defined. A flow component manages a process repository which stored process information distributed to this peer.

As briefly indicated above, to enable direct communication and coordination for teamwork, a peer needs to manage a peer repository, which stores the information of cooperative peers, such as the roles the associated team member can play, the location of a peer, and so on. In SwinDeW, peers are logically connected with each other so as to form communities according to the roles their associated team members play. For example, all peers associated with QA engineers form a QA engineer community. If a team member is capable of playing multiple roles in a project, the associated peer is involved in multiple communities correspondingly. Normally, a peer is initially informed of the information about other peers involved in the same communities. A peer in a community can also have the information about other peers in another community via a peer involved in both communities. In addition, some external means like organisational management are used to assure that no single peer is isolated. As a result, all peers are connected eventually and capable of locating one another.

3 Decentralised Process Enactment Support

Based on the framework described in the previous section, apparently, the first issue that needs to be addressed is the process definition storage in a decentralised process support environment without a centralised data repository. Obviously, replicating the

complete process definition at each peer, which is clearly resource-consuming and error-prone, is not an ideal solution. Although having a basic understanding of the whole process is important, most individual peers are concerned about the detail of only a part of the process during the process enactment. SwinDeW separates essential information about various tasks from the software process, and only distributes appropriate information to relevant peers. Basically, essential information of a task, which helps peers to understand the work to be done, includes the full specification of this task such as team member's role in enactment, tools to be used, control and data dependencies with other tasks. After being extracted from the software process, essential information about a task only goes to a peer associated with a team member who is capable of playing a role in the enactment of this task. In the case that a task is not strictly associated with a single peer, all the relevant peers share the information about the task and negotiate for the right to carry out this task later. All of the above offer the flexibility in choosing work items for the purposes of meeting team member's personal satisfaction and balancing workloads, therefore leading to efficient process enactment.

Once the software process definition is properly partitioned and distributed, the software process enactment can be conducted with the support of decentralised coordination. This decentralised coordination is the result of efforts of all the peers involved and is actually realised by the direct communication among various peers. To enable work to be done and passed from one peer to another, there are two kinds of coordination: *data coordination* and *control coordination*. Correspondingly, there are two types of messages flowing among the peers, i.e., *data messages* and *control messages* [13]. A data message transfers real data related to the particular software process to coordinate data dependency between tasks. A control message delivers information to coordinate control dependency, which is emphasised here. In general, three kinds of control messages perform different functions: (1) *Request and report*. These messages are transmitted to enquiry or indicate the current state of software process enactment. (2) *Instruction***.** This command-like control message instructs peers to take actions. In reality, instruction messages are used for coercive coordination, where the software process is enacted as defined and there is no doubt about the work to be done. (3) *Suggestion*. Suggestion messages are used for cooperative coordination, where the software process is not well defined or some exceptions happen.

Put simply, this p2p-based decentralised process enactment support gives more flexible information and control to team members, thus allowing them to function more independently. From the coordination viewpoint in general, a peer, which is in charge of the enactment of a task, receives data and control messages from its preceding peers directly, then helps the associated team member to conduct the enactment of the task, and finally transmits relevant data and control information to its succeeding peers directly. This distributed coordination is regarded light-weighted in contrast to heavy-weighted centralised coordination. It builds a relatively flexible control/negotiation platform for software process enactment, which reflects the increasingly distributed application nature better. In addition, it may remove performance bottleneck and single-point failures.

4 Process Refinement Support

In an incompletely modelled software process, the basic software development procedure is specified as partially-ordered tasks while details of some tasks remain uncertain. This uncertainty should be determined on-the-fly so that the process enactment support is required to enable incomplete process refinement. For example, at a high level of process abstraction, an incompletely-defined task could be decomposed into more refined and specific tasks during process enactment.

To support run-time refinement, SwinDeW introduces a special task called *refinement*, which is associated with a task of abstraction or uncertainty known as a to-be-refined task. A refinement task manages the refinement of an associated to-be-refined task during enactment.

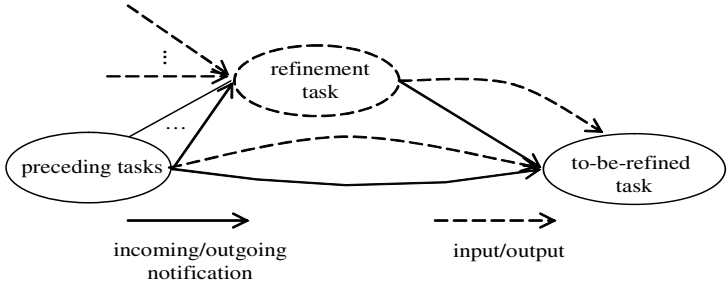


Fig 2. Model of a refinement task

As shown in Figure 2, a refinement task is modelled as follows at modelling time:

- *Description*: A refinement task refines the associated task at run-time, which may include the decomposition of a task, the finalisation of some uncertain attributes, and remodelling of a task.
- *Responsibility*: A refinement task is carried out by an authorised person such as a process engineer or project manager.
- *Inputs*: The inputs of a refinement task are very flexible. A refinement task may have multiple inputs such as feedbacks from relevant peers, available resources, history experience, specifics of the software process, and so on.
- *Output*: The single output of a refinement task is a detailed description of the associated to-be-refined task, which is one of the inputs of the to-be-refined task.
- *Incoming notifications*: A refinement task receives notifications from the preceding tasks of the to-be-refined task upon their completion, i.e., the preceding tasks of the to-be-refined task are also preceding tasks of this refinement task.
- *Outgoing notification*: A refinement task notifies the to-be-refined task upon its completion, i.e., this refinement task is a preceding task of the to-be-refined task.
- *Constraints*: Normally a refinement task can begin at anytime during the process enactment while the latest start time is when it has received all incoming notifications.

A refinement task is distributed to a peer associated with an authorised person to refine a to-be-refined task. This peer precedes the peer associated with the to-be-refined task because the refinement task should be completed before the start of the to-be-refined task. Normally, a refinement task should be done before the associated peer receives the notifications from the peers in charge of the preceding tasks. This kind of refinement is regarded as a "pull" model where a task is refined in advance and hence can be enacted without real delay. However, in the worst case, the refinement depends on the up to date status of process enactment and the peer associated with a refinement task only starts when the preceding work has been done. This is the reason why a refinement task receives the notifications from the preceding tasks of the to-be-refined (see Figure 2). This kind of refinement is regarded as a "push" model where the enactment of a refinement task is triggered passively and may block the whole process. To enact a refinement task in an organised manner, the associated peer may monitor the progress of process enactment through communications with other relevant peers, take various inputs into account, and use analysis and modelling tools. After the completion of a refinement task, the associated peer notifies its succeeding peer, i.e., the peer associated with the to-be-refined task and transmits a new task description to update its succeeding peer's task repository. The to-be-refined task is eventually enacted according to the refined model.

When a task can be decomposed into low-level sub-tasks, this task is known as a sub-process. After decomposition of a sub-process, the sub-tasks may need to be distributed to appropriate peers again for execution. Task distribution and enactment coordination are supported with the same mechanisms described in Section 3. Note that some of the sub-tasks are atomic tasks and specified completely, while a sub-task can be a sub-process or with some uncertainties again, which need to be refined further. In this situation, some new refinement tasks should be facilitated. By these means, incomplete tasks can be refined at multiple levels of abstraction, which allows for improved flexibility and adaptation in process enactment.

In short, an incompletely specified process can be elaborated at run-time. Uncertainties can be determined on-the-fly. Both process level refinement and instance level refinement need to be supported. The former is permanent refinement and applicable to the process instances enacted later. The refinement in the latter case only applies to a particular process instance. Again, this run-time process refinement is carried out in a decentralised fashion.

5 Discussion

The previous section has described a mechanism supporting process refinement on-the-fly. This proposed mechanism is based on an underlying assumption that the refinement of a task will not affect other tasks, which is known as *"closed-box" refinement*. Closed-box refinement details a task when its data and control interfaces with the outside are presented completely. In this situation, the artifacts this task consumes and produces are defined and the execution dependencies with other tasks are determined. Closed-box refinement will not affect a task's interfaces as well as external software behaviour.

However, the refinement of an incomplete process may be more complicated. Sometimes, a task's data or control interfaces with the outside cannot be defined completely at modelling time. The refinement of a task may change its interfaces and affect external software behaviour as well. In this situation, the refinement is known as *"open-box" refinement*, which "opens the box" and advises the internal changes to the outside. In general, open-box refinement of a task may affect other tasks which:

- have not yet been enacted;
- are being enacted currently; and
- have been enacted already.

Therefore, open-box refinement should be treated more carefully as some changes may result in problems like inconsistency and deadlock. For example, when a refined task tries to modify some artifacts which have been consumed or are being consumed by another task, the artifacts may be inconsistent. Open-box refinement can be addressed together with process changes. Relevant peers are informed of changes by *instruction* messages. At the same time, in the case that an inconsistency occurs, relevant peers would negotiate automatically to handle the inconsistency. Peers may allow the inconsistency to remain, or attempt to achieve consistency by means like rollback, suspension, and so on. Hence, the process enactment support should facilitate inconsistency detection, automatic negotiation, decision making and evaluation, etc.

6 Conclusions and Future Work

This paper has argued that the enactment of a software process is in essence distributed teamwork, thus decentralised coordination may be more appropriate in support of light-weighted flexible software process enactment than its centralised counterpart. To provide decentralised coordination, this research utilises the peer-to-peer distributed model to develop a p2p-based process support environment, in which team members are supported with coordination by direct communication among associated peers. In addition, this research also addresses the run-time process refinement by introducing the refinement task. The main contribution of this research is that the approach proposed can give team members more control and flexibility thus may achieve more efficiency and human satisfaction.

In the future, further investigation on decentralised process support will be conducted to improve the framework proposed. Mechanisms supporting open-box refinement at run-time, which incorporate with inconsistency detection and recovery, will be explored further. At the same time, dynamic process evolution and process enactment exceptions will also be addressed.

Acknowledgements. Work reported in this paper is supported partially by Swinburne Vice Chancellor's Strategic Research Initiative Grant 2002-2004.

References

1 K. Aberer and M. Hauswirth, "Peer-to-Peer Information Systems: Concepts and Models, State-of-the-Art, and Future Systems", In Proc. of 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), 326–327, Vienna, Austria, Sept. 2001.

2 T. Bauer and P. Dadam, "Efficient Distributed Workflow Management Based on Variable Server Assignments", In Proc. of 12th Conference on Advanced Information Systems Engineering (CAiSE'00), 94109, Stockholm, Sweden, June 2000.

3 R. Conradi, A. Fuggetta, and M. L. Jaccheri, "Six Theses on Software Research", In Proc. of 6th European Workshop on Software Process Technology (EWSPT'98), 100–104, Weybridge, UK, Sept. 1998.

4 P. H. Feiler and W. S. Humphrey, "Software Development and Enactment: Concepts and Definitions", In Proc. of 2nd International Conference on Software Process, 28–40, Berlin Germany, Feb. 1993.

5 G. L. Ferrari**,** C. Montangero**,** L. Semini and S. Semprini, "Mobile Agents Coordination in Mob_{adtl}", In Proc. of 4th International Conference on Coordination Languages and Models (Coordination'00), 232–248, Limassol, Cyprus, Sept. 2000.

6 A. Fuggetta, "Software Process: A Roadmap", In Anthony Finkelstein, ed. The future of Software Engineering. ACM Press, pp27–34, 2000.

7 C. Ghezzi**,** "Ubiquitous, Decentralized, and Evolving Software: Challenges for Software Engineering", In Proc. of 1st International Conference on Graph Transformation (ICGT'02), Lecture Notes in Computer Science**,** Vol. 2505, 1–5, Springer, Barcelona, Spain, Oct. 2002.

8 G. Valetto, G. E. Kaiser, and Gaurav S. Kc, "A Mobile Agent Approach to Process-Based Dynamic Adaptation of Complex Software Systems", In Proc. of 8th European Workshop on Software Process Technology (EWSPT'01), 102–116, Witten, Germany, June 2001.

9 N. Glaser**,** J-C. Derniame, "Software Agents: Process Models and User Profiles in Distributed Software Development", In Proc. of 7th Workshop on Enabling Technologies Infrastructure for Collaborative Enterprises (WETICE'98), 45–50, Palo Alto, CA, USA, June 1998.

10 I. Gorton and S. S. Motwani, "Issues in Cooperative Software Engineering using Globally Distributed Teams", Information and Software Technology Journal, 38(10):647–655, 1996.

11 J. Grundy, M. Apperley, J. Hosking and W. Mugridge, "A Decentralized Architecture for Software Process Modeling and Enactment", IEEE Internet Computing, 2(5):53–62, Sept/Oct. 1998.

12 J. Yan, Y. Yang and G. K. Raikundalia, "A Decentralised Architecture for Workflow Support", In Proc. of 7th International Symposium on Future Software Technology (ISFST'02), CD ISBN: 4-916227-14-X, Wuhan, China, Oct. 2002.

13 J. Yan, Y. Yang and G. K. Raikundalia, "Enacting Business Processes in a Decentralised Environment with p2p-based Workflow Support", In Proc. of 4th International Conference on Web-Age Information Management (WAIM'03), Lecture Notes in Computer Science, Springer, 2003, to appear.

14 Y. Yang, "Issues on Supporting Distributed Software Processes", In Proc. of 6th European Workshop on Software Process Technology (EWSPT'98), 143–147, Weybridge UK, Sept. 1998.

Author Index