Leszek A. Maciaszek
Pericles Loucopoulos (Eds.)

Communications in Computer and Information Science 230

Evaluation of Novel Approaches to Software Engineering

5th International Conference, ENASE 2010
Athens, Greece, July 2010
Revised Selected Papers

Springer

Communications
in Computer and Information Science 230

Leszek A. Maciaszek
Pericles Loucopoulos (Eds.)

Evaluation of Novel Approaches to Software Engineering

5th International Conference, ENASE 2010
Athens, Greece, July 22-24, 2010
Revised Selected Papers

Springer

Volume Editors

Leszek A. Maciaszek
Wrocław University of Economics, Institute of Business Informatics
53-345 Wrocław, Poland, and
Macquarie University, Department of Computing
Sydney, NSW 2109, Australia
E-mail: leszek@science.mq.edu.au

Pericles Loucopoulos
Loughborough University, The Business School
Loughborough, Leicestershire, LE11 3TU, UK
E-mail: p.loucopoulos@lboro.ac.uk

ISSN 1865-0929 e-ISSN 1865-0937
ISBN 978-3-642-23390-6 e-ISBN 978-3-642-23391-3
DOI 10.1007/978-3-642-23391-3
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011941861

CR Subject Classification (1998): D.2, F.3, D.3, C.2, H.4, K.6

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Software systems are complex and difficult to build, maintain and evolve. They need to integrate with systems already in existence and to conform to changing social contexts, information technology and business conditions. Accordingly, software engineering is unlike traditional engineering, such as mechanical, electrical or building engineering.

Many principles of software engineering cannot be formalized in mathematical models, but this does not free the software engineer from rigor. Software engineering processes and practices have to be rigorous. They also need to be novel, while not disregarding proven traditional approaches. By merging novel approaches with established traditional practices and by evaluating them against software quality criteria, software researchers and practitioners advance knowledge and practice, identify the most hopeful trends and propose new directions for consideration in large-scale software development and integration.

This observation epitomizes the mission of the ENASE (Evaluation of Novel Approaches to Software Engineering (ref. http://www.enase.org/)) conference series in its desire to be a prime international forum to discuss and publish research findings and IT industry experiences with relation to evaluation of novel approaches to software engineering. This book contains revised and extended versions of full papers of the 5th edition of ENASE held in Athens, Greece. The previous four conferences took place in Erfurt, Germany (2006), Barcelona, Spain (2007), Madeira, Portugal (2008), and Milan, Italy (2009). The next 2011 ENASE conference will take place in Beijing, China.

More than 70 papers were submitted to ENASE 2010. A few of these papers were rejected on formal grounds and all other papers were sent to at least three PC members for review. After the careful consideration of research contributions, 19 papers were accepted as full papers and 11 as short papers. The acceptance rate confirms the desire of the ENASE Steering Committee to ensure the high quality of the conferences. All five ENASE conferences had the acceptance rate for full papers at or below 30%.

At the conference in Athens, the papers were presented in the following eight categories. In this volume we have dispensed with this categorization, but the reader may find it useful to evaluate the breadth and depth of the coverage.

1. Quality and Metrics
2. Service and Web Engineering
3. Process Engineering
4. Patterns, Reuse and Open Source
5. Process Improvement
6. Aspect-Oriented Engineering
7. Service and Web Engineering
8. Requirements Engineering

December 2010

Leszek A. Maciaszek
Pericles Loucopoulos

Organization

Conference Chair

Joaquim Filipe	Polytechnic Institute of Setúbal / INSTICC, Portugal

Program Co-chairs

Pericles Loucopoulos	Loughborough University, UK
Leszek Maciaszek	Macquarie University, Australia

Organizing Committee

Patrícia Alves	INSTICC, Portugal
Sérgio Brissos	INSTICC, Portugal
Helder Coelhas	INSTICC, Portugal
Vera Coelho	INSTICC, Portugal
Andreia Costa	INSTICC, Portugal
Patricia Duarte	INSTICC, Portugal
Bruno Encarnação	INSTICC, Portugal
Mauro Graça	INSTICC, Portugal
Liliana Medina	INSTICC, Portugal
Elton Mendes	INSTICC, Portugal
Carla Mota	INSTICC, Portugal
Raquel Pedrosa	INSTICC, Portugal
Vitor Pedrosa	INSTICC, Portugal
Daniel Pereira	INSTICC, Portugal
Filipa Rosa	INSTICC, Portugal
Mónica Saramago	INSTICC, Portugal
José Varela	INSTICC, Portugal
Pedro Varela	INSTICC, Portugal

Program Committee

Colin Atkinson,Germany
Franck Barbier, France
Giuseppe Berio, France
Maria Bielikova, Slovak Republic
Nieves R. Brisaboa, Spain
Dumitru Burdescu, Romania
Ismael Caballero, Spain
Wojciech Cellary, Poland
Panagiotis Chountas, UK
Rebeca Cortazar, Spain
Massimo Cossentino, Italy
Schahram Dustdar, Austria

Ulrich Eisenecker, Germany
Angelina Espinoza, Spain
Maria João Ferreira, Portugal
Ulrich Frank, Germany
Tudor Girba, Switzerland
Cesar Gonzalez-Perez, Spain
Hans-Gerhard Gross, The Netherlands
Ignacio García Rodríguez De Guzmán, Spain
Jo Hannay, Norway
Brian Henderson-Sellers, Australia
Charlotte Hug, France
Zbigniew Huzar, Poland
Stefan Jablonski, Germany
Slinger Jansen, The Netherlands
Wan Kadir, Malaysia
Robert S. Laramee, UK
George Lepouras, Greece
Cuauhtemoc Lopez-Martin, Mexico
Graham Low, Australia
André Ludwig, Germany
Leszek Maciaszek, Australia
Cristiano Maciel, Brazil
Lech Madeyski, Poland
Tom McBride, Australia
Stephen Mellor, Azerbaijan
Sascha Mueller-Feuerstein, Germany
Johannes Müller, Germany
Andrzej Niesler, Poland
Janis Osis, Latvia
Mieczyslaw Owoc, Poland
Eleutherios Papathanassiou, Greece
Marcin Paprzycki, Poland
David Parsons, New Zealand
Oscar Pastor, Spain
Juan Pavon, Spain
Naveen Prakash, India
Lutz Prechelt, Germany
Elke Pulvermueller, Germany
Gil Regev, Switzerland
Félix García Rubio, Spain
Francisco Ruiz, Spain
Krzysztof Sacha, Poland
Motoshi Saeki, Japan
Heiko Schuldt, Switzerland
Manuel Serrano, Spain
Jan Seruga, Australia
Andreas Speck, Germany
Stephanie Teufel, Switzerland
Rainer Unland, Germany
Antonio Vallecillo, Spain
Jean Vanderdonckt, Belgium
Olegas Vasilecas, Lithuania
Benkt Wangler, Sweden
Igor Wojnicki, Poland
Kang Zhang, USA

Auxiliary Reviewer

Valeria Seidita, Italy

Invited Speakers

Pericles Loucopoulos	University of Loughborough, UK
Stephen Mellor	Freeter, UK
Cesar Gonzalez-Perez	LaPa - CSIC, Spain
David Marca	University of Phoenix, USA
Nikolaos Bourbakis	Wright State University, USA

Table of Contents

Pluggable Programming Language Features for Incremental Code Quality Enhancement

Bernhard G. Humm and Ralf S. Engelschall

Darmstadt University of Applied Sciences, Darmstadt, Germany
Capgemini, CSD Research, Munich, Germany
bernhard.humm@h-da.de, ralf.engelschall@capgemini.com

Abstract. Evolutionary prototyping is an incremental software development method in which a proof of concept is, step by step, extended towards the final product. This article pleads for a programming approach termed "incremental code quality enhancement" when developing software incrementally. However, current programming languages are not well suited for incremental code quality enhancement. They are inflexible regarding their use of language features like typing, access control, contracts, etc. In some languages, the programmer is forced to use them, in others he may not. This article introduces pluggable programming language features, a concept that allows greater flexibility for application programmers without losing control over the use of those features. The approach is demonstrated exemplary by interface specifications for a business information system.

Keywords: Programming language features, Aspects, Flexibility, Evolutionary prototyping, Plug-in.

1 Introduction

Flexibility is one of the most basic and important design goals in software engineering [8] [21]. Flexibility allows for adaptation of applications to different and possibly varying needs. This not only applies to the resulting application, but also to the tools for creating them.

However, when analyzing current programming languages and their features concerning their flexibility of use, the result is rather disappointing. Consider just the following two examples.

– Current mainstream programming languages like C/C++, Java, and C# are all statically typed. Static typing is mandatory there and the programmer has no flexibility as to omit type specifications where sensible. Contrarily, dynamic languages like Smalltalk, Scheme, Python and PHP are all dynamically typed and the programmer has no option whatsoever to explicitly specify type declarations statically where sensible.
– Access control in Java is mandatory. For all classes, interfaces and members, accessibility must be declared (public, protected, private or package local as default). The programmer has no option of omitting access control specifications where sensible. On the other hand, declaring access control for packages is not possible at

all in Java. Also, in dynamic languages like Smalltalk, explicit access control on class and method level is not possible. The programmer has no option of specifying access control where sensible.

This inflexibility causes a number of problems:

Coding Overhead. In many industrial software projects, the implementation technology is pre-defined, e.g., Java. Implementing application parts like scripts, code generators, or data migration routines — for which scripting languages are most suitable — in Java result in unnecessary coding overhead due to mandatory language features, whose use is not necessary in this context.

Poor Quality. Contrarily, implementing critical application parts in a dynamic language like Smalltalk may reduce quality — in this case safety — due to missing compile-time checks [18].

One might argue that in such a case, the language choice is simply wrong and an industrial strength language like Java should have been used. This leads us to the next problem.

Incremental Development Impeded. In many project situations it is sensible to develop software incrementally [16], e.g., with evolutionary prototyping [6,2,9,10]. This means that an application or a part of it is quickly prototyped first and then, incrementally, the code quality is being enhanced. If the target application is critical and an industry-scale language like Java is chosen, then quick prototyping is impeded due to many mandatory language features.

The overall picture, today, is that programming languages define a fixed set of features that are to be used. The programmer has no flexibility as to use less strict features where acceptable or even to specify more advanced features where necessary, e.g., access control on package level in Java.

To alleviate those problems, we plead for a concept that we call *“pluggable programming language features”* and argue from the application programmer’s point of view, i.e., from the view of the user of a programming language. Pluggable programming language features are particularly useful for a development method which we call *“incremental code quality enhancement”*.

The article is structured as follows. Sect. 2 motivates for the topic with a discussion on typing. In Sect. 3 we present the concept of pluggable programming language features. Sect. 4 and 5 present a sample application and a research prototype of the concept. Sect. 6 introcudes incremental code quality enhancement and how this can be achieved via pluggable programming language features. Sect. 7 discusses the results. Sect. 8 concludes this article.

2 Static versus Dynamic Typing

In statically typed programming languages, variables and operation parameters are assigned a type which is checked at compile type. In contrast, in dynamically typed languages type checks are performed at run-time. While typing is only one of many programming language features, there is, currently, a strong correlation between dynamically typed and RAD languages and statically typed and mainstream application development languages, respectively. Languages that allow static as well as dynamic typing are rare. Examples are VisualBasic, Perl6, Common Lisp (in part), and, recently, C# 4.0.

The differences between statically and dynamically typed languages are sometimes exaggerated as "language war". Advocates of static typing claim:

– Earlier detection of programming mistakes, e.g. preventing adding an integer to a boolean
– Better documentation in the form of type signatures, e.g. incorporating types of arguments when resolving operation names)
– More opportunities for compiler optimizations, e.g. replacing virtual calls by direct calls when the exact type of the receiver is known statically
– A better design time developer experience, e.g. via auto-completion by the development environment

Advocates of dynamic typing claim:

– Higher coding efficiency since the resulting code is less verbose
– Higher expressiveness via language features like closures, typically found in dynamically typed languages
– Better reusability since variables and operations are not (unnecessarily) restricted in use by types

Language wars are not at all necessary – we fully agree with Meiyer and Drayton in their article "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages" [18].

Typing is only one aspect of a whole spectrum of contracts [19] between providers and users of components – all with the purpose of reducing errors and, therefore, increasing quality. But why stop with typing? We claim that every intrinsic aspect of an operation should be expressed explicitly as part of the contract and should be checked as early as possible: value restrictions, pre- postconditions, invariants, exceptions, usage protocol, etc.

In contrast, statically typed languages allow – and even require – to specify the types of all parameters but do not provide language features to specify more[1]. Static typing alone seems like an arbitrary point in the whole spectrum – for some interfaces it may be adequate to specify constraints more strictly, for others less.

While the adequate degree of contract specification relates to the application in its final, production-ready state, we add another dimension to the argument: the development time. In early development stages, particularly during prototyping, it is not necessary to specify full-featured contracts. The demand for precise contracts increases gradually during the development process.

[1] Eiffel does allow for specifying pre- and postconditions but Eiffel is, currently, not in widespread use.

3 Pluggable Programming Language Features

Before introducing the concept of pluggable programming language features, we need to distinguish two kinds of programming language features.

Core programming language features are essential for implementing applications at all. Examples are objects, classes, operations, variables, and control constructs like loops.

Additional programming language features specify aspects of core language features. Examples are access control for classes, type declaration of variables, and pre- and postconditions of operations. Additional language features are not essential in the sense that it is possible to implement applications without using additional programming language features.

Only additional programming language features may be pluggable. For a programming language to adhere to the concept of pluggable programming language features we postulate the following requirements.

Optional Language Features. The language must allow for implementing applications without using any additional programming language features at all. In particular, static typing must not be mandatory.

Independent Language Features. The language must allow for specifying additional programming language features independently and to check for their conformance at an adequate point in time. In particular, static type checking must be possible.

Extensible Language Features. The programming language must be extensible to allow for the implementation of new additional language features.

Language Feature Configuration. The programming language must allow for configuring the use of additional language features in an application or parts of them. The use may be enabled mandatory or optionally, or disabled. Enabled language features will be checked, e.g., by the compiler.

In total, the concept allows for plugging in additional language features, either pre-defined ones or new ones. Arbitrary use of language features is avoided via language feature configuration.

We see two major use cases for pluggable programming language features.

Customizing Features According to Requirements. Pluggable programming language features allows system architects to customize a programming language with respect to the quality requirements of an application to be developed. Depending on the criticality, more language features may be plugged in — even additional ones that have not been pre-defined in the programming language. The configuration enforces the use of those language features by the programmers.

Customizing Features per Development Stage. Pluggable programming language features allows for efficient incremental software development, particularly with evolutionary prototyping. In an early stage of development, additional language features may be omitted completely by programmers. This allows for rapid prototyping. Such a rapid prototype may be used to get user feedback quickly, as well as checking for architectural integrity of the application. Gradually, the code quality of the application may be enhanced by plugging in additional language features. Language feature configuration gives control over this process. For different stages in the development process, e.g., "Proof of Concept", "Alpha Release", "Beta Release", and "Final Product", specific language features may be enforced.

We now demonstrate the concept via a research prototype and its use via a sample application that focuses on the second use case, namely customizing features per development stage.

4 Example Domain: Customer Management Component Interfaces

4.1 Customer Management

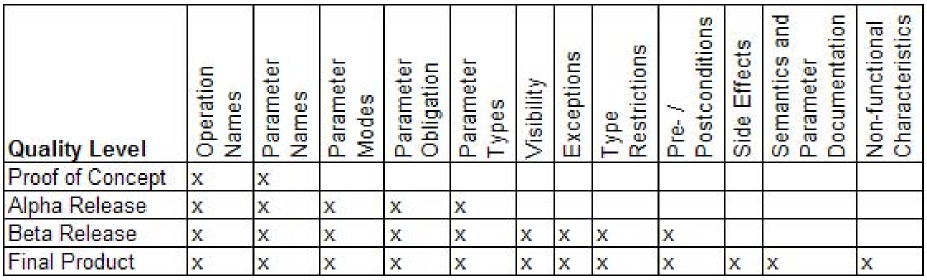
We demonstrate pluggable programming language features exemplary via interfaces for a customer management component of a business information system. We use the term *component* as a functionally coherent unit of software with specified interfaces (provided and required). An *interface* represents the external view of a component. It consists of operations. An *operation* provides functionality of a component. It is specified via syntax (signature) and semantics (behavior). See, e.g., [22].

Our example is the `create-customer` operation with parameters `name`, `address`, and `date-of-birth` for adding a new customer object to a customer management data store. The example seems trivial but may be quite complex in practice. For instance, `address` may be checked for validity syntactically as well as semantically via city map data.

4.2 Interface Specification Aspects

An interface is specified by a *name* and its *operations*. An operation's signature is minimally specified by its *name* and the *parameter names*. Additionally, the following aspects may be specified:

– Access control, e.g., public, private
– Parameter mode: input, output, input/output
– Parameter obligation: mandatory, optional (e.g., expressed by `null` values)
– Parameter types, e.g., primitive types like `Integer` and complex types like `Customer`
– Type restrictions, e.g., only positive `Integer` values for a bank transfer. Note: Type restrictions may be implemented as separate types, e.g., `Positive-Integer`
– Pre- and postconditions: constraints before and after operation execution, respectively — e.g., `date-of-birth < now`. Note: parameter modes, obligations, types and type restrictions may all be specified as pre- and postconditions
– Exceptions: specification of exceptional situations that are externally visible, e.g., duplicate customer
– Side effects specification, e.g., read-only, modifying



Quality Level	Operation Names	Parameter Names	Parameter Modes	Parameter Obligation	Parameter Types	Visibility	Exceptions	Type Restrictions	Pre-/ Postconditions	Side Effects	Semantics and Parameter Documentation	Non-functional Characteristics
Proof of Concept	x	x										
Alpha Release	x	x	x	x	x							
Beta Release	x	x	x	x	x	x	x	x	x			
Final Product	x	x	x	x	x	x	x	x	x	x	x	x

Fig. 1. Language Feature Configuration

- Semantics documentation: specification of the operation's behavior and documentation of its parameters, usually informally in prose. Note: pre- and postconditions are part of the semantic specification, too
- Parameter Documentation: describing the meaning of the operation parameters
- Non-functional characteristics: specifying (formally or informally) performance and other non-functional characteristics

Generally speaking, for a production-grade business information system, the more complete the interface specification, i.e., the more intrinsic information is specified explicitly, the better.

4.3 Language Feature Configuration

Consider, for instance, the development stages "Proof of Concept", "Alpha Release", "Beta Release", and "Final Product". Then, language features may be assigned to the development stages as shown in Fig. 1. For the proof of concept, operation names and parameter names are sufficient. For Alpha Release and Beta Release, the architect demands additional language features like parameter typing, visibility, and exceptions. For the final product, full documentation is mandatory.

In the following section, we describe language features for specifying some of the interface specification aspects exemplary.

5 Language Features for Interface Specification

5.1 Research Prototype in Lisp

We have chosen Lisp[2] [17] as the implementation language for our research prototype to demonstrate pluggable programming language features.

The following features make Lisp ideal for experimenting with language extensions:

Typing. Lisp is dynamically typed yet provides a powerful type system as well as features for object-oriented programming.

[2] More specifically: Allegro Common Lisp, a professional implementation of the ANSI Common Lisp standard.

Code is Data. Lisp has a minimalistic syntax with the list as the basic data structure. Lists are not only used to express application data but also to express Lisp code itself. This makes it particularly convenient to transform Lisp programs via Lisp programs.

Macro Processor. The built-in macro processor allows introducing new language features efficiently and with limited effort.

Unlike Java and C#, Lisp does not provide an explicit language feature for interfaces. But as in other languages like C++, the concept may be emulated.

We have implemented a custom macro `define-function` that extends the basic built-in `defun` macro for defining an operation. `define-function` is a real extension of `defun` in the sense that it accepts all declarations of `defun` but, additionally, optional aspects.

In the next sections, we show some of the language features exemplary step by step by means of the example `create-customer`, thereby incrementally enhancing code quality by the use of pluggable language features.

5.2 Operation and Parameter Naming

In the simplest form (development stage “Proof of Concept”), the *name of an operation* and its *parameter names* are specified only.

```
(define-function create-customer (name address date-of-birth))
```

This expression declares the operation `create-customer` with input parameters `name`, `address`, and `date-of-birth`. No additional language features need to be specified at this stage.

5.3 Parameter Typing

The *type of an input parameter* (necessary for development stage “Alpha Release”) is specified via the keyword `:type` in a list per parameter. The *type of the operation result* (out parameter) is specified via the keyword `:result-type` in an options list following the parameter list.

```
(define-function create-customer
 ((name          :type Structured-Name)
  (address       :type Structured-Address)
  (date-of-birth :type Date))
 (:result-type Customer))
```

The parameter `name` is of type `Structured-Name`, the parameter `address` of type `Structured-Address`, etc.

5.4 Pre- and Postconditions

Pre- and postconditions (necessary for development stages “Beta Release” and “Final Product”) are specified via the keywords `:pre` and `:post` in the options list, followed by a Lisp boolean expression that can be evaluated at run-time.

```
(define-function create-customer
 ((name          :type Structured-Name)
  (address       :type Structured-Address)
  (date-of-birth :type Date))
 (:result-type   Customer
  :pre         (is-valid? address)
  :pre         (lies-in-past? date-of-birth)
  :pre         "No duplicate of previously created customer"
  :post        (get-id result)))
```

The first precondition is satisfied if the operation `is-valid?` with the actual parameter `address` evaluates to true. This checks for valid addresses. `lies-in-past?` checks whether the birth date is plausible. The third pre-condition regarding duplicate checking is treated as an informal comment. The postcondition specifies that the resulting `Customer` object contains a non-`nil` identifier.

5.5 Documentation of Semantics

To document the semantics of the operation and the input and output parameters, the keywords `:documentation` and `:result-documentation` are used in the options list and the parameters lists.

```
(define-function create-customer
 ((name          :type Structured-Name
                 :documentation "Customer name consists of ...")
  (address       :type Structured-Address
                 :documentation "Postal address consists of ...")
  (date-of-birth :type Date
                 :documentation "Customer birth date"))
 (:result-type   Customer
                 :result-documentation "New Customer object"
  :pre           (is-valid? address)
  :pre           (lies-in-past? date-of-birth)
  :pre           "No duplicate of previously created customer object"
  :post          (get-id result)
                 :documentation "Creates a new Customer object"))
```

5.6 Additional Language Features

Analogously, we have implemented the following additional language features: access control, modes, obligations, exception specification, and non-functional characteristics. None of those are natively provided in the core language feature set of Lisp. With our extensions, application programmers may optionally and independently use all of those additional programming language features.

5.7 Conformance Checking

It is not enough to provide language features for specifying interface aspects — the specification conformance has to be checked, too. Therefore, we have implemented the macro `define-function` to generate conformance checks. Type specifications are, if possible, checked at compile time. Pre- and postconditions are checked at runtime. All specification aspects are compiled into the built-in function documentation of Lisp.

But checking the specified aspects is only one kind of conformance check. The macro also checks the conformance of the application code with the language feature configuration during compilation. In case of violations, warnings are being generated. For example, static parameter type checking is enforced from development stage "Beta Release" on as in any statically typed language like Java.

Note: not all all conformance checks can be fully automated. For example, a conformance checker can not decide whether or not there are meaningful preconditions for an operation.

The architect can configure the conformance levels per application or per application parts, e.g., components, and adapt the configuration over development time. Developers get direct feedback whether their code complies to the current conformance level.

6 Incremental Code Quality Enhancement

6.1 Code Quality

Software engineering [8,5], in essence, aims at developing *high-quality* software at *reasonable cost*. *Software quality* [15] can be assessed via quality models like ISO 9126 [11] and has internal and external aspects. Internal aspects address the application developer via ease of development and maintenance. *Code quality* is an important internal aspect which is produced during programming.

In a software development project, the implementation usually is based on an upfront design of a particular architecture. Unfortunately, during implementation one often has to discover deficiencies in this architecture the first time: insufficient separation of concerns, cyclic dependencies, inconsistent interfaces, unsuitable couplings, violated layering, etc. To really fix these kinds of deficiencies, an extensive *refactoring* [7] would be required. Due to time and budget constraints this is either refused and the resulting software is shipped in time but with lower quality, or it is performed and the resulting high-quality software causes the the project to suffer from time and budget overrun. High-quality results and reasonable costs are two contrary goals which are hard to bring into balance.

6.2 Incremental Software Development

One can tackle the problem in advance in two ways: either by investigating more into the design discipline to avoid expensive refactorings at all, or by following an *incremental software development* approach where "merciless refactoring" [1] results in small and less expensive steps all the time. The latter approach is a central aspect in agile software engineering methods [16,12]. One variant is *evolutionary prototyping* [6]. In evolutionary prototyping, an application is implemented prototypically first as a proof of

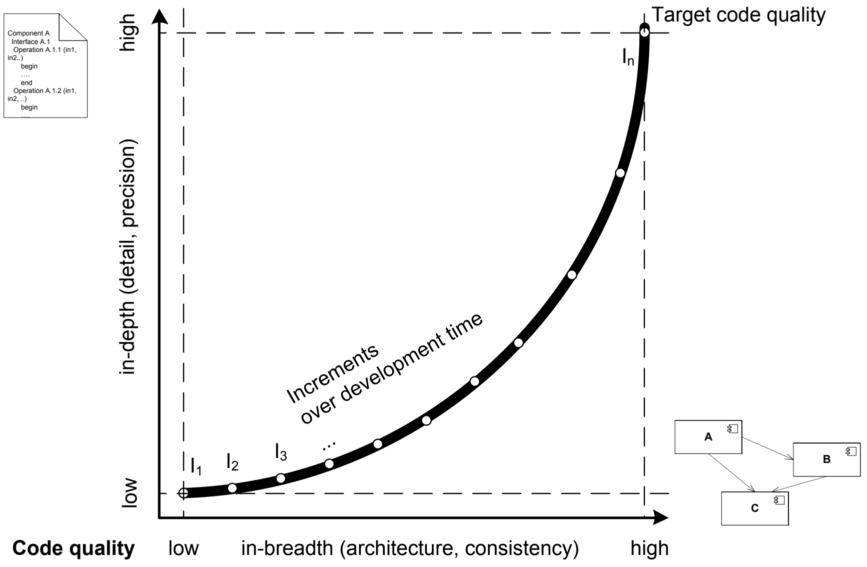

Component A
Interface A.1
Operation A.1.1 (in1, in2..)
begin
end
Operation A.1.2 (in1, in2, ..)
begin
high
in-depth (detail, precision)
low
Target code quality
I_n
Increments over development time
I_1
I_2
I_3
...
A
B
C
Code quality
low
in-breadth (architecture, consistency)
high

Fig. 2. Code Quality Dimensions

concept which is then, incrementally, refined towards the final product, thus constantly enhancing code quality. Evolutionary prototyping may reduce development time and costs while improving user involvement. While evolutionary prototyping is not suitable for all kinds of software projects, its benefits have been proven in numerous projects of different sizes [2,9,14,10].

But how to incrementally enhance code quality in evolutionary prototyping *effectively* and *efficiently* during implementation? To answer this question, we distinguish two *dimensions* (see Fig. 2) of *internal code quality* [11].

In-breadth code quality concerns the structural consistency of the entire application, i.e., the application architecture. Example: conformance of application components to a layer concept.

In-depth code quality concerns correctness and precision of code in detail. Example: specification and validation of an operation's pre-condition.

We plead for optimizing in-breadth internal code quality *before* in-depth internal code quality during incremental development (see Fig. 2). This allows for effective and efficient quality enhancement – for the following reasons.

Efficiency. Optimizing in-breadth quality of relatively small code reduces the costs for refactoring [7] compared to refactoring voluminous code with all details implemented and documented already. For example, splitting a component that does not conform to a layering concept may induce a a lot of refactoring if all operations have already been implemented in detail.

Effectiveness. In-breadth code quality can be optimized largely independently of details. E.g., conformance to a layering concept is independent of individual preconditions. Thus, the effectiveness is not compromised by optimizing in-breadth quality before in-depth quality.

6.3 A Method for Incremental Code Quality Enhancement

When developing incrementally, particularly via evolutionary prototyping, then the architect and the programmers should proceed according to the following method.

1. Select a programming language and programming environment that fulfills the requirements from Sect. 3.
2. Define a language feature configuration according to the quality requirements of the final product. Provide automatic conformance validators where feasible and economically worthwhile.
3. Define language feature configurations for development stages and possibly per application part (e.g., component). Provide automatic conformance validators where feasible and economically worthwhile.
4. Develop the application incrementally. Develop a prototype quickly and with as little effort as possible. Spend sufficient time for optimizing in-breadth code quality and this way improve the architecture.
5. Continue incrementally implementing the application, optimizing in-depth code quality. Control the quality of the increments via conformance checks.
6. Put the application into operation only after the conformance checks for the final product have passed.

6.4 Pluggable Programming Langauge Features and Incremental Code Quality Enhancement

In a way, incremental code quality enhancement "happens" implicitly in most software development projects, today. However, it is not explicitly and as vigorously pursued in a controlled manner as described in the method above. Pluggable programming language features ideally support incremental code quality enhancement. This is because language features can be added optionally and independently during development and, at the same time, there is full control via language feature configurations.

7 Discussion

7.1 Evaluation

This article is a *plea* for pluggable programming language features. We cannot empirically prove the usefulness of the approach. However, our confidence stems from our long-time experience in developing large-scale business information systems and the promising results of our research prototype and sample implementation. Furthermore, we qualitatively justify our approach by evaluating it and the sample implementation against the problems identified in Sect. 1.

Coding Overhead. Pluggable programming language features allow to reduce coding overhead by omitting unnecessary language features in certain application contexts, like scripts, code generators, or data migration routines. A language switch towards a scripting language is not necessary since the programming language itself offers the necessary flexibility.

Poor Quality. Pluggable programming language features allow critical applications to be implemented in a strict manner thus improving code quality. Not only language features common in industrial-strength programming languages can be used. Additionally, even more strict language features may be plugged in. Examples are pre- and postconditions or advanced access control which extends towards packages and components.

Incremental Development Impeded. Pluggable programming language features particularly boost incremental application development, e.g., with evolutionary prototyping. An application or a part of it may be quickly prototyped first and then, incrementally, the code quality may be enhanced (incremental code quality enhancement). Language feature configuration prevents arbitrary use of language features at the programmers' goodwill. Certain quality levels at certain development stages can be enforced.

7.2 Language Support Today

Current programming languages, both in industry and academia, only poorly support pluggable programming language features. Today, there is a strict demarcation of languages focusing either on rapid application development (RAD) or on industry scale development.

Industry Scale Languages, Statically Typed. Languages like Java and C# are currently in mainstream use for developing large-scale, high-quality applications. They are all statically typed and are not well suitable for rapid application development (RAD). More advanced features like pre- and postconditions are not directly provided and may only indirectly be provided, e.g.,via byte-code injection.

RAD Supporting Languages, Dynamically Typed. RAD supporting languages like Perl, Smalltalk, Python, Ruby, Groovy, Scala and F#, conversely, are currently not in mainstream use for developing industry-scale applications. They are either used for throw-away prototyping or for developing special-purpose applications like web sites. Most of them are dynamically typed and do not allow for static typing. Language features may possibly be added – e.g., with Lisp macros as demonstrated in this article – but this is not commonly done.

Hybrid Typing Languages. A few languages like VisualBasic, Perl 6 and Lisp (partially) exist that allow for static as well as dynamic typing. They also allow, in limited ways, for extending the language by new quality features. Neither is in mainstream use. However, with C# 4.0, the first mainstream programming language has recently incorporated dynamic typing optionally — one important step towards pluggable programming language features.

7.3 Related Work

In [18], Meijer and Drayton plead for typing as a pluggable programming language feature. We extend their point of view in three ways. Firstly, we regard typing as one language feature only. Although most important, it represents only one point in a whole spectrum between flexible prototype development and extremely strict development of critical applications. Secondly, we allow for true plugging of programming language features in the sense that new features may be added to the language. Finally, we add the concept of language feature configuration which gives control over the use of language features.

The comparison with Bracha's article "Pluggable Type Systems" [3] is similar. His implementation of Strongtalk [4] on the basis of Smalltalk is an example of a pluggable language feature, namely typing.

With the Scala programming language [20], Odersky targets at scalability and flexibility, too. He tries to reduce the set of language features as much as possible and, instead, provides features in libraries. However, on the level of additional language features like typing and access control, Scala is still inflexible. Scala uses type inference to ease the programmer from the burden of specifying types unnecessarily often but is still statically typed at any time.

Finally, we see a strong relationship between Aspect-Oriented Programming (AOP) [13] and pluggable programming language features. While not inherently tied to it, AOP in practice is used for implementing functionality for the end-user like, e.g., logging. On the other hand, pluggable programming language features target the application programmer by addressing internal code quality like maintainability, stability, reliability, etc. Hence, our approach follows the tradition of AOP, but with a different focus.

8 Conclusions and Future Work

In this article, we plead for pluggable programming language features, a concept that adds flexibility to programming languages. It allows for using or omitting programming language features with full control via language feature configurations. It is particularly suited for incremental code quality enhancement, a development method in which in-breadth code quality is optimized before in-depth code quality.

We demonstrated the concept via a research prototype and a sample application in Lisp. While the concept has obvious benefits, it is not well supported by current programming languages. Furthermore, we agree with Meijer and Drayton, who identify a "huge cultural gap" between the communities of statically and dynamically typed languages [18].

However, we see a new trend towards dynamic programming languages in the last decade that are implemented on top of mainstream platforms. Examples are implementations of Python, Ruby, Groovy, and Scala on the Java Platform or F# and C# 4.0 on the .NET platform. Furthermore, there are a number of Lisp implementations on the Java platform, e.g., ABCL, Clojure, Jatha, and CLForJava.

This may allow for pluggable language features and incremental code quality enhancement to eventually break through — for two reasons. Firstly, the technical integration of languages of different styles eases the implementation of pluggable language features. Optional typing in C# 4.0 is a perfect example for that. Secondly, a growing community of programmers who are proficient in both language styles will help closing the cultural gap. Additionally, if mainstream languages already had real support for pluggable programming language features, the necessity for numerous special languages would be reduced.

Our plea for pluggable programming language features and incremental code quality enhancement is from the application programmers' point of view. We see future work in the following areas. Pluggable programming language features need to be implemented in programming languages on top of mainstream platforms. Integrated development environments need to support pluggable programming language features, particularly their configuration. Experience needs to be gained in industrial projects of different sizes.

References

1. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison Wesley, Reading (2005)
2. Berger, H., Beynon-Davies, P., Cleary, P.: The Utility of a Rapid Application Development (RAD) approach for a large complex Information Systems Development. In: Proceedings of the 13th European Conference on Information Systems (ECIS 2004), Turku, Finland (2004)
3. Bracha, G.: Pluggable type systems. In: OOPSLA Workshop on Revival of Dynamic Languages (2004)
4. Bracha, G., Griswold, D.: Strongtalk: Typechecking Smalltalk in a production environment. In: Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications, OOPSLA 1993 (1993)
5. Broy, M., Jarke, M., Nagl, M., Rombach, H.D.: Dagstuhl-Manifest zur Strategischen Bedeutung des Software Engineering in Deutschland. In: Perspectives Workshop Dagstuhl, Germany (2006)
6. Floyd, C.: A systematic look at prototyping. In: Approaches to Prototyping, pp. 1–18 (1984)
7. Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
8. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall PTR, Upper Saddle River (2002)
9. Gordon, V.S., Bieman, J.M.: Reported Effects of Rapid Prototyping on Industrial Software Quality (1993)
10. Hekmatpour, S.: Experience with evolutionary prototyping in a large software project. SIGSOFT Softw. Eng. Notes 12(1), 38–41 (1987)
11. ISO. TR 9126-4: Software Quality (2004), http://www.iso.org/iso/catalogue_detail.htm?csnumber=39752
12. Kelter, U., Monecke, M., Schild, M.: Do we need 'agile' Software Development Tools? In: NetObjectDays (2002)
13. Kiczales, G., Lamping, J., Mendhekar, Videira Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, Springer, Heidelberg (1997)
14. Lichter, H., Schneider-Hufschmidt, M., Züllighoven, H.: Prototyping in industrial software projects—bridging the gap between theory and practice. In: ICSE 1993: Proceedings of the 15th International Conference on Software Engineering, pp. 221–229. IEEE Computer Society Press, Los Alamitos (1993)

15. Liggesmeyer, P.: Software-Qualität. Testen, Analysieren und Verifizieren von Software. Spektrum Akademischer Verlag (2002)
16. Martin, R.C.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall, Englewood Cliffs (2002)
17. McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Communications of the ACM 3(4), 184–195 (1960)
18. Meijer, E., Drayton, P.: Static Typing Where Possible, Dynamic Typing When Needed. In: Workshop on Revival of Dynamic Languages (2005)
19. Meyer, B.: Object-Oriented Software Construction, 1st edn. Prentice-Hall, Inc., Upper Saddle River (1988)
20. Odersky, M.: An Overview of the Scala Programming Language: EPFL Technical Report IC/2004/64 (2004)
21. Sommerville, I.: Software Engineering, 7th edn. International Computer Science Series. Addison Wesley, Reading (2004)
22. Szyperski, C.: Component software. Addison-Wesley, Harlow (1998)

A Survey on How to Manage Specific Data Quality Requirements during Information System Development

César Guerra-García, Ismael Caballero, and Mario Piattini Velthius

Alarcos Research Group, Department of Information Technologies and Systems, University of Castilla-La Mancha, Paseo de la Universidad 4, Ciudad Real, Spain
CesarArturo.Guerra@alu.uclm.es
{Ismael.Caballero,Mario.Piattini}@uclm.es

Abstract. More and more companies and organizations currently consider that supporting the data in their Information Systems (IS) with an appropriate level of quality is a critical factor for making sound decisions. This has motivated the inclusion of specific mechanisms during IS development, which allow the data to be managed and ensure acceptable levels of quality. These mechanisms should be implemented to satisfy specific data quality requirements which are defined by a user at the moment of using an IS functionality. Since our ultimate research goal is to establish that these mechanisms are necessary for the management of data quality in IS development, we first decided to conduct a survey on related methodological and technical issues in order to determine the current state-of-the-art in this field. This was achieved through the use of a systematic review technique. This paper presents the principal results obtained after conducting the survey, in addition to the principal conclusions reached.

Keywords: Data quality, Requirements specification, Systematic literature review.

1 Introduction

Several authors have reported problems caused by inadequate levels of data quality (DQ) in the use of IS [1]. These problems may negatively affect an organization's performance and additionally involve, among other things, certain types of damage, manifested as an increasingly higher cost in economical terms [2-7]. Once organizations become aware of this situation, they are willing to eradicate these kinds of problems in order to avoid losses.

As an approach towards obtaining data with adequate levels of quality, and consequently reducing the chance of losses, Karel et al. stated that it is necessary to implement mechanisms by means of specific Data Quality Software [8]. The capabilities of Data Quality Software include data cleansing, standardization, matching, merging, enrichment and data profiling. However, these solutions are "post-mortem", and although widely used, they are costly to buy and to implement. In addition, they are not focused on specific users' data quality requirements, which could embrace different data quality dimensions, such as those proposed by Strong et al. in [9] or those that appear in ISO/IEC 25012 [10]. However, they are solely focused on what are commonly called intrinsic requirements such as completeness, or accuracy. The latter are necessary but are not sufficient for a broader kind of related data quality problems, in which data quality requirements go beyond these intrinsic data quality dimensions, as Wang et al. demonstrate in [11].

We propose that, if possible, these kinds of problems require preventive action (such as avoiding storing data which are not reliable or believable) rather than corrective actions (such as the use of data cleansing tools). We agree that corrective actions are necessary [12], but they imply greater costs to organizations (e.g. the time and money needed to execute cleansing processes on data that could already be used within a business process), in which unproductive and out-of-time costs might be incurred for nothing.

The goal of our research is, therefore, to discover how to identify and introduce the appropriate mechanisms by implementing preventive actions at the point-of-entry of organizational IS software. These mechanisms might make it possible to attain a trade-off between preventive and "post-mortem" corrective action, thus minimizing investment or unnecessary costs.

We are conscious that preventive mechanisms must exist as part of the IS development, which requires bringing together specific software requirements with specific data quality requirements for the software being developed: it is important to delimit how software can be enhanced in order to satisfy new data quality requirements. To achieve this goal, we must first identify existing proposals (both *methodological* and *technological*) that have dealt with some kind of solution for the introduction of DQ requirements management as part of IS software development.

In order to conduct a rigorous and complete survey, we decided to use Systematic Review (SR) techniques to attain a strict view of the relevant literature. More precisely, we decided to follow the formal Systematic Review protocol template proposed by Biolchini et al. in [13], since it is one of the most widely used techniques in the field of Software Engineering.

An SR focuses on integrating empirical research with the aim of creating generalizations. This integration challenge involves specific objectives which allow the researcher to critically analyze the data found, thus resolving conflicts in the literary material involved and identifying aspects which need to be researched in the future. The descriptions of the different phases of Biolchini et al.'s protocol are:

1. *Planning*, which primarily focuses on defining the research objectives, the selection of information sources, and the definition of the inclusion and exclusion criteria of studies.
2. The *Execution* phase, which focuses on the selection and evaluation of the studies found, along with extracting information from the selected studies.
3. The *results analysis* phase, which is responsible for analyzing and presenting the results according to the different criteria and perspectives defined above that will facilitate their understanding and subsequent use.

Fig. 1 shows the principal phases of this protocol.

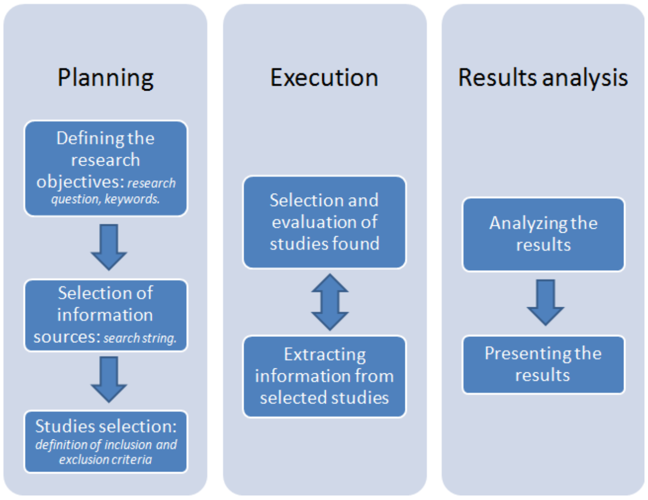

Planning
Defining the research objectives: research question, keywords.
Selection of information sources: search string.
Studies selection: definition of inclusion and exclusion criteria
Execution
Selection and evaluation of studies found
Extracting information from selected studies
Results analysis
Analyzing the results
Presenting the results

Fig. 1. Main phases of Biolchini et al.'s protocol

In summary, it could be said that the principal contribution of this paper is, on one hand, to show the results of the execution of an instance of this protocol for dealing with the *specification and modeling of DQ requirements;* and on the other hand, the consequent conclusions reached after analysing these results.

We have structured the paper into several sections, each of which corresponds with the steps in the different phases of Biolchini et al.'s protocol. . The paper concludes with a section presenting our conclusions, and with an Appendix containing information about the most important works found.

2 Planning the Systematic Review

The expected outcome of the SR is a presentation of the state-of-the-art of existing research proposals for the specification and modelling of DQ requirements, published in the resources available. Once the results have been obtained, the principal beneficiaries of this work will be those people related to software development, such as: systems analysts, designers, programmers and project managers, in addition to academics and researchers related to the data quality area, and other relative areas such as quality in Information Systems and Requirements Engineering.

2.1 Question Formularization

The research question which has motivated the SR is: *"Are there any works that propose mechanisms (both methodological and technological) for the specification, representation and incorporation of data quality requirements during the process of developing an Information System?"*

According to Biolchini et al.'s protocol, and in order to seek a response to our research question, we elaborated a list with keywords, which could be used when querying the different search engines of the bibliographic resources in hand. These keywords are shown in Table 1.

Table 1. Keywords

Data Quality Dimension	Data Quality Metadata
Data Quality Requirement	Data Quality Framework
Data Quality Metamodel	Data Quality Methodology
Data Quality Modeling	Data Quality Dimension
Data Quality Representation	

2.2 Resource Selection

In this section, we show the set of criteria for the selection of resources, the search methods and the specification of the search strings (based on the aforementioned keywords) used in the SR protocol. According to the recommendations of experts in the SR field, the searches should be conducted by using the search engines in the electronic databases of leading publishers. We have also added several specialized resources to the set of these databases: on the one hand, since it is the most important conference in the field, we have included the conference proceedings of the "*International Conference on Information Quality*" (ICIQ, http://mitiq.mit.edu/) since it is the most important international event in the DQ area; and on the other hand we have included the contents of the only two journals dealing with this area: *International Journal of Information Quality "IJIQ"* and *Journal of Data and Information Quality "JDIQ"* (note that the search in both journals was carried out manually). All of these resources contain works of great importance, and most of them offer search engines. The complete list of resources is presented in Table 2.

Table 2. List of resources

ACM Digital Library
IEEE Computer Society
International Conference on Information Quality (ICIQ)
Science Direct
Wiley InterScience
International Journal of Information Quality (IJIQ)
Journal of Data and Information Quality (JDIQ)

Upon considering the list of keywords mentioned above (see Table 1), and combining them through the logical connectors "AND" and "OR" we decided to coin the following search string: *("Data Quality") AND ("requirements" OR "dimensions" OR "Metamodel" OR "Modeling" OR "Model" OR "metadata" OR "framework" OR "Methodology" OR "Modelling" OR "representation" OR "accuracy" OR "completeness" OR "consistency" OR "credibility" OR "currentness" OR "accessibility" OR "compliance" OR "confidentiality" OR "efficiency" OR "precision" OR "traceability" OR "understandability" OR "availability" OR "portability" OR "recoverability").*

Please note that the syntax of the search string may differ according to the specific requirements of the different search engines of the available resources. It is worth noting that it was decided incorporate each of the different DQ dimensions defined in

ISO/IEC 25012 standard in a particular way, with the aim of broadening the range of the search. Table 3 describes each of the data quality dimensions proposed by this standard.

Table 3. Data Quality dimensions proposed by standard ISO/IEC 25012

Dimension	Description
	Inherent
Accuracy	The degree to which data have attributes that correctly represent the true value of the intended attribute of a concept or event in a specific context of use.
Completeness	The degree to which subject data associated with an entity have values for all expected attributes and related entity instances in a specific context or use.
Consistency	The degree to which data have attributes that are free from contradiction and are coherent with other data in a specific context of use.
Credibility	The degree to which data have attributes that are regarded as true and believable by users in a specific context of use.
Currentness	The degree to which data have attributes that are of the right age in a specific context of use.
	Inherent and system dependent
Accessibility	The degree to which data can be accessed in a specific context of use, particularly by people who need supporting technology or a special configuration owing to a disability.
Compliance	The degree to which data have attributes that adhere to standards, conventions or regulations in force and similar rules relating to data quality in a specific context of use.
Confidentiality	The degree to which data have attributes that ensure that they are only accessible and interpretable by authorized users in a specific context of use.
Efficiency	The degree to which data have attributes that can be processed and provide the expected levels of performance by using the appropriate amounts and types of resources in a specific context of use.
Precision	The degree to which data have attributes that are exact or that provide discrimination in a specific context of use.
Traceability	The degree to which data have attributes that provide an audit trail of access to the data and of any changes made to the data in a specific context of use.
Understanda-bility	The degree to which data have attributes that enable them to be read and interpreted by users, and are expressed in appropriate languages, symbols and units in a specific context of use.
	System dependent
Availability	The degree to which data have attributes that enable them to be retrieved by authorized users and/or applications in a specific context.
Portability	The degree to which data have attributes that enable them to be installed, replaced or moved from one system to another, thus preserving the existing quality in a specific context of use.
Recoverability	The degree to which data have attributes that enable them to maintain and preserve a specified level of operations and quality, even in the event of failure, in a specific context of use.

2.3 Studies Selection

Once the resources in which we intended to carry out the searches had been selected, we defined the procedure for selecting the studies, which also included criteria for the inclusion and exclusion of the studies (works) found during the SR.

The procedure used to select studies was basically as follows: initially a researcher read only the title and the abstract of the set of papers found in each of the searches in order to select the most *relevant studies* from each set. After an initial coarse-grained filtering, the researcher analyzed the complete article, deciding which works she or he judged to be unsuitable since they did not make a particularly notable contribution to the DQ requirement field. A list of those studies that were considered to be very important (typically named *primary studies*) was then made. Once this list was considered to be complete, other researchers with a higher expertise in the field were encouraged to verify that the studies actually did provide important knowledge with regard to that area.

The procedure for the selection of primary studies consists of an iterative and incremental process. It is said to be *iterative* because some of the main activities such as searching, reading and information extraction are carried out for each of the selected resources. What is more, if a search does not produce a minimal set of results, it is possible that the search string must be progressively refined to obtain more accurate results. It is said to be *incremental*, because we perform the searches to extract information from a set of potential studies that grows from scratch until the completion of the Systematic Review. In our case, the first author of the paper presented here acted as a researcher, and the remaining authors were those who had greater expertise in the area of DQ.

The inclusion and exclusion criteria defined for this work are explained below. We considered that a work could be included in the results of the SR (namely *Inclusion criteria*) if, and only if:

- The articles described proposals or strategies for the specification and/or modelling of data quality requirements as a software specification.
- The articles were written in English.
- There was an analysis of the title, keywords and summary of each of the studies found.
- There were no restrictions with regard to the date of publication.

On the contrary, we considered that works should not be included in the results of the SR (namely *Exclusion criteria*), if:

- They did not propose any methodology, strategy or model (or metamodel) with which to specify data quality requirements.

As a result of the application of these criteria, we were able to decide which studies found by the searchers could be considered as *primary studies*.

3 Execution of the Selection

After executing the search procedure on the different resources, a total of 820 studies were found (once different versions of the same works had been eliminated). After applying the inclusion and exclusion criteria, only 42 were considered to be important, while only 8 were eventually considered as *primary studies* by the experimental researchers. Table 4 summarizes a report of our findings.

Table 4. Distribution of studies by resource

		Studies		
Resources	***Search Date***	***Found***	***Relevant***	***Primary***
ACM Digital Library	Sept ´10	164	6	4
IEEE Computer Society	Sept ´10	169	9	1
ICIQ	Oct ´10	44	7	2
JDIQ	Oct ´10	12	0	0
IJIQ	Oct ´10	34	0	0
Science Direct	Nov ´10	100	16	1
Wiley InterScience	Nov ´10	297	4	0
	Total	**820**	**42**	**8**

Of all the studies reviewed, only the following were considered as primary studies:

1. *Toward quality data: An attribute-based approach* [14].
2. *Data Quality Requirements Analysis and Modeling* [15].
3. *A flexible and generic data quality metamodel* [16].
4. *IP-UML: Towards a Methodology for Quality Improvement Based on the IP-MAP Framework* [17].
5. *A Product Perspective on Total Data Quality Management* [18].
6. *DQRDFS: Towards a Semantic Web Enhanced with Data Quality* [19].
7. *Quality Views: Capturing and Exploiting the User Perspective on Data Quality* [20].
8. *A Data Quality Metamodel Extension to CWM* [21].

Once the primary studies had been identified, the next step was to extract the relevant information (e.g. technology used, model representation or a proposed methodology) from each one of them. A form with which to better guide this process of extracting relevant information was designed. All the information from the studies is shown in Tables 6 to 13, in the Appendix.

4 Analysis of Obtained Results

Once the information had been extracted from all the primary studies, our principal aim was to address the usability of the primary studies identified in accordance with our interest in discovering proposals dealing with methodological and technological issues. In this section, we show the results of the corresponding analysis. It is worth highlighting that the number of proposals is significantly poor in comparison to the degree of interest that data quality and the information quality field has motivated in recent years. Our concern about this led us to enquire of various DQ researchers and practitioners from different countries and organizations why we had not found more works. Most of them agreed that since data quality is dealt with as a specific issue rather than an organizational issue, many organizations are not yet aware of the possible benefits of our research topic. Most of the researchers interviewed also agreed that the topic is quite relevant because the results could help organizations that develop software to improve the usability of their products at a relatively low cost.

Table 5 summarizes the information extracted from each proposal: the technology or data model used, the existence of a tool or prototype supporting it, the methodology proposed, the inclusion of an example or study case, and reports concerning whether the proposed results have already been tested in a real environment.

Table 5. Relevant information from selected studies

Studies	Model	Tool	Methodology	Example	Tested
Data Quality Requirements Analysis and Modeling [15]	Relational	No	No	Yes	No
Toward quality data: An attribute-based approach [14]	Relational	No	No	Yes	No
A Product Perspective on Total Data Quality Management [18]	Relational	Yes	No	Yes	No
IP-UML: Towards a Methodology for Quality Improvement Based on the IP-MAP Framework [17]	Object Oriented	Yes	No	Yes	No
Quality Views: Capturing and Exploiting the User Perspective on Data Quality [20]	XML	No	No	Yes	No
A flexible and generic data quality metamodel [16]	Relational	No	No	Yes	No
A Data Quality Metamodel Extension to CWM [21]	Object Oriented	No	No	No	Yes
DQRDFS: Towards a Semantic Web Enhanced with Data Quality [19]	XML	No	No	Yes	No

Upon studying the analysis in greater depth, we noted that none of the existing works provide a methodology for obtaining and managing DQ requirements. We had hoped to find a methodology that could, at some point, lead analysts and developers to implement a correct management of data quality requirements from the earliest stages, and throughout the process of developing an Information System. This lack of works addressing methodological and technological issues consequently motivates the challenging research goal of depicting a methodology with which to manage data quality software requirements and combine them with other requirements. On the other hand, and with regard to the technology used, we concluded that since many different kinds of applications could be developed by using different kinds of technologies, some sort of generalization should be used in order to make different kinds of developments possible. This generalization can be achieved by working with models and metamodels. Our most important conclusion in relation to this issue is, therefore, that we should consider the foundations of Model Driven Engineering, MDE [22] and Model Driven Architecture, MDA [23]. The greatest motivation for this is to better generalize our findings so that they will be valid for any kind of possible development by using the same concepts concerning data quality requirements.

It is also worth highlighting that once all the studies found had been analyzed, we discovered that none of them showed a clear and specific definition of the term "*DQ Requirement*". This concept is only mentioned by Wang et al. in [15]. However, its definition is focused on specifying certain indicators of quality that should be related to certain data at the moment of modeling. The definition of this term is, therefore, mandatory if we are to gain a better understanding of it and its subsequent applicability.

In order to coin this concept, we first considered the definition of "*software requirement"* published in the IEEE 610.12-1990 standard [24]:

1. *A condition or capability needed by a user to solve a problem or achieve an objective.*
2. *A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents.*
3. *A documented representation of a condition or capability as in 1 or 2.*

These definitions cover the points of view of both users and developers. The ISO/IEC 25012 standard is also focused on data quality as part of a computational system and it defines quality characteristics for the data used by users and others software systems [10].

After analyzing the above definitions, we thus propose to define a "**DQ requirement**" as follow: "the specification of a set of dimensions or characteristics of DQ that a set of data should meet for a specific task performed by a determined user".

At present, our principal purpose is to define a DQ software requirement which satisfies the specific needs of quality in the data that each user requires at a specific moment to carry out his/hers tasks or functionalities with an Information System.

5 Conclusions

Conducting an SR is a highly intensive task in comparison to that of a conventional literature search. However, if the complete protocol of an SR is followed step by step, then a better validation of the results is generated, and the efforts are worthwhile. The principal goal of this paper is to show the results obtained after the application of the protocol, along with the conclusions reached after carrying out an SR to discover how well the management of data quality requirements (at both the methodological and technological levels) is dealt with in specialized literature. After analyzing the results, it is evident that there is a need for new proposals dealing with methodological issues, owing to the scarcity of existing initiatives aimed at this particular area. Technological issues must be also dealt with. To do this, we can conclude that MDA foundations might be the best environment in which to carry out research into this area.

In this respect, we are currently working on a proposal for a methodology and a metamodel in order to specify, analyze and model DQ-specific requirements. We are considering the incorporation of elements for the management of DQ requirements from the early stage, and their propagation throughout the entire development cycle of any kind of software.

Acknowledgements. This research is part of the PEGASO-MAGO (TIN2009-13718-C02-01), and IQMNet (TIN2010-09809-E (subprogram TIN)) projects, both of which are supported by the Spanish Ministerio de Educación y Ciencia, ENGLOBAS

(PII2I09-0147-8235), and TALES (HITO-2009-14), both supported by the Consejería de Educación y Ciencia of Junta de Comunidades de Castilla-La Mancha.

References

1. Caballero, I., et al.: IQM3: Information Quality Maturity Model. Journal of Universal Computer Science 14, 1–29 (2008)
2. Eppler, M., Helfert, M.: A Classification and Analysis of Data Quality Costs. In: International Conference on Information Quality. MIT, Cambridge (2004)
3. Laudon, K.C.: Data Quality and Due Process in Large Interorganizational Record System. Communications of the ACM 29(1), 4–11 (1986)
4. Mehmood, K., Si-Said, S., Comyn-Wattiau, I.: Data Quality Through Conceptual Model Quality - Reconciling Researchers and Practitioners through a Customizable Quality Model. In: International Conferece on Information Quality, ICIQ 2009, Potsdam, Germany (2009)
5. Thi, T.T.P., et al.: InfoGuard: A Process-Centric Rule-Based Approach for Managing Information Quality. In: European Research Consortium for Informatics and Mathematics ERCIM, pp. 55–56 (2010)
6. Reuters, T., Lepus: Thomson Reuters And Lepus Survey Reveals Data Quality and Consistency Key to Risk Management And Transparency (2010)
7. Wang, R., Storey, V., Firth, C.: A Framework for Analysis of Data Quality Research. IEEE Transactions on Knowledge and Data Engineering 7(4) (1995)
8. Karel, R., Moore, C., Coit, C.: Forrester's report for Business Process and Application Professionals on Trends 2009: Master Data Management, Forrester (2009)
9. Strong, D.M., Lee, Y.W., Wang, R.Y.: Data Quality in Context. Communications of the ACM 40(5), 103–110 (1997)
10. ISO-25012, ISO/IEC 25012: Software Engineering-Software product Quality Requirements and Evaluation (SQuaRE)-Data Quality Model (2008)
11. Wang, R., Strong, D.: Beyond accuracy: What data quality means to data consumers. Journal of Management Information Systems 12(4), 5–33 (1996)
12. Bertino, E., Dai, C., Kantarcioglu, M.: The Challenge of Assuring Data Trustworthiness. In: Zhou, X., Yokota, H., Deng, K., Liu, Q. (eds.) DASFAA 2009. LNCS, vol. 5463, pp. 22–33. Springer, Heidelberg (2009)
13. Biolchini, J.C.D.A., et al.: Scientific research ontology to support systematic review in software engineering. Advanced Engineering Informatics 21(2), 133–151 (2007)
14. Wang, R.Y., Reddy, M., Kon, H.: Towards quality data: An attribute-based approach. Journal of Decision Support Systems 13(3-4), 349–372 (1995)
15. Wang, R.Y., Madnick, S.: Data Quality Requirements: Analysis and Modelling. In: Ninth International Conference on Data Engineering (ICDE 1993). IEEE Computer Society, Vienna (1993)
16. Becker, D., McMullen, W., Hetherington-Young, K.: A Flexible and Generic Data Quality Metamodel. In: International Conference on Information Quality (2007)
17. Scannapieco, M., Pernici, B., Pierce, E.: IP-UML: Towards a Methodology for Quality Improvement Based on the IP-MAP Framework. In: International Conference on Information Quality, ICIQ 2002 (2002)
18. Wang, R.Y.: A Product Perspective on Total Data Quality Management. Communications of the ACM 41(2), 58–65 (1998)

19. Caballero, I., et al.: DQRDFS:Towards a Semantic Web Enhanced with Data Quality. In: Web Information Systems and Technologies, Funchal, Madeira, Portugal (2008)
20. Missier, P., et al.: Quality views: capturing and exploiting the user perspective on data quality. In: Proceedings of the 32nd International Conference on Very Large Data Bases, vol. 32 (2006)
21. Gomes, P., Farinha, J., Trigueiros, M.J.: A data quality metamodel extension to CWM. In: Proceedings of the Fourth Asia-Pacific Conference on Comceptual Modelling, vol. 67, pp. 17–26. Australian Computer Society, Inc., Ballarat (2007)
22. Bézivin, J.: In Search of a Basic Principle for Model Driven Engineering. UPGRADE 2(2), 21–24 (2004)
23. OMG, MDA Guide Version 1.0.1., Object Management Group, p. 62 (2003)
24. IEEE, IEEE Std 610.12-1990 IEEE Standard Glossary of Software Engineering Terminology -Description (1990)
25. Shankaranarayan, G., Wang, R.Y., Ziad, M.: IP-MAP: Representing the Manufacture of an Information Product. In: Fifth International Conference on Information Quality (ICIQ 2000). MIT, Cambridge (2000)
26. Ballou, D.P., Wang, R.Y., Pazer, H.: Modelling Information Manufacturing Systems to Determine Information Product Quality. Management Science 44(4), 462–484 (1998)
27. Bernes-Lee, T., Hendler, J., Lassila, O.: The Semantic Web. Scientific American, Singapore (2001)
28. OMG. Common Warehouse Metamodel (CWM) Specification v1.1. (2003), (cited October 2008) `http://www.omg.org/docs/formal/03-03-02.pdf` (Consulted: 29-09-2008)

Appendix

Table 6. Primary study by Wang et al. [15]

	Data Extraction of the Study
Publication	Richard Wang, Henry Kon, and Stuart Madnick. April, 1993. *Data Quality Requirements Analysis and Modeling*. In: Proceedings of the Ninth International Conference of Data Engineering. Austria.
	Objective Results of the Study
Proposal	The article is focused on: (1) establishing a set of premises, terms and definitions for the management of DQ, and (2) developing a step by step methodology for defining and documenting DQ parameters for users. The requirements analysis methodology proposed by the authors is based on two principal approaches: - Specification of *labels* needed for users with the objective of assessing, determining or improving data quality. - Obtaining, from the user viewpoint, the general aspects of DQ non-sensitive to labeling, for example, the features of completeness and response time. A series of views (view of application, view of parameters and quality view) is also proposed which should be included as part of the documentation of quality requirements specification, the authors jointly refer to a list of possible data quality candidates.
Results	Methodology for collecting and documenting data quality requirements.
Model	It uses a “Relational” type of model.
Used methodology	None.
Difficulties	There is no definition and standardization of quality dimensions.

Table 7. Primary study by Becker et al. [16]

	Data Extraction of the Study
Publication	David Becker, William McMullen y Kevin Hetherington-Young. November, 2007. *A flexible and generic data quality metamodel.* In: Proceedings of the 12th. International Conference on Information Quality, ICIQ 2007. U.S.A.
	Objective Results of the Study
Proposal	Analyze and describe three generic metamodels mentioning some of their most important capabilities: Common Warehouse Metamodel (CWM), Data Warehouse Quality (DWQ) and Universal Meta Data Model. The authors propose an architecture and a basic metamodel for DQ, which meets the objectives of adequately providing flexibility, generality and ease of use of the requirements in situations of potential use. This metamodel adequately represents the information products (IP), data objects, and metrics, measurements, requirements, evaluations and actions of DQ.
Results	Proposal for architecture and a generic metamodel for DQ.
Model	It uses a "Relational" type of model.
Used methodology	None.
Difficulties	No mention of any.

Table 8. Primary study by Wang et al. [14]

	Data Extraction of the Study
Publication	Richard Wang, Reddy, M., Kon, H.. March, 1995. *Toward quality data: An attribute-based approach.* In: Journal of Decision Support Systems. U.S.A.
	Objective Results of the Study
Proposal	The authors propose a quality perspective using labeled data in a cell level with quality indicators, which are objective characteristics of the data and its manufacturing process. Based on these indicators, the user can evaluate the quality of data for a specific application. The authors additionally investigate how these quality indicators can be specified, stored, retrieved and processed. They propose a data model based on attributes, query algebra and integrity rules that facilitate cell-level tagging, along with data processing of the application that is augmented with quality indicators.
Results	A methodology for analyzing data quality requirements based on an entity-relationship model, for the specification of the types of quality indicators to be modelled.
Model	It uses a "Relational" type of model.
Used methodology	None.
Difficulties	Study and research object-oriented approach, since the relational model that represents the schema of quality may be restrictive. An object-oriented approach seems simpler to model the data and its quality indicators, because many of the quality control mechanisms are oriented towards procedures and this approach could manage them without any problem.

Table 9. Primary study by Wang [18]

Data Extraction of the Study	
Publication	Richard Wang. February, 1998. *A Product Perspective on Total Data Quality Management*. In: Communications of the ACM. U.S.A.
Objective Results of the Study	
Proposal	This article presents the Total Data Quality Management (TDQM) methodology, whose main purpose is to deliver High quality information products (IP) to information consumers, along with introducing the concepts of TDQM cycle and information products. It explains the stages of the TDQM related to the information products: Definition, Measurement, Analysis and Improvement, with particular emphasis on defining the characteristics of information products and quality requirements of the information. The author also shows a software tool with which to conduct surveys to assess the quality of information, with which it may be possible to evaluate a list of quality dimensions defined by the author.
Results	It shows the TDQM methodology and illustrates how it can be put into practice in a wide range of organizations.
Model	It uses a "Relational" type of model.
Used methodology	None.
Difficulties	None.

Table 10. Primary study by Caballero et al. [19]

Data Extraction of the Study	
Publication	Ismael Caballero, Eugenio Verbo, Coral Calero y Mario Piattini . May, 2008. *DQRDFS: Towards a Semantic Web Enhanced with Data Quality*. In: 4th. International Conference on Web Information Systems and Technologies, WEBIST ´08. Portugal.
Objective Results of the Study	
Proposal	This article introduces a new view of the Semantic Web, based on the concept of quantity of data quality (QDQ), in which DQ aspects are used as a base to enable machines to process documents from the Semantic Web for different activities such as information retrieval or document filtering. The Semantic Web is an extension of the current Web in which the information is provided with a well-defined meaning, thus enabling computers and people to cooperate [27]. This article has a twofold goal: (1) it shows the readers a brief introduction to DQ, and (2) it shows how the DQ fundamentals have been applied with the aim of highlighting the quality of Web documents for the Semantic Web. The first step in order to permit DQ in the semantic web is to identify the set of elements that need to be studied from the User Requirements Specification for the DQ (DQ-URS). The second step is to identify the DQ dimensions and their related metadata. The third step is to obtain and record the values for the metadata. This information is represented by using XML-type documents.
Results	It shows a proposal of the concept of QDQ oriented towards the Semantic Web.
Model	It uses the XML language (Extensible Markup Language) for its representation.
Used methodology	None.
Difficulties	None.

Table 11. Primary study by Missier et al. [20]

	Data Extraction of the Study
Publication	Paolo Missier, Suzanne Embury, Mark Greenwood, Alun Preece y Binling Jin. September, 2006. *Quality Views: Capturing and Exploiting the User Perspective on Data Quality*. In: International Conference on Very large databases, VLDB ´06. Korea.
	Objective Results of the Study
Proposal	This article presents a quality user-centered model and a software environment, which domain experts can use to easily and rapidly code and test their own heuristics quality criteria. As a core of the model, they propose the concept of "quality view", similar to customized "*lenses"*, through which the data can be observed. The main contributions of this work are: (1) An extensible semantic model for the user for concepts of quality information in e-science. (2) A process model and a declarative language with which to specify abstract views of quality in terms of a few logical operators. (3) An architecture for implementing quality views within many data processing environments.
Results	It proposes a framework for specifying requirements for quality processing by the user, called "quality views".
Model	It uses the XML language (Extensible Markup Language) for its representation.
Used methodology	None.
Difficulties	None.

Table 12. Primary study by Gomes et al. [21]

	Data Extraction of the Study
Publication	Pedro Gomes, José Farinha, Maria José Trigueiros. February, 2007. *A Data Quality Metamodel Extension to CWM*. In: 4th. Asia-Pacific Conference on Conceptual Modelling, APCCM 2007. Australia.
	Objective Results of the Study
Proposal	This paper proposes a metamodel for data quality and data cleaning, both concepts being applicable to the context of data warehouses. This metamodel is integrated with the "Common Warehouse Metamodel" [28],thus providing an extension of this standard towards data quality. It also provides a set of modelling guidelines for the storage of formal specifications of DQ rules. The main purpose of the metamodel is to provide support to profiling and data cleaning activities, with rules that can be established with the aim of detecting data quality problems. It also establishes data cleaning solutions. In relation to data cleaning, a "metadata" holder is provided, with the objective of enabling the ultimate goal of achieving the highest possible level of automation. However, a metadata support is also provided when the user's participation is required in the cleaning process.
Results	It displays a metamodel for quality and data cleaning, both concepts being applied to the context of data warehouses.
Model	It uses an "Object Oriented" type of model.
Used methodology	None.
Difficulties	None.

Table 13. Primary study by Scannapieco et al. [17]

Data Extraction of the Study	
Publication	Monica Scannapieco, Barbara Pernici y Elizabeth Pierce. November, 2002. *IP-UML: Towards a Methodology for Quality Improvement Based on the IP-MAP Framework.* In: Proceedings of the Seventh International Conference on Information Quality, ICIQ´02. U.S.A.
Objective Results of the Study	
Proposal	It proposes a UML profile for data quality in order to sustain the quality improvement within an organization. This profile is based on the IP-MAP Framework [25], but differs from it, mainly because: (1) it specifies the artifacts for production during the improvement process in terms of diagrams drawn using UML elements defined in the data quality profile, (2) it uses the IP-MAP not only to evaluate the quality and think about the improving actions, but also as a schematic means to design and implement improving actions. The IP-MAP is an extension of a Information Manufacturing System (IMS) proposed by [26]. This Framework has the advantage of combining both data analysis and process analysis, with the aim of assessing the quality of the data. The data quality profile consists of three different models: Data Analysis Model, Quality Analysis Model and Quality Design Model. The data analysis model specifies which data are important to consumers because their quality is critical to organizations' success. The quality analysis model consists of modelling the elements that permit the representation of the data quality requirements, a quality requirement can be related to a dimension of quality or features that are typically defined for data quality. The quality design model incorporates the perspective of IP-MAP, which helps in understanding the details associated with the manufacturing process of the information products.
Results	Shows a profile and a methodology for producing UML artifacts designed by the data quality profile.
Model	It uses an "Object Oriented" type of model.
Used methodology	None.
Difficulties	None.

Constructing a Catalogue of Conflicts among Non-functional Requirements

Dewi Mairiza and Didar Zowghi

School of Software, Faculty of Engineering and Information Technology,
University of Technology, Sydney (UTS), Australia
{Dewi.Mairiza,Didar.Zowghi}@uts.edu.au

Abstract. Non-Functional Requirements (NFRs) are recognized as a critical factor to the success of software projects because they address the essential issue of software quality. NFRs tend to interfere, conflict, and contradict with one another and this conflict is widely acknowledged as one of the key characteristics of NFRs. Several models of NFRs conflicts have been proposed and the interacting nature of NFRs has been characterized as either positive or negative inter-relationships among NFRs. Positive relationship represents a pair of NFRs that are supporting each other while negative relationship represents those NFRs that are conflicting with one another. Furthermore, as NFRs are also relative, the interpretation of NFRs may vary depending on many factors such as the context of the system being developed and the extent of stakeholders' involvement. The multiple interpretations of NFRs may lead to positive or negative inter-relationships that are not always obvious. These relationships may change depending on the meaning of NFRs in the system being developed. Hence, the existing potential conflicts models remain in disagreement with one other. This paper presents the result of an extensive and systematic investigation of the extant literature over the notion of NFRs and the conflicts among them. Rigorous synthesis of the carefully reviewed literature has resulted in the construction of a catalogue of NFRs conflicts with respect to NFRs relative characteristic. The relativity of conflicts is characterized by three categories: *absolute conflict; relative conflict; and never conflict.* This comprehensive catalogue could assist software developers with identifying the NFRs conflicts, performing conflicts analysis, and suggesting potential strategies to resolve these conflicts.

Keywords: Non-functional requirements, Relationship, Conflict, Relative, Catalogue.

1 Introduction

In the early eighties, the term Non-Functional Requirements (NFRs) was introduced as those requirements that restrict the type of solutions that a software system might consider [1]. However, although this term has been in use for almost three decades, studies to date indicate that currently there is no general consensus in the software or systems engineering community regarding the notion of NFRs. In the literature, the term NFRs is considered within two different perspectives: (1) NFRs as the requirements that describe the properties, characteristics or constraints that a software system must exhibit; and (2) NFRs as the requirements that describe the quality attributes that the software product must have [2].

In software development, NFRs are recognized as a critical factor to the success of software projects. NFRs address the essential issue of the quality of the system [3-5]. Without well-defined NFRs, a number of potential problems may occur, such as a software which is inconsistent and of poor quality; dissatisfaction of clients, end-users, and developers toward the software; and causing time and cost overrun for fixing the software [5]. NFRs are also considered as the constraints or qualifications of the operations [6]. They place restrictions on the product being developed, development process, and specify external constraints that the product must exhibit [7]. Charette [8] claims that NFRs are often more critical than individual Functional Requirements (FRs) in the determination of a system's perceived success or failure [9, 10]. Neglecting NFRs has led to a series of software failures. For example systemic failure in London Ambulance System [11, 12], performance and scalability failure in the New Jersey Department of Motor Vehicles Licensing System [13], failure in the initial design of the ARPANet Interface Message Processor Software [14], and some other examples as described in [11, 13-15].

Although NFRs are widely recognized to be very significant in the software development, a number of empirical studies reveal that NFRs are often neglected, poorly understood and not considered adequately in developing the software applications. In the development of software systems, users naturally focus on specifying their functional or behavioral requirements, i.e. the things the product must do [5, 9]. NFRs are often overlooked in the software development process [3, 16]. A number of studies investigating practices of dealing with NFRs in the software industry also reported that commonly software developers do not pay sufficient attention to NFRs [3, 16-18]. NFRs are not elicited at the same time and the same level of details as the FRs and they are often poorly articulated in the requirements documents [17, 18]. Furthermore, in the requirements engineering literature, NFRs have received less attention and not as well understood as FRs [5]. Majority of software engineering research, particularly within requirements engineering area only deal with FRs, i.e. ensuring that the necessary functionality of the system is delivered to the user [19]. Consequently, capturing, specifying, and managing NFRs are still difficult to perform due to most of software developers do not have adequate knowledge about NFRs and little help is available in the literature [20].

NFRs tend to interfere, conflict, and contradict with one another. Unlike FRs, this inevitable conflict arises as a result of inherent contradiction among various types of NFRs [3, 5]. Certain combinations of NFRs in the software system may affect the inescapable trade offs [3, 9, 13]. Achieving a particular type of NFRs can hurt the achievement of the other type(s) of NFRs. Hence, this conflict is widely acknowledged as one of many characteristics of NFRs [5].

Prior studies reveal that dealing with NFRs conflicts is essential due to several reasons [2]. First of all, conflicts among software requirements are inevitable [5, 21-23]. Conflicting requirements are one of the three main problems in software development in term of the additional effort or mistakes attributed to them [23]. A study of two-year multiple-project analysis conducted by Egyed & Boehm [24, 25] reports that between 40% and 60% of requirements involved are in conflict, and among them, NFRs involved the greatest conflict, which was nearly half of requirements conflicts [26]. Lessons learnt from industrial practices also confirm that one of the essential aspects during NFRs specification is management of conflicts among interacting NFRs [3]. Experience shows most systems suffer with severe tradeoffs among the major groups of NFRs. For example: the tradeoffs between security and performance requirements; or between security and usability requirements. In fact, conflicts resolutions for handling NFRs conflicts often result in changing overall design guidelines, not by simply changing one module [3]. Therefore, since conflicts among NFRs have also been widely acknowledged as one of NFRs characteristics, managing these conflicts as well as making them explicit is essential [19]. NFRs conflicts management is important for finding the right balance of attributes satisfaction, in achieving successful software products [9, 13].

A review of various techniques to manage the conflicts among NFRs have been presented in the literature [2]. Majority of these techniques provide a documentation, catalogue, or list of potential conflicts. These catalogues represent the interrelationships among various types of NFRs. Apart from strength and weaknesses of each technique, however, NFRs are also relative [5]. This means that the interpretation and importance of NFRs may vary depending on many factors, such as the particular system being developed as well as the extent of stakeholder involvement. NFRs can be viewed, interpreted, and evaluated differently by different people and different contexts within which the system is being developed. Consequently, the positive or negative relationships among them are not always obvious. These relationships might change depending on the meaning of NFRs in the context of the system being developed. Due to this relative characteristic of NFRs, existing potential conflicts models that represent the relationship among NFRs are often in disagreement with each other. For example, according to Wiegers [9] efficiency requirements have negative relationship (conflict) with usability requirements, but according to Egyed & Grünbacher [27] these two types of NFRs have positive relationship (support). Given that none of the existing conflicts catalogues deal with the relative characteristics of NFRs, we are motivated to pose the following research question:

"Can a catalogue of conflicts among NFRs be developed with respect to the relative characteristic of NFRs?"

The catalogue of conflicts with respect to the NFRs relative characteristic that has been developed from a rigorous synthesis of the literature from several disciplines is presented as the novel contribution of this paper. This catalogue is built as a two-dimensional matrix that represents the conflict-relationships between various types of NFRs, i.e. how each type of NFRs is associated with the other types of NFRs considering the NFRs relative characteristic. The conflict-relationships are represented in three categories: *absolute conflict; relative conflict; and never conflict.*

This article is organized in six sections. The first section is the introduction to NFRs and conflicts among them. The second section describes the research framework and source of information used in this study. The superset list of NFRs is presented in section three continued by presenting the catalogue of NFRs conflicts in section four. Section five describes the benefits and potential applications of the conflicts catalogue in the software development projects. Then, section six concludes this paper by highlighting some open issues that are acquired from the investigation.

2 Catalogue Framework

To get a significant and comprehensive snapshot of the NFRs conflicts model, an extensive investigation of the literature over the last three decades has been performed. This investigation was conducted by exploring the articles from academic resources and documents from software development industry. Four general types of sources of information have been identified: (1) journal papers; (2) conference proceedings; (3) books; and (4) documents from software industry. Selection of those sources is made in order to confirm the completeness of the information by obtaining the academics and practitioners perspectives related to the notion of NFRs and conflicts among them. The study conducted by Chung et al. [5] was used as the starting point for selection of the papers to be reviewed.

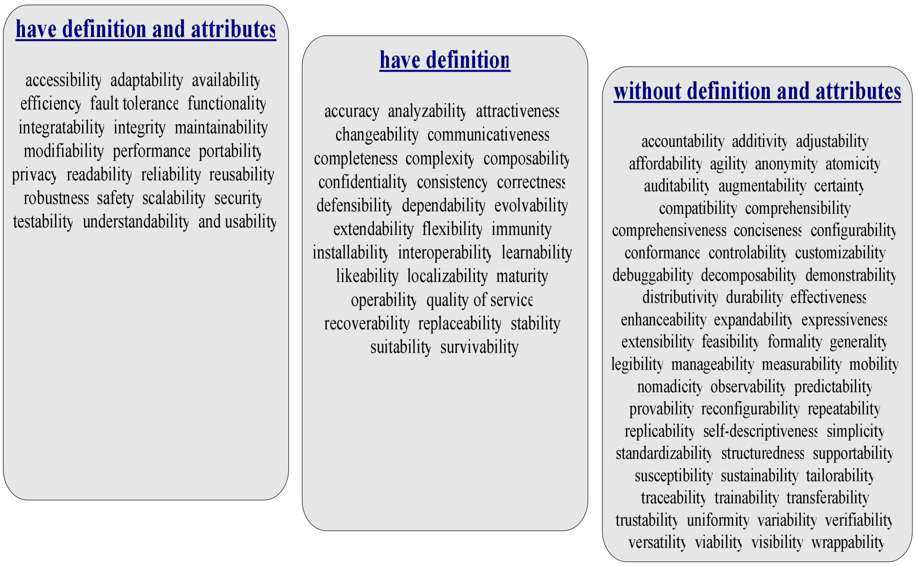

have definition and attributes
accessibility adaptability availability
efficiency fault tolerance functionality
integratability integrity maintainability
modifiability performance portability
privacy readability reliability reusability
robustness safety scalability security
testability understandability and usability
have definition
accuracy analyzability attractiveness
changeability communicativeness
completeness complexity composability
confidentiality consistency correctness
defensibility dependability evolvability
extendability flexibility immunity
installability interoperability learnability
likeability localizability maturity
operability quality of service
recoverability replaceability stability
suitability survivability
without definition and attributes
accountability additivity adjustability
affordability agility anonymity atomicity
auditability augmentability certainty
compatibility comprehensibility
comprehensiveness conciseness configurability
conformance controlability customizability
debuggability decomposability demonstrability
distributivity durability effectiveness
enhanceability expandability expressiveness
extensibility feasibility formality generality
legibility manageability measurability mobility
nomadicity observability predictability
provability reconfigurability repeatability
replicability self-descriptiveness simplicity
standardizability structuredness supportability
susceptibility sustainability tailorability
traceability trainability transferability
trustability uniformity variability verifiability
versatility viability visibility wrappability

Fig. 1. NFRs Types in the Literature

Our study has examined 182 sources of information. All of them are literatures within the discipline of software engineering. They cover various issues of NFRs and conflicts among them. The research articles reviewed are published in key journals and conference proceedings of the software engineering literature, such as the Journal of Systems and Software; IEEE Transaction on Software Engineering; IEEE Software; Lecture Notes in Computer Science; Journal of Information and Software Technology; Requirements Engineering Journal; Requirements Engineering Conference, International Conference on Software Engineering, and Requirements Engineering Foundations of Software Quality Workshop.

Each source was then systematically analyzed using content analysis technique. Content analysis is a research technique that uses a set of procedures to make valid inferences from texts or other meaningful matter [28, 29]. This technique is well founded and has been in used for over sixty years. The analysis covers three essential issues: the NFRs types, the definition and attributes[1] of each type, and the conflict interdependencies among them. Content analysis technique was selected because it enables researchers to identify trends and patterns in the literature through the frequency of keywords, and by coding and categorizing the data into a group of words with similar meaning or connotations [29, 30]. Furthermore, this technique is also applicable to all domain contexts [28, 31].

To develop a catalogue of NFRs conflicts, a research framework was followed. This framework consists of three research stages:

(1) to create a comprehensive catalogue of NFRs types, their definition and attributes characterization
(2) to identify the interdependencies among NFRs
(3) to perform a normalization process to standardize the NFRs in the conflicts catalogue

Since there is no standard catalogue of NFRs types available in the literature and previous studies [32-34] also claimed that many types of NFRs were introduced without definition or attributes characterization, the first stage of the research was creating a comprehensive catalogue of NFRs types. Each type of NFRs discussed in the literature was recorded. The definitions and attributes correspond to each of NFRs type were also documented. Conflicting terminologies and definitions were handled through the frequency analysis technique and keywords identification.

Table 1. NFRs Types in the Initial Catalogue

	NFRs Types	
Accuracy	Interoperability	Reliability
Analyzability	Legibility	Reusability
Availability	Maintainability	Robustness
Compatibility	Performance	Safety
Confidentiality	Portability	Security
Dependability	Privacy	Testability
Expresiveness	Provability	Understandability
Flexibility	Recoverability	Usability
Functionality		Verifiability

The second stage of the research was creating an initial catalogue of the conflicts among NFRs. In this stage, NFRs conflict relationships were used as the criteria to develop the catalogue. This stage was initiated by identifying the interdependencies

[1] In this paper, the term attribute is considered as the major components of each NFRs type. In the literature, attribute is also referred as NFRs subtype [5] or quality sub factors [4].

among various types of NFRs. These interdependencies represent the typical interrelationships of a particular type of NFRs towards another type of NFRs (e.g. positive, negative, or neutral interrelationships). This investigation produced the initial catalogue that presents the conflict relationships among 26 types of NFRs. These NFRs types are listed in Table 1.

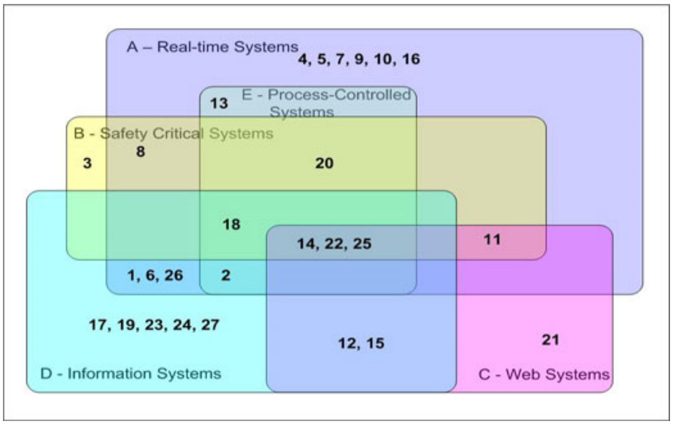
Table 2. NFRs Definition and Attributes [34]

NFRs	Definition	Attributes
Performance	requirements that specify the capability of software product to provide appropriate performance relative to the amount of resources needed to perform full functionality under stated conditions	response time, space, capacity, latency, throughput, computation, execution speed, transit delay, workload, resource utilization, memory usage, accuracy, efficiency compliance, modes, delay, miss rates, data loss, concurrent transaction processing
Reliability	requirements that specify the capability of software product to operates without failure and maintains a specified level of performance when used under specified normal conditions during a given time period	completeness, accuracy, consistency, availability, integrity, correctness, maturity, fault tolerance, recoverability, reliability, compliance, failure rate/critical failure
Usability	requirements that specify the end-user-interactions with the system and the effort required to learn, operate, prepare input, and interpret the output of the system	learnability, understandability, operability, attractiveness, usability compliance, ease of use, human engineering, user friendliness, memorability, efficiency, user productivity, usefulness, likeability, user reaction time
Security	requirements that concern about preventing unauthorized access to the system, programs, and data	confidentiality, integrity, availability, access control, authentication
Maintainability	requirements that describe the capability of the software product to be modified that may include correcting a defect or make an improvement or change in the software	testability, understandability, modifiability, analyzability, changeability, stability, maintainability compliance

The next stage was performing a normalization process against 26 types of NFRs that have been identified in the initial catalogue. This normalization was conducted in order to standardize the data obtained in the previous stage. Normalization is the process of removing the irrelevant NFRs, i.e. the types of NFRs that do not have definition and/or attributes, from the initial catalogue. The objective is to produce a conflicts catalogue of the well-defined NFRs types. In this normalization, the catalogue of NFRs types, their definitions, and their attributes are utilized as the basis of removing those irrelevant NFRs. This process has removed six NFRs from the initial catalogue. They are *compatibility, expressiveness, legibility, provability, verifiability* and *analyzability*. Therefore, the final conflicts catalogue is a two-dimensional matrix that represents the conflict interrelationships among 20 types of “normalized” NFRs.

3 NFRs Types

Various authors (e.g. [5, 35, 36]) define the term NFRs as the requirements that specify the desired quality attributes of the system. According to this definition, our analysis of NFRs types in the literature has resulted in identification of 114 types of NFRs. The superset list of these 114 NFRs types can be found in our previous publication [34].



Legend:

1	Accuracy	10	Installability	19	Reusability
2	Availability	11	Integrity	20	Safety
3	Communicativeness	12	Interoperability	21	Scalability
4	Compatibility	13	Maintainability	22	Security
5	Completeness	14	Performance	23	Standardizability
6	Confidentiality	15	Privacy	24	Traceability
7	Conformance	16	Portability	25	Usability
8	Dependability	17	Provability	26	Verifiability
9	Extensibility	18	Reliability	27	Viability

Fig. 2. Mapping of Concerned NFRs and Types of Systems [34]

Further investigation to the superset list indicates that 23 NFRs types (20.18%) have definition and attributes, 30 types (26.32%) only have definition, and the rest 61 types (53.50%) were introduced without definition or attributes. Since this finding indicates that more than 50% of NFRs listed in the literature do not have any definitions and attributes characterization, therefore, it confirms the previous claim made by Glinz [32, 33] that stated that "in the literature, many NFRs were introduced without definition or clarifying examples". The detailed list of this classification is presented in Fig. 1. In addition, the top five of the most frequently discussed NFRs types in the literature are presented in Table 2 and the concerned NFRs in various types of systems are presented in Fig. 2.

4 Catalogue of Conflicts

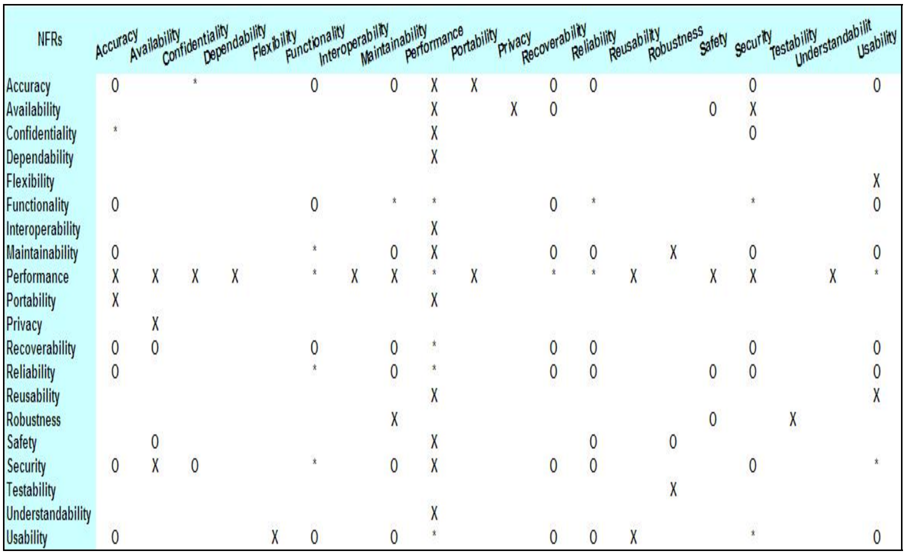
The catalogue of conflicts is a two-dimensional matrix that represents the typical interrelationships among 20 types of normalized NFRs, in term of the conflicts emerge among them. In this catalogue, the relativity of NFRs conflicts is presented in three categories: *absolute conflict; relative conflict;* and *never conflict* (as presented in Fig. 3).

- *absolute conflict.* this relationship represents a pair of NFRs types that are always in conflict. In the catalogue, this conflict relationship is labeled as 'X'.
- *relative conflict.* this relationship represents a pair of NFRs types that are sometimes in conflict. It consists of all pairs of NFRs that are claimed to be in conflict in a certain case but they are also claimed as not being in conflict in the other cases. This disagreement occurs due to several factors, such as the different interpretation/meaning of NFRs in the system being developed, the context of the system, the stakeholders' involvement, and the architectural design strategy implemented in that system. In the conflicts catalogue, this type of conflict relationship is labeled as '*'.
- *never conflict.* this relationship represents a pair of NFRs types that in the software development projects are never in conflict. It consists of all pairs of NFRs who have never been declared as being in conflict with each other. They may contribute either positively (e.g. support [37] or cooperative [27]) or indifferent to one another (e.g. low or very little impact on the other [9]).

Further analysis of the conflicts catalogue indicates that 36 pairs of NFRs are absolute conflict (e.g. accuracy and performance; security and performance; and usability and reusability); 19 pairs are relative conflict (e.g. reliability and performance; usability and security; and performance and usability); and 50 pairs are never conflict (e.g. accuracy and maintainability; security and accuracy; and usability and recoverability). The rest of relationships are not known due to there is no information available in the literature about how those pairs of NFRs contribute to each other. In the conflicts catalogue, this unknown conflict is presented as "the blank spaces".

Furthermore, this catalogue shows that NFRs with the most conflict with other NFRs is performance. Performance has absolute conflict with accuracy, availability, confidentiality, dependability, interoperability, maintainability, portability, reusability, safety, security, and understandability, and it has relative conflict with functionality, recoverability, reliability, and usability.

The investigation also indicates that certain attributes of a particular type of NFR can be in conflict with each other. This conflict points to the self-conflicting relationships for a particular type of NFR. Self-conflicting relationship is defined as a situation where the attributes of a single type of NFRs are in conflict. One of the examples is the relative conflict between performance and performance requirements. Performance requirements can be characterized among others by “response time” and “capacity”. In many systems, these two attributes are in conflict. For example in a road traffic pricing system [38, 39], multi-user attribute[2] has negative contribution to the response time of the system. This means that increasing the number of concurrent users in the system may diminish the response time of the system.



NFRs	Accuracy	Availability	Confidentiality	Dependability	Flexibility	Functionality	Interoperability	Maintainability	Performance	Portability	Privacy	Recoverability	Reliability	Reusability	Robustness	Safety	Security	Testability	Understandabilit	Usability
Accuracy	0		*			0		0	X	X		0	0				0			0
Availability									X		X	0				0	X			
Confidentiality	*								X								0			
Dependability									X											
Flexibility																				X
Functionality	0					0		*	*			0	*				*			0
Interoperability									X											
Maintainability	0					*		0	X			0	0		X		0			0
Performance	X	X	X	X		*	X	X	*	X		*	*	X		X	X		X	*
Portability	X								X											
Privacy		X																		
Recoverability	0	0				0		0	*			0	0				0			0
Reliability	0					*		0	*			0	0			0	0			0
Reusability									X											X
Robustness								X								0		X		
Safety		0							X				0		0					
Security	0	X	0			*		0	X			0	0				0			*
Testability															X					
Understandability									X											
Usability	0				X	0		0	*			0	0	X			*			0

Fig. 3. Catalogue of Conflicts Among NFRs

Table 3. Conflicting NFRs in Literature

Conflicting NFRs	Nature of Conflict	%
Security and Performance	absolute	33%
Security and Usability	relative	23%
Availability and Performance	absolute	20%
Performance and Portability	absolute	17%
Reusability and Performance	absolute	17%
Interoperability and Performance	absolute	10%
Maintainability and Performance	absolute	10%
Reliability and Performance	relative	10%
Usability and Performance	relative	10%
Usability and Reusability	absolute	3%

[2] In these papers [38, 39], the term “attribute” is considered as “concern”.

The investigation by using frequency analysis technique also indicates that conflict between security and performance requirements are the most frequently conflicts discussed in the literature. 33.33% of the reviewed articles talk about this conflict, followed by conflict between security and usability requirements (23.33%) and conflict between availability and performance requirements (20%). This result indicates that those three types of conflicts (i.e. conflict between security and performance, between security and usability, and between availability and performance) are the three most frequent conflicts in the software projects and the most considered and essential to deal with in the software development process. The top ten conflicting NFRs that are often discussed in the literature are presented in Table 3.

5 Using the Catalogue

The catalogue of conflicts among NFRs, as presented in Fig. 3, extends and complements previously published NFRs conflicts models. Our work focuses on the extent and relativity of NFRs conflicts, that is, on negative links between NFRs and their corresponding-levels. Most of the existing conflicts models in the literature, however, concentrate on both positive and negative interrelationships. For example, Wiegers [9] has developed a matrix that represents the positive and negative relationships between particular type of NFRs; Egyed & Grünbacher [27] created a model of potential conflicts and cooperations among NFRs; and Sadana & Liu [37] have also defined conflict and support as the two types of contribution of a particular type of NFRs on the other types of NFRs.

Utilizing our NFRs catalogue of conflicts in conjunction with the existing conflicts models extends the overall understanding of how NFRs associate with each other (positive or negative) and how this negative association can be characterized in term of the relative characteristic of NFRs.

Software developers can use the conflicts catalogue to deal with various aspects of managing the conflicts among NFRs. For example, the conflicts catalogue can be used to identify which NFRs of the system that are really in conflict, including how relative the conflict is. If the identified conflict is an "absolute conflict", then software developers may need to identify the potential strategies to resolve this conflict, such as prioritization strategy. On the other hand, if it is a "relative conflict", then software developers need to understand and evaluate this particular NFRs in term of numerous factors involve in the development project (e.g. the meaning of particular type of NFRs in the context of the system being developed; the stakeholder involvement; or system development methodology used in the project) in order to further investigate whether those NFRs are really in conflict.

Furthermore, this catalogue can also be used to perform the NFRs conflicts analysis. By using this catalogue in conjunction with the framework presented by Sadana & Liu [37], software developers would be able to develop a structural hierarchy of functional and non-functional requirements affected by each conflict type. Therefore, this catalogue could further assist in the analysis of NFRs conflicts from the perspective of functional requirements. By utilizing this catalogue in conjunction with the "NFR Prioritizer" method presented by Mala & Uma [40], this catalogue could assist software developers to analyze the tradeoffs among NFRs and prioritize the NFRs. In term of analyzing the NFRs tradeoff, this catalogue can be used as the basis to develop the "NFR Taxonomy" that will be used to identify the type of relationships among NFRs. The NFR Taxonomy represents the conflicting or dependable association between each NFRs type. The example of NFR taxonomy is presented as follow [40]:

Usability#Accessibility+#Installability+#Operability+#Maintainability-

The above taxonomy represents that usability contributes positively to accessibility, installability, operability while it also contributes negatively to maintainability. Then, by combining the weight of user preference on each NFR type and the level of NFRs tradeoff derived from the NFR Taxonomy, software developers would be able to prioritize the NFRs of the system in term of the existence of conflicts among them.

Furthermore, this catalogue can also be used in conjunction with the "Trace Analyzer" technique developed by Egyed & Grünbacher [27]. The aim of this technique is to identify the true conflicts among NFRs of the system. By tracing the relationships between the system test cases and the software program codes, trace analyzer can characterize whether the conflicts listed in the NFRs conflicts catalogue are "really in conflict" in the developed system.

In term of conflicts resolution, the proposed catalogue of conflicts can also be used as the basis to execute a conflicts resolution technique. For example, by using this catalogue in conjunction with the "Non-Functional Decomposition (NFD)" framework developed by Poort & de With [41], software developers would be able to decompose the NFRs of the system when the NFRs conflicts identified.

6 Conclusions

Majority of techniques to manage the conflicts among NFRs present the documentation, catalogue, or list of potential conflicts. None of them deal with relative characteristic of NFRs. This relative characteristic means that the interpretation and importance of NFRs may vary depending on the particular system being developed as well as the extent of stakeholders' involvement. NFRs can be viewed, interpreted, and evaluated differently by different people and different contexts within which the system is being developed. Consequently, the positive or negative interrelationships among them are not always obvious.

In this paper we presented a catalogue of conflicts among NFRs by considering this relative characteristic. We presented the relativity of conflicts based on three categories: *absolute conflict; relative conflict;* and *never conflict*. This distinction would assist developers to perform further analysis of the identified conflicts and investigate the potential strategy to resolve the conflicts.

Furthermore, this catalogue can also be used to identify the NFRs conflicts in various phases of software development projects. For example, in the requirements engineering phase, during the elicitation process, system analysts would be able to identify which NFRs of the system will be in conflict and how relative this conflict is. This analysis would allow developers to identify the conflicts among NFRs early, so they would be able to discuss the potential conflicts with the system's stakeholder before specifying the software requirements. As another example, during the architecture design process, system designers could be able to use this catalogue to analyze the potential conflicts in term of the architectural decisions (e.g. layering, clustering, and modularity). The relativity of conflict relationships presented in the catalogue, would allow system designers to investigate the potential architecture strategies to get the best solution based on the type of conflicts among NFRs. Furthermore, by using this catalogue as the basis of conflicts identification, we can adopt numerous existing conflicts analysis and conflicts resolution techniques presented in the literature, such as [27, 37, 40, 41] to further investigate and evaluate the NFRs conflicts. Some examples of the existing techniques and the potential utilization of the catalogue in each technique have been described in Section 5 – "Using the Catalogue".

In the process of investigating conflicts and developing the conflicts catalogue, we also identified 114 NFRs types listed in the literature. Among these 114 types, more than 50% of the NFRs were introduced without any definition or attributes characterization while only 20% were provided with definition and attributes. This statistic and the list of NFRs types without definitions and attributes presented in this paper are expected to encourage software engineering community, particularly requirements engineering researchers to further investigate the unclear NFRs types and establish the a clear concept of them.

Further research will focus on collecting data from software practitioners to complete the catalogue. Those NFRs that have been removed from the initial catalogue due to lack of definitions and/or attributes will also be further investigated to improve the completeness of the catalogue. Also, the catalogue from industry can be compared with the one developed from the content analysis.

Moreover, besides collecting data to improve the conflicts catalogue, we would also perform further research on investigating the relative conflicts among NFRs. This study would not only investigate how those NFRs dynamically generate conflicts with each other in term of the system context, but also to develop a framework to assist developers in identifying in which situations those NFRs are in conflict and in which situations are not. The self-conflicting relationships will be covered in this study.

This study is conducted as part of a long term project of investigating conflicts among NFRs. Findings of this investigation, especially the conflicts catalogue, will be used as the basis to select those NFRs that are known to be frequently in conflict. The ultimate goal is to develop an integrated framework to effectively manage the conflicts between a pair of NFRs by considering the NFRs relative characteristic. This framework should be able not only to identify the existence and the extent of conflicts, but also to characterize and find the potential strategies to resolve the conflicts.

In this study, we do not claim that the catalogue of conflicts presented is an exhaustive and complete list. However, this catalogue represents what could be found in the current literature. We propose to conduct further research to compare and contrast our findings from the comprehensive review of research literature and the state of the practice.

Acknowledgements. This paper is an extended version of [42]. We would like to thank The International Schlumberger Foundation Paris for funding this research through Faculty for the Future Award Program.

References

1. Yeh, R.T.: Requirements analysis - a management perspective. In: IEEE Computer Software and Applications Conference (COMPSAC 1982), Los Alamitos, pp. 410–416 (1982)
2. Mairiza, D., et al.: Managing conflicts among non-functional requirements. In: 12th Australian Workshop on Requirements Engineering (AWRE 2009), Sydney, Australia (2009)
3. Ebert, C.: Putting requirement management into praxis: dealing with nonfunctional requirements. Information and Software Technology 40, 175–185 (1998)
4. Firesmith, D.: Using quality models to engineer quality requirements. Journal of Object Technology 2, 67–75 (2003)
5. Chung, L., et al.: Non-functional requirements in software engineering. Kluwer Academic Publishers, Massachusetts (2000)
6. Mittermeir, R.T., et al.: Modern software engineering, foundations and current perspectives. Van Nostrand Reinhold Co, New York (1989)
7. Kotonya, G., Sommerville, I.: Non-functional requirements (1998)
8. Charette, R.N.: Applications strategies for risk analysis. McGraw-Hill, New York (1990)
9. Wiegers, K.E.: Software requirements, 2nd edn. Microsoft Press, Washington (2003)
10. Sommerville, I.: Software Engineering, 7th edn. Pearson Education Limited, Essex (2004)
11. Breitman, K.K., et al.: The world's a stage: a survey on requirements engineering using a real-life case study. Journal of the Brazilian Computer Society 6, 1–57 (1999)
12. Finkelstein, A., Dowell, J.: A comedy of errors: the London ambulance service case study. In: Eigth International Workshop Software Specification and Design, pp. 2–5 (1996)
13. Boehm, B., In, H.: Identifying quality-requirements conflict. IEEE Software 13, 25–35 (1996)
14. Boehm, B., In, H.: Aids for identifying conflicts among quality requirements. IEEE Software (March 1996)
15. Leveson, N.G., Turner, C.S.: An investigation of the Therac-25 accidents. IEEE Computer 26, 18–41 (1993)
16. Grimshaw, D.J., Draper, G.W.: Non-functional requirements analysis: deficiencies in structured methods. Information and Software Technology 43, 629–634 (2001)
17. Heumesser, N., et al.: Essential and requisites for the management of evolution - requirements and incremental validation. Information Technology for European Advancement, ITEA-EMPRESS Consortium (2003)
18. Yusop, N., et al.: The impacts of non-functional requirements in web system projects. International Journal of Value Chain Management 2, 18–32 (2008)
19. Paech, B., Kerkow, D.: Non-functional requirements engineering - quality is essential. In: 10th International Workshop on Requirements Engineering: Foundation for Software Quality, pp. 27–40 (2004)
20. Lauesen, S.: Software requirements: styles and techniques. Addison-Wesley, Reading (2002)
21. Chung, L., et al.: Using non-functional requirements to systematically support change. In: The Second International Symposium on Requirements Engineering, York, pp. 132–139 (1995)
22. Chung, L., et al.: Dealing with change: an approach using non-functional requirements. Requirements Engineering 1, 238–260 (1996)
23. Curtis, B., et al.: A field study of the software design process for large systems. Communication of the ACM 31, 1268–1287 (1988)

24. Boehm, B., Egyed, A.: WinWin requirements negotiation processes: a multi-project analysis. In: 5th International Conference on Software Processes (1998)
25. Egyed, A., Boehm, B.: A comparison study in software requirements negotiation. In: 8th Annual International Symposium on Systems Engineering, INCOSE 1998 (1998)
26. Robinson, W.N., et al.: Requirements interaction management. ACM Computing Surveys 35, 132–190 (2003)
27. Egyed, A., Grünbacher, P.: Identifying requirements conflicts and cooperation: how quality attributes and automated traceability can help. IEEE Software 21, 50–58 (2004)
28. Krippendorff, K.: Content analysis: and introduction to its methodology, 2nd edn. Sage Publications, Inc., Thousand Oaks (2004)
29. Weber, R.P.: Basic content analysis. Sage Publications, Inc., Thousand Oaks (1989)
30. Stemler, S.: An overview of content analysis. Practical Assessment, Research & Evaluation 7 (2001)
31. Neuendorf, K.A.: The content analysis guidebook, 1st edn. Sage Publications, Inc., Thousand Oaks (2001)
32. Glinz, M.: Rethinking the notion of non-functional requirements. In: Third World Congress for Software Quality, Munich, Germany, pp. 55–64 (2005)
33. Glinz, M.: On non-functional requirements. In: 15th IEEE International Requirements Engineering Conference (RE 2007), pp. 21–26 (2007)
34. Mairiza, D., et al.: An investigation into the notion of non-functional requirements. In: 25th ACM Symposium On Applied Computing, Switzerland (2010)
35. Alexander, I., Maiden, N.: Scenarios, stories, use cases: through the systems development life-cycle. John Wiley & Sons, Ltd., Chichester (2004)
36. Robertson, S., Robertson, J.: Mastering the requirements process, 2nd edn. Addison-Wesley, Boston (2006)
37. Sadana, V., Liu, X.F.: Analysis of conflict among non-functional requirements using integrated analysis of functional and non-functional requirements. In: 31st International Computer Software and Applications Conference, COMPSAC 2007 (2007)
38. Brito, I., Moreira, A.: Integrating the NFR framework in a RE model. Presented at the Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, Lancaster, UK (2004)
39. Moreira, A., et al.: Crosscutting quality attributes for requirements engineering. In: 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy (2002)
40. Mala, G.S.A., Uma, G.V.: Elicitation of non-functional requirements preference for actors of usecase from domain model. In: Hoffmann, A., Kang, B.-h., Richards, D., Tsumoto, S. (eds.) PKAW 2006. LNCS (LNAI), vol. 4303, pp. 238–243. Springer, Heidelberg (2006)
41. Poort, E.R., de With, P.H.N.: Resolving requirement conflicts through non-functional decomposition. In: Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA 2004 (2004)
42. Mairiza, D., et al.: Towards a catalogue of conflicts among non-functional requirements. In: Maciaszek, L.A., Loucopoulos, P. (eds.) ENASE 2010. CCIS, vol. 230, pp. 33–46. Springer, Heidelberg (2011)

Applying AspectJ to Solve Problems with Persistence Frameworks

Uwe Hohenstein and Michael C. Jaeger

Siemens AG, CT T DE IT 1, Otto-Hahn-Ring 6, D-81730, Munich, Germany
{Uwe.Hohenstein,Michael.C.Jaeger}@siemens.com

Abstract. This work reports on problems we had with persistence frameworks in an industrial project. Most problems occurred when replacing the persistence framework Hibernate with OpenJPA. Such a substitution basically means exchanging API calls and dealing with functional differences. But the replacement involved challenging problems since some important Hibernate functionality was missing in OpenJPA and could not be emulated, and other functionality did not work appropriately in OpenJPA. Conventional techniques such as wrapping code are not sufficient to tackle those points. However, we found powerful mechanisms in the aspect-oriented programming language AspectJ to solve problems fast, easily, and in a straightforward manner. All the problems are well-motivated and the aspect-oriented solutions are explained in detail.

1 Introduction

Whenever Java and relational database systems (DBS) are used, object-relational (O/R) persistence frameworks or tools such as Hibernate, Java Data Objects (JDO) or Java Persistence API (JPA) come into play: Application programmers can store and retrieve Java objects in relational tables without knowing about the underlying table structure and/or how to formulate SQL queries. Programming can be done at an object-oriented level, i.e., by storing and retrieving Java objects. The O/R framework translates those object-oriented operations into SQL.

We were involved in an industrial project with Siemens Enterprise Communications (SEN), where the Hibernate persistence framework was used. The project develops a Java-based service-oriented telecommunication middleware which serves as an open service platform for the deployment and provision of communication services [1]. Examples for such services are the capturing of user presence, the management of calling domains, administration functionality for the underlying switch technology, and so forth. The technical basis is OSGi.

Hibernate was used for managing persistent data in a relational DBS. Hibernate is a widely used and popular O/R framework. It is open-source software and provides only a thin layer upon the Java Database Connectivity API (JDBC), offering developers much control on performance-relevant settings. Hibernate was used for two reasons: First, to be independent of various DBSs to be supported in the product, namely solidDB, MySQL, and PostgreSQL. And second, to benefit from the higher, object-oriented level of database programming.

Some time ago, the owner of Hibernate was accused of violating a patent on O/R frameworks in the United States. This patent infringement claim seemed to be a problem of Hibernate at a first glance. However, every software product that is shipped to the United States with Hibernate inside is affected as well; any redistribution of Hibernate implies the role of a supplier. To avoid the risk of a patent infringement, the project management decided to replace Hibernate with another O/R framework. An additional business issue was the GNU Lesser General Public License (LGPL) used by Hibernate. LGPL was not fully compatible with agreements that SEN has with its business partners. As a consequence, the project management decided to replace such LGPL software in general.

The Hibernate replacement started with a first brief evaluation, where several substitute candidates were roughly assessed: Proprietary frameworks such as iBATIS and tools conforming to the JDO or JPA standards. As a quick result, the OpenJPA framework was chosen because it is open-source and implements the JPA specification. The JPA standard seems to be appropriate because it is part of the EJB 3.0 specification and is more recent than JDO. Thus, OpenJPA could easily be replaced with other JPA-conforming tools if OpenJPA would also be gripped by the patent. Moreover, OpenJPA is provided with the more convenient Apache software license.

Migrating from Hibernate to OpenJPA is merely straightforward at a first glance: It is possible to wrap OpenJPA by still offering Hibernate interfaces; changes are thus minimal. However, during the replacement effort, severe problems raised that were difficult to detect in an OpenJPA evaluation. Those issues occurred lately and endangered the success of replacement. In order to cope with them, we found and applied solutions using Aspect-Orientation (AO).

AO has been proposed for developing software to eliminate crosscutting concerns, i.e., functionalities that are typically spread over several classes. Those lead to code tangling and scattering [2] in conventional programming [3]. Research has shown its usefulness: Hannemann and Kiczales [4] identify several crosscutting concerns in the GoF patterns [5] and extract them into aspects. [3,6] use aspects for designing and building flexible middleware. Rashid [7] discusses several facets of AO in the context of databases, in particular implementing DBSs in a more modular manner and an AO-based persistence framework [8]. Others use AO to maintain database statistics [9] or to implement ACID properties [10]. It turns out in all these studies that aspect-orientation increases programming productivity, quality and traceability, degree of code reuse, software modularity, and is better supporting evolution [11].

In this paper, we discuss another application of AO, to apply aspects to existing 3rd party software libraries in order to add missing functionality or to change internal behavior. Our intent is to show that AO provides a straightforward solution being suitable for software migrations in enterprise settings. The essential and novel value of our AO approach is a method to address the challenges of integrating 3rd party software, keeping the original software untouched and being able to manage the concerns of replacement in a maintainable manner.

In Section 2, we summarize the general strategy for replacing Hibernate and outline a selection of replacement issues that we solved by applying conventional methods. Section 3 presents some critical problems that occurred during the replacement, for which conventional solutions are hard to find and apply. After having introduced the fundamentals of AO and AspectJ [12], Section 4 explains our solutions using AO. Our lessons learned during the replacement are summarized in Section 5. The paper ends with Section 6 that gives a summary on our experiences and our conclusions.

2 Replacement Strategy

In order to perform the Hibernate replacement, a master plan was established in the beginning. This plan consists of the following steps:

1. The goal was to start a practical replacement as early as possible. A selection and brief assessment of potential Hibernate substitutes leads to an early decision for the JPA standard with OpenJPA as implementation, because obvious similarities exist between OpenJPA and Hibernate.
2. A checklist was established for those Hibernate concepts that were seen specific or critical. A short evaluation of the checklist let appear OpenJPA appropriate.
3. We transformed the project's central persistence infrastructure to OpenJPA, particularly its configuration and deployment.
4. As a proof of concept, the most complicated software project was migrated first in a sandbox environment. By this step, we expected to identify as many problems as early as possible.
5. The real replacement on the affected software projects was scheduled and planned.
6. Finally, we performed the replacement in coordination with the affected development teams. Training and coaching was also necessary.

The short theoretic evaluation of Step 2 was successful, and no major problems have been detected at that time. Of course, several differences between Hibernate and OpenJPA APIs exist. For instance, we have to use an `EntityManager` instead of a `Session.EntityManager.persist()` instead of `Session.save()`, etc. But since most concepts of Hibernate seemed to have an equivalent counterpart in OpenJPA, we got an optimistic impression about the replacement. This first impression was also confirmed by [13] who state that it is no problem to migrate from Hibernate to OpenJPA.

It became clear that obvious differences are easy to cope with a wrapper approach. Implementing the Hibernate interface on top of OpenJPA has the advantage that the old Hibernate interface in use can still be retained. Only import statements have to be changed. Even the change of import packages is not really mandatory, but useful since Hibernate and OpenJPA could thus run in parallel in an OSGi container during the replacement phase. This allows for a step by step replacement of services. Ongoing development work on the middleware is not really affected.

Despite several conceptual similarities, the practical evaluation of Step 4 brought up some differences which we would like to mention briefly (see also [13] for further topics).

One problem is that JPQL delete-by-queries do not work correctly because OpenJPA generates an SQL query with a self-reference which cannot be executed by most DBSs:

```
DELETE FROM Tab
WHERE key IN (SELECT key FROM Tab WHERE <condition>)
```

A solution is to omit delete-by-queries by implementing the functionality manually, i.e., by querying the objects to be deleted first and then deleting each object one by one. This poses a performance problem due to lots of DELETE operations. A sustainable solution is to correct the query generation by avoiding the unnecessary subquery. The relevant translation is part of so-called `Dictionary` classes. Hence, the change can simply be done by defining a dictionary class `MyMySQLDictionary` that extends the predefined `MySQLDictionary` in such a way.

Furthermore, the life cycle of the persistent objects is different. For example, it is possible in Hibernate to overwrite an existing persistent object in the database by creating a new object having the same key values; saving that object overwrites the existing one. However, OpenJPA treats the (temporary) object as a new one, which let the database system complain about duplicates.

Hibernate's `Criteria` interface for queries is not supported in OpenJPA release 1.1.0. Thus, `Criteria` queries must be re-formulated in the JPQL language.

Smaller differences exist between the query languages HQL and JPQL, e.g., an explicit alias `t` has to be used at any place, as in `SELECT t FROM Type t WHERE t.attr=1` instead of Hibernate's short form `FROM Type WHERE attr=1`. This affects conditions that could be composed as `attr=1` in the GUI and now need to be extended with an alias `t`.

Hibernate has a special delete-orphan cascade option: While the ordinary delete-cascade removes with a father object all depending son objects, delete-orphan removes son objects in addition when the association with the father object is destroyed; a son object cannot exist without a father. Despite being not supported by the JPA standard, OpenJPA provides such a feature by means of an extended mapping annotation. If one stays with XML mapping files, those cascades must be resolved and implemented manually.

OpenJPA comes with an easy integration of the Apache DBCP connection pool, while we used Hibernate with the C3P0 pool. DBCP behaves differently and performance tests brought up different connection pool settings for DBCP.

Although those issues represent a very individual effort, such a correction did not pose any problems to the progress of the replacement.

3 Harder Problems

The differences between Hibernate and OpenJPA explained in the previous section are easy to solve. However, some problems – being detected in later phases of the replacement unfortunately – endangered the success of the overall replacement and were hard to solve with conventional programming techniques. This section discusses those problems in detail. Corresponding AO solutions are presented in Section 4.

3.1 Lack of Key Generation

An O/R framework requires mapping information on how to map classes onto database tables, attributes to table columns, associations to foreign keys etc. This can either be done by means of XML mapping files or by Java-5 annotations in the entity classes. Our project used XML mapping files. The following Hibernate mapping example relates a class `MyClass` (`<class>`) to a table `MyTable` (`table=…`), fields `id` and `p2` to table columns `pk` and `c2`, respectively.

```
<class name="MyClass" table="MyTab">
  <id name="id" column="pk">
    <generator class="sequence"/> </id>
  <property name="p2" column="c2"/> ...
</class>
```

Thereby, `<id>` defines a key field that uniquely identifies objects in a class; the corresponding column `pk` is used as a database primary key.

Indeed, the mapping specification in OpenJPA is different; a file `orm.xml` specifies mappings with a different syntax. The transformation of Hibernate mapping files into OpenJPA syntax is straightforward and can be achieved by an XSLT script for most differences. However, some differences are fundamental. For example, there are various alternatives for providing `<id>` values in Hibernate, e.g., to let the application be responsible for providing the key values and ensuring their uniqueness (`<generator class= "assigned"/>`), to let Hibernate generate an id by means of creating a globally unique identifier, or to use mechanisms that DBSs offer such as `sequence` generators (in solidDB) or auto-increment columns (in MySQL). These strategies are supported by OpenJPA, too. But Hibernate also offers a more abstract `native` key generation: Depending on what the underlying DBS supports, either `sequence` or `identity` (for auto-increment columns) is used. Since the project must support several DBSs, especially solidDB, MySQL, and PostgreSQL, and since the type of DBS should be invisible, such an abstract strategy is required.

OpenJPA has a similar `auto` strategy that lets OpenJPA decide what to do, but it uses a table for maintaining highest values instead of taking auto-increment columns or sequences. This is not appropriate as database installations already exist at customers, containing keys generated by either sequences or auto-increment columns. For these, the probability is high that `auto` generates already existing values. Hence, value clashes are most likely when upgrading to an OpenJPA-based implementation.

One solution is certainly to maintain three XML mapping files, one for each DBS with the supported strategy. A simple model-driven approach that generates DBS-specific variants with `sequence` or `identity`, respectively, could help here. This was regarded as an inappropriate solution as it causes a problem for deployment. OpenJPA expects the mapping file in a JAR. The overall project strategy is to have one unchangeable deployment JAR: All parameters that might vary from one installation to another, such as the database URL, its port, user and password, must be placed outside the deployed JAR file. This is because only parts of the JDK are installed on target machines and unzip/zipping of JAR files is not available to exchange parts such as mapping files. Hence, the resulting installation procedure would now need to handle several JAR files for deployment, one for each DBS.

The issue with providing different mappings becomes even worse, since we were forced to use mapping annotations in some cases. Some OpenJPA features are only available as annotations, but not in XML mappings, e.g., a “delete-orphan” cascade (cf. Section 2): This is a special option that removes son objects when their association with the father object is destroyed. On the one hand, using the delete-orphan option with annotations means that also several code variants have to be maintained, since the mapping is part of the source code. On the other hand, implementing delete-orphan behavior manually, i.e., deleting objects explicitly whenever they become parentless can be very cumbersome since cascades go over several levels in the object model.

Any of both proposals would require massive changes in the implementation and deployment infrastructure.

3.2 Failover Problem

The main DBS to be supported in our project is solidDB. solidDB is not as popular as other DBSs. However, it is often used in telecommunication projects. One reason is its hot-standby failover concept: It is possible to install two DBSs, one primary and one secondary, the databases of both being synchronized. If the primary solidDB server crashes, the secondary becomes the new primary and silently takes over the work immediately. To apply failover, applications have to use a specific *dual-node* URL of the form `jdbc:solid://h1:1315,h2:1315/usr/pw`. This URL specifies two database servers on host h1 and host h2.

The failover concept is important for our project and certainly one of the first priority requirements. We knew that Hibernate and the solidDB JDBC driver can handle the dual-node URL. Since, the O/R framework is supposed to pass this URL through to the JDBC driver, no particular problems were expected. But since the setup and accomplishment of failover test scenarios involves many steps, the final check has been postponed in the first assessment of OpenJPA.

When it came to test deployments, the failover feature of the solidDB DBS did not work for OpenJPA; connections to the database could not be established at all with the given URL. The first problem occurred: How can we find out why no connections are possible? Debugging was very tedious as the problem occurred in the depth of OpenJPA and the JDBC driver. As we are describing later, AO helped us to detect the cause for the problem.

It turned out that the dual-node URL was damaged by OpenJPA: Only the first part `jdbc:solid://h1:1315` arrived at the solidDB server. The reason is that a string is used to set several facets of connection properties in one `openjpa.ConnectionProperties`, the URL, the driver class name etc.:

```
String str = "Url=jdbc:solid://h1:1315,h2:1315/usr/pw,
              DriverClassName=solid.jdbc.SolidDriver,
              ...";
props.setProperty("openjpa.ConnectionProperties",str);
EntityManagerFactory emf
        = persProvider.createEntityManagerFactory("mydb",props);
```

A deeper investigation brought up that OpenJPA takes the comma as a separator during the analysis of `openjpa.ConnectionProperties` and thus derives the following units from the properties:

```
Url=jdbc:solid://h1:1315
h2:1315/usr/pw
DriverClassName=solid.jdbc.SolidDriver
…
```

That is, `h2:1315/usr/pw` is taken as a unit of its own, and since it does not satisfy the form `property=value`, it is simply ignored; and the URL degrades to `jdbc:solid://h1:1315`.

To solve the problem and to leave the dual-node URL intact, we obviously have to change the internal behavior of OpenJPA.

3.3 Missing Connection Property

Unfortunately, the previous solution solves only half of the failover problem: It allows establishing connections to solidDB, but no failover occurs. Indeed, the solidDB JDBC driver requires a special failover property `solid_tf_level` to be set for any database connection. OpenJPA allows passing additional properties, but only OpenJPA properties starting with "`openjpa.`", are analyzed and passed to the JDBC driver; others are ignored.

A solution must somehow change the behavior of the solidDB driver, the source code of which is unavailable.

3.4 Possible Solutions

What are possible solutions to solve the above problems? There is no easy workaround such as wrapping OpenJPA or JDBC methods because we have to intervene in the internal behavior.

We can certainly ask the vendor of solidDB to change its JDBC driver. This is in general expensive and must be done again and again when a new version is launched. For patches of OpenJPA, the open source community could provide solutions. However, the problem affects the interplay between OpenJPA and the rather specific solidDB DBS. We require solidDB-specific patches to the OpenJPA source code, but solidDB is not officially supported by OpenJPA. We reported those solidDB specific issues to the OpenJPA project, but we could not wait for a solution because this would have caused a significant delay.

Patching source code is possible, if the code is available. This is not always the case, e.g., the sources of the solidDB JDBC driver are unavailable. In case of OpenJPA, a deeper understanding of the complete source code is necessary because several logical parts are involved: The XML parser for mapping files, the handling of annotations, storing and using meta-data, interpreting the meta-data to perform database operations etc. One technical difficulty is then to patch the code in such a way that changes apply only for solidDB, but not for other DBSs. OpenJPA knows the JDBC driver and can derive the used DBS. However, this information is needed in a different class. Hence, we have to let unrelated classes exchange this kind of information, which means the change cannot be done locally.

Moreover, the build process must be understood in order to produce a new OpenJPA JAR file. This could also cause trouble with integrating two different build approaches such as Ant and Maven.

Aspect-orientation provides simpler solutions.

4 AspectJ Solutions

Aspect-orientation is a solution for our problems, especially if 3rd party tools behave in a wrong manner and if no source code is available. We applied AO to change the internal behavior of OpenJPA and JDBC drivers in order to achieve in OpenJPA some missing Hibernate functionality.

The most popular AO language is certainly AspectJ [14]. Special extensions to Java enable separating the definition of crosscutting concerns. Programming with AspectJ is essentially done by Java and by new *aspects*. The main purpose of aspects is to change the program flow. An aspect can intercept certain points of the program flow, called *join points*. Examples of join points are method calls or executions, and attribute accesses.

Join points are syntactically specified by means of *pointcuts*. Pointcuts identify join points in the program flow by means of a signature expression. For example, a specification can determine exactly one method. Or it can use wildcards to select several methods of several classes by `* MyClass*.get*(..,String)`. A star "`*`" in names denotes any character sequence, hence, `get*` means any method that starts with "get". A type "`*`" denotes any type. Parameter types can be fixed or left open (`..`). Interception of methods can be done at the caller or callee side. An `execution(...)` pointcut intercepts at the callee side, i.e., any caller is affected. In contrast, `call(...)` intercepts at the caller side.

Once join points are captured, *advices* specify weaving rules involving those joint points, such as taking a certain action before or after the join points. Pointcuts can be specified in such a way that they expose the context at the matched join point, i.e., the object on which the intercepted method is invoked. Parameter values can be accessed in advices as well.

The AspectJ language requires a compiler of its own. Usually, the AJDT plug-in will be installed in Eclipse. However, a new compiler requires changes in the build process, which is often not desired, so for us. Then, using Java-5 annotations such as `@Aspect` is an alternative: Aspects can be written in pure Java. This was important for us, because we could rely on standard Eclipse with an ordinary Java compiler, without AJDT. In order to use annotations, the AspectJ runtime JAR is required in the classpath. To make the aspect active, we also have to start the JVM (e.g., in Eclipse) with a `-javaagent` argument referring to the AspectJ weaver. Annotations are then evaluated and become really active, because load-time weaving takes place: Aspects are woven whenever a matching class is loaded.

We now show AspectJ examples that solve our problems.

4.1 Solving the Lack of Key Generation

The basic idea to remedy the lack of key generation is to accept both strategies `sequence` and `identity`, but to change the internal OpenJPA behavior in such a way that it uses the strategy available in the DBS. Hence, if `identity` has been chosen, but if the DBS does not supply auto-increment columns, then let OpenJPA internally switch to the `sequence` strategy. This is much easier than adding a new `native` strategy for mapping specifications and/or annotations, which requires a corresponding modification of the XML parser, the analysis of annotations, the use of this kind of meta-data to derive SQL operations adequately etc.

Changing the OpenJPA behavior to handle `identity` appropriately according to the type of DBS can easily be done by the following aspect.

```
@Aspect
public class KeyGenerationAspect {
  private String db = null;
  @Before("execution(*org.apache.openjpa.persistence
         .PersistenceProviderImpl.createEntityManagerFactory(..))
          && args(.., p)")
  public void determineDBS(Properties p) {
    String str = p.getProperty("openjpa.ConnectionProperties");
    if (str != null) {
      if (str.contains("Solid"))
         db = "SOLID";
      else if (str.contains("mysql"))
         db = "MYSQL";
      else if (str.contains("postgresql"))
         db = "POSTGRES";
  }
  @Around("call(* org.apache.openjpa.meta.FieldMetaData
    .getValueStrategy(..)) && !within(com.siemens.ct.aspects.*)")
  public Object changeStrategy(JoinPoint jp) {
    FieldMetaData fmd = (FieldMetaData) jp.getTarget();
    int strat = fmd.getValueStrategy();
    if (db.equals("SOLID") && strat == STRATEGY_IDENTITY) {
      fmd.setValueSequenceName("system");
      return STRATEGY_SEQUENCE;
    } ... // similar for other DBSs
    return strat;
} }
```

A `@Aspect` annotation lets the Java class `KeyGenerationAspect` become an aspect. Annotations are used instead of the AspectJ language. This was important for us because we could rely on a standard Eclipse setup with an ordinary Java compiler.

There are two advices: The first one `determineDBS` determines the DBS and the second one `changeStrategy` changes the strategy if necessary. Both advices exchange information about the DBS in use by means of an aspect-local variable `db`.

Since the method `determineDBS` is annotated with `@Around`, it defines an advice to be executed around those join points that are specified by the pointcut string: Any execution of the method `PersistenceProviderImpl.createEntityManager-Factory` with a `Properties` parameter. The `args(..,p)` clause requires at least a `Properties` parameter and binds a variable `p` to that parameter. The variable also occurs in the method signature and allows the advice to access the value. Thus, `p.getProperty("openjpa.ConnectionProperties")` yields the connection properties, i.e., the comma-separated list we are interested in so that we can extract the type of DBS. The result is stored in an internal variable `db`.

The `changeStrategy` advice uses this information about the DBS to switch from strategy `identity` to `sequence` in case of solidDB. Hence, the aspect can simply be used to share and exchange information even if different parts of code, even of different JARs, are intercepted. The technical problem how to determine the type of DBS is solved in an easy way.

The `@Around` advice `changeStrategy` intercepts any call of `FieldMetaData .getValueStrategy`, which returns the strategy. Due to `@Around`, the original logic is replaced in such a way that we decide when to switch the strategy in the advice.

Please note that `!within(com.siemens.ct.aspects.*)` is necessary: Whenever `getValueStrategy` is called, the call is implicitly changed to calling the `@Around` method, which performs `strat = fmd.getValueStrategy()` inside. This means this call is again intercepted, resulting in an infinite recursion. `!within` excludes any call within the aspect from being intercepted.

The parameter `JoinPoint jp` gives access to context information about the join point, especially the target object on which the method is invoked (`jp.getTarget()`). This is a `FieldMetaData` object in this case, which allows determining the current strategy by means of `getValueStrategy()`. Instead of returning the original strategy, e.g., `identity`, we can switch for solidDB to `sequence` and set the sequence name to the system sequence.

4.2 Solving the Failover Problem

As explained in Section 3.2, OpenJPA is unable to connect to the solidDB DBS with a dual-node URL `jdbc:solid://h1:1315,h2:1315/usr/pw`. Our first problem was to detect the reason why.

Refining the log4j level especially for OpenJPA produces an overwhelming but useless output of OpenJPA activities such as initialization activities, analyzing mapping specifications, named queries etc.

Debugging works only, if the source code is available. Even with IDE support, the problem is hard to detect with debugging, especially since several dynamic method invocations interrupt the execution flow: OpenJPA has a pluggable connection pool and loads dynamically the one chosen. And the connection pool dynamically invokes the JDBC driver for the selected DBS.

According to Laddad [14], one myth about AOP is to be good only for logging and tracing. AOP is indeed useful for tracing (but we disagree with the word "only"). We want to show how AO allows for a better and spontaneous controlling of tracing that is more dedicated to the problem to solve; overwhelming and useless trace output can be avoided. Thanks to load-time weaving in Eclipse, tracing can be done in a few minutes: Add the `aspectjrt` JAR-file to the classpath, provide an `aop.xml` file specifying relevant packages, use `-javaagent` in Eclipse, and implement the following advice:

```
@Before("execution(* *.*(..,String,..))")
public void myTrace(final JoinPoint jp) {
  Object[] args = jp.getArgs();
  for (Object a : args) {
    if (a instanceof String && arg!=null
    && ((String)a).contains("jdbc:solid:"))
    System.out.println("* In: " +   jp.getSignature() + "->"
                                               + a.toString());
} } }
```

This `@Before` advice intercepts any execution of any method (`execution(* *.*)`) with a `String` parameter (`(..,String,..)`) and checks whether the string contains a solidDB URL. If it does, it prints out that URL. The parameter `JoinPoint jp` gives access to context information about the join point. For instance, `jp.getSignature()` can be used to print out the intercepted method signature, and `jp.getArgs()` returns the passed parameter values.

These simple changes are done in a few minutes and lead to the following clear output:

```
* In: void org.apache.openjpa.lib.conf.Value setString(String)
->  DriverClassName=solid.jdbc.SolidDriver  ,Url=jdbc:solid://h1:
   1315,h2:1315/usr/pw,defaultAutoCommit=false,initialSize=35
...
* In: Options org.apache.openjpa.lib.conf.Configurations.parse
  Properties(String)
-> DriverClassName=solid.jdbc.SolidDriver,Url=jdbc:solid://h1:
   1315,h2:1315/usr/pw,defaultAutoCommit=false,initialSize=35
* In: boolean solid.jdbc.SolidDriver.acceptsURL(String)
-> jdbc:solid://h1:1315
* In: Connection solid.jdbc.SolidDriver.connect(String,
  Properties)
-> jdbc:solid://h1:1315
...
```

The bold parts are important: They show the transition from a good to a bad URL. Hence, the problem lies in the method `Configurations.parseProperties()`: The URL is correct before execution, but truncated afterwards. To detect this problem, AO tracing is much more effective than debugging. Thanks to a problem-specific tracing, the reason for problems can be detected immediately.

Since the problematic method is now known, we can fix the problem in a second step. Looking at the OpenJPA code, we see what goes wrong in method `parse-Properties`. As already explained in Section 3.2, the code separates the units by using a comma. Then, if no "=" is found in a unit, the unit is ignored, what exactly happened to the second part of the dual-node URL.

An aspect can correct the URL. Having a pointcut trapped the execution of this `parseProperties` method, an `@Around` advice can implement an instead-of behavior: Instead of executing the original method, we use our "corrected" implementation without touching the original source code directly:

```
@Around("execution(public static Options org.apache.openjpa.lib
    .conf.Configurations.parseProperties(String)) && args(s)")
public Object parseProperties(String s) {
  Options opts;
  parse properties string s correctly and set the return value
opts;
  return opts;
}
```

4.3 Missing Connection Property

Similarly, we can add the `solid_tf_level` connection property by modifying the JDBC driver: The following advice intercepts the execution of `SolidDriver.connect(...)` and adds the required `solid_tf_level` property to the `Properties` parameter:

```
@Before("execution(* solid.jdbc.SolidDriver.connect
                (..,String,..,Properties,..)) && args(url, pr)")
public void addSolidTfLevel(String url, Properties pr) {
  if (url != null && url.contains("solid"))
    pr.setProperty("solid_tf_level","1");
}
```

The part `(..,String,..,Properties,..)` specifies the parameters of interest. The `args` clause binds variables `url` and `pr` to them. The variable `url` is used to determine the DBS platform and `pr` to set the `solid_tf_level` property. Again, the JDBC driver, an external JAR file, is modified.

4.4 Further Problems

We applied AspectJ in a similar manner to solve other deficits of OpenJPA. We are not going into technical details, because the techniques are similar.

One problem occurred with class loading in OpenJPA. In some use cases, we ran into out-of-memory exceptions sporadically. Our analysis showed that thousands of class loader objects are created by OpenJPA. Unfortunately, the garbage collector places those objects in the system space, which means that the objects are destroyed too seldom. Using AspectJ, we detected the places where the class loaders are created and where they are used. The result was surprising: OpenJPA effectively uses only one of those class loaders. To solve the useless creation of class loaders, we defined an aspect that intercepts any constructor call. Instead of calling the original constructor, an around advice creates a class loader object only for the first time. Any further call returns that singleton.

Another memory problem is concerned with OpenJPA's query compilation cache. This cache is indispensable for achieving an acceptable performance since it relieves OpenJPA from analyzing and transforming JPQL queries again and again. Its size is configurable. If the cache is exceeded, an old query is dropped, however, this query is still kept in a second hidden cache with a fixed upper size of 1000. Since we have several database projects, each obtaining such a cache with hundreds of old queries, we again ran into high memory consumption. An aspect helped us to reduce the second cache to 0.

Furthermore, we also had some performance problems due to wrong connection pool settings. An aspect helped us to monitor whenever a JDBC connection is requested and released; the difference determines the number of currently active connections. Moreover, the aspect detects whenever a connection is requested directly via JDBC, thus bypassing OpenJPA; there is a danger of not having closed the connection. This monitoring is done for all databases in the system. Hence, we get detailed statistics of connection usage.

5 Experiences

5.1 General Experiences

The first lesson we learned is not really an experience, but rather a confirmation of our approach: The recommendation is to start doing as early as possible, not spending too much time on product selection. We decided to quickly choose a Hibernate substitute because the real problems are anyway hard to detect even with an extensive evaluation of products. The problems are occurring when starting the doing – and they will certainly arise. In our case, we checked the most important issues carefully and early. However, the severe problems came up quite late during the replacement. It is nearly impossible, in our opinion, to check all problems for several candidates.

Anyway, there is no need to worry about potential or suddenly arising problems. Even if hard problems occur unexpectedly, AO is a very powerful mechanism to overcome them.

The wrapping approach, i.e., implementing the "old" Hibernate interface on top of OpenJPA turned out to be very helpful and reduced the replacement time drastically. But there is a difference between syntactic and semantic success. It is quite easy to get the replacement compile-clean. The harder problems occur at runtime during the testing, e.g., the different behavior in Hibernate and OpenJPA when storing new objects with an existing key. And performance is not portable anyway.

Especially for achieving the same semantic behavior, testing turned out to be important. Without a huge test suite with several thousands of JUnit test cases, the replacement would presumably have failed. Thanks to the test suite, we could immediately check the correct behavior after replacement. We can remember only very few errors that came up after finishing and testing the replacement.

5.2 Convincing Project Management

Unfortunately, our project managers are not keen on using AO or having AspectJ in their projects: There is always the fear of having uncontrollable behavior if several developers use AOP. Our experiences go along with a recent study of AO adoption [15] within non-academic projects, which indicates that the majority of the interviewed developers are "early adopters" (according to [16]) of this technology. The current stage of adoption is that occasionally developers learn the AO concepts and try to apply them *in non-critical* phases of development projects, e.g., for architectural checks or performance monitoring, as in [17]. Very rarely the project management deliberately decides to use AO technologies in a project. This keeps the obstinate myths living: "AO is good only for logging/tracing" [14].

Well, we were able to convince our project management of using our AspectJ-based solution. Since we represented a focused team, we did not use the approach of

[18] and other authors who describe several stages for the adoption of AOP in order to guide single developers getting familiar with AO. This approach suits well, if a critical mass of developers can be convinced, which then in turn influence decisions of their management. We acknowledge the practical benefit of this approach, but it did not apply for our case. Even the approach we proposed in [19] could not be applied, because the advantages of AO we showed are not relevant in this project.

Rather, we faced the lucky situation that we had to tackle critical problems which imposed a lot of pressure: The replacement must have been successful in a short time, switching to yet another candidate than OpenJPA was not feasible because it could pose again uncertainties. Moreover, there was a lack of adequate alternative solutions to overcome the explained problems. The only alternative seemed to patch source code: This implies that the sources are available and that building the 3rd party library is feasible. This could go for a single version of OpenJPA, but did not work with the solidDB JDBC driver. Hence, our project management was slightly forced to accept AO.

However, AspectJ in its "originally intended" form is still unacceptable, because the infrastructure would require a lot of significant changes: As a new language, AspectJ requires a special compiler, for instance given by the Eclipse AJDT plug-in. Nonetheless, we have used AspectJ, but it is important that we have used aspects that are implemented as ordinary Java classes. All the AspectJ concepts such as aspects, pointcuts and advices are specified as annotations. Instead of using load-time weaving (cf. Section 4), which caused some problems with the class loading of the underlying OSGi container, we preferred an explicit instrumentation. The aspect classes are compiled with the Java compiler and then applied to existing JAR files in a separate step, particularly to 3rd party JAR files such as OpenJPA or JDBC drivers. Both steps require the predefined `iajc` taskdef to invoke the AspectJ compiler in Ant build scripts. The result is a new JAR, e.g., `myopenjpa.jar`, which must be used instead of the original one. Please note building the new JAR file requires only a single build file and a single additional build step. As a consequence, no source code and no knowledge about the build process is required for modifications to a 3rd party tool's JAR file. Integration into an external build process, for example by using a tool like Cruise Control with daily builds and overnight test reports, does not pose any problems and can be done by exchanging the JAR files. And finally, scaling problems with AspectJ for large projects such as long compile-times, as reported by [17], are avoided.

6 Conclusions

This paper reports on problems that occurred in a concrete replacement scenario in an industrial telecommunication project where the object-relational persistence framework Hibernate has been replaced with OpenJPA due to licensing and patent problems.

At a first glance, the Hibernate replacement has appeared as a straightforward task, because there are only syntactic differences in the APIs and in the mapping specifications of both frameworks. In fact, putting the Hibernate interface on top of OpenJPA reduced code changes to simply exchanging packages. This kept the replacement effort low. However, harder problems occurred and endangered the success of the replacement. For example, OpenJPA does not offer Hibernate's native key generation strategy and OpenJPA prevents a failover between two solidDB database servers. This functionality is important for the telecommunication middleware, and hence, solutions are indispensable!

For these harder problems, we have presented the successful adoption of aspect-orientation (AO), especially AO programming with AspectJ [12]. Particularly, with this approach we bridged the gap of functionality and handled deficits of internal functionality. The key to success was not only AspectJ, but the special capability to apply aspects to external JAR files the source code of which is unavailable. By this technique, we were able to correct the behavior of OpenJPA and JDBC drivers. Aspects can change the behavior, however, leave the source code and original JARs intact. Thus, the essential and novel value of our AO approach is a method to address the challenges of integrating 3rd party software, keeping the original software untouched and being able to manage the concerns of replacement in a maintainable manner.

It is AspectJ that let the replacement succeed with simple solutions in short time. In contrast to [20], we were satisfied with the power of the AspectJ language. Indeed, AspectJ is a powerful language and we are simply using this power to easily solve critical problems quickly. Moreover, there is a lack of adequate alternative solutions. The only alternative seems to patch the source code explicitly – if available at all. The effort for changing the source code, adding data exchange between unrelated classes, and building the JAR leads to more complexity, error proneness, and effort than our AO-based approach. Moreover, we are unsure whether the problems could be solved with conventional techniques since the source code of JDBC drivers is usually not available.

Another advantage becomes obvious. Although we exchanged the solidDB JDBC driver twice and switched from OpenJPA version 0.9.7 to 1.1.0 during the effort, we did not touch the aspects, they are stable and still work correctly with the newer versions.

In future work, we want to apply AO for other purposes in the project. For example, we currently use a model-driven approach to generate code from XML specifications, i.e., several Java classes are generated by XSL-T transformations. We want to investigate whether AspectJ could be an alternative, although others decline appropriateness [21]. We hope that such a solution could be easier to use, better understandable, and evolvable.

References

1. Strunk, W.: The Symphonia Product-Line. In: Java and Object-Oriented (JAOO) Conference (2007)
2. Elrad, T., Filman, R., Bader, A.: Theme Section on Aspect-Oriented Programming. CACM 44(10) (2001)
3. Murphy, G., Walker, A.R., Robillard, M.: Separating Features in Source Code: An Exploratory Study. In: Proc. of 23rd Int. Conf. on Software Engineering (2001)
4. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: Proc. of the 17th Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2002 (2002)

5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
6. Burke, B.: Implementing Middleware Using AOP. In: Proc. 4th Conf. on Aspect-Oriented Software Development (AOSD), Chicago (2005)
7. Laddad, R.: Aspect-Oriented Database Systems. Springer, Heidelberg (2004)
8. Rashid, A.: Persistence as an Aspect. In: [22]
9. Hohenstein U.: Using Aspect-Orientation to Manage Database Statistics. In: [23]
10. Kienzle, J., Gélineau, S.: AO Challenge – Implementing the ACID Properties for Transactional Attributes. In: Proc. of 5th Int. Conf. on Aspect-Oriented Software Development, Bonn, Germany (2006)
11. Coady, Y., Kiczales, G.: Back to the Future: A Retrospective Study of Aspect Evolution in Operating System Code. In: [22]
12. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming, 2nd edn. Manning, Greenwich (2008)
13. Vines, D., Sutter, K.: Migrating Legacy Hibernate Applications to OpenJPA and EJB 3.0., `http://www.ibm.com/developerworks/websphere/techjournal/0708_vines/0708_vines.html`
14. Laddad, R.: AOP@Work: Myths about AOP, `http://www-128.ibm.com/developerworks/java/library/j-aopwork15`
15. Duck, A.: Implementation of AOP in Non-Academic Projects. In: [23]
16. Joosen, W., Sanen, F., Truyen, E.: Dissemination of AOSD expertise support documentation. AOSD-Europe Deliverable No.: AOSD-Europe-KUL-8
17. Wiese, D., Meunier, R.: Large Scale Application of AOP in the Healthcare Domain: A Case Study. In: Industry Track of 7th Int. Conf. on Aspect-Oriented Software Development (AOSD), Brussels (2008)
18. Kiczales, G.: Adopting AOP. In: Proc. 4th Conf. on Aspect-Oriented Software Development; AOSD 2005, Chicago. ACM Press, New York (2005)
19. Wiese, D., Hohenstein, U., Meunier, R.: How to Convince Industry of Aspect-Orientation? In: Industry Track of 6th Int. Conf. on Aspect-Oriented Software Development, AOSD 2007, Vancouver (2007)
20. Ostermann, K., Mezini, M., Bockisch, C.: Expressive Pointcuts for Increased Modularity. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 214–240. Springer, Heidelberg (2005)
21. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features Using AspectJ. In: Proc. Int. Software Product Line Conference (SPLC), Kyoto. IEEE Computer Society, Los Alamitos (2007)
22. Aksit, M.: Proc. of 2nd Int. Conf. on Aspect-Oriented Software Development. In: AOSD 2003 (2003)
23. Chapman, M., Vasseur, A., Kniesel, G.: Proc. of Industry Track of 3rd Conf. on Aspect-Oriented Software Development (AOSD), Bonn (2006)

Trends in Harmonization of Multiple Reference Models

César Pardo[1], Francisco J. Pino[1], Félix García[2]
Mario Piattini Velthius[2], and Maria Teresa Baldassarre[3]

[1] IDIS Research Group, Electronic and Telecommunications Engineering Faculty
University of Cauca, Calle 5 No. 4 – 70, Kybele Consulting Colombia (Spinoff)
Popayán, Cauca, Colombia
[2] Alarcos Research Group, Institute of Information Technologies & Systems
University of Castilla-La Mancha, Paseo de la Universidad 4, Ciudad Real, Spain
[3] Department of Informatics, University of Bari
SER & Practices, SPINOFF, Via E. Orabona 4, 70126, Bari, Italy
{cpardo,fjpino}@unicauca.edu.co
{Felix.Garcia,Mario.Piattini}@uclm.es
baldassarre@di.uniba.it

Abstract. Diverse models currently exist in the field of Software Engineering which help organizations to apply recommended practices in order to support ther multiple needs in the areas of software development, maintenance and operation, security, IT government, etc. Examples of such models are CMMI, ISO 9001, ISO 12207, ISO 27001, COBIT, ITIL. Nevertheless, many differences exist between these models, since each model defines its own structure, terminology, definitions and quality systems, amongst other aspects. This issue increases the complexity when an organization is required to apply two or more models in order to satisfy its needs. Organizations must, therefore, define the most appropriate means of choosing and implementing multi-models, and harmonization may be one solution. This paper presents a systematic literature review with the aim of analyzing the state of the art with regard to inititatives concerning the harmonization of multiple reference models. As a result, it has been concluded that there is currently a lack of guidelines with which to help organizations to implement the harmonization of multiple models, and of a unified terminology with which to homogenize the diversity of the structure of the different models and the harmonization techniques which can be applied. In order to address these issues, a framework to support the harmonization of multiple models is outlined.

Keywords: Multi-model, Multiple, Reference models, Harmonization, Software process improvement, Systematic review.

1 Introduction

There is currently a wide range of models that can be taken as references for the improvement of an organization's processes, e.g. models to improve quality management such as ISO 9001, models for software quality management such as CMMI, ISO 12207 and ISO 90003, models for IT governance such as ITIL, PMBOK and COBIT, models for security management systems such as 27000, models for IT Service Management such as ISO 20000 and Bodies of Knowledge such as SWEBOK, amongst others. According to [1], it would be imprudent to think that any of the models defined at present provides a total solution for process management in the context of: Information Security Management System (ISMS), Information Technology Governance Processes (IT Governance), or processes of development, software maintenance and operation.

The great diversity and heterogeneity of available reference models, together with the need to solve problems from many dimensions and organizational hierarchies, provides organizations with a positive environment which enables them to choose different solutions to various problems and needs [2]. However, each of these approaches defines its own structure of process entities, definitions and quality systems, which increases the complexity in the implementation of multi-models in a single organization. Organizations must, therefore, define the most appropriate means of choosing and implementing multi-models in the face of this huge quantity. Harmonization may be one solution towards working simultaneously with multiple models [2]. The multi-model environments in software process improvement are present when an organization decides or needs to integrate into its processes different practices or characteristics that are present not in one, but in several models [3].

At present, although the number of related works on the harmonization of multiple models is small, in the last 4 years there is within the software engineering community an ever-increasing interest in defining solutions for this type of environments. This is evidenced by the initiatives and projects performed or being carried out, such as: the PrIME project of the SEI [4], ARMONÍAS project of the research group ALARCOS [5], Enterprise SPICE [6], among other publications and works analyzed in this paper.

In this article, we present a systematic review of the literature which deals with the proposals that exist to support the harmonization of reference models for process improvement. In accordance with the general goals of systematic reviews, our aim is to provide an up-to-date state of the art which synthesizes the work in this area of knowledge and which can be used to identify gaps from which to formulate innovative research activities. The works found are classified and analyzed taking into account the trends of publication, the models used and the methods and techniques proposed. Some factors that influence the work with multiple models, as identified from the studies analyzed, are set out.

This paper proceeds as follows. The systematic review itself is presented is presented in Section 2. Section 3 presents the results obtained along with a discussion of them. Section 4 outlines a framework with which to address the principal issues identified with regard to the harmonization of multiple models, and finally, our conclusions and future work are described.

2 Systematic Review on the Harmonization of Reference Models

To carry out the systematic review on the harmonization of reference models we followed the guidelines presented in [7], the protocol template defined in [8] and the field procedure proposed in [9].

The research question is: *What works and initiatives related to the harmonization and integration of reference models have been carried out?* The list of keywords used to find an answer to the research question is shown in the basic search string presented in Table 1.

Table 1. Basic search strings

(integration OR integrating OR integrated OR unification OR unifying OR unified OR combination OR combining OR combined OR mapping OR mapped OR harmonization OR harmonizing OR harmonized OR) AND (standards OR models OR frameworks OR technologies) AND ("process improvement" OR "software process")

The planned list of sources with which the systematic review was carried out is:

- Science@Direct, on the subject of Computer Science,
- Wiley InterScience, on the subject of Computer Science,
- IEEE Digital Library,
- ACM Digital Library, and
- As grey literature, the reports of the PRIME project from the SEI were reviewed. In addition, some papers and works delivered by experts were reviewed.

The inclusion criterion of the primary studies obtained focused on the analysis of the title, abstract and keywords. This allowed us to determine whether the articles found were related to software process improvement, and moreover whether they perform or propose a strategy for carrying out the harmonization of multiple-models.

The exclusion criterion focused on the reading and detailed analysis of the abstract and conclusions. In certain cases where this was not enough, it was necessary to extend the analysis to other parts of the document.

The selection of studies followed an iterative and incremental procedure. This procedure was implemented by searching, extracting and visualizing results from each search source iteratively. In this way the revision report grew and evolved more and more until it was complete, thereby obtaining the final revision report.

3 Results and Discussion

On the basis of information extracted from the studies found, a statistical analysis to show relevant findings of the systematic review was performed. Below are the results from different points of view.

3.1 Trends of the Publications Multi-model Environments in Software Process Improvement

As shown in Figure 1, we may note that there has been increasing interest in recent years on the part of the software engineering community with regard to process improvement environments where multiple models are involved.

Figure 1 shows an increase of the publications found in the last years. From the analysis of the 32 studies found (see all references of the studies selected in references section), it is possible to classify them into six categories. Figure 2 illustrates a summary of the categorized studies.

A brief summary of the studies categorized is presented below:

a. *Studies where only two process reference models are harmonized.* These models can be from the same organization, or different. It is possible to see that 38% (12) of the works found harmonize only two models. In these proposals models are harmonized based on internationally recognized standards, e.g. ISO 9001 and CMM [10-12], and ISO 9001 and CMMI [13-16]. These proposals seek to integrate the processes of the models from ones that have been previously institutionalized. Other studies attempt to integrate CMM or CMMI with other models different and apart from ISO 9001. These are: CMM and Cleanroom model [17], CMMI and SWEBOK model [18], CMMI and Six-Sigma model [19], CMMI and ITIL [20] and CMMI and ISO 12207 [21].

b. *Studies that harmonize more than two process reference models.* 9% (3) of the works found harmonize more than two models, e.g. the high-level comparison between EIA IS 731, the CMMI^{SM} and SECM [22], the analysis performed to identify the problems of interoperability and harmonization of the models ISO/IEC 15288, EIA 632, IEEE 1220 and other related ISO standards [23], and the aligning of Cobit 4.1, ITIL V3 and ISO/IEC 27002 for Business Benefit [24].

c. *Studies that harmonize two or more process reference models and assessment models.* 22% (7) of the studies analyze the integration of the assessment models and their implementation in different process reference models. Some of the related studies include: analysis of compatibility between SPICE and CMM [25], analysis of the compatibility of CMMI as Process Assessment Model, ISO 12207 as Process Reference Model and ISO 15504-2 as Measurement Framework [21, 26], integration of ISO/IEC 15504 and CMMI-SE/SW [26, 27], defining support structures and comparison between CMMI and SPICE [28, 29], among others.

d. *Studies that propose unique and/or universal models.* 3% (1) of the works found correspond to a study that proposes a unique and/or universal model, but which does not describe the solutions used, e.g. steps, activities or process performed carried out. The study found presents the lessons learnt from the definition of the Capability Maturity Model (iCMM), as a new approach that integrates multiple approaches, including: ISO 9001, Malcolm Baldridge National Quality Award criteria, International lifecycle and assessment standards and processes, and several CMMs [30].

e. *Studies that provide a solution for supporting multi-model harmonization.* 25% (8) of the works proposed provide solutions (methodology, process, framework, activities, tasks, steps, amongst other elements) for supporting the harmonization of multiple models, these being the following: the VM XT project, which is applied as the standard in harmonizing the different approaches and projects of Information technology (IT) under a specific model [31], an ontology for the integration of quality standards in ISO 9001:2000 and CMMI is taken for collaborative projects [32]. The PRIME project presents the value of harmonization process improvement in organizations when different models are in use [3, 33, 34] and Infosys Project defines a path for the transition from ISO 9001 to SW-CMM level 4, based on the experience of an organization [35]. Enterprise SPICE is an initiative to establish an Enterprise Integrated Standards-Based model for use with international standard ISO/IEC 15504 (SPICE) [6]. In [36] a work is presented that identifies principles and process characteristics for designing a system of processes at the architectural level and in [37] we can discover research that defines a method for process-based unification of different approaches to multiple process-oriented software quality.

f. *Studies that provide analysis of multiple models or related concepts.* 3% (1) of the works found correspond to a study that recognizes the value of having processes that are drawn from widely accepted and proven quality models e.g. CMMI-DEV, ISO 9000, ISO 20000, eSCMSP, ITIL, Lean Six Sigma and ISO 27001 [38].

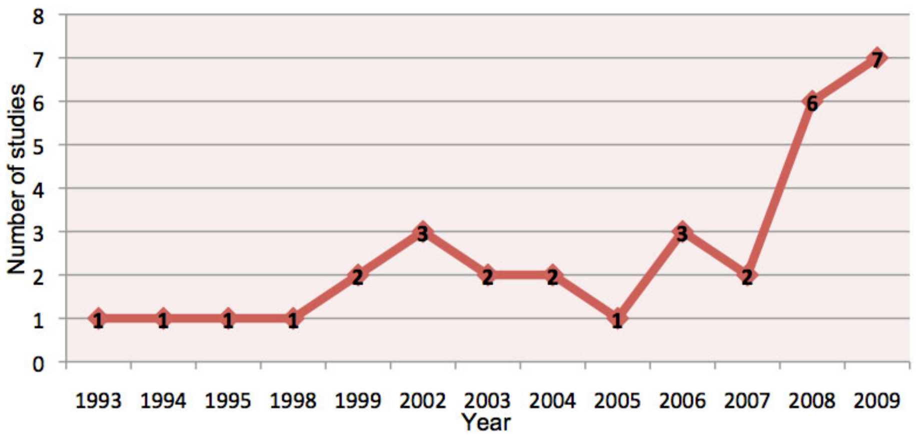


Fig. 1. Trends of the publications on Multi-Model environments in Software Process Improvement

3.2 Models Used

On the basis of the analysis and classification performed above, it is significant to highlight that in the harmonization of models, different types of models are involved. In Table 2 the process reference models and reference models for assessment used in the studies are shown in alphabetical order. As can be seen in the Table, the models for assessment that are most frequently used in the integration with other models are the ISO/IEC 15504 or SPICE, at 11%. Likewise, it can be seen that the process reference models which are most frequently used are CMM (13%), CMMI (25%) and ISO 9001 (18%). On the other hand, models such as ITIL and ISO 27000 (Part 1 or 2) are used in a lesser percentage; 5% each one, respectively. The ISO 12207 and Sigma and Lean Six-Sigma have 4% use compared to other models such as CSE, COBIT, EIA IS 731, eSCMSP, ISO 20000, SECM, SWEBOK, V-Modell XT, Six- and other ISO standards have a 2% usage each.

With regard to process reference models, those which are most widely used are the ISO models at 41%, of which ISO 9001 is the most frequently used, at 18%, and the SEI models at 39%, of the which the CMMI is the most frequently used at 25%. Other models are used in smaller percentage (20%); see Figure 3(a). Likewise, we can observe that in most of the studies that involve these models, the way of achieving CMM or CMMI is analyzed starting from ISO 9001. Although the major aim is to reuse parts of the ISO standards in a CMM or CMMI environment, it is difficult for an ISO-certified organization to implement CMMI easily because of the differences in the language, structure, and details of the two sets of documents; see [14].

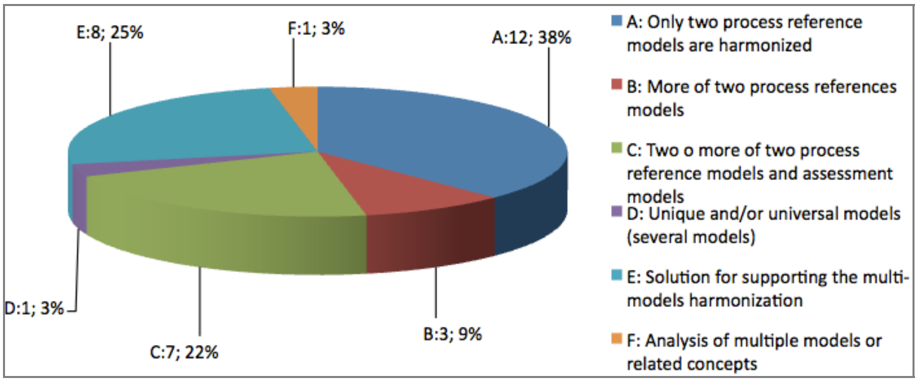


Fig. 2. Classification of the works found

Table 2. Models used

Models	Total	%
Cleanroom Software Engineering (CSE)	1	2
CMM	7	13
CMMI	14	25
COBIT	1	2
EIA IS 731	1	2
eSCMSP	1	2
ISO 12207	2	4
SPICE or ISO 15504	6	11
ISO 20000	1	2
ISO 9001	10	18
ISO/IEC 15288, EIA 632, EEE 1220 and other related ISO standards	1	2
ISO 27000 Part 1 and Part 2	3	5
ITIL	3	5
SECM	1	2
Six-Sigma or Lean Six-Sigma	2	4
SWEBOK	1	2
V-Modell XT	1	2
TOTAL	56	100%

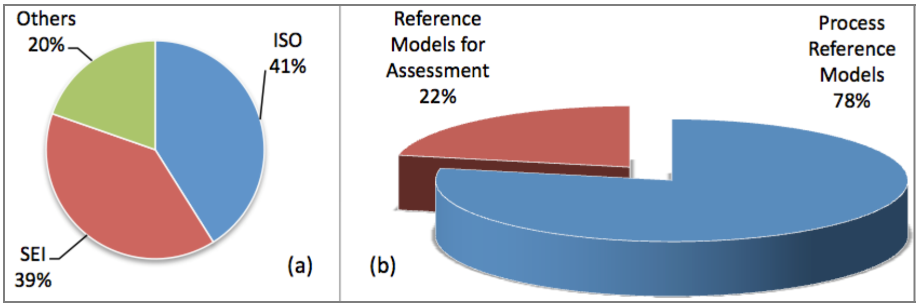


Fig. 3. Reference model for the assessment and process reference models involved

With regard to process reference models and reference model for assessment, Figure 3(b) shows that: (i) 22% of the studies involve the harmonization of reference models for assessment and process reference models and (ii) 78% only involve the study of process reference models. This suggests that there is a special interest in analyzing the compatibility and the relationships between two approaches, e.g. the relationships established between CMMI, as a candidate conformant Process Assessment Model, relative to the Measurement Framework defined in ISO/IEC 15504-2, and the Process Reference Model described in ISO/IEC 12207, e.g. [26].

3.3 Methods and Techniques Proposed

With regard to the analyses carried out above, this section provides a brief summary of some of the methods and techniques used in works found. Table 3 shows those techniques used. Likewise, it shows that several attempts have been made to define solutions for the harmonization of multi-models. These works propose various techniques with solutions to support harmonization. The techniques used are classified in different ways, e.g. the activity used to discover related elements in several models may be called *comparison* or *mapping*. Other works use terms such as *synergy* or *compatibility* to identify the level of relationship between models. However, most related comparison techniques do not use a *comparison scale* that

Table 3. Models used

Technique, term or concept used	Studies	Quantity	%
Integration, Unification	[3, 33, 34], [6], [13, 39], [35], [31], [32]	10	31
Comparison, Mapping, Align	[31], [10-12], [15, 18], [25, 26], [22], [24], [17], [16], [21], [22], [27], [28]	17	50
Combine, Combination, Merger, Single model, Universal model	[19], [20], [40]	3	9
Harmonization	[23],	1	3
Neither	[36], [37]	2	6

allows a range for the relations identified among the models compared to be established. This would allow the subjectivity in the comparison to be minimized. Similarly, *combining* and *merger* are used to refer to several *integrated* or *unified* models, but with the difference that the steps followed for their integration are not shown. Some works use the term *single model* or *universal model*. Likewise, *complementarily* is used to refer to models that take elements of other models to maximize their qualities.

It may be seen that of the techniques used in 50% of the studies analyzed, some kind of comparison, alignment or mapping is used as a technique leading to the harmonization of multiple models. Only some of the studies propose different harmonization techniques. However, we believe that the techniques or terms used in the other studies correspond to general or related concepts. In that sense, we believe that the terms found can be classified into *methods* and *techniques*. The *methods* are general procedures and the *techniques* are specific procedures applied to the definition or framework of a method. That is, a *method* is a procedure which is generally oriented towards a specific purpose, while the *techniques* are different ways of applying the method. Based on the techniques found, in Table 4 we have ordered the techniques, terms or concepts used in the studies analyzed into a general concept called harmonization, along with methods, techniques and the possible objective or result.

Table 4. Methods and techniques

Methods	Techniques	Objective
Comparison	Align, Mapping [21], [15, 18], [31],	Complement Homogenization
Integration or Unification	Combine, Merger [35], [13, 39], [31],	Single model, Universal model

3.4 Factors That Influence the Work with Multiple Models

The primary studies were also used to search for and extract the information that reported the factors that may influence an organization in needing to work with more than one assessment or process reference model. The following can be highlighted as some of these:

- *Market niches with specific models.* It is possible that in some market niches the groups of organizations prefer certain models or fact standards, e.g. according to the literature analyzed, CMMI or ISO 9001, respectively.
- *Improvement of practices from legacy process models.* It is possible that is necessary to carry out the complementarily of the process and practices which have been institutionalized from specialized models or more detailed ones, e.g. to obtain a certification in CMMI from an ISO certification obtained previously, see [14].
- *Business positioning.* Although certification on a specific model does not entail an increase in sales for an organization, at a commercial level it increases confidence among its customers, allowing a better business positioning.
- *Leveraged or merger corporate.* It is possible that in a corporate merger the organizations do not use the same model. Taking into account that in a merger

an organization can be absorbed by other, it is necessary to identify and define rules to lead the merger adequately.

- *Systematic search of the capability of the processes.* For the organizations interested in performing a continual and ever–more-complete improvement of their processes, it is possible that the harmonization of multiple models may allow them to carry out substantial growth in the capacity of their processes from other models.
- *Business growth.* Business growth involves more mature and complex processes. At any specific time in their business growth, organizations can require integration of models and practices that support the performing of activities and the process of management and/or development.

4 A Framework to Harmonize Multiple Models

In order to address the principal issues identified in this systematic review, a framework to support the harmonization of multiple models has been developed, which is composed of the following elements:

- A set of support guidelines for the determination of the harmonization goals, along with criteria for the selection and configuration of the harmonization strategy.
- An ontology for the harmonization of multiple reference models called H2mO [44], which presents a set of terms, concepts and relationships to support the harmonization and integration of models. This is also related to another ontology, which is an extension of the above, and establishes and clarifies the key elements with which to express process-based approaches of any reference model. This ontology is called *Ontology of Process-reference Models* (OPrM). A *Common Schema or Common Structure of Process Entities* (CSPE) has been was developed, which is used along with a harmonization technique to facilitate the harmonization of different models. A detailed summary of the CSPE template and homogenization technique is presented in [2].
- A set of techniques and methods to support the identification and definition of the harmonization strategies to be implemented in the harmonization process. This is comprised of three different techniques: harmonization, comparison and integration, see [2] and [41] respectively. The integration technique is currently being developed from the results obtained from the integration of six models to support the improvement procedures and Information Technology (IT) Government in the banking sector. The integrated models are: COBIT 4.1, BASEL II, RISK IT, ITIL V.3, VAL IT and ISO/IEC 27002, and a detailed summary of the model obtained, called *IT Government to the Banking Sector Model (ITGSM),* the harmonization strategy defined is presented in [45], [42].
- A Process for driving harmonization of multi-models. This process describes a set of activities, tasks and roles, which permit the configuration of a

suitable harmonization strategy needed to drive the harmonization of multiple reference models [43].

Figure 4 shows a summary and the relationships between the elements described above.

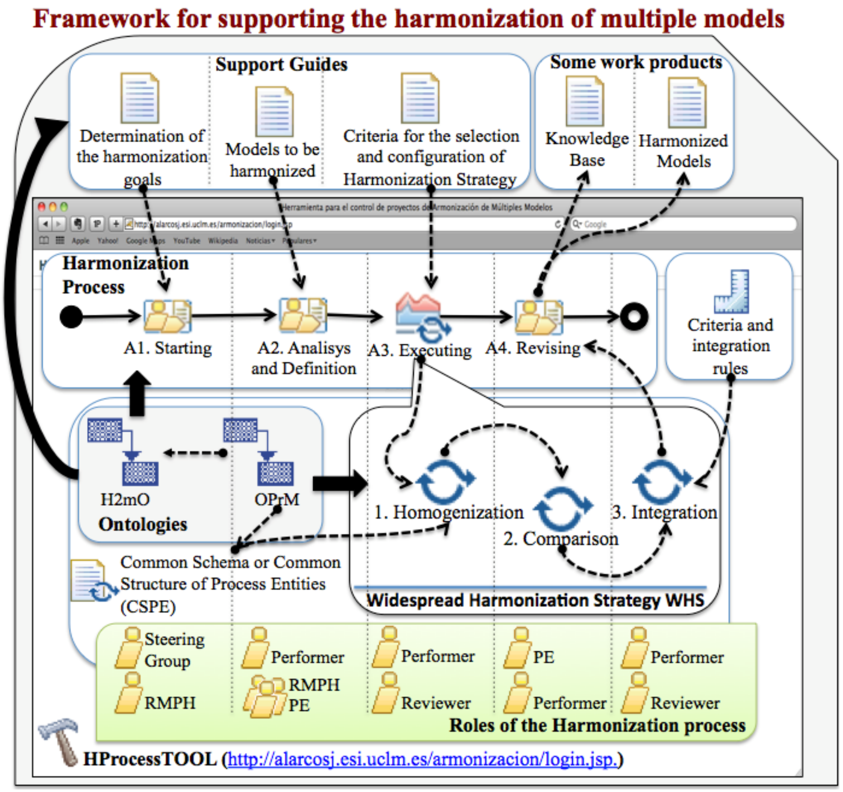

Framework for supporting the harmonization of multiple models
Support Guides
Determination of the harmonization goals
Models to be harmonized
Criteria for the selection and configuration of Harmonization Strategy
Some work products
Knowledge Base
Harmonized Models
Harmonization Process
A1. Starting
A2. Analisys and Definition
A3. Executing
A4. Revising
Criteria and integration rules
H2mO
OPrM
Ontologies
Common Schema or Common Structure of Process Entities (CSPE)
1. Homogenization
2. Comparison
3. Integration
Widespread Harmonization Strategy WHS
Steering Group
RMPH
Performer
RMPH
PE
Performer
Reviewer
PE
Performer
Performer
Reviewer
Roles of the Harmonization process
HProcessTOOL (http://alarcosj.esi.uclm.es/armonizacion/login.jsp)

Fig. 4. Framework to support the harmonization of multiple reference models

5 Conclusions

Undoubtedly, the effort required in systematic reviews is considerably greater than for a conventional review of the literature. The way systematic reviews are performed allows us to summarize the evidence found on a specific topic. In this article a systematic review of the literature on the harmonization of multi-models for software process improvement has been presented, which has allowed us to obtain a view of the initiatives and related works.

From the results obtained in the current review, the first observation from the study that was carried out is that in the last 4 years there has been an ever-increasing interest on the part of the software engineering community in harmonizing multiple models. Currently, software development organizations may need more than one model to support and achieve the organization's strategic goals. Nevertheless, there is a lack of proposals, so for the organizations it is no easy task to carry out the implementation and management of the different situations which must be taken into account in order to harmonize more than two approaches or models as references for software process improvement.

With regard to the most frequently used models, it can be seen that the CMMI as process reference model is the one most used by the SEI. We note that the models defined by the ISO are the ISO 9001 as reference model and the ISO 15504 as process assessment method, while a smaller percentage of studies involve other models.

Another relevant fact is that the systematic review carried out has allowed us to identify that various techniques have been defined to support the harmonization of multiple models, e.g. comparison, mapping, integration, unification, merger, amongst others. However, some of them may be similar despite having different names, or they may be different although they have the same name. There is no formal consensus or a single glossary with which to identify the techniques identified.

Another fact to highlight is that there are significant differences between the structures, terminology and approaches; these hinder the harmonization of multiple models. Likewise, it has been possible to identify several factors which influence the work with multi-model environments. These factors or needs we have identified can influence the approach to implementation or selection of the models when carrying out a multi-model project.

Bearing in mind the shortcomings found in this current research stream, we have presented a detailed summary of our research proposal, which defines a set of elements with which to facilitate the harmonization of multiple reference models.

Acknowledgements. This work has been funded by the projects: ARMONÍAS (JCCM of Spain, PII2I09-0223-7948), PEGASO/MAGO (MICINN and FEDER of Spain, TIN2009-13718-C02-01) and ARCA (CEC – JCCM of Spain - and FEDER, HITO-2009-06).

References

1. Piattini, M., Vidal, F.H.: Gobierno de las tecnologías y los sistemas de información, Ra-Ma, Madrid, España (2007)
2. Pardo, C., Pino, F., García, F., Piattini, M.: Homogenization of Models to Support multi-model processes in Improvement Environments. In: 4th International Conference on Software and Data Technologies, Sofía, pp. 151–156 (2009)
3. Siviy, J., Kirwan, P., Morley, J., Marino, L.: Maximizing your Process Improvement ROI through Harmonization. Technical report, Software Engineering Institute (SEI). Carnegie Mellon University (2008)
4. SEI: The PrIME Project (2010), `http://www.sei.cmu.edu/process/research/prime-details.cfm`
5. ARMONÍAS: A Process for Driving Multi-models Harmonization, ARMONÍAS Project (2009), `http://alarcos.esi.uclm.es/armonias/`
6. SPICE: Enterprise SPICE. An enterprise integrated standards-base model (2008), `http://www.enterprisespice.com/`
7. Kitchenham, B., Charters, S.: Guidelines for performing systematic literature reviews in software engineering: Version 2.3. EBSE Technical Report (2007)

8. Biolchini, J., Gomes, P., Cruz, A., Travassos, G.: Systematic Review in Software Engineering. Technical report, Systems Engineering and Computer Science Department, UFRJ (2005)
9. Pino, F., Garcia, F., Piattini, M.: Software Process Improvement in Small and Medium Software Enterprises: A Systematic Review. Software Quality Journal 16, 237–261 (2008)
10. Paulk, M.C.: Comparing ISO 9001 and the Capability Maturity Model for Software. Software Quality Journal 2, 245–256 (1993)
11. Paulk, M.C.: A Comparison of ISO 9001 and the capability maturity model for software. Technical report, Software Engineering Institute (1994)
12. Paulk, M.C.: How ISO 9001 compares with the CMM. IEEE Software 12, 74–83 (1995)
13. Yoo, C., Yoon, J., Lee, B., Lee, C., Lee, J., Hyun, S., Wu, C.: An integrated model of ISO 9001:2000 and CMMI for ISO registered organizations. In: Proceedings - Asia-Pacific Software Engineering Conference (APSEC), Busan, pp. 150–157 (2004)
14. Yoo, C., Yoon, J., Lee, B., Lee, C., Lee, J., Hyun, S., Wu, C.: A unified model for the implementation of both ISO 9001:2000 and CMMI by ISO-certified organizations. Journal of Systems and Software 79, 954–961 (2006)
15. Mutafelija, B., Stromber, H.: ISO 9001:2000 - CMMI V1.1 Mappings. Technical report, Software Engineering Institute (2003)
16. Kitson, D.H., Vickroy, R., Walz, J., Wynn, D.: An Initial Comparative Analysis of the CMMI Version 1.2 Development Constellation and the ISO 9000 Family. Technical report, Software Engineering Institute. Carnegie Mellon (2009)
17. Oshana, R.S., Linger, R.C.: Capability maturity model software development using cleanroom software engineering principles - results of an industry project. In: Hawaii International Conference on System Sciences, Maui, p. 260 (1999)
18. Mutafelija, B., Stromber, H.: Architecting Standard Processes with SWEBOK and CMMI. In: SEPG 2006 Conference on Systems and Software Consortium, Nashville, p. 38 (2006)
19. Lin, L.-C., Li, T.-S., Kiang, J.P.: A continual improvement framework with integration of CMMI and six-sigma model for auto industry. Quality and Reliability Engineering International 25, 551–569 (2009)
20. CITIL: CMMI+ITIL (2010), `http://www.wibas.de/publikationen/referenzmodelle/was_ist_cmmi/index_de.html`
21. Pino, F., Balssarre, M.T., Piattini, M., Visaggio, G.: Harmonizing maturity levels from CMMI-DEV and ISO/IEC 15504. Software Process: Improvement and Practice (2009) (in press)
22. Minnich, I.: EIA IS 731 compared to CMMI^{SM}-SE/SW. Systems Engineering 5, 62–72 (2002)
23. Croll, P.R.: Interoperability of Systems Engineering Standards-Harmonizing World and National Perspectives. In: 5th Annual Systems Engineering Conference, Tampa, p. 30 (2002)
24. ITGI: Aligning Cobit 4.1, ITIL V3 and ISO/IEC 27002 for Business Benefit. Technical report, IT Governance Institute (ITGI) and Office of Government Commerce (OGC) (2008)
25. Rout, T.P.: SPICE and the CMM: is the CMM compatible with ISO/IEC 15504? AquIS, Venice, Italy 12 (1998)
26. Rout, T.P., Tuffley, A.: Harmonizing ISO/IEC 15504 and CMMI. Software Process: Improvement and Practice 12, 361–371 (2007)
27. Wangenheim, C.G.v., Thiry, M.: Analyzing the Integration of ISO/IEC 15504 and CMMI-SE/SW. Technical report, LQPS - Laboratorio de Qualidade e Productividade de Software. Universidad do Vale do Itajaí - UNIVALI (2005)

28. Lepasaar, M., Mäkinen, T., Varkoi, T.: Structural comparison of SPICE and continuos CMMI. In: The Proceedings of SPICE 2002, Venice, Italy, pp. 223–234 (2002)
29. Foegen, M., Richter, J.: CMM, CMMI and ISO 15504 (SPICE). Technical report, IT Maturity Services (2003)
30. Ibrahim, L., Pyster, A.: A Single Model for Process Improvement. IT Professional 6, 43–49 (2004)
31. Biffl, S., Winkler, D., Höhn, R., Wetzel, H.: Software process improvement in Europe: potential of the new V-modell XT and research issues. Software Process: Improvement and Practice 11, 229–238 (2006)
32. Ferchichi, A., Bigand, M., Lefebvre, H.: An Ontology for Quality Standards Integration in Software Collaborative Projects. In: First International Workshop on Model Driven Interoperability for Sustainable Information Systems, Montpellier, pp. 17–30 (2008)
33. Siviy, J., Kirwan, P., Marino, L., Morley, J.: The Value of Harmonization Multiple Improvement Technologies: A Process Improvement Professional's View. Technical report, Software Engineering Institute, Carnegie Mellon (2008)
34. Siviy, J., Kirwan, P., Renato, V., Peter, K., Gerhard, G.: SEPG Europe 2008. In: Multimodel Improvement in Practice, Munich, p. 23 (2008)
35. Jalote, P.: CMM in Practice: Processes for Executing Software Projects at Infosys, vol. 1. Addison-Wesley Professional, Massachusetts (1999)
36. Ferreira, A., Machado, R.J.: Software Process Improvement in Multimodel Environments. In: Fourth International Conference on Software Engineering Advances (ICSEA 2009), Porto, pp. 512–517 (2009)
37. Kelemen, Z.D.: A Process Based Unification of Process-Oriented Software Quality Approaches. In: Proceedings of the 2009 Fourth IEEE International Conference on Global Software Engineering (2009)
38. Heston, K.M., Phifer, W.: The Multiple Quality Models Paradox: How Much 'Best practice' is Just Enough? Software Process: Improvement and Practice (2009) (in press)
39. Yoo, C., Yoon, J., Lee, B., Lee, C., Lee, J., Hyun, S., Wu, C.: A unified model for the implementation of both ISO 9001:2000 and CMMI by ISO-certified organizations. Journal of Systems and Software 79, 954–961 (2006)
40. Ibrahim, L., Pyster, A.: A Single Model for Process Improvement. IT Professional 6, 43–49 (2004)
41. Pino, F., Balssarre, M.T., Piattini, M., Visaggio, G.: Harmonizing maturity levels from CMMI-DEV and ISO/IEC 15504. Software Process: Improvement and Practice (2009) doi:10.1002/spip.443
42. Lemus, S.M., Pino, F.J., Piattini, M.: Towards a Model for Information Technology Governance applicable to the Banking Sector. In: V International Congress on IT Governance and Service Management (ITGSM 2010), Alcalá de Henares, pp. 1–6 (2010)
43. Pardo, C., Pino, F.J., García, F., Piattini, M., Baldassarre, M.T.: A Process for Driving the Harmonization of Models. In: The 11th International Conference on Product Focused Software Development and Process Improvement (PROFES 2010), Second Proceeding: Short Papers, Doctoral Symposium and Workshps, Limerick, pp. 53–56 (2010)
44. Pardo, C., Pino, F.J., García, F., Piattini, M., Baldassarre, M.T.: An ontology for the harmonization of multiple standards and models. Computer Standards & Interfaces (in press, accepted manuscript, 2011), doi: 10.1016/j.csi.2011.05.005
45. Pardo, C., Pino, F.J., García, F., Piattini, M., Baldassarre, M.T., Lemus, S.: Homogenization, Comparison and Integration: A Harmonizing Strategy for the Unification of Multi-Models in the Banking Sector. In: Caivano, D., Oivo, M., Baldassarre, M.T., Visaggio, G. (eds.) PROFES 2011. LNCS, vol. 6759, pp. 59–72. Springer, Heidelberg (2011)

Prioritization of Stakeholder Value Using Metrics

Lindsey Brodie and Mark Woodman

School of Engineering and Information Sciences, Middlesex University
The Burroughs, Hendon, London, NW4 4BT, U.K.
`{L.Brodie,M.Woodman}@mdx.ac.uk`

Abstract. Given the reality of resource constraints, software development always involves prioritization to establish what to implement. Iterative and incremental development methods increase the need to support dynamic prioritization to identify high stakeholder value. In this paper we argue that the current prioritization methods fail to appropriately structure the data for stakeholder value. This problem is often compounded by a failure to handle multiple stakeholder viewpoints. We propose an extension to an existing prioritization method, impact estimation, to move towards better capture of explicit stakeholder value and to cater for multiple stakeholders. A key feature is the use of absolute scale data for stakeholder value. We use a small industry case study to evaluate this new approach. Our findings argue that it provides a better basis for supporting priority decision-making over the implementation choices for requirements and designs.

Keywords: Stakeholder value, Impact estimation, Requirements prioritization, Design prioritization, Metrics, Value-based software engineering.

1 Introduction

Research into prioritization has increased in recent years with many new prioritization methods and variants being put forward. Much has been achieved in identifying the prioritization factors and the issues of concern when structuring prioritization data. However, existing prioritization methods and the prioritization data they utilize (in content and structure) continues to be insufficient to support the type of prioritization process that ideally needs to be adopted. Specifically, progress in improving the prioritization process seems hampered by inadequate conceptualizations of stakeholder value, in particular by the use of implicit notions of value. This is often compounded by an additional failure to support multiple stakeholder viewpoints. Note the term "stakeholder" is used here to mean any group of people with an interest in the system, and they can be identified by role and/or location.

In this paper, to move towards addressing the problems identified above, we propose capturing stakeholder value by stakeholder role, and using absolute scale data (as opposed to using, for example, ordinal scale data) for stakeholder value. We consider the explicit "real world" data captured by using absolute scales normally provides a better basis for supporting priority decision-making. For example, as we shall discuss later, it supports arithmetic calculations such as return on investment (ROI).

To present the argument for our proposals for stakeholder value, this paper is structured in the following way. Section 2 outlines the need for prioritization explaining why the prioritization process is important. Section 3 provides an overview of the existing research on prioritization and analyses how it relates to the problems we perceive impacting the prioritization of stakeholder value. Section 4 then investigates in detail how stakeholder value is currently expressed within the prioritization data and explains some of the resulting weaknesses. Finally, Section 5 briefly describes initial validation of using explicit absolute scale data for stakeholder value: a case study using value impact estimation (VIE). We have developed VIE as a simple extension to an existing method, impact estimation (IE). IE [1] uses absolute scale data and captures the impact of each of the potential designs on each of the requirements. VIE extends this to additionally capture explicit stakeholder value by stakeholder role. Our initial findings are that use of absolute scales is indeed beneficial for capturing stakeholder value, and that capturing stakeholder value by stakeholder role is helpful for decision-making. However, there remains considerable future work to develop adequate theory on stakeholder value and stakeholder viewpoints, and improve understanding of the prioritization process.

2 The Need for Prioritization

2.1 Lack of Guidance

Prioritization can be considered something of a "gap" in current software engineering. Certainly within the most commonly used system development methods, it has had far too low a profile in the past. Also industry standards such as the Integrated Capability Maturity Model (CMMI) [2] and SWEBOK (Software Engineering Book of Knowledge) [3] fail to offer specific guidance on the prioritization process. This lack of attention matters because of the "bigger picture": the main purpose of prioritization is to help ensure projects are implementing the "right thing" at the "right time", while making good use of the always limited human, monetary and time resources. Opportunities to assist project planning, and so improve project delivery, are being lost if prioritization is not intelligently executed.

In addition, the demand to move towards value-based software engineering (VBSE) [4] raises the need for greater attention to be paid to the delivery of stakeholder value. Indeed, Sullivan [5] reports on a lack of "formal, testable and tested theories, methods, and tools to support economic-based analysis and decision-making (and value-based analysis more broadly)".

2.2 Changing Needs for Prioritization

Moreover, recent developments in software development mean that prioritization can be seen today as having a more central, on-going role to play throughout systems development. In Waterfall methods, prioritization only has to be carried out once, early on in the systems development process, and involves deciding what requirements are to be in the system and what are not. However, prioritization processes now have to support iterative and incremental development [6]. Such development requires on-going communication to capture data from the external environment, accept changing requirements, and receive feedback from each incremental delivery, in order to then establish what the stakeholders agree is of high value and should be in the next increment. This means the prioritization process has to cater for reuse of data while also accommodating changing data. Moreover, dynamic prioritization has to occur with each increment to determine what to implement next. Also that on-going identification of high stakeholder value is essential.

An additional demand comes from the recognition of the need for improved stakeholder understanding, especially the handling of multiple stakeholder viewpoints [7]. There is a need to not only capture and present the different viewpoints, but also to enable stakeholder negotiation and tradeoffs, and to help achieve stakeholder consensus and buy-in [8]. The prioritization process has a major part to play in providing better support to the system/product owners, who decide what shall be implemented.

3 Existing Research on Prioritization

Research in the area of prioritization has increased in recent years. In 1997, Karlsson and Ryan [9] wrote an influential paper describing their Cost-Value Approach based on the Analytic Hierarchy Process (AHP) [10], which acted as a springboard for much subsequent research. In this section, we briefly review the existing literature on prioritization: listing the existing prioritization methods, the identified prioritization factors and some of the identified issues with structuring prioritization data. Concurrently, we analyse how this existing research relates to the problems we perceive in prioritizing stakeholder value.

3.1 Positioning of Prioritization

Aspects of prioritization are discussed in the IT literature under several subject areas including requirements prioritization [11], [12], [13], release planning [14], architecture selection [15], COTS (Commercial Off-The-Self) selection [16], financial management [17], [18], and decision-making and negotiation methods [7]. There appears to be compartmentalization in the literature, which we argue needs questioning. While specialist areas for prioritization exist, it is essential that an overall view be considered because any given system encompasses many of these subject areas: there has to be interaction and integration at the system level. Accordingly, the stance taken by this research is that a holistic view should be taken: any overall prioritization process must include consideration of a wide range of prioritization data, which includes the fundamental software engineering concepts that we have termed here as “objective”, “requirement”, “design” and “increment”. All these four concepts impact on the concept of stakeholder value. For example, carrying out a prioritization process using just the requirements without consideration of, say, the potential designs and the operational impacts, both of which affect the costs, needs to be questioned. See Figure 1, which shows an increment delivery cycle with iteration around these concepts as software development progresses.

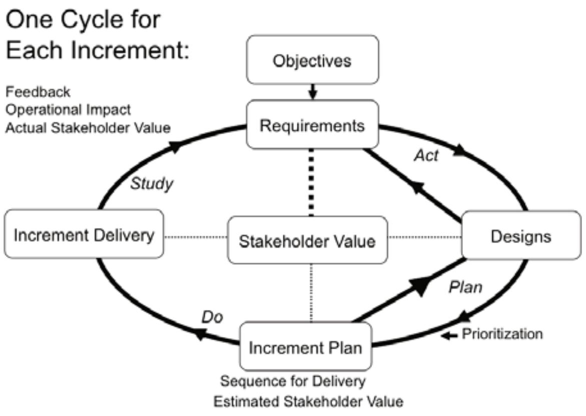


Fig. 1. Increment delivery cycle based on Deming's Plan-Do-Study-Act (PDSA) Cycle

Despite the previous argument, responsibility for the prioritization process and data model probably should reside within requirements engineering because it interfaces with the majority, if not all, of the stakeholders, and because the system requirements form the primary (but not sole) data for prioritization. However, care needs to be taken that there is adequate consideration of the wider aspects of the prioritization process that fall within other viewpoints, such as strategy management and operations management.

3.2 Existing Prioritization Methods

To date, we have identified over 60 different prioritization methods in the literature. For brevity, full discussion of these is not given. A selection of those found categorized by subject area is as follows:

Requirements Prioritization: MoSCoW [19], the Hundred-Dollar Test [20] and Requirements Prioritization Tool (RPT) [12].

Requirements (and Effort) Prioritization: Cost-Value Approach [9] and Wiegers' Method [21].

Requirements (and Design) Prioritization: Analytic Hierarchy Process (AHP) [10], Quality Function Deployment (QFD) [22], [23] and Impact Estimation (IE) [1].

Architecture (Design) Prioritization: Cost Benefit Analysis Method (CBAM) [15] and Reasoning Frameworks [24].

COTS (Design) Prioritization: Procurement-Orientated Requirements Engineering (PORE) [16] and Mismatch Handling for COTS Selection (MiHOS) [25].

Release Planning: Planning Game [26], EVOLVE/EVOLVE* [14], [27] and Requirements Triage [8].

Financial Prioritization: Business Case Analysis/ROI [28], Incremental Funding Method (IFM) [29] and Real Options Analysis [17].

Negotiation Prioritization: Quantitative WinWin [30] and Distributed Collaborative Prioritization Tool (DCPT) [7].

Others: Conjoint Analysis [31].

The prioritization methods given most coverage in the literature include AHP, QFD the Cost–Value Approach, and more recently, the Planning Game.

However, it is not clear to what extent all these methods are used by software development in industry, or indeed how successful they have been [32]. Indeed, there appear to be some problems with the take-up and continued use of the well-known prioritization methods, such as QFD [33] and AHP [21].

3.3 Prioritization Factors

There are many prioritization factors (also sometimes called "criteria" [34], [35] or "aspects" [20], [32]) that can be considered in the prioritization process. We have identified a list of over 50 prioritization factors from the literature; the main sources include [12], [13], [14], [21], [35], [36], [37]. See Table 1, in which we chose to subdivide the factors into three categories by stakeholder viewpoint and note the similarity to the choices of Lehtola [32] and Barney*, et al.* [35].

For brevity here, we have limited discussion of our work to just three stakeholder viewpoints that are representative of the mandatory viewpoints in any systems development prioritization process: strategy management, systems development and operations management. (Clearly there are many more stakeholder roles than these in a system.) We added a further sub-division under the four software engineering concepts used earlier in Figure 1. We decided that strategy management has responsibility for the objectives, systems development is primarily responsible for the requirements and designs, and operations management has responsibility for accepting the planned and delivered increments. In other words, the data associated with the selected system concepts would be of prime interest to the stakeholder viewpoint when establishing priorities. Furthermore, we introduced grouping of the prioritization factors by concept area, for example, strategy, cost and risk. Several of these groups are also identified by Berander [20] as "aspects". Note that, due to space limitations, any explanations of individual prioritization factors and relevant references have been omitted. Note also that these prioritization factors are not complete; this table only reflects the main prioritization factors found in the literature. The following observations can be made:

A general set of prioritization factors that could be proposed as "a starter" for a prioritization process emerges from the table. The prioritization factors span all the four software engineering concepts. This argues for a prioritization process that offers support for all these concepts. If more narrowly focused, specialized, prioritization methods are to exist then they need to integrate into an overarching prioritization process/method. The table provides support for the existence of different stakeholder viewpoints in the mappings between the stakeholder viewpoints and the prioritization factors: different stakeholder viewpoints are interested in and knowledgeable about different prioritization factors. This means any prioritization process or prioritization method must cater for different stakeholder viewpoints.

Table 1. Prioritization factors by stakeholder viewpoint and software engineering concept

STAKEHOLDERS	**Strategic management**	**Systems development**		**Operations management/ customers**
CONCEPTS	**Organizational objectives (objective)**	**Systems requirements (requirement)**	**Design solutions (design)**	**Delivery plans (increment/delivery)**
PRIORITIZATION FACTORS				
OPINION	Vision/intuition/gut feeling/preference/ bias	Preferences/ bias/ importance	Intuition/ preferences/ bias	Preferences/ bias
STRATEGY	Strategic alignment/ business objectives/ product strategy		Long-term Strategy for systems architecture	
	Competition	Quality		
	Customer demand	Originator of requirement		End user value
	New business potential			
TIME	Urgency/time to market/lead time		Time schedule/ time constraints	
	Long term versus short term		Long term versus short term	
LEGAL	Legal mandate/ regulations	Legal mandate/ regulations		
	Contracts in place			
FINANCIAL BENEFIT	Market value/price			
	Financial benefits			
	Financial penalties			
	Benefit/cost ratio			
	Cost of not implementing			
COST	Development costs/ implementation costs/ support costs		Development costs/ support costs	Implementation costs/ support costs
	Operational costs			Operational costs
FIT	Fit with operational context: . business processes . skills/training . delivery timing		Staff competence Balanced workload	Fit with operational context: . business processes . skills/training . delivery timing
			Resource availability/ effort constraints	Resource availability/ effort constraints
	Fit with other products		Change impact/ base code dependencies	Change impact
			Logical implementation order	
			Reuse potential	
EXTERNAL DEPENDENCY	Intermediary channels	External dependencies	External dependencies	
RISK	Business risk Sales barriers	Volatility of requirements	Technical risk in: . current system . proposed system . implementation process	
			Difficulty of implementation/ complexity	Difficulty of implementation/ complexity

A tentative observation can be made that the prioritization factor groupings (for example, strategy, legal, cost and risk), map across to the dimensions for stakeholder value.

3.4 Known Issues in Structuring Prioritization Data

A list of issues encountered when structuring the prioritization data to support the prioritization process was identified by extrapolating from discussions in the literature, for example from [8], [11], [12], [21], [36], [37], [38], [39]. The issues considered relevant to expressing prioritization data include:

Explicit Stakeholder Value: This is the often the expression of stakeholder priority to reflect the stakeholder value as well as the capture of explicit value.

Multiple Stakeholder Viewpoints: There is a need to handle different areas of interest/expertise and capture the different viewpoints together with their associated stakeholder value.

Requirements Abstraction: This is the ability to handle requirements captured at different levels of refinement.

Interdependencies: The ability to express interdependencies among the requirements and also the designs. This becomes increasingly important with iterative and incremental development.

Dynamic Prioritization: The priority data must be captured in order that it can be reused in subsequent prioritizations (future increments) without needing further inputs from stakeholders (unless something significant has changed in the system and/or its environment that they need to provide additional data on).

Scaling-up: This is the ability to scale up to cope with large numbers of items. Some existing prioritization methods become impractical when the number of requirements begins to grow to sizes typical of modern systems. In fact, for most large-scale projects, prioritization can tend to be carried out at a fairly high level of abstraction.

4 Analysis of Existing Prioritization Data

4.1 Expressing Prioritization Data

How prioritization data is expressed is a key factor in a prioritization process. We argue in this section that the prioritization data that the prioritization methods currently utilize (in content and structure) is insufficient to support the type of enhanced prioritization process that ideally needs to be adopted. Specifically, the lack of use of quantified data captured on absolute scale types is hindering progress.

The type of scale being used to capture the data is specifically important as it identifies the extent to which arithmetic calculations can validly be carried out. Only the ordinal and ratio scale types are commonly used in existing prioritization methods. The absolute scale type is only occasionally used at present, but we propose it should be much more widely used and in fact, that it should replace much of the use of the ordinal and ratio scale types.

Table 2. Mapping of prioritization technique(s) and scale type(s) to prioritization methods

Prioritization Method	**Prioritization Technique(s)**	**Scale Type(s)**
QFD	Weighting and Grouping	Ratio Ordinal
AHP	Weighting (Pair-wise comparison)	Ratio
IE	Metrics	Absolute
Cost-Value Approach	Weighting (Pair-wise comparison)	Ratio
MoSCoW	Grouping	Ordinal
Planning Game	Grouping	Ordinal
Requirements Triage	Grouping and Weighting	Ordinal Ratio

4.2 Prioritization Techniques

Several different ways (sometimes termed "prioritization techniques" [20]) of expressing prioritization data can be identified [13], [37]. We have reduced the number of different categories to four main ones as follows:

Grouping: The individual items are each categorized into one of a set of priority groups, for example the MoSCoW prioritization method demands each requirement is categorized as either "must have", "should have", "could have" or "would like, but wouldn't have this time" [19]. The results are on an ordinal scale.

Ranking: Requirements are ranked in order of preference. Ranking is carried out by bubble sort or by binary search tree [11]. This is an ordinal scale of measure as there is no information about the differentials amongst the ranked items.

Weighting: Stakeholders assign their preferences and relative weightings are calculated. The results are on a ratio scale. One means of obtaining the weightings is by using voting [13]: stakeholders are requested to distribute some fixed number of votes (say 100 or 1000 dollars) amongst the different items being prioritized. Another means is by using pair-wise comparison: priorities are calculated by creating a hierarchy with branches of up to seven comparable items and then the items within each branch are pair-wise compared using a scale of 1 to 9 where 1 equates to "equally important" and 9 equates to "extremely more important" [12]. The scales are then converted to normalized weightings, which are then carried up the hierarchy. In AHP, pair-wise comparison is used to first weight the requirements, and then the designs.

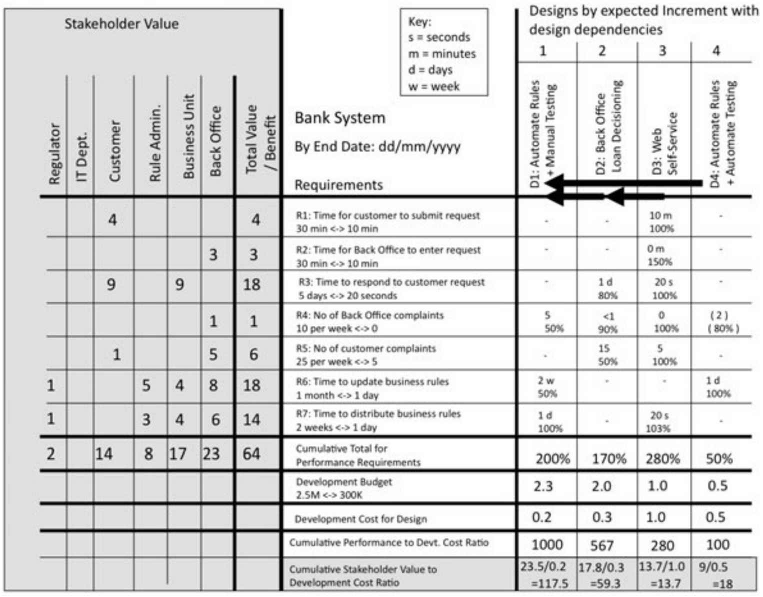
Metrics: Absolute scales of measure are used to express certain attributes and these metrics form the basis for selection, for example by enabling calculation of ROI figures [37]. ROI calculation needs data on the amount of benefit (stakeholder value) that would be achieved by implementing a given design and the implementation cost associated with it. Only absolute scale data enables such ROI estimates to be calculated, as explicit stakeholder value data such as "a cost saving over the next year of 220,000 monetary units" would be captured. This contrasts to the ordinal scale data of say, the MoSCoW method, which simply captures requirements identified as of high stakeholder value into a "must have" priority group. In this paper, we are using Planguage [40] to express metrics, which captures the performance and resource requirements, as required levels on scales of measure.

See Table 2, which gives some examples of how the scale types and prioritization techniques map to a selection of prioritization methods.

See also Table 3, which shows how the prioritization techniques cope with a selection of prioritization data issues. Some example data has been inserted in the top row. From this row, it can be seen that use of metrics with absolute scale types results in real data that is much easier to understand and say, discuss with another stakeholder. It is less ambiguous than trying to work out what "Medium" should be interpreted to mean. An observation can be made that all the techniques, apart from metrics, are generating additional data that captures some indirect notion of stakeholder value (such as "must have"), but not any explicit value (such as 220,000 monetary units).

Table 3. How prioritization techniques cope with a selection of data structuring issues

Prioritization Technique >	**Grouping**	**Ranking**	**Weighting**	**Metrics**
Example of prioritization data	"High", "Medium" or "Low"	1, 2, 3, ... N	30/100	Time to carry out task to be reduced from 1 day to 5 minutes
Data Structuring Issue				
Stakeholder value	Implicit; value is say, "Medium"	Implicit; value is say, ranked as "2"	Implicit; value is 30% of whatever 100% equates to	Depends on metric. For this metric, an estimate of value is able to be derived if say, monetary rate of pay is known.
Multiple stakeholder viewpoints (Note assuming 2 stakeholders)	N Would be represented as say, "High" and "Medium"	N Would be represented as say, "2" and "20"	N Would be represented as say, 30/100 and 2/100	Y Time to carry out task to be reduced from 1 day down to say, 5 minutes and to 2 hours
Requirements Abstraction	N	N	Y Create hierarchy	Y Create hierarchy
Interdependencies	(Y) Would have to work by selecting an item and then seeing if there were any prior dependencies that would override	(Y) Ditto	(Y) Ditto	(Y) Ditto
Dynamic prioritization	Y Add any new data to an existing data grouping. No extra effort (unless something has changed)	Y Would need to re-examine existing ranks	(Y) Considerable effort needed by stakeholders	Y Add to existing data and reprocess
Scaling up	N Too many in a group	N Difficult to keep track of numerous rankings	N Considerable effort to carry out all the additional pair-wise comparisons	(Y) Would use high-level hierarchical data to reduce numbers



Key:
s = seconds
m = minutes
d = days
w = week

Stakeholder Value							Bank System By End Date: dd/mm/yyyy	Designs by expected Increment with design dependencies			
								1	2	3	4
Regulator	IT Dept.	Customer	Rule Admin.	Business Unit	Back Office	Total Value / Benefit	Requirements	D1: Automate Rules + Manual Testing	D2: Back Office Loan Decisioning	D3: Web Self-Service	D4: Automate Rules + Automate Testing
		4				4	R1: Time for customer to submit request 30 min <-> 10 min	-	-	10 m 100%	-
					3	3	R2: Time for Back Office to enter request 30 min <-> 10 min	-	-	0 m 150%	-
		9		9		18	R3: Time to respond to customer request 5 days <-> 20 seconds	-	1 d 80%	20 s 100%	-
					1	1	R4: No of Back Office complaints 10 per week <-> 0	5 50%	<1 90%	0 100%	(2) (80%)
		1			5	6	R5: No of customer complaints 25 per week <-> 5	-	15 50%	5 100%	-
1			5	4	8	18	R6: Time to update business rules 1 month <-> 1 day	2 w 50%	-	-	1 d 100%
1			3	4	6	14	R7: Time to distribute business rules 2 weeks <-> 1 day	1 d 100%	-	20 s 103%	-
2		14	8	17	23	64	Cumulative Total for Performance Requirements	200%	170%	280%	50%
							Development Budget 2.5M <-> 300K	2.3	2.0	1.0	0.5
							Development Cost for Design	0.2	0.3	1.0	0.5
							Cumulative Performance to Devt. Cost Ratio	1000	567	280	100
							Cumulative Stakeholder Value to Development Cost Ratio	23.5/0.2 =117.5	17.8/0.3 =59.3	13.7/1.0 =13.7	9/0.5 =18

Fig. 2. VIE table for bank case study. The shaded area represents the extensions to IE.

5 Some Examples from a Case Study

5.1 Choice of Prioritization Method

By comparing how prioritization methods handled the prioritization factors and the data structure issues [41], and by considering the usage of software engineering concepts and scale types, we determined that IE offered an initial sound basis for this research: it spans the concepts of requirement, design and increment, and uses absolute scale data [1]. However, the IE method lacks consideration of explicit stakeholder value and stakeholder viewpoint, so we extended it to cater for stakeholder value by stakeholder role. We chose to link stakeholder value to requirement. See Figure 2 for an example of an extended IE table, which we term a value impact estimation (VIE) table. The non-shaded area is a basic IE table and the shaded area represents the extensions to IE.

5.2 Case Study Description

The case study examples are from a customer business rules "decisioning" system for a bank. The bank's objectives are customer satisfaction and, more efficient and effective internal processes. The main problems perceived by the bank are the time, effort and accuracy of updating and using the business rules, and the elapse time taken and the accuracy of dealing with customer requests. Of course, having up-to-date business rules in place impacts the accuracy of the handling of the customer requests. As the intention is to demonstrate that absolute scale data helps prioritization reasoning, a detailed discussion of all the requirements is not given here. We also limit our comments here about the use of performance requirements (also known as non-functional requirements) apart from recognizing that this is an additional reason why IE merits attention (given very few prioritization methods handle performance requirements [37]).

For brevity, a very restricted, cut down sample of the system specification is presented below. Note the data highlighted in bold in this specification is captured in the VIE table in Figure 2.

Stakeholders: Regulator, IT Department, Customer, Rules Administration, Business Units, Back Office.

Requirements:
Function: Submit request.
Performance requirement: **Reduce time for customer to submit request**.
Scale: Average time taken for defined [request type: Default = Loan].
Past: **30 minutes**.
Goal: **10 minutes**.

Function: Enter customer request details.
Performance requirement: **Reduce time for Back Office to enter request**.
Past: **30 minutes**.
Goal: **10 minutes**.

Function: **Process a customer request.**
Performance requirement: **Reduce time to process customer request**.
Past: **5 days**.
Goal: **20 seconds**.
Performance requirement: Reduce number of complaints.
Scale: Average number of complaints in defined [Time] from defined [Stakeholder].
Past **[Back Office]: 10 per week.**
Goal: **0 per week.**
Past **[Customer]: 25 per week**.
Goal: **5 per week**.

Function: **Update the business rules**.
Performance requirement: Reduce time to update rules.
Scale: Average time taken for defined [request type].
Past: **1 month**.
Goal: **1 day.**

Function: **Distribute business rules**.
Performance requirement: Reduce time taken. Scale: Average time taken.
Past: **2 weeks**.
Goal: **1 day**.

Designs:
APTM: **Automate the rules & test manually**.
Rationale: Speed up the distribution to Back Office staff.

BD: **Back Office loan decisioning system**.
Rationale: Automating applying the rules will save time.
Dependency: **APTM**.

WSS: **Web self-service.**
Rationale: Customers can get a rapid response.
Dependency: **BD, APTM**.

```
APAT: Automate the rules & test automatically.
Rationale: Speed up the distribution to Back Office staff.
Dependency: APTM.
```

5.3 Description of a Basic IE Table

To create a basic IE table, the performance requirements are placed down the left-hand column. Each performance requirement shows its current baseline (Past) level and the required target (Goal) level. Beneath the performance requirements, the resource requirements are listed. Next, the designs are placed on the top row, and the estimated impact of each of the designs on each of the performance and resource impacts can be filled in as a level on the scale of measure. As discussed earlier, this involves estimating the level on the scale of measure that will result for a requirement if the design is implemented. If the baseline level is taken as 0% and the target level is taken as 100%, then the estimated percentage impact of the design on achieving the requirement can be calculated and the percentage change can be determined.

By looking down a column for a given design, you can see which requirements it is contributing towards meeting. By summing the estimated percentage changes in the performance requirements down a column for a given design, and then dividing by the estimated development cost of the design, an estimated cumulative performance to cost ratio for each design can be calculated. The performance to cost ratios for the potential designs within an IE table can be compared to determine which design offers the most impact given its cost. In this case study, the designs are complementary so the aim is to sequence the implementation order to deliver the highest value as early as possible. Looking at Figure 2, it can be seen that the designs are in the right order regarding the figures for the estimated cumulative performance to (development) cost ratios. Note the totals for the performance impacts are calculated based on the estimated *additional* percentage impact over the estimated percentage impact achieved after the last design was implemented. So design BD contributes an additional 40% (90% - 50%) over the 50% estimated for APTM, so its total estimated percentage impact is 80% + 40% + 50% = 170%. A further refinement was to cap any percentage impact at 100% for the calculations. As implementation proceeds and increments are completed, the actual results can be measured and captured in the IE table alongside the estimates. This allows deviations from the planned levels to be identified and future plans adjusted as appropriate.

5.4 Extending IE to Cater for Multiple Stakeholders and Stakeholder Value

To address the problems with multiple stakeholders and stakeholder value, as already outlined earlier, we extended the basic IE table to capture on its left-hand side the different stakeholders of interest and their associated stakeholder value, see the shaded areas of Figure 2. The figures for stakeholder value given in the stakeholder columns represent the stakeholder value of achieving 100% of each requirement. In this VIE table stakeholder value was estimated on the financial value of the estimated time saving and the estimated additional sales. Note the actual financial values are not given here due to the commercial sensitivity of this information. Instead the financial figures for stakeholder value were all divided by the lowest figure and then rounded to the nearest integer.

In turn, the extension of adding explicit stakeholder value enables cumulative stakeholder value to development cost ratios to be calculated for the different designs as shown in the shaded bottom row of Figure 2. The calculation is worked out on the basis that an estimated percentage impact of a design on a requirement will result in the same percentage of the stakeholder value of the requirement being achieved. In other words, an assumption that the utility curve [42] is linear. From the results for the value to cost ratios, it appears that maybe the implementation order of the designs, WSS and APAT should be reversed.

6 Conclusions

Despite much research in the last decade on prioritization in software engineering projects, progress is being hampered by inadequate representation of stakeholder value. The issue is becoming more urgent because dynamic prioritization of stakeholder value is increasingly needed as iterative and incremental development methods become more widely used. We have argued in this paper that the use of absolute scale data is essential to address the problems with the current prioritization processes: specifically, to provide unambiguous prioritization data that stakeholders can understand and relate to, and to support arithmetic calculations.

This paper has briefly reviewed existing research on prioritization. Our findings include:

- By categorizing and analysing the existing prioritization methods, that many (but not all of) the existing prioritization methods are restricted in their scope (for example, some methods are just considering the requirements).
- By investigation of the prioritization factors discussed in the literature, we have shown that the scope of the prioritization process spans system-wide data from organizational objectives to increment delivery. An additional finding from this data is that different stakeholders have different viewpoints on the prioritization factors, and that therefore, multiple stakeholder viewpoints need to be supported.
- By identifying the known issues with structuring prioritization data and analyzing how the prioritization techniques and scale types used in prioritization methods tackle these issues, we determine that the techniques of grouping, ranking and weighting are weaker than metrics in addressing the issues. Specifically, expression of stakeholder value is implicit in the prioritization data, and that arithmetic calculations are often impossible or problematic, apart from when metrics are used.

Further, we demonstrate the validity of the use of absolute scale data in prioritization by using VIE, an extended version of the IE prioritization method with some examples from a case study. We specifically extended IE to cater for stakeholder value for multiple stakeholders. We show the ability to carry out calculations to investigate requirement and design priorities.

Work is underway to investigate further extending the IE method to represent additional aspects of stakeholder value. Future work plans to make the detailed decision-making of a rational prioritization process explicit.

References

1. Gilb, T.: Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage. In: Brodie, L. (ed.). Butterworth-Heinemann, London (2005) ISBN 0750665076
2. Boehm, B., Port, D., Basili, V.R.: Realizing the Benefits of the CMMI with the CeBASE Method. Systems Engineering (2002)
3. Bourque, P., Dupuis, R. (eds.): SWEBOK: Guide to the Software Engineering Body of Knowledge 2004 Version. IEEE Computer Society, Los Alamitos (2004)
4. Boehm, B.: Value-Based Software Engineering. Software Engineering Notes 28(2) (2003)
5. Sullivan, K.: Introduction to the First Workshop on the Economics of Software and Computation. In: Companion to the Procs. of the 29th International Conf. on Software Engineering. IEEE, Los Alamitos (2007)
6. Larman, C., Basili, V.R.: Iterative and Incremental Development: a Brief History. IEEE Computer 36(6) (2003)
7. Park, J., Port, D., Boehm, B.: Supporting Distributed Collaborative Prioritization for Win-Win Requirements Capture and Negotiation. In: Procs. of the International Third World Multi-Conference on Systemics, Cybernetics and Informatics (SCI 1999). International Institute of Informatics and Systemics (1999)
8. Davis, A.: The Art of Requirements Triage. IEEE Computer 36(3), 42–49 (2003)
9. Karlsson, J., Ryan, K.: A Cost-Value Approach for Prioritizing Requirements. IEEE Software (1997)
10. Saaty, T.L.: How to Make a Decision: The Analytic Hierarchy Process. European Journal of Operational Research 48, 9–26 (1990)
11. Karlsson, J., Wohlin, C., Regnell, B.: An Evaluation of Methods for Prioritizing Software Requirements. Information and Software Technology (1998)
12. Moisiadis, F.: The Fundamentals of Prioritising Requirements. In: Procs. of the Systems Engineering, Test and Evaluation Conference (2002)
13. Berander, P., Andrews, A.: Requirements Prioritization (ch. 4). In: Aurum, A., Wohlin, C. (eds.) Engineering and Managing Software Requirements. Springer, Heidelberg (2005) ISBN 3540250433
14. Greer, D., Ruhe, G.: Software release planning: an evolutionary and iterative approach. Information and Software Technology 46(4), 243–253 (2004)
15. Kazman, R., Asundi, J., Klein, M.: Quantifying the Costs and Benefits of Architectural Decisions. In: Procs. of the 23rd International Conference on Software Engineering (ICSE 2001). IEEE, Los Alamitos (2001)
16. Mohamed, A., Ruhe, G., Eberlein, A.: COTS Selection: Past, Present, and Future. In: Procs. of the IEEE Intl. Conf. and Workshops on the Engineering of Computer-Based Systems (ECBS 2007). IEEE, Los Alamitos (2007)
17. Favaro, J.: Managing Requirements for Business Value. IEEE Software (March/April 2002)
18. Sivzattian, S.V.: Requirements as Economic Artifacts: A Portfolio-Based Approach, Ph.D. Thesis. Department of Computing, Imperial College of Science, Technology and Medicine, London (2003)
19. Stapleton, J. (ed.): DSDM: Business Focused Development, 2nd edn. Addison Wesley, Reading (2003)
20. Berander, P.: Evolving Prioritization for Software Product Management. Blekinge Institute of Technology. Doctoral Dissertation Series (2007)

21. Lehtola, L., Kauppinen, M.: Suitability of Requirements Prioritization Methods for Market-driven Software Product Development. In: Software Process Improvement and Practice. Wiley, Chichester (2006)
22. Cohen, L.: Quality Function Deployment: How to Make QFD Work for You. Addison Wesley, Reading (1995)
23. Akao, Y.: QFD: Past, Present, and Future. In: Procs. of the International Symposium on QFD 1997, Linkoping (1997)
24. Bass, L., Ivers, J., Klein, M., Merson, P.: Reasoning Frameworks (CMU/SEI-2005-TR-007). Software Engineering Institute, CMU (2005)
25. Mohamed, A., Ruhe, G., Eberlein, A.: Decision Support for Handling Mismatches between COTS Products and System Requirements. In: Procs. of the COTS-Based Software Systems (ICCBSS 2007) Conf. (2007)
26. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley, Reading (2000)
27. Saliu, O., Ruhe, G.: Supporting Software Release Planning Decisions for Evolving Systems. In: Procs. of the 2005 29th Annual IEEE/NASA Software Engineering Workshop (SEW 2005). IEEE, Los Alamitos (2005)
28. Favaro, J.: Value-Based Management and Agile Methods. In: Marchesi, M., Succi, G. (eds.) XP 2003. LNCS, vol. 2675. Springer, Heidelberg (2003)
29. Denne, M., Cleland-Huang, J.: Software by Numbers: Low-Risk, High-Return Development, 190 pages. Prentice-Hall, Englewood Cliffs (2004) ISBN 0131407287
30. Ruhe, G., Eberlein, A., Pfahl, D.: Trade-off Analysis for Requirements Selection. International Journal of Software Engineering and Knowledge Engineering 13(4), 345–366 (2003)
31. Green, P.E., Wind, Y.: New Way to Measure Consumers Judgments. Harvard Business Review (1975)
32. Lehtola, L.: Providing value by prioritizing requirements throughout software product development: State of practice and suitability of prioritization methods. Licentiate thesis, Helsinki University of Technology (2006)
33. Martins, A., Aspinwall, E.: Quality Function Deployment: an empirical study in the UK. Total Quality Management (August 2001)
34. Wohlin, C., Aurum, A.: Criteria for Selecting Software Requirements to Create Product Value: An Industrial Empirical Study. In: Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Grunbacher, P. (eds.) Value-Based Software Engineering. Springer, Heidelberg (2005)
35. Barney, S., Aurum, A., Wohlin, C.: A product management challenge: Creating software product value through requirements selection. Journal of Systems Architecture (2008)
36. Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., Natt och Dag, J.: An Industrial Survey of Requirements Interdependencies in Software Product Release Planning. In: Procs. of the 5th IEEE International Symposium on RE (2001)
37. Firesmith, D.: Prioritizing Requirements. Journal of Object Technology (2004)
38. Gorschek, T., Wohlin, C.: Requirements Abstraction Model. Requirements Engineering 11, 79–101 (2006)
39. Mead, N.: Requirements Prioritization Introduction. Software Engineering Institute (SEI), Carnegie Mellon University (CMU) (2006)
40. Gilb, T.: Principles of Software Engineering Management. Addison Wesley, Reading (1988) ISBN 0201192462
41. Brodie, L., Woodman, M.: Towards a Rational Prioritization Process for Incremental and Iterative Systems Engineering. In: Procs. of the 1st International Workshop on Requirements Analysis. Pearson, London (2008)
42. Daniels, J., Werner, P.W., Bahill, A.T.: Quantitative Methods for Tradeoff Analyses. Systems Engineering 4(3) (2001)

ProMISE: A Process Metamodelling Method for Information Systems Engineering

Charlotte Hug[1], Agnès Front[2], and Dominique Rieu[2]

[1] Centre de Recherche en Informatique,
Université Paris 1 Panthéon-Sorbonne
[2] Laboratoire d'Informatique de Grenoble, Grenoble University
220 rue de la Chimie, 38041 Grenoble cedex 9, France
Charlotte.Hug@univ-paris1.fr
{Agnes.Front,Dominique.Rieu}@imag.fr

Abstract. Processes play a great part in information systems engineering projects success. There are a lot of process models and metamodels; however, the "one size fits all" motto has to be moderated: models have to be adapted to the specificities of the organizations or the projects. In order to help method engineers building adapted process models, we propose a method to build process metamodels and to instantiate them according to the organizations context. Our method consists of selecting the concepts needed from a conceptual graph, gathering the current knowledge of metamodelling concepts for information systems engineering processes, and integrating them in a new process metamodel that will be instantiated for any project in an organization. This method is supported by a tool.

Keywords: Process engineering, Information systems engineering, Metamodelling, Graph, Tool.

1 Introduction

To design and produce information systems, project managers focus on the quality of the deliverables or on the intermediary support documents produced all along the project life (analysis models, test procedures, for example); as such, they focus on the quality of their definition, formalization, level of detail and completeness. The quality of the products highly depends on the processes followed [1], as the processes define the way products have to be created. A development process can be roughly defined as a sequence of activities that create and update products. The objective for an organization is to properly define the processes, formalize them, adjust them to the different projects and reproduce the optimized processes. The Capability Maturity Model Integration [2] specifies different degrees of maturity of the development processes in an organization, the supreme goal being following repeatable and optimized processes. The information systems engineering (ISE) processes quality is then essential.

Many information systems/software engineering processes or methods have been defined. They appeared in the 1970's with the Waterfall model [3], the Spiral Model

[4], then the RUP [5] and more recently Agile methods as XP [6] and SCRUM [7]. They are based on different process models: they propose different lifecycles and activities, specify distinct kinds of deliverables and assign roles differently. Thus, each method proposes its own way to build IS: each method is based on a different process metamodel that uses different concepts.

In order to produce information systems, process models have to be efficient and fitted to the organizations specific constraints. An unsuitable method or process model will not be followed by the development teams, create tensions between team members and generate delay or bad IS design. Existing methods or process models have then to be adapted, customized to the organizations context; this is the method engineer's role.

As the process models flexibility depends on their process metamodel flexibility, we state that the key to build adapted process models lies in adapted process metamodels. However, existing process metamodels are hardly adaptable and are defined independently of one another [8], [9], [10]. Upon modelling the process models of their organizations, method engineers have to use those already predefined process models or to instantiate process metamodels without adaptation possibilities; the resulting models might be partially inadequate to the organizations specificities and constraints and to their business activities.

In this paper, we present the ProMISE method (Process Metamodelling for Information Systems Engineering) that allows method engineers building their own process metamodels according to their organization specificities and technologies. The method consists of selecting the needed concepts from a conceptual graph and integrating them in a new adapted process metamodel. The construction of the process metamodel is hidden to the method engineers: they use a conceptual graph that builds the process metamodel and checks its consistency. The produce process metamodels are multi-points of view as they integrate various points of view of the existing process metamodels, they are adapted to the constraints and specificities of the organization as only the needed concepts are integrated and the process metamodel is federated as all the knowledge of ISE processes is defined in one metamodel.

The paper is organized as follows. In the next section, we present the conceptual graph, base of our adaptive method to build process metamodels for ISE. We introduce the method in Section 3. Section 4 presents an example of the Grenoble's University Hospital. Section 5 is devoted to discussion and Section 6 presents the tool that supports our method. Section 7 concludes this paper.

2 The Base of the Method: The Conceptual Graph

In this section, we present the base of our approach that is a conceptual graph. It was built from a Process Domain Metamodel and a 3D Space [8], [9], [10]. A study [10], [11] of the different existing process metamodels (activity oriented [12]; [13]; [14]; [15]; [16]) such as SPEM, product oriented [17]; [18];[19]; [1] such as Statechart and State Machines, decision oriented [20]; [21]; [22]; [23] like Ibis and Daida, context oriented [24] such as NATURE and strategy oriented [25] like MAP), allowed us to define a Process Domain Metamodel which only contains the main classes of existing process metamodels and the associations between the concepts. In order to facilitate the classes' selection from the Process Domain Metamodel, we propose the use of a conceptual graph that allows method engineers to easily navigate between the concepts. The concepts are organized according to a 3D space.

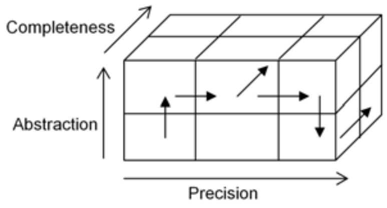


Fig. 1. The Completeness – Precision – Abstraction 3D space

2.1 The 3D Space

The 3D space represented in Figure 1 guides method engineers through a methodological frame to build process metamodels for ISE. The three axes [26] help method engineers in the selection of the concepts: completeness, precision and abstraction. Completeness is the coverage of the metamodel of one or more points of view (activity, product, decision, context and strategy). Precision is the level of detail of the metamodel and abstraction is the intentional and/or operational level of concern of the metamodel. The intentional level represents the objectives of the ISE process while the operational level represents the actions required to concretize these objectives. Method engineers will build their process metamodels depending on these three axes: each engineering activity has for objective to: extend the Process Metamodel Under Construction (PMUC) (completeness axis), precise the PMUC (precision axis) or abstract (inv. concretize) the PMUC (abstraction axis).

2.2 The Conceptual Graph

The conceptual graph (Figure 2) is the base of our method. It organizes the recognized concepts for ISE process metamodelling, representing the actual knowledge base of the domain. The purpose of such conceptual graph is to guide method engineers in the Completeness – Precision – Abstraction 3D space while selecting the concepts they need to represent in their metamodels. The conceptual graph defines the set of possibilities: it restrains method engineers in the selection and the use of the defined concepts only, in order to maintain the consistency of the PMUC.

2.2.1 The Concepts

The concepts of the conceptual graph are used in ISE processes and are usually represented in process metamodels. The concepts of the graph represent two types of elements:

- Classes that represent the main concepts (concepts in bold in Figure 2) defined in the Process Domain Metamodel and are linked to each other by the completeness and abstraction relations. Those concepts are Work Unit, Condition and Role (activity point of view) [12]; [13]; [14]; [15]; [16], Work Product (product point of view) [17];

[18]; [19];[1], Issue, Alternative, Argument (decision point of view) [20]; [21]; [22]; [23], Situation, Context, Intention (context point of view) [24] and Strategy (strategy point of view) [25]. Figure 3 presents a close-up on a few of those. A Work Unit represents an action that is executed during the ISE process. A Work Product is something that is produced, used or modified during the ISE process and a Role is someone/thing that carries out an action during the ISE process. A Strategy represents how an intention is achieved.

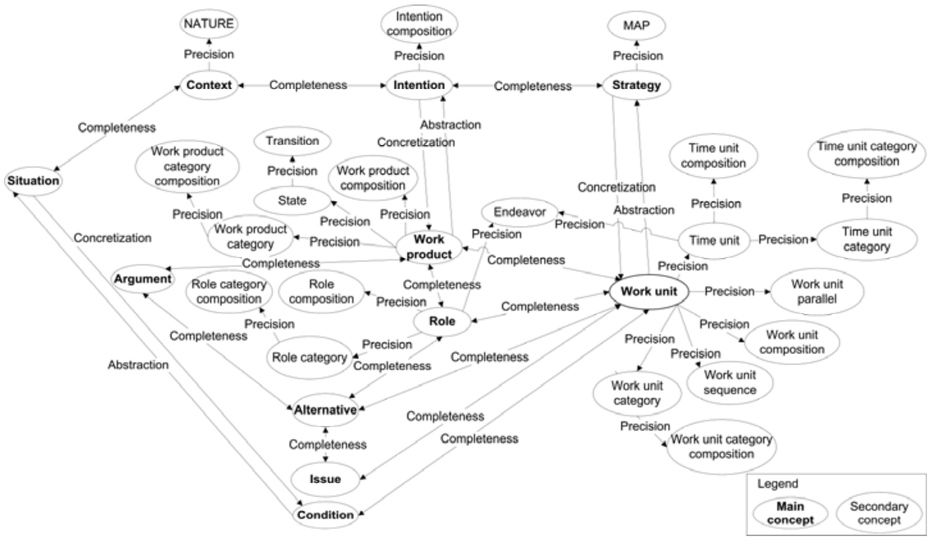


Fig. 2. The conceptual graph

- Classes that decompose the previous classes, linked by the precision relation (secondary concepts). For example, in Figure 3, the Work Unit Category concept refines the Work Unit concept to express the fact that there are different categories of work unit, as activity or task for example. The Work Unit Composition concept refines the Work Unit concept to represent a Work Unit class with a reflexive composition, to express that the "Design components" activity is composed of the tasks "Class design" and "Subsystem design" [5], for example.

2.2.2 The Relations

The relations represent conceptual links between concepts in the Completeness – Precision – Abstraction 3D space as presented in section 2.1.

The completeness relation links one concept to another that extends it. This relation is symmetric, non-transitive and non-reflexive. For example, in Figure 3 (on the left), the Work Unit concept can be completed by the Work Product and Role concepts. As the Work Product concept can also be completed by the Work Unit concept (symmetry), the represented link is bidirectional.

The precision relation specifies that a concept can be refined by another concept. Such relation is non-symmetric, non-reflexive and non-transitive. For example, the

Work Unit concept can be refined using the Work Unit Category or Work Unit Composition concepts (but the Work Unit concept does not refine the Work Unit Category concept – non symmetry) (cf. Figure 3 in the centre).

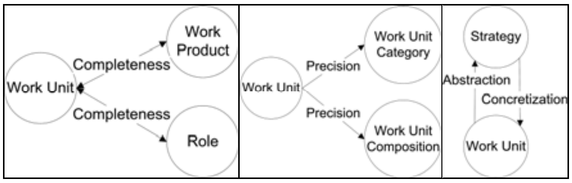


Fig. 3. Examples of the Completeness, Precision and Abstraction relations

The abstraction relation specifies that one concept can be abstracted by another concept; it is non-symmetric, non-reflexive and non-transitive. For example, the Work Unit concept is abstracted by the Strategy concept (cf. Figure 3 on the right). The inverse relation of Abstraction is Concretization. We can say that the Work Unit concept is the concretization of the Strategy concept.

On the one hand, the relations help method engineers selecting the concepts in the conceptual graph and on the other hand, they assure the coherency of the selected concepts. For example, the Work Unit Category Composition concept can not be selected before the Work Unit Category concept (Figure 2). The consistency of the process metamodels produced is then ensured, as the conceptual graph was designed in such a way as the concepts were coherently linked to each others.

2.2.3 Example

The conceptual graph in the Completeness – Precision – Abstraction 3D space is dynamically built: the perspective evolves depending on the node the method engineer is considering. Figure 4 shows a part of the 3D perspective that method engineers would see from the Work Unit concept.

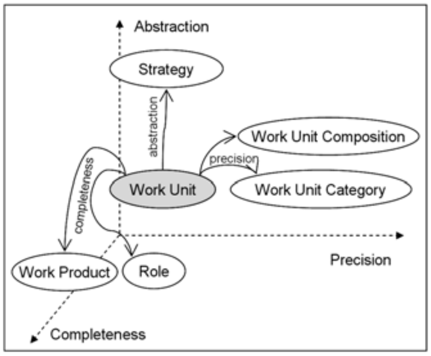


Fig. 4. Part of the perspective from the Work Unit concept in the conceptual graph

If method engineers want to extend their PMUC, it will lead to the Work Product and Role concepts thanks to the completeness relation defined in the conceptual graph. If they want to precise their PMUC, it will lead to the Work Unit Category and Work Unit Composition concepts, using the precision relation and if they want to abstract it, it will lead to the Strategy concept thanks to the abstraction relation.

We now describe the method that uses the conceptual graph to build process metamodels for ISE.

3 The Method

In this section, we present the method based on the conceptual graph to build process metamodels for ISE. The two-step method consists of: (i) concepts selection within the conceptual graph, (ii) concepts integration in the PMUC, according to the Process Domain Metamodel. These two steps are iterated until method engineers obtain the complete process metamodel they need.

3.1 Concept Selection

The first action of the Concept selection activity is the *Definition selection* that will lead to get a *Concept* (left part of Figure 5). A definition is composed of a short description, synonyms of the concept and examples (Table 1). It enables method engineers to select definitions from the *Concepts dictionary* corresponding to their needs. Each definition is associated to a concept appearing as a node in the conceptual graph. The next step is the Concept integration (section 3.2).

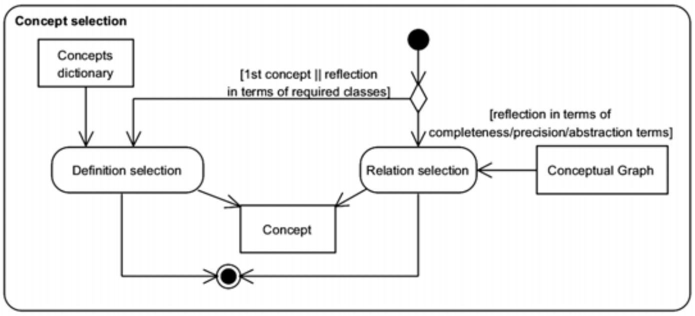

Concept selection
Concepts dictionary
[1st concept || reflection in terms of required classes]
[reflection in terms of completeness/precision/abstraction terms]
Definition selection
Relation selection
Conceptual Graph
Concept

Fig. 5. The Concept selection

After the first loop, method engineers go back to the *Concept selection*. They may refine the PMUC in terms of concepts attainable through relations with the previously integrated concept (completeness, precision and abstraction relations) or in terms of integration of classes thanks to the definitions. The *Relation selection* activity consists of selecting one of the relations that starts from the integrated concepts. For example, if the method engineer just integrated the Work Unit concept to his/her PMUC and if he/she wants to extend it, he/she could select Role, Work product and all the concepts linked through the completeness relation to the Work Unit concept in the conceptual graph. It works in the same way through the precision and abstraction relations.

Table 1. Some definitions examples

Description	Synonyms, AKA, examples	Concept
Represents how an intention is achieved	Tactics, approach, manner	Strategy
Objective of the ISE process	Goal	Intention
Task that is executed during the ISE process	Activity, task, work definition	Work Unit
Work Unit that is composed of other work units	Activity composed of tasks	Work Unit composition
Something that is produced, used or modified by a work unit during the ISE process	Product, document, model, program	Work Product
Someone/thing that carries out a work unit during the ISE process	Actor, developer, analyst, system	Role

3.2 Concept Integration

Once the concept is selected, it has to be integrated in the *PMUC*. The integration activity is rather complex (Figure 6): it has to take into account the different types of concepts (main or secondary).

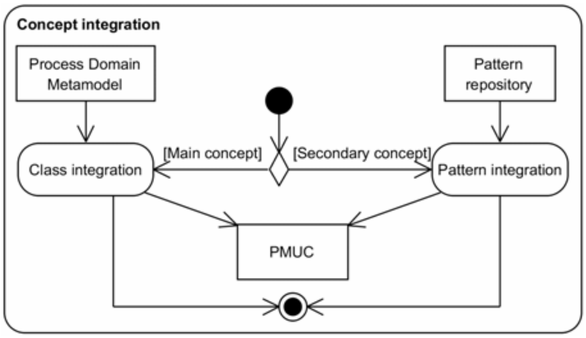


Fig. 6. The Concept integration

The main concepts of the Conceptual Graph correspond to classes in the Process Domain Metamodel. These classes have then to be integrated in the PMUC with the associations between the integrated classes. The secondary concepts correspond to design or business patterns that are applied on the classes of the PMUC. The patterns that can be used are stored in a *Pattern Repository*. According to the selected concept, one of the patterns is applied on the PMUC. The PMUC is thus built by adding classes and applying patterns. The integration process is fully described in [27].

Method engineers can then choose either to continue the process or to stop it if the PMUC is complete. If the PMUC is not complete, they go back to the *Concept selection* activity.

The ProMISE method allows method engineers to build process metamodels according to the constraints and specificities of their organization as they only select the needed concepts from the conceptual graph. The conceptual graph allows guiding method engineers in the construction and checking the consistency of their PMUC. The guiding is done thanks to the relations defined between the concepts that method engineers will select according to their intention (abstract, complete, precise a concept). The consistency of the produced PMUC is continuously checked as method engineers can only select concepts according to the conceptual graph which have been built in order to verify the consistency at any time. Some concepts cannot be selected until other concepts have been integrated. Moreover, the construction of the process metamodel itself is hidden to the method engineers as they only manipulate the conceptual graph and the concepts definition.

We will now present an example of the ProMISE method use.

4 Grenoble's University Hospital Example

This section describes an example of the information system centre of Grenoble's University Hospital (http://www.chu-grenoble.fr/). This example has not a purpose of validating our method but illustrating it. We specifically conducted qualitative evaluations to validate the method with an academic focus group and semi-structured interviews with industrialists [28].

4.1 Requirements

The information system centre (ISC) manages approximately forty different applications that need to be regularly updated to meet new users' requirements (medical assistants, hospital doctors and administration staff).

The ISC managers want to model the ISE processes to achieve a more rigorous project management, defining a unified and optimal way to manage projects regardless of the development team. They also want to collect and reuse knowledge for a more efficient production in terms of resources and time use and therefore costs. A method engineer is in charge of the study of the ISE processes and their modeling. The method engineer in this example is one of the project managers of the ISC.

We have worked with this project manager who determined the various aspects of the ISE processes (this example only presents an extract of the problem):

- A part of the process is defined in terms of goals and sub-goals; this part is intended primarily for hospital services managers (services are for example the surgical unit or the accounting department) who are more interested in the results and impacts of new system functionalities on their service (intentional part),
- The second part of the process is defined by phases, activities and products produced during these activities (operational part).

The problems met by the method engineer are the following: how can he represent these concepts? What are the existing models? Which models meet these requirements? At the present time, these representation choices are made difficult because of the numerous existing process models and metamodels, their lack of mutual complementarity and the complexity to adapt them to specific needs of organizations.

Our method enables the method engineer to model the process metamodel that corresponds to the information system centre ISE processes. The method guides him through the selection of concepts he needs to represent and through their assembly in order to create a specific process metamodel including all the concepts at the intentional level concerning the services managers and at the operational level concerning the activities and the products.

4.2 Method Use

The first step of our method is the Concept selection. The method engineer must select one of the definitions that correspond to the concepts he wants to model. The definition "Goal or objective of the ISE process" corresponds to the part of the process defined in terms of goals. The engineer chooses this definition and the corresponding Intention class from the Process Domain Metamodel is integrated in the new PMUC. The method engineer examines then the relations of the Intention concept in the conceptual graph; the *precision* relation permits him to select the Intention Composition concept that will allow him to decompose the goals into sub-goals. This concept is integrated in the PMUC as a reflexive composition on the Intention class, which corresponds to the use of the Composition pattern on the Intention class. Figure 7 presents this part of the path in the conceptual graph and the corresponding PMUC.

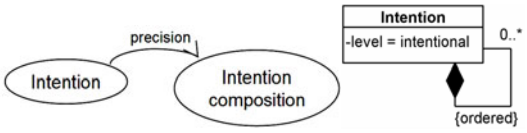


Fig. 7. First part of the path in the conceptual graph and the PMUC

Then, the relation *concretization* starting from the Intention concept in the conceptual graph allows the method engineer to get the Work Product concept that will represent the products produced during the ISE process. The corresponding class is integrated in the PMUC, as well as the "concretizes" dependency linked to the Intention class. In order to model the fact that a work product can be composed of other work products (for example, "Functional specifications" is composed of "Simplified requirements" and "Actors diagram"), the method engineer refines the Work Product concept thanks to the Work Product Composition concept. To specify that work products are of different types (for example, "Functional specifications" is a document and "Actor diagram" is a UML diagram), the method engineer refines the Work Product concept by the Work Product Category concept. The Work Product Category class is added into the PMUC. Similarly to what was done with the Work Product, the method engineer wants to specify that a document is composed of UML diagrams, texts and graphics. He refines the Work Product Category concept by the Work Product Category Composition concept. Figure 8 presents the corresponding part of the path in the conceptual graph and the corresponding PMUC.

Thanks to the *completeness* relation, the method engineer can extend the PMUC with the Work Unit concept to represent activities and steps. The Work Unit class and its associations "In" and "Out" defined in the Process Domain Metamodel are integrated to the PMUC. By using the *precision* relation, the method engineer can refine the Work Unit concept to represent the sequence and the composition of work units, the work unit categories and the composition of work unit categories. Figure 9 presents the complete path carried out in the conceptual graph.

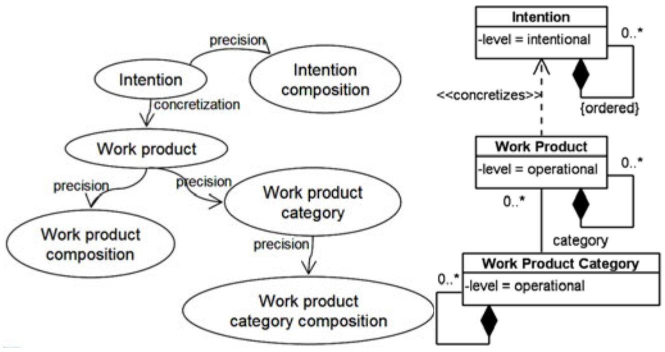


Fig. 8. Second part of the path in the conceptual graph and the PMUC

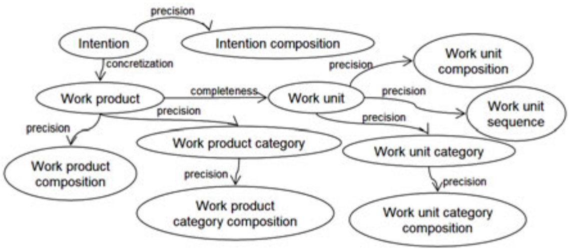


Fig. 9. Complete path in the conceptual graph

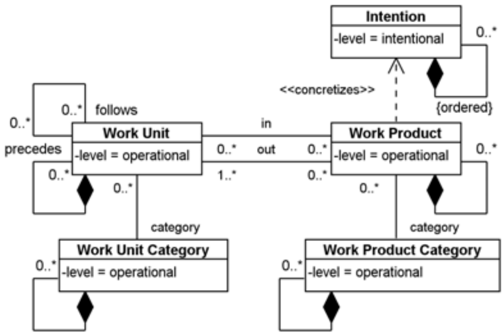


Fig. 10. The final process metamodel

Figure 10 presents the final process metamodel obtained. It represents the classes defined in the requirements and the associations between them. The link between the classes of intentional and operational level is represented by the dependency link stereotyped as “concretizes”. The abstraction level of each class is represented as an attribute *level.* The process metamodel is multi-points of view as it focuses on the activity, product and strategy points of view. The complementarity and the connection between the points of view are modeled by the “concretizes” dependency and the associations.

The method engineer can then instantiate the metamodel to represent the various ISE process models of the ISC. Figure 11 is a partial instantiation of the final process metamodel to represent the ISE processes. The method engineer wants to model the intentions and sub-intentions of service managers. One of the intentions of the service managers is to know the level of impact of a new functionality and the changes on the services organization. This can be represented as the object “Define the level of impact of the change in the service”, instance of the Intention class. This intention can be decomposed into two sub-intentions. Service managers want to define the impact of the change in the service organization and the persons that will be impacted by the change. These estimations will be useful to define the costs of the IS change, as costs of business process modifications.

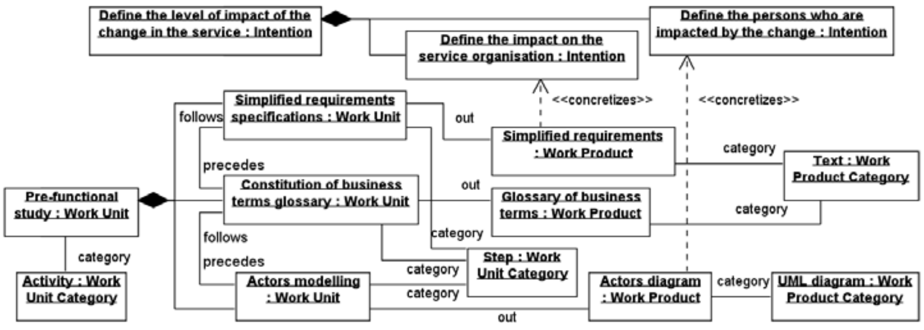

Define the level of impact of the change in the service : Intention
Define the impact on the service organisation : Intention
Define the persons who are impacted by the change : Intention
<<concretizes>>
<<concretizes>>
Simplified requirements specifications : Work Unit
follows
out
Simplified requirements : Work Product
category
Text : Work Product Category
precedes
Constitution of business terms glossary : Work Unit
out
Glossary of business terms : Work Product
category
Pre-functional study : Work Unit
category
follows
precedes
Step : Work Unit Category
category
category
Actors modelling : Work Unit
Actors diagram : Work Product
category
UML diagram : Work Product Category
out
category
Activity : Work Unit Category

Fig. 11. The process model represented as an object diagram

The operational abstraction level of the process model represents the detail of the “Pre-functional study” activity composed of three steps. First, “Simplified requirements specifications” produces the “Simplified requirements” work product that is a text. Second, the “Constitution of business terms glossary” step produces a glossary and finally, “Actors modeling” produces a UML diagram “Actors diagram”. All the work products produced during the Pre-functional study form a document called “Functional specifications” (not represented in Figure 11). The two sub-intentions “Define the impact on the service organization” and “Define the persons who are impacted by the change” are concretized by the “Simplified requirements” and “Actors Diagram” work products.

The process model represented as an object diagram is not easily and quickly understandable. Our method proposes a graphical representation (formalism) depending on the concepts in the PMUC. For example, if concepts of the operational level as work unit and work product are defined in the metamodel, the method will propose to use activity diagrams [18]. If intentions and strategies are used, the method will propose the MAP formalism [25], if there are only intentions, the KAOS formalism [29] will be proposed.

The top part of Figure 12 shows how the intentions and sub-intentions of the intentional level defined in Figure 11 can be modeled using the KAOS formalism. They are represented as parallelograms. The composition is modeled thanks to a circle. Figure 12 also presents the concepts of the operational level defined in Figure 11 as an activity diagram. The activities and steps are represented with rounded rectangles. All the work products are represented by rectangles. Stereotypes are used to specify their category. The "concretizes" dependencies are defined between the different work products and intentions of the models: the method engineer, the service managers and project managers can switch from the intentional level to the operational level.

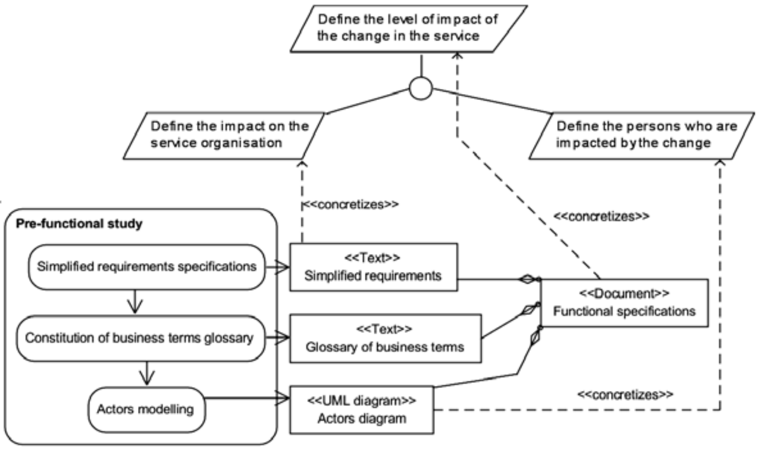

Define the level of impact of the change in the service
Define the impact on the service organisation
Define the persons who are impacted by the change
<<concretizes>>
<<concretizes>>
<<concretizes>>
Pre-functional study
Simplified requirements specifications
Constitution of business terms glossary
Actors modelling
<<Text>> Simplified requirements
<<Text>> Glossary of business terms
<<UML diagram>> Actors diagram
<<Document>> Functional specifications

Fig. 12. Intentions and sub-intentions defined at the intentional level and their concretization at the operational level in the ISE process

5 Discussion

Our proposition offers method engineers to build process metamodels for ISE depending on the specificities, the context of the projects or organizations. Our purpose differs from Situational Method Engineering, as its aim is to define IS development methods by reusing and assembling different existing method fragments [30], but it is set in the same trend of situational engineering. We may name our domain SPME (Situational Process Metamodelling Engineering).

Let us note that we do not reconsider the existing process metamodels. They all play a part in ISE processes and have their legitimacy. However, they do not define their concepts complementarity in respect to the other process metamodels. Our proposition does not consist of yet another process metamodel, but it proposes a method allowing method engineers to build process metamodels including complementarity between the concepts. Our method uses some part of the existing process metamodels. Therefore, method engineers can reuse knowledge they acquired from their experience in ISE process metamodelling. There lies the real contrast between our proposal and currently available process models, such as RUP [5] or SCRUM [7], process models that are hardly adaptable. Applying these, method engineers must follow them as described and have a little or no mean of customization. Our method, on the other hand, proposes method engineers to instantiate process models according to their needs from process metamodels they have defined themselves but still using widely accepted concepts and formalism of ISE process models.

The existing process metamodels are also fixed [10]. They do not allow method engineers to extend them or customize them. Their use is therefore limited as they do not provide all needed concepts. For example, adding the intention concept to the RUP model would be difficult as it is not defined in the RUP metamodel. Using it without defining it in the metamodel could lead to misuses and the relations with the other concepts would not be defined.

Finally, new process metamodels as ISO/IEC 24744 [16] are more flexible and provide more concepts than previous process metamodels thanks to metamodelling mechanisms as the Powertype. However, the strategy, intention and decision concepts are not taken into account here.

To conclude, we can say that our method allows more flexibility, more personalized adaptation and allows building process metamodels with less limitation than the existing one.

6 The ProMISE Tool

In this section, we present the ProMISE tool that supports our method. It has been built using Java. The two main supports of the method, the conceptual graph and the Process Domain Metamodel are defined independently from the tool in XMI files. XMI [31] is a standard format that allows storing UML models as structured text files. The main benefit of having the supports outside the tool is to permit more flexibility and scalability as the guiding will be generated thanks to the conceptual graph file and not the tool it-self. The guiding evolves as the conceptual graph evolves. Method engineers can interact with a visual conceptual graph, thanks to Prefuse [32]. Prefuse is a powerful toolkit for creating rich interactive data visualizations, such as graphs. The PMUC is displayed as a class diagram using the API UMLJGraph [33] that allows displaying UML diagrams in Java. The PMUC can be exported as an XMI file. This allows method engineers importing their process metamodels in any CASE tool, to instantiate them for example. The imports and exports are done thanks to JDom [34], a Java API able to read and write both XML and XMI files.

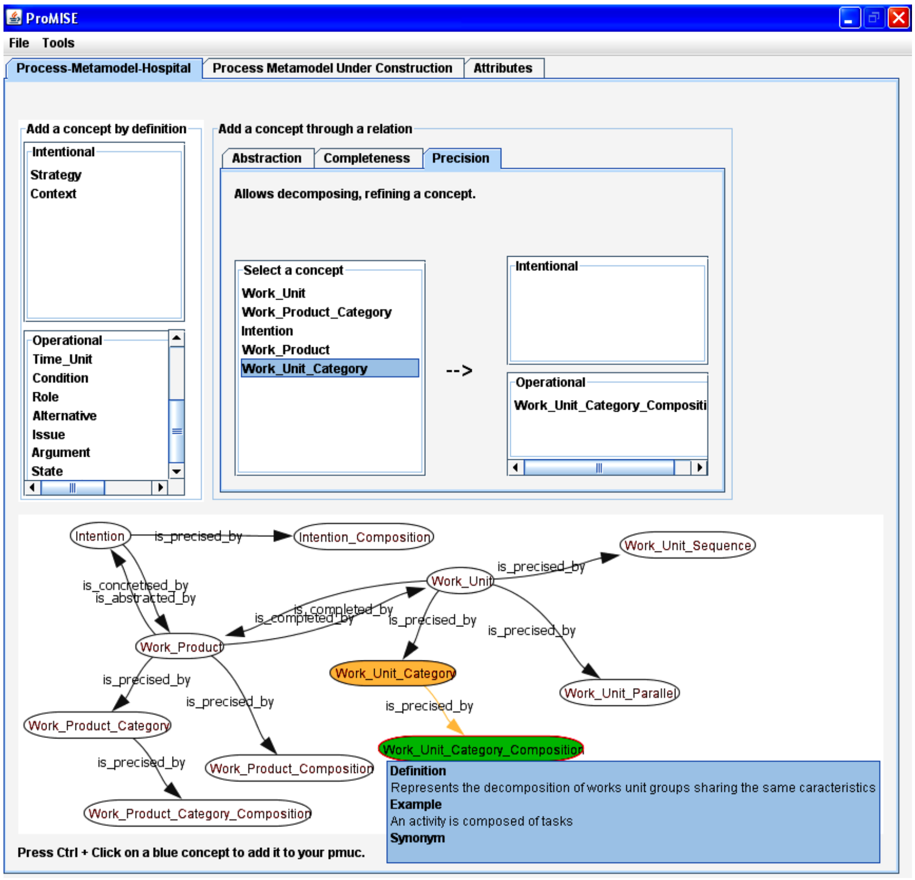
The tool allows method engineers to build process metamodels through the use of the concepts definition and the relations. Figure 13 presents a global view of the interface. It is composed of three tabs:

– The first tab (here called "Process-Metamodel-Hospital) allows method engineers to build their PMUC for a particular organization or project through the use of the definitions and the conceptual graph.

– The second tab, “Process Metamodel Under Construction”, allows method engineers to view their PMUC as a UML class diagram.
– The third tab, “Attributes”, allows method engineers to add attributes to their PMUC classes, we will not detail this functionality here.

The first tab that allows the construction of the PMUC is decomposed in two parts:

– The top part of the interface permits to select concepts by definition or by relation. Concepts are displayed according to their abstraction level which facilitates their selection. The definition, examples and synonyms of each concept can be seen by mouse over. Each relation (completeness, precision, abstraction) is represented by a tab. By selecting one tab, the concepts that can be integrated through the corresponding relation are displayed in the lists. For example, in Figure 13, the Precision tab is selected. Work Unit Category is a concept that can be refined; this allows selecting the Work Unit Category Composition concept.



Allows decomposing, refining a concept.

Definition
Represents the decomposition of works unit groups sharing the same caracteristics
Example
An activity is composed of tasks
Synonym

Press Ctrl + Click on a blue concept to add it to your pmuc.

Fig. 13. Interface of the ProMISE tool

– The lower part of the interface shows the conceptual graph with the already integrated concepts in the PMUC and the concepts that can be reached by the relations and that can be integrated in the PMUC (Work Unit Category Composition in Figure 13). By selecting a relation tab, the conceptual graph is updated with the concepts that can be integrated.

The construction of the process metamodel itself is done by the tool that uses the Process Domain Metamodel, the patterns to add new classes to the PMUC. Method engineers do not see the "dirty" part of the process metamodel construction and only interact with the conceptual graph.

7 Conclusions

In this paper, we present a method that allows method engineers to build process metamodels for ISE. The method is based on two steps: (i) the selection of concepts meeting the specificities and constraints of the projects or organizations, using a conceptual graph to help the concepts selection in a completeness – precision – abstraction 3D space; (ii) the integration of the concepts to build an adapted process metamodel called PMUC. The produced process metamodels are multi-points of view as they integrate different points of view (activity, product, decision, context and strategy). The metamodels are also adapted to the context of the organizations as only the needed concepts were selected. At last, all the knowledge of ISE processes of the project or the organization is modeled in only one process metamodel and related process models. There is a better consistency of the manipulated concepts and a better understanding of the links between intentional and operational levels in the projects.

The ProMISE tool has been implemented to allow method engineers building process metamodels according to our method. The construction of the process metamodel itself is hidden to the method engineers as they only "play" with the conceptual graph: the process metamodel is built automatically by the tool.

Further step is to allow the instantiation of the process metamodels until the monitoring of particular information systems engineering projects. Another part of perspectives concerns the formalism that method engineers should use to represent the process models instantiated from the metamodels produced by this method. It would be useful to guide method engineers in the use of such or such formalism, depending on the concepts selected in their PMUC.

The Process Domain Metamodel may evolve, with the publications by the community of new process models and metamodels for ISE. The conceptual graph will also evolve, in order to propose method engineers the largest choice of possibilities taking into account the latest evolutions in terms of ISE process metamodelling.

References

1. Humphrey, W.S., Kellner, M.I.: Software process modeling: principles of entity process models. In: ICSE 1989, pp. 331–342. ACM, New York (1989)
2. Software Engineering Institute: CMMI for Development, Version 1.2 (2006)

3. Royce, W.W.: Managing the development of large software systems: concepts and techniques. In: ICSE 1987, pp. 328–338. IEEE Computer Society Press, Los Alamitos (1987)
4. Boehm, B.: A spiral model of software development and enhancement. SIGSOFT Software Engineering Notes 11(4), 14–24 (1986)
5. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley, Longman Publishing, Co., Inc., Boston (2000)
6. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, Longman Publishing Co., Inc., Boston (1999)
7. Schwaber, K., Beedle, M.: Agile Software Development with SCRUM. Prentice Hall, Upper Saddle River (2001)
8. Hug, C., Front, A., Rieu, D.: A Process Engineering Method Based on a Process domain Model and Patterns. In: MoDISE International Workshop, pp. 126–137 (2008)
9. Hug, C., Front, A., Rieu, D.: Process Engineering Method Based on Ontology and Patterns. In: ICSOFT 2008, pp. 29–36 (2008)
10. Hug, C., Front, A., Rieu, D., Henderson-Sellers, B.: A Method to build Information Systems Engineering Process Metamodels. J. of Sys. & Soft. 82(10), 1730–1742 (2009)
11. Hug, C., Front, A., Rieu, D.: Ingénierie des processus. Une approche à base de patrons. Revue RSTI. Série ISI 13(4), 11–34 (2008)
12. OMG: Software Process Engineering Meta-Model. Version 2.0 (2008)
13. Open Process Framework, `http://www.opfro.org`
14. OOSPICE, Software Process Improvement and Capability Determination for Object-Oriented/ Component-Based Software Development, `http://www.oospice.com`
15. Australian Standard: Standard Metamodel for Software Development Methodologies. AS, 4651–2004 (2004)
16. ISO/IEC: 24744 Software Engineering - Metamodel for Development Methodologies (2007)
17. Harel, D.: Statecharts: A Visual Formulation for Complex Systems. Science of Computer Programming 8(3), 231–274 (1987)
18. OMG: Unified Modeling Language: Superstructure. Version 2.2 (2009)
19. Finkelstein, A., Kramer, J., Goedicke, M.: ViewPoint oriented software development. Third International Workshop on Software Engineering and Its Applications, pp. 374–384 (1990)
20. Kunz, W., Rittel, H.W.J.: Issues as elements of information systems. WP 131, Heidelberg, Berkeley (1970)
21. Potts, C., Bruns, G.: Recording the Reasons for Design Decisions. In: ICSE 1988, pp. 418–427. IEEE Computer Society Press, Los Alamitos (1988)
22. Potts, C.: A generic model for representing design methods. In: ICSE 1989, pp. 217–226. IEEE Computer Society/ ACM Press (1989)
23. Jarke, M., Mylopoulos, J., Schmidt, J.W., Vassiliou, Y.: DAIDA: An Environment for Evolving Information Systems. ACM Trans. on Inf. Sys. 10(1), 1–50 (1992)
24. Rolland, C., Souveyet, C., Moreno, M.: An Approach for defining ways-of-working. Information System Journal 20(4), 337–359 (1995)
25. Rolland, C., Prakash, N., Benjamen, A.: A Multi-Model View of Process Modelling. Requirements Engineering 4(4), 169–187 (1999)
26. Panet, G., Letouche, R.: Merise/2 Modèles et techniques Merise Avancés. Les Editions d'Organisation, Paris (1994)
27. Hug, C.: Méthode, modèles et outil pour la méta-modélisation des processus d'ingénierie de systèmes d'information. PhD Thesis, Grenoble I University (2009)

28. Hug, C., Mandran, N., Front, A., Rieu, D.: Qualitative Evaluation of a Method for Information Systems Engineering Processes. In: RCIS 2010, pp 257–268 (2010)
29. Objectiver: A KAOS tutorial. Respect-It (2007)
30. Ralyté, J., Rolland, C.: An Assembly Process Model for Method Engineering. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 267–283. Springer, Heidelberg (2001)
31. OMG.: MOF 2.0 / XMI Mapping Specification. Version 2.1.1 (2007)
32. Prefuse, http://prefuse.org/
33. UMLJGraph, http://umljgraph.sourceforge.net/
34. JDOM, http://www.jdom.org/

Investigating the Use of Object-Oriented Design Patterns in Open-Source Software: A Case Study[*]

Apostolos Ampatzoglou, Sofia Charalampidou, and Ioannis Stamelos

Aristotle University of Thessaloniki, Thessaloniki, Greece
{apamp,stamelos}@csd.auth.gr

Abstract. During the last decade open source software communities are thriving. Nowadays, several open source projects are so popular that are considered as a standard in their domain. Additionally, the amount of source code that is freely available to developers, offer great reuse opportunities. One of the main concerns of the reuser is the quality of the code that is being reused. Design patterns are well known solutions that are expected to enhance software quality. In this paper we investigate the extent to which object-oriented design patterns are used in open-source software, across domains.

Keywords: Open source software, Design patterns, Empirical study.

1 Introduction

Open source software (OSS), a term introduced in 1998 [9], has been expanding rapidly in recent years. There exist several successful projects developed as open source software, such as Linux, Mozila Firefox and Apache Server.

Collaboration is the basis of the development of an open source project. A first version of the project is developed by a single developer or a group of developers and is released over the internet, freely available, so that the members of the open source community can extend and maintain the project. In open source software development, there are both advantages and disadvantages. One disadvantage of open source software is that there is no documentation and technical support. On the other hand, the advantages of this software development type are its low cost, its reliability and the availability of the source code in order to customize the project according to once special needs [18].

Moreover, another feature of open source software is the potential reuse of the source code, which is freely available to the open source developers. A code segment should have certain characteristics, such as understandability, maintainability and flexibility, in order to be easily and successfully reused in another project.

Gamma et.al have introduced, in 1995, design patterns as common solutions to common design problems [10]. The main incentive to introduce patterns was the creation of a common vocabulary for developers, which provide flexible, reusable and

[*] This paper is an extended and revised version of the paper entitled “An Empirical Study on Design Pattern Usage on Open-Source Software”, published in ENASE 2010.

maintainable design solutions. Furthermore, Meyer et.al explains how object-oriented design patterns can be transformed to reusable components [16].

In literature, many empirical studies have attempted to examine how design pattern application affects software quality. The main conclusion of these studies is that object oriented design patterns can not be considered as universally good or bad. In section 2, we provide a more detailed presentation of the current state of the art, discussing the effect of design pattern use on software quality.

This paper is an extended and revised version on authors' previous work [1] that aims at examining the application of object-oriented design patterns in open source software. More specifically, an empirical study has been performed, in order to investigate which patterns are more frequently used in open source software, which differences exist within software domains and the size of design patterns. The main extension and revision points are concluded below:

- The number of case study subjects is increased
- Added two software categories and revised two others by exploring broader software categories (i.e. replaced e-commerce applications with business applications)
- One more research question dealing with design pattern size has been added.

In the next section of the paper, a literature review on design patterns influence on software quality is provided, In section 3 we present the methodology of our work, i.e. research questions, case study process and data analysis methods. In section 4, the findings of our empirical study are presented, while in section 5 we provide a discussion on the results, categorized according to the research question they address to. Finally, at the end of the paper, possible threats to validity, future work and conclusions are presented.

2 Design Patterns

In this section of the paper we present the findings of a literature review on the influence of design pattern application on software quality. A common division of software quality is between internal and external quality [4]. Software internal quality is measurable and estimates software features such as complexity, cohesion, coupling, inheritance etc. that are not easy to understand for the end-user or the developer. External software quality can not be easily measured, but it is closer to the end-user's and the developer's sense. Functionality, reliability, usability, efficiency, maintainability and portability, are the best known external quality characteristics, as described in ISO/IEC 9126.

The effect of design pattern application on software internal quality has been examined by Ampatzoglou et.al [2] and Huston [12]. According to Huston, the application of the Mediator pattern reduces coupling, the Bridge pattern reduces size and inheritance metrics and finally the use of the Visitor pattern reduces the project's complexity with respect to number of methods [12]. Ampatzoglou et.al suggests that the application of the State and the Bridge pattern reduces coupling and complexity, with respect to cyclomatic complexity and increases cohesion among methods. As a side-effect, the project size concerning the number of classes increases [2].

Furthermore, the effect of design patterns on external quality has been investigated in several studies. The influence of design patterns i.e. Abstract Factory, Observer, Decorator, Composite and Visitor to software maintainability has been investigated by Vokac et.al and Prechelt et.al [17 and 21], by conducting controlled experiments. According to the results of the experiment, the employment of a design pattern is usually more useful than the simpler solution. The software engineer has to choose between applying a design pattern or a simple solution in line with common sense. Besides, Hsueh et.al investigate how design patterns impact on one quality attribute, which is the most obvious attribute that the pattern affects [11]. The selection of the quality attribute is made according to the pattern's non functional requirements, whereas the metric is selected according to [4].

Wendorff presents an industrial case study, where inappropriate pattern use has caused severe maintainability problems. The reasons of inappropriate design pattern use is classified into two categories (1) software engineers have not understood the reasoning behind the patterns that they have employed (2) the patterns that they have applied have not fulfilled the project's requirements. Moreover, the paper emphasizes the need for documenting design pattern application and that pattern removal leads to extreme cost [22]. In [13], an analysis on software maintenance, with professional engineers, is performed. According to the empirical study, design patterns do not always have positive impact on software quality. In particular, it is concluded that when patterns are applied, the simplicity, the learnability and the understandability are negatively affected.

In [6], an industrial case study is conducted, in order to examine the correlation among code changes, reusability, design patterns, and class size. On the report of the results of the study, the number of changes is highly correlated to class size. Additionally, classes that play roles in design patterns or that are reused through inheritance are more change prone than others. Despite the study's good structure and validation, it investigates an individual maintainability aspect, change proneness, and does not mention maintainability issues such as change effort and design quality.

In [8], the authors present the investigation of correlations among class change proneness, the role that a class holds in a pattern and the kind of change that occurs. They use three open source projects in order to perform the empirical study. Concerning the majority of design patterns, the results of the study comply with common sense. However, in some cases, the conclusions differ from those expected.

3 Methodology

Wholin et.al suggests that there are three major empirical investigation approaches, surveys, case studies and experiments [23]. In this paper we have conducted a case study, exploiting the plethora of open source. On the contrary, surveys are not suitable for our research because in this case we would miss the patterns that were employed without intention by programmers. Finally, an experiment with open-source programmers would decrease the number of subjects in our research. In this section of the paper we describe the methodology of our case study. The case study of our research was based on the guidelines described in [14], and consisted of the following steps:

(a) Define hypothesis
(b) Select projects
(c) Method of comparison selection
(d) Minimization of confounding factors
(e) Planning the case study
(f) Monitoring the case study and
(g) Analyze and report the results

The hypotheses, i.e. step (a), are defined in section 3.1. Steps (b) and (d) which deal with project selection protocol and minimizing confounding factors are presented in section 3.2, accompanied with step (e). The methods used in analyzing the data, i.e. step (c), is presented in section 3.3, step (f), described in [14], is discussed in section 6. Finally, concerning step (g), we report the results on section 4 and discuss them in section 5.

3.1 Research Questions

In this section of the paper we state the research questions that are investigated in our study.

RQ1: Which is the frequency of design pattern application?
RQ2: Are there any differences in pattern application within the software categories under study?
RQ3: Are there any differences in the number of pattern participant classes across pattern types and software categories?

3.2 Case Study Plan

In this section of the paper we present the case study plan. According to [5] planning a case study is an important step for the validity of the study. Our plan involved a five step procedure described below:

(a) choose open source project categories
(b) identify a number of projects that fulfil certain selection criteria, for each a category
(c) perform pattern detection for every selected project. The pattern detection was conducted with an automated tool [20] that identifies instances of eleven (11) patterns of all GoF pattern categories (i.e. *Creational*, *Behavioural* and *Structural*)
(d) tabulate data
(e) analyze data with respect to the research questions

In this study the OSS project categories that have been considered are development tools, office/business applications, internet application, databases and computer games. These categories have been selected as highly active topics in open source communities [19]. From these categories we have selected projects that fulfilled the following criteria:

(a) Software written in java, due to limitations of pattern detection tool [20]. However, java is probably the most widely used programming language.
(b) software that provides binary code, due to limitations of pattern detection tool.
(c) software should be ranked in the fifty most successful projects of their category, according to sourceforge.net rating.
(d) software binary size should be larger than 100KB, in order not to be considered trivial.

In case studies, factors, other than the independent variables, which influence the value of the dependent variable, are considered confounding factors. The most important confounding factors in our research are considered to be the experience of the developer on design patterns and object-oriented programming in general. In our study we limit our analysis to automatically collected data. On the other hand, it is expected that in a random developer sample of a large developers' community, the distribution of skill and experience are closely near to the distribution of the population.

3.3 Data Analysis Methods

The dataset that has been created after design pattern detection consisted mainly of numerical data. On the completion of the pre-processing phase each project was characterized by 28 variables:

- name
- category
- number of downloads
- number of factory method instances
- number of prototype instances
- number of singleton instances
- number of creational pattern instances
- number of adapter instances
- number of composite instances
- number of decorator instances
- number of proxy instances
- number of structural pattern instances
- number of observer instances
- number of state-strategy instances
- number of template method instances
- number of visitor instances
- number of behavioural pattern instances
- average number of pattern participants per pattern (11 variables)

The analysis phase of our study has employed descriptive statistics, independent sample t-test and paired sample t-test. Concerning RQ_1, we have employed descriptive statistics and paired sample t-tests so as to compare the mean number of instances for each design pattern. In the investigation of RQ_2 and RQ_3, for similar reasons we have used descriptive statistics and independent sample t-tests. The statistical analysis has been performed with SPSS©.

According to [23], one of the first steps during statistical analysis of the dataset is the elimination of outliers. In our study we identified and erased seventeen outliers. In most cases the observed extreme values where identified as maximum values, that is software that exhibit a very large number of pattern instances.

4 Results

In Table 1, the mean number of design pattern instances is presented. The data refer to the whole data set without discrimination across software categories. In addition to that, standard deviation of each variable is presented.

Table 1. Average Number of Pattern Instances

	Mean	**Std. Deviation**
Factory	3.21	7.21
Prototype	5.80	18.98
Singleton	13.99	19.16
Creational	*23.01*	*36.55*
Adapter	34.71	53.66
Composite	0.48	2.22
Decorator	2.53	6.50
Proxy	1.58	4.50
Structural	*39.30*	*61.17*
Observer	1.44	2.55
State	37.70	58.96
Template	5.93	8.52
Visitor	0.50	2.50
Behavioural	*45.65*	*66.19*

The results of Table 1 provide indications on the employment rate of each pattern in OSS. In order to be able to compare the mean values of each variable in a more elaborate way, we have performed 55 paired sample t-tests, i.e. one test for every possible pair of design patterns. The results of a t-test between two variables are interpreted by two numbers, the mean difference (diff) and the t-test significance (sig). The *diff* variable represents the difference of subtracting the mean value of the second variable, from the mean value of the first. Whereas, *sig* represents the possibility, that *diff* is not statistically significant. In Table 2, we present the statistically significant differences in pattern application.

Table 2. Significant paired sample t-tests on pattern employment difference

	diff	**sig**		**diff**	**sig**
Factory – Singleton	-10.78	0.00	*Decorator – Template*	-3.40	0.00
Factory – Adapter	-31.50	0.00	*Decorator – Visitor*	2.03	0.00
Factory – Composite	2.74	0.00	*Proxy – State*	-36.10	0.00
Factory - Proxy	1.64	0.03	*Proxy – Template*	-4.36	0.00
Factory - Observer	1.78	0.01	*Proxy - Visitor*	1.08	0.04
Factory – State	-34.45	0.00	*Observer – State*	-36.25	0.00
Factory - Template	-2.72	0.00	*Observer – Template*	-4.50	0.00
Factory – Visitor	2.71	0.00	*Observer - Visitor*	0.94	0.01
Prototype – Singleton	-8.19	0.00	*State – Template*	31.71	0.00
Prototype – Adapter	-28.91	0.00	*State – Visitor*	37.19	0.00
Prototype – Composite	5.33	0.00	*Template – Visitor*	5.43	0.00
Prototype - Decorator	3.27	0.04	*Adapter – Composite*	34.23	0.00
Prototype - Proxy	4.22	0.01	*Adapter – Decorator*	32.18	0.00
Prototype – Observer	4.36	0.02	*Adapter – Proxy*	33.13	0.00
Prototype – State	-31.84	0.00	*Adapter – Observer*	33.27	0.00
Prototype – Visitor	5.30	0.01	*Adapter - State*	-2.66	0.00
Singleton – Adapter	-20.72	0.00	*Adapter – Template*	28.77	0.00
Singleton – Composite	13.51	0.00	*Adapter – Visitor*	34.21	0.00
Singleton – Decorator	11.46	0.00	*Composite – Decorator*	-2.06	0.00
Singleton – Proxy	12.41	0.00	*Composite - Proxy*	-1.10	0.01
Singleton – Observer	12.55	0.00	*Composite – Observer*	-0.96	0.00
Singleton – State	-23.58	0.00	*Composite – State*	-37.22	0.00
Singleton – Template	8.06	0.00	*Composite – Template*	-5.46	0.00
Singleton – Visitor	13.49	0.00	*Decorator – State*	-35.14	0.00

In Table 3, the mean numbers of instances of each design patterns within the software categories under study are presented.

In order to statistically validate the results of the above table, we performed 42 independent sample t-tests, i.e. one test for each pattern for all the possible pairs of software categories. In Table 4, we provide the statistically significant results on comparing pattern application between software categories. The results are presented similarly to those of Table 2.

Table 3. Average Number of Pattern Instances among Software Categories

	Office / Business	**Internet**	**Development Tools**	**Database**	**Games**
Factory	9.24	3.00	1.00	2.64	0.50
Prototype	20.05	1.85	2.45	3.45	1.58
Singleton	29.43	13.55	8.30	7.36	11.67
Creational	*58.71*	*18.40*	*11.75*	*13.45*	*13.75*
Adapter	91.95	18.30	19.80	24.77	19.83
Composite	1.52	0.15	0.10	0.32	0.29
Decorator	6.95	1.50	1.50	2.14	0.75
Proxy	4.95	0.25	0.20	0.73	1.67
Structural	*105.38*	*20.20*	*21.60*	*27.95*	*22.54*
Observer	2.57	1.20	1.35	0.77	1.33
State	94.57	29.25	23.84	27.14	15.63
Template	11.71	5.80	4.10	6.05	2.42
Visitor	0.00	1.70	0.25	0.27	0.37
Behavioral	*108.86*	*37.95*	*29.84*	*34.23*	*19.75*

Table 4. Significant independent sample t-tests

	Pattern	**diff**	**sig**
Office/Business - Internet	Factory	6.24	0.05
Office/Business - Internet	Prototype	18.20	0.04
Office/Business - Internet	Singleton	15.88	0.03
Office/Business - Internet	Adapter	73.65	0.00
Office/Business - Internet	Decorator	5.45	0.06
Office/Business - Internet	Proxy	4.70	0.02
Office/Business - Internet	State	65.32	0.01
Office/Business - Internet	Template	5.91	0.08
Office/Business – Development Tools	Factory	8.24	0.01
Office/Business – Development Tools	Prototype	17.60	0.05
Office/Business – Development Tools	Singleton	21.13	0.01
Office/Business – Development Tools	Adapter	72.15	0.00
Office/Business – Development Tools	Decorator	5.45	0.06
Office/Business – Development Tools	Proxy	4.75	0.02
Office/Business – Development Tools	State	70.73	0.01
Office/Business – Development Tools	Template	7.61	0.01
Office/Business – Database	Factory	6.60	0.04
Office/Business – Database	Prototype	16.59	0.07
Office/Business – Database	Singleton	22.07	0.00
Office/Business – Database	Adapter	67.18	0.00
Office/Business – Database	Proxy	4.23	0.03
Office/Business – Database	Observer	1.80	0.05
Office/Business – Database	State	67.44	0.01

Table 4. (*Continued*)

	Pattern	**diff**	**sig**
Office/Business – Database	Template	5.67	0.08
Office/Business – Games	Factory	8.74	0.01
Office/Business – Games	Prototype	18.46	0.04
Office/Business – Games	Singleton	17.76	0.02
Office/Business – Games	Adapter	72.12	0.00
Office/Business – Games	Decorator	6.20	0.03
Office/Business – Games	State	78.95	0.02
Office/Business – Games	Template	9.30	0.00
Internet – Development Tools	Factory	2.00	0.09
Internet - Database	Singleton	6.19	0.09
Internet – Games	Factory	2.50	0.03
Internet – Games	State	13.63	0.09
Database – Games	Factory	2.14	0.07
Database – Games	Template	3.63	0.07

In Table 5, we present the mean numbers of classes that participate in each design patterns within the software categories under study. Additionally, Table 6 presents the statistically significant differences between the mean values of number of classes that participate in design patterns, among software categories.

Table 5. Average Number of Pattern Participating Classes among Software Categories

	Office / Business	**Internet**	**Development Tools**	**Database**	**Games**	**Overall**
Factory	6.35	5.93	6.77	6.93	6.21	6.46
Prototype	7.84	7.69	10.80	7.23	8.74	8.16
Singleton	1.00	1.00	1.00	1.00	1.00	1.00
Adapter	2.00	2.00	2.00	2.00	2.00	2.00
Composite	7.42	20.50	12.00	6.20	9.33	9.84
Decorator	11.77	17.12	10.33	11.93	17.23	13.46
Proxy	2.20	2.29	2.07	3.20	2.11	2.32
Observer	10.39	11.90	11.00	6.65	10.67	10.33
State	7.15	9.04	6.92	6.40	8.68	7.47
Template	6.83	5.79	5.82	6.35	7.88	6.45
Visitor	0.00	7.15	3.24	4.72	2.00	5.21

Table 6. Significant independent sample t-tests

	Pattern	diff	sig
Office/Business – Development Tools	Prototype	-2.96	0.00
Internet – Development Tools	Prototype	-3.11	0.00
	Pattern	**diff**	**sig**
Databases – Development Tools	Prototype	-3.57	0.00
Office/Business – Internet	Composite	-13.08	0.01
Databases – Development Tools	Composite	-5.8	0.00
Databases – Internet	Composite	-14.3	0.02
Internet – Games	Composite	11.17	0.03
Office/Business – Internet	Decorator	-5.35	0.00
Office/Business – Games	Decorator	-5.46	0.00
Internet – Development Tools	Decorator	6.79	0.00
Internet – Databases	Decorator	5.19	0.00
Games – Development Tools	Decorator	6.9	0.00
Games – Databases	Decorator	5.3	0.00
Internet – Development Tools	Proxy	0.22	0.04
Databases – Office/Business	Observer	-3.74	0.01
Databases – Internet	Observer	-5.25	0.00
Databases – Development Tools	Observer	-4.35	0.00
Databases – Games	Observer	-4.02	0.00
Office/Business – Internet	State	-1.89	0.00
Internet – Development Tools	State	2.12	0.00
Databases – Office/Business	State	-0.75	0.00
Databases – Internet	State	-2.64	0.00
Databases – Development Tools	State	-0.52	0.01
Office/Business – Games	State	-1.53	0.00
Games – Development Tools	State	1.76	0.00
Games – Databases	State	2.28	0.00
Games – Development Tools	Template	2.06	0.01
Games – Databases	Template	1.53	0.01
Internet – Games	Visitor	5.15	0.00
Games – Databases	Visitor	-2.72	0.00
Games – Development Tools	Visitor	-1.24	0.00
Databases – Internet	Visitor	-2.53	0.00
Databases – Development Tools	Visitor	1.03	0.00
Internet – Development Tools	Visitor	3.91	0.00

5 Discussion

This section of the paper discusses the results of our case study. The discussion is organized in subsections according to the research questions that have been introduced in the beginning of the paper. Thus, section 5.1 discusses which design patterns are more frequently used in open source software development, section 5.2 discusses the usage of each design pattern on three software categories and section 5.3 discusses the size of the design patterns used in open source software in general.

5.1 Design Pattern Application

The results of Table 1, clearly suggest that some patterns are more frequently applied in open source than others. In addition to that, Table 2 suggests that pattern usage intensity classifies patterns in seven categories as shown in Figure 1. Patterns on the top of Figure 1 are statistically significantly employed more times in open source software projects than those closer to the bottom of Figure 1.

Some of the results that are presented in Figure 1 are reasonable, whereas some findings are surprising. As one would expect, the *Adapter* pattern is frequently used, because reusing classes of others is a common practice in open source software communities. In such cases, adapter provides a mechanism for adopting the new class in the existing system without modifying the existing code. In addition to that, the Adapter's rationale is akin to the basic concepts of object - oriented programming and thus it might be explicitly used by the developers. Furthermore, the *State* pattern as expected ranks high, because its background requires just the proper use of inheritance. Finally, more difficult to understand patterns, according to authors' opinion, such as *Visitor* and *Observer*, are not often employed by open source developers.

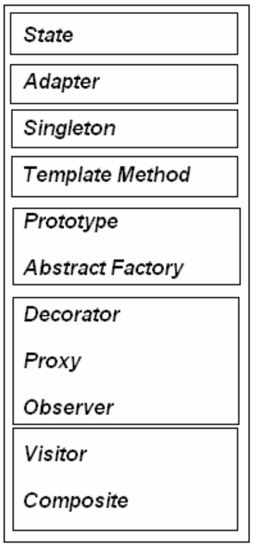


Fig. 1. Design Pattern Usage Levels

On the contrary, although the *Singleton* pattern is quite complex in its structure [7] and it was expected not to be as popular, it is ranked as the 3rd most used pattern. A possible reason for this is the limitation of the case study subject to the Java languages, where Singleton is implemented by a simple instantiation mechanism. Another bizarre observation is that the *Decorator* pattern is more frequently used than the *Composite* pattern. The Composite pattern is the base of the Decorator pattern and therefore it was expected to be more frequently employed. Summing up the above, open source developers employ easy to understand patterns more than more elaborate ones. A possible reason for this is that typically there are no detailed formal design activities before programming in open source.

5.2 Design Patterns and Software Categories

As it is observed in Table 3, design pattern usage within every category follows similar distribution as in open source software development in general. However, comparing pattern application across software categories, the results suggest that some patterns are more frequently applied in one category, than another. From Tables 3 and 4, we observe that *Office/Business* applications employ statistically significantly more patterns than any other category. Furthermore, rather surprising is the fact that in general *Development Tools* employ a relative limited number of pattern instances w.r.t the other software categories. One would expect that developers of this category would be familiar with patterns and use them. Figure 2 presents the ranking of pattern usage among software categories.

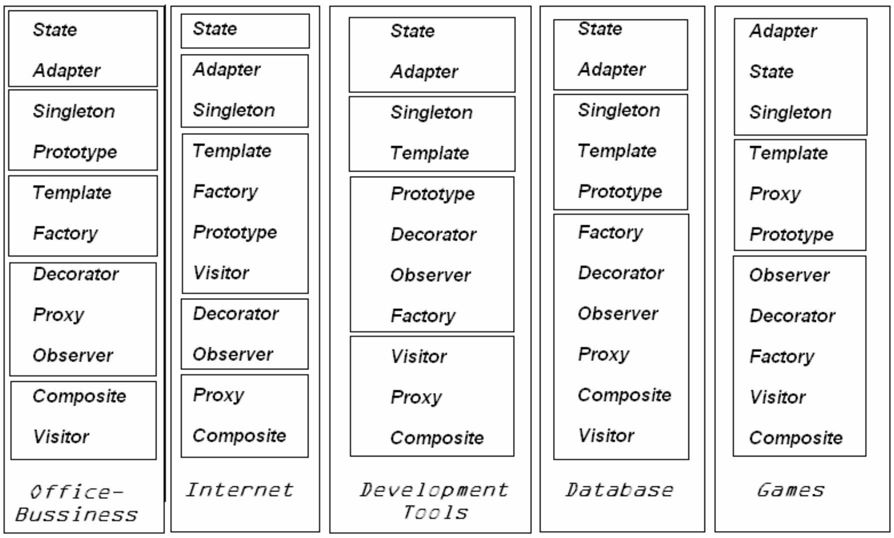


Fig. 2. Design Pattern Usage Levels across Categories

From Figures 1 and 2, it is suggested that *Decorator and Observer* patterns are more highly ranked in *Development Tools* than in the other categories. This fact can be justified by the expectation that developers of this category are more likely to be aware of the pattern, which is not easily applied by chance. In addition to that, the *Adapter* pattern is the most frequently employed pattern in the *Games* category. This fact suggests that game developers might perform more “as is” reuse activities than other programmers. This observation is interesting and deserves further investigation.

Additionally, the *Visitor* pattern appears to be more applicable in *Internet* application and the *Proxy* pattern more applicable in *Games*. Thus, we can assume that domain specific requirements (functional or non-functional) of this category might be implemented with the use of these patterns.

5.3 Design Pattern Size among Software Categories

This section of the paper discusses the most important findings on the variation of the size of deign patterns among software categories. The “largest” patterns, with respect to number of classes appear to be *Decorator* and *Observer*, whereas the pattern with the least pattern, apart from Singleton and Adapter that employ a standard number of classes, appears to be *Visitor*.

Within software categories, we found that when the *Prototype* pattern is applied in *Development Tools*, it appears to employ statistically significant more classes than when applied in any other software category. Similarly, the *Composite* pattern instances in *Internet* applications are larger than the Composite instances in other software categories. Concerning *Decorator*, we identified that the larger pattern instances can be found in *Games* and *Internet* applications. Finally, the smallest *Observer* instances can be identified in *Database* applications.

These findings can be used in studies that investigate pattern effect on software quality, with respect to their size, the role that each class plays in a pattern and for case study construction.

6 Threats to Validity

This section of the paper presents the internal and external threats to the validity of our case study. Firstly, since the subjects have been open-source projects, the results may not apply to closed source software. Concerning the empirical study internal validity, the existence of confounding factors is analyzed in section 3.4. The most confounding factor is that the study cannot take into account the knowledge of developer’s on design patterns, but it can be reasonably assumed that the familiarity degree with pattern knowledge across different application domains, corresponds to the distribution of the population.

In addition to that, the sample size is quite small with respect to the total number of open source software and generalizing the results from the sample to the population is risky. In addition, the dataset consisted only from Java projects, since the tool we used was able to detect design patterns only in binary java files. Moreover, only one repository, namely Sourceforge, has been mined.

7 Conclusions

This study is an extension of a previous work of the authors. It empirically investigates the usage of object oriented design patterns in open source software development. For this reason the authors have explored 129 open source software from five categories, i.e. development tools, business/office application, internet applications, database applications and computer games.

The results of the study confirm that “easy to use” design patterns, such as *Adapter*, *State* and *Singleton* are more frequently applied in open source. More elaborate patterns such as *Visitor* and *Observer* are more frequently employed by development tool programmers, most probably due to their better understanding and knowledge on software engineering issues. Additionally, the frequent application of the *Adapter* pattern in computer games might indicate higher reuse levels in this type of software applications. Finally, the results suggested that among software categories, *Office/Business* application employ statistically significantly more design patterns than other categories and that the size of design patterns vary among software genres.

As future work we are about to create a web repository on the findings of the design pattern detection process, so as to enhance design pattern reuse opportunities. In addition to that, we are going to explore projects written in other programming languages, such as C++. More software categories and open source projects are going to be investigated. Finally, the most important findings of the study, such as the reuse increased reuse opportunities in games, the limited number of pattern instances in development tools and the factors that influence design pattern usage are going to be investigated.

References

1. Ampatzoglou, A., Charalampidou, S., Savva, K., Stamelos, I.: An empirical study on design pattern employment in open-source software. In: 5th Working Conference on the Evaluation of Novel Approaches in Software Engineering, pp. 275–284. INSTICC, Athens (2010)
2. Ampatzoglou, A., Chatzigeorgiou, A.: Evaluation of object-oriented design patterns in game development. Information and Software Technology 49(5), 445–454 (2007)
3. Arnout, K., Meyer, B.: Pattern componentization: the factory example. Innovations in Systems and Software Technology 2(2), 65–79 (2006)
4. Bansiya, J., Davis, C.: A Hierarchical Model for Object-Oriented Design Quality Assessment. IEEE Transaction on Software Engineering 28(1), 4–17 (2002)
5. Basili, V.R., Selby, R.W., Hutchens, D.H.: Experimentation in Software Engineering. IEEE Transactions on Software Engineering 12(7), 733–743 (1986)
6. Bieman, J.M., Jain, D., Yang, H.J.: OO design patterns, design structure, and program changes: an industrial case study. In: 17th International Conference on Software Maintenance, ICSM 2001, pp. 580–591. IEEE Computer Society, Florence (2001)
7. Chatzigeorgiou, A.: Object-Oriented Design: UML, Principles, Patterns and Heuristics, 1st edn. Kleidarithmos, Athens (2005)
8. Di Penta, M., Cerulo, L., Gueheneuc, Y.G., Antoniol, G.: An Empirical Study of Relationships between Design Pattern Roles and Class Change Proneness. In: 24th International Conference on Software Maintenance, ICSM 2008, pp. 217–226. IEEE Computer Society, Beijing (2008)
9. Feller, J., Fitzgerald, B.: Understanding open source software development, 1st edn. Addison-Wesley Longman, Boston (2002)

10. Gamma, E., Helms, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, 1st edn. Addison-Wesley Professional, Reading (1995)
11. Hsueh, N.L., Chu, P.H., Chu, W.: A quantitative approach for evaluating the quality of design patterns. Journal of Systems and Software 81(8), 1430–1439 (2008)
12. Huston, B.: The effects of design pattern application on metric scores. Journal of Systems and Software 58(3), 261–269 (2001)
13. Khomh, F., Gueheneuc, Y.G.: Do design patterns impact software quality positively? In: 12th European Conference on Software Maintenance and Reengineering, CSMR 2008, pp. 274–278. IEEE Computer Society, Athens (2008)
14. Kitchenham, B., Pickard, L., Pfleeger, S.L.: Case Studies for Method and Tool Evaluation. IEEE Software 12(4), 52–62 (1995)
15. McShaffry, M.: Game Coding Complete. Paraglyph Press, Arizona (2003)
16. Meyer, B., Arnout, K.: Componentization: The Visitor Example. IEEE Computer 39(7), 23–30 (2006)
17. Prechelt, L., Unger, B., Tichy, W.F., Brossler, P., Votta, L.G.: A controlled experiment in maintenance comparing design patterns to simpler solutions. IEEE Transactions on Software Engineering 27(12), 1134–1144 (2001)
18. Samoladas, I., Stamelos, I., Angelis, L., Oikonomou, A.: Open source software development should strive for even greater code maintainability. Communications of the ACM 47(12), 83–87 (2004)
19. Sowe, S.K., Angelis, L., Stamelos, I., Manolopoulos, Y.: Using Repository of Repositories (RoRs) to Study the Growth of F/OSS Projects: A Meta-Analysis Research Approach. In: OSS 2007, Open Source Software Conference, pp. 147–160. Springer, Limerick (2007)
20. Tsantalis, N., Chatzigeorgiou, V., Stephanides, G., Halkidis, S.T.: Design Pattern Detection using Similarity Scoring. IEEE Transaction on Software Engineering 32(11), 896–909 (2006)
21. Vokác, M., Tichy, W., Sjøberg, D.I.K., Arisholm, E., Aldrin, M.: A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns - A Replication in a Real Programming Environment. Empirical Software Engineering 9(3), 149–195 (2003)
22. Wendorff, P.: Assessment of Design Patterns during Software Reengineering: Lessons Learned from a Large Commercial Project. In: 5th European Conference on Software Maintenance and Reengineering, CSMR 2001, pp. 77–84. IEEE Computer Society, Lisbon (2001)
23. Wohlin, C., Runeson, P., Host, M., Ohlsson, M.C., Regnell, B., Wesslen, A.: Experimentation in Software Engineering, 1st edn. Kluwer Academic Publishers, Boston (2000)

Requirements Engineering via Non-monotonic Logics and State Diagrams

David Billington[1], Vladimir Estivill-Castro[2], René Hexel[1], and Andrew Rock[1]

[1] ICT/IIIS, Griffith University, Nathan, QLD, 4111, Australia
[2] Visiting Scholar, Universitat Popeu Fabra, Barcelona, Spain
{d.billington,v.estivill-castro,r.hexel,a.rock}@griffith.edu.au
www.mipal.net.au

Abstract. We propose to model the behaviour of embedded systems by finite state machines whose transitions are modelled by predicates of non-monotonic logics. We argue that this enables modelling the behaviour in close parallelism to the requirements. Such requirements engineering also results in direct and automatic translation to implementation, minimising software faults. We present our method and illustrated with a classical example. We also compare our approach with other state diagram methods, as well as Petri nets and Behavior Trees.

Keywords: Requirements engineering, Non-monotonic logics, Automatic code generation, Finite state machines, Behaviour modelling.

1 Introduction

We extend state transition diagrams in that we allow transitions to be labelled by statements of a non-monotonic logic, in particular Plausible Logic. We show that this has several benefits. First, it facilitates requirements engineering. Namely, we show this approach can be more transparent, clear, and succinct than other alternatives. Therefore, it enables better capture of requirements and this leads to much more effective system development. Furthermore, we show that such diagrams can be directly, and automatically translated into executable code, i.e. no introduction of failures in the software development process.

Finite automata have a long history of modelling dynamic systems and consequently have been a strong influence in the modelling of the behaviour of computer systems [25, Bibliographical notes, p. 113-114]. System analysis and design uses diagrams that represent behaviour of components or classes. State diagrams (or state machines) constitute the core behaviour modelling tool of object-oriented methodologies. In the early 90s state machines became the instrument of choice to model the behaviour of all the objects of a class. The Object-Modeling Methodology (OMT) [25, chapter 5] established state diagrams as the primary dynamic model. The Shlaer-Mellor approach established state models to capture the life cycle of objects of a given class [28]. Class diagrams capture the static information of all objects of the same class (what they know, what they store), but behaviour is essentially described in models using states and transitions. The prominence of OMT and Shlaer-Mellor has permeated into the most popular modelling language for object-orientation, the Unified Modeling Language (UML).

"A state diagram describes the behaviour of a single class of objects" [25, p. 90]. Although the state diagrams for each class do not describe the interactions and behaviour of several objects of diverse classes in action (for this, UML has collaboration diagrams and sequence diagrams), they constitute a central modelling tool for software engineering.

Although UML and its variants have different levels of formality, in the sense of having a very clear syntax and semantics, they aim for the highest formality possible. This is because their aim is to remove ambiguity and be the communication vehicle between requesters, stakeholders, designers, implementors, testers, and users of a software system. In particular *Executabel UML* [21] suggest the formal semantics enables models that are testable, and can be compiled into a less abstract programming language to target a specific implementation. Thus, they offer constraints very similar to the formal finite state machine models. For example, in a deterministic finite state machine, no two transitions out of the same state can be labelled with the same symbol. This is because formally, a deterministic finite state machine consists of a finite set of states, an input language (for events), and a transition function. The transition function indicates what the new state will be, given an input and the current state. Other adornments include signalling some states as *initial* and some as *final*. However, a fundamental aspect of finite state machines is that the transition function is just that, a function (mathematically, a function provides only one value of the codomain for each value in the domain).

Granted that the model can be extended to a non-deterministic machine, where given an input and a state, a set of possible states is the outcome of the transition. In this case, the semantics of the behaviour has several interpretations. For example, in the theory of computation, the so-called *power set construction* shows that non-deterministic and deterministic finite state machines are equivalent. However, other semantics are possible, such as multi-threaded behaviour. Therefore, as a modelling instrument in software engineering, it is typically expected that the conditions emanating from a state are mutually exclusive and exhaustive. "All the transitions leaving a state must correspond to different events" [25, p. 89]. Namely, if the symbol c_i is a Boolean expression representing the guard of the transition, then $\bigvee_{i=1}^{n} c_i = \mathsf{true}$ (the exhaustive condition), and $c_i \wedge c_j = \mathsf{false},\ \forall\, i \neq j$ (the exclusivity condition). In fact, Shlaer-Mellor also suggest the analysis should make use of the *State Transition Table* (STT) [28] "to prevent one from making inconsistent statements" [28, p. 52] and they provide an illustration where two transitions out of the same state and labelled by the same event are corrected using the table.

Recently, the software engineering community has been pushing for *Requirements Engineering* (RE) [16], concerned with identifying and communicating the objectives of a software system, and the context in which it will be used [23]. Hence, RE identifies and elicits the needs of users, customers, and other stakeholders in the domain of a software system. RE demands a careful systematic approach. Significant effort is to be placed on rigorous analysis and documented specification, especially for security or safety critical systems. RE is important because a requirement not captured early may result in a very large effort to re-engineer a deployed system.

2 Declarative Requirements

An ambition of both artificial intelligence (AI) and software engineering is to be able to only specify *what* we want, without having to detail *how* to achieve this. The motivation for our approach has a similar origin. We aim at producing a vehicle of communication that would enable the specification of behaviour without the need for imperative programming tools. Therefore, we want to use a declarative formalism (a similar ambition has lead to the introduction of logic programming and functional programming). Non-monotonic logic is regarded as quite compatible with the way humans reason and express the conditions and circumstances that lead to outcomes, as well as a way to express the refinements and even exceptions that polish a definition for a given concept. In fact, non-monotonic reasoning is regarded as one of the approaches to emulate common-sense reasoning [26]. We illustrate that the addition of this declarative capability to state transition diagrams for capturing requirements is beneficial because the models obtained are much simpler (a fact necessary to ensure that the natural language description has indeed been captured). This way, we only need to specify the *what* and can have all of the *how* within an embedded system generated automatically.

With our approach, modelling with state-diagrams is sufficient to develop and code behaviours. The semantics of a *state* is that it is lasting in time, while a *transition* is assumed to be instantaneous. The state-diagram corresponds closely to the formalism of finite state machines (defined by a set S of states, a transition function $t : S \times \Upsilon \rightarrow S$, where Υ denotes a possible alphabet of input symbols). In our case, we can specify the behaviour by the table that specifies the transition function t (analogously to an STT [28]).

We still use the notion of an initial state s_0, because, in our infrastructure that implements these ideas [3], in some *external-state* transition, behaviours must be able to reset themselves to the initial state. A final state is not required, but behaviours should be able to indicate completion of a task to other modules. Now, current practice for modelling with finite state machines assumes that transitions are labelled by events. In both the Shlaer-Mellor approach and OMT [25], transitions are labelled by events only. For example, in Fig. 1, a transition is labelled by the event `ball_visible` (e.g. a sensor has detected a ball). A slight extension is to allow labels to be a decidable Boolean condition (or expression) in a logic with values true or false (that is, it will always be possible to find the value of the condition guarding the transition). This easily captures the earlier model because rather than labelling by event e, we label by the Boolean expression e_HAS_OCCURRED. Our extension to behaviour modelling extends this further with the transition labels being any sentence in the non-monotonic logic. Replacing the guarding conditions with statements in the non-monotonic logic incorporates reasoning into the reactive[1] nature of state machines. Since our logics model reasoning (and beliefs like "in this frame vision believes there is no ball"), they are better suited to model these transitions (they may even fuse contradicting beliefs reported by many sensors and modules in a deterministic way), and gracefully handle situations

[1] In the agent model *reactive* systems are seen as an alternative to logic-based systems that perform planning and reasoning [37].

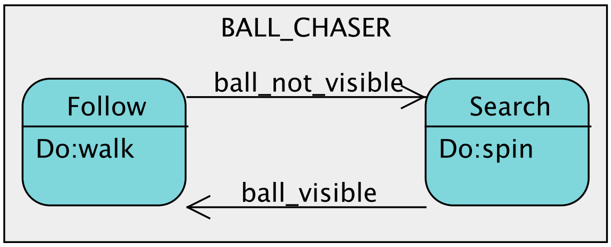


Fig. 1. Simple Finite State Diagram

with incomplete (or superfluous) information without increasing the cognitive load of the behaviour designer.

The designer can separate the logic model from the state-transition model. Moreover, the designer would not be required to ensure the exhaustive nature of the transitions leading out from a state; as priorities can indicate a default transition if conditions guarding other transitions cannot be decided.

"State diagrams have often been criticized because they allegedly lack expressive power and are impractical for large problems" [25, p. 95]. However, several techniques such as nesting state diagrams, state generalisation, and event generalisation were used in OMT to resolve this issue. We have shown elsewhere [4] (1) how the technique of nested state diagrams (e.g. team automata [29,10]) handle complexity, and (2) that there is an equivalence between Behavior Trees and state machines, mitigating the problem of expressive power of state diagrams for larger systems. In fact our approach follows the very successful modelling by finite state machines [30] that has resulted in *stateWORKS*, a product used for over a decade in the engineering of embedded systems software [31]. In *stateWORKS*, transitions are labelled by a small subset of propositional logic, namely *positive logic algebra* [31], which has no implication, and no negation (only OR and AND). Thus our use of a non-monotonic logic is a significant variation.

3 Plausible Logic

Non-monotonic reasoning [1] is the capacity to make inferences from a database of beliefs and to correct those as new information arrives that makes previous conclusions invalid. Although several non-monotonic formalisms have been proposed [1], the family of non-monotonic logics called Defeasible Logics has the advantage of being designed to be implementable. Billington [2] compared the main members of this family, including *Plausible Logic* (PL), indicating their uses and desirable properties. Although the most recent member of this family, CDL [2], has some advantages over PL [5], the differences are not significant for the purposes this paper. We shall therefore use PL as its corresponding programming language, DPL, is more advanced. If only factual information is used, PL essentially becomes classical propositional logic. But when determining the provability[2] of a formula, the proving algorithms in PL can deliver three

[2] Provability here means determining if the formula can be verified or proved.

values (that is, it is a three-valued logic), $+1$ for a formula that has been proved, -1 for a formula that has been disproved, and 0 when the formula cannot be proved and attempting so would cause an infinite loop. Another very important aspect of PL is that it distinguishes between formulas proved using only factual information and those using plausible information. PL allows formulas to be proved using a variety of algorithms, each providing a certain degree of trust in the conclusion.

In PL all information is represented by three kinds of rules and a priority relation between those rules. The first type of rules are strict rules, denoted by the strict arrow $\rightarrow$ and used to model facts that are certain. For a rule $A \rightarrow l$ we should understand that if all literals in A are proved then we can deduce l (this is simply ordinary implication). A situation such as *Humans are mammals* will be encoded as $human(x) \rightarrow mammal(x)$. Plausible rules $A \Rightarrow l$ use the plausible arrow $\Rightarrow$ to represent a plausible situation. If we have no evidence against l, then A is sufficient evidence for concluding l. For example, we write *Birds usually fly* as $bird(x) \Rightarrow fly(x)$. This records that when we find a bird we may conclude that it flies unless there is evidence that it may not fly (e.g. if it is a penguin). Defeater rules $A \rightharpoonup \neg l$ say if A is not disproved, then it is too risky to conclude l. An example is *Sick birds might not fly* which is encoded as $\{sick(x), bird(x)\}$ $\rightharpoonup \neg fly(x)$. Defeater rules prevent conclusions that would otherwise be too risky (e.g. from a chain of plausible conclusions).

Finally, a priority relation $>$ between plausible rules $R_1 > R_2$ indicates that R_1 should be used instead of R_2. The following example demonstrates the expressive power of this particular aspect of the formalism:

$\{\} \rightarrow quail(Quin)$	*Quin is a quail*
$quail(x) \rightarrow bird(x)$	*Quails are birds*
$R_1 : bird(x) \Rightarrow fly(x)$	*Birds usually fly*

From the rule R_1 above, one would logically accept that $Quin$ flies (since $Quin$ is a $bird$).

$\{\} \rightarrow quail(Quin)$	*Quin is a quail*
$quail(x) \rightarrow bird(x)$	*Quails are birds*
$R_2 : quail(x) \Rightarrow \neg fly(x)$	*Quails usually do not fly*

However, from R_2, we would reach the (correct) conclusion that Quin usually does not fly. But what if both knowledge bases are correct (both R_1 and R_2 are valid)? We perhaps can say that R_2 is more informative as it is more specific and so we add $R_2 > R_1$ to a knowledge base unifying both. Then PL allows the agent to reach the proper conclusion that Quin usually does not fly, while if it finds another bird that is not a quail, the agent would accept that it flies. What is important to note here is that if the strict rules are consistent, all proofs within PL will also be consistent. I.e. it will never be possible to prove both a literal l and its negation $\neg l$ at the same time.

Note that Asimov's famous Three Laws of Robotics are a good example of how humans describe a model. They define a general rule, and the next rule is a refinement. Further rules down the list continue to polish the description. This style of development is not only natural, but allows incremental refinement. Indeed, the knowledge elicitation mechanism known as *Ripple Down Rules* [8] extracts knowledge from human experts

Table 1. One-Minute Microwave Oven Requirements

Req.	Description
R1	There is a single control button available for the use of the oven. If the oven is closed and you push the button, the oven will start cooking (that is, energise the power-tube) for one minute.
R2	If the button is pushed while the oven is cooking, it will cause the oven to cook for an extra minute.
R3	Pushing the button when the door is open has no effect.
R4	Whenever the oven is cooking or the door is open, the light in the oven will be on.
R5	Opening the door stops the cooking.
R6	Closing the door turns off the light. This is the normal idle state, prior to cooking when the user has placed food in the oven.
R7	If the oven times out, the light and the power-tube are turned off and then a beeper emits a warning beep to indicate that the cooking has finished.

by refining a previous model by identifying the rule that needs to be expanded by detailing it more.

4 A Classical Example

We proceed here to illustrate our approach with an example that has been repeatedly used by the software engineering community, e.g. [9,22,28,33,20]. This is the so called one-minute microwave oven [28]. Table 1 shows the requirements as presented by Myers and Dromey [22, p. 27, Table 1]. Although this is in fact not exactly the same as the original by Shlaer and Mellor [28, p. 36], we have chosen the former rather than the latter because we will later compare with Behavior Trees regarding model size and direct code generation.

4.1 Microwave in Plausible Logic

Because we have a software architecture that handles communication between modules through a decoupling mechanism named the *whiteboard* [3], we can proceed at a very high level. We assume that sensors, such as the microwave button, are hardware instruments that deposit a message on the whiteboard with the signature of the depositing module and a time stamp. Thus, events like a button push or actions such as energising the microwave tube are communicated by simply placing a message on the whiteboard[3]. Thus, knowledge of an event like a button push simply exists because a corresponding message has appeared on the whiteboard. Similarly, an action like energising the microwave tube is triggered by placing a different message on the whiteboard. The driver for the corresponding actuator then performs an action for this particular message as soon as it appears on the whiteboard.

However, here the label `cook` for the transition of the state NOT_COOKING to the state COOKING and the label `~cook` from COOKING to NOT_COOKING are not necessarily events. They are consequents in a logic model. For example, `~cook` is an output of such a model that acts as the cue to halt the cooking. The logic model will specify

[3] Matters are a bit more complex, as messages on the whiteboard expire or are consumed, and for actuators, they could have a priority and thus actuators can be organised with "subsumption" [7].

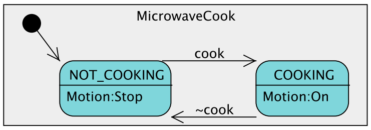
```
% MicrowaveCook.d

name{MICROWAVECOOK}.

input{timeLeft}.
input{doorOpen}.

C0: {}        => ~cook.
C1: timeLeft =>  cook. C1 > C0.
C2: doorOpen => ~cook. C2 > C1.

output{b cook, "cook"}.
%
```

(a) DPL for 2-state machine controlling tube, fan, and plate.

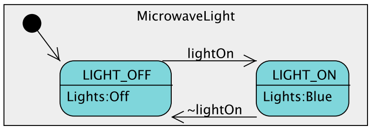
```
% MicrowaveLight.d

name{MICROWAVELIGHT}.

input{timeLeft}.
input{doorOpen}.

L0: {}        => ~lightOn.
L1: timeLeft =>  lightOn. L1 > L0.
L2: doorOpen =>  lightOn. L2 > L0.

output{b lightOn, "lightOn"}.
output{b ~lightOn, "lightOff"}.
```

(b) DPL for 2-state machine controlling the light.

Fig. 2. Simple theories for 2-state machines



(a) A 2-state machine for controlling tube, fan, and plate.

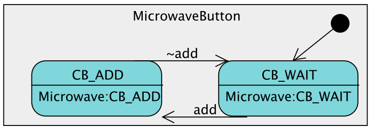


(b) A 2-state machine for controlling the light.

Fig. 3. Simple 2-state machines control most of the microwave

the conditions by which this cue is issued. Fig. 2a shows the logic model in the logic programming language DPL that implements PL. In the case of the microwave oven requirements, for the purposes of building a model, a system analyst or software engineer would first identify that there are two states for various actuators. When the oven is cooking, the fan is operating, the tube is energised and the plate is rotating. When the oven is not cooking, all these actuators are off. The approach can be likened to arranging the score for an orchestra: all these actuators will need the same cues from the conductor (the control) and, in this example, all switch together from the state of COOKING to the state of NOT_COOKING and vice versa. They will all synchronously consume the message to be off or to be on. Thus, we have a simple state diagram to model this (Fig. 3a). By default, we do not have the conditions to cook. This is Rule `C0` in the logic model (called a *theory*) relevant to the cooking actuators. However, if there is time left for cooking, then we have the conditions to cook (Rule `C1`) and this rule takes priority over `C0`. However, when the door opens, then we do not cook: `C2` takes priority over `C1`.

The light in the microwave is on when the door is open as well as when the microwave is cooking. So, the cues for the light are not the same as those for energising the tube. However, the light is in only one of two states LIGHT_OFF or LIGHT_ON. The default state is that the light is off. This is Rule `L0` in the *theory* for the light (see Fig. 2b). However, when cooking the light is on. So Rule `L1` has priority over `L0`. There is a further condition that overwrites the state of the light being off, and that is when the door is open (Rule `L2`). Note that in this model, the two rules `L1` and `L2` override the default Rule `L0`, while in the model for cooking the priorities caused each new rule

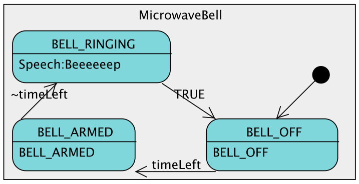


(a) State machine

```
% MicrowaveButton.d

name{MICROWAVEBUTTON}.

input{doorOpen}.
input{buttonPushed}.

CB0: {}           => ~add.
CB1: buttonPushed =>  add. CB1 > CB0.
CB2: doorOpen     => ~add. CB2 > CB1.

output{b add, "add"}.
```

(b) DPL theory

Fig. 4. The modelling of the button's capability to add to the timer



(a) Bell state machine

```
% MicrowaveBell.d

name{MicrowaveBell}.

input{timeLeft}.
```

(b) DPL theory for the bell.

Fig. 5. The modelling of the bell's capability to ring when the time expires

to refine the previous rule. The control button (Fig. 4) also has two states. In one state, CB_ADD, the time left can be incremented, while in the other state, pushing the button has no effect. Again, between these two states we place transitions labelled by an expression of PL (in all cases, simple outputs of a theory). The control button does not add time unless the button is pushed. This is reflected by Rule `CB0` and Rule `CB1` below and the priority that `CB1` has over `CB0`. When the door is open, pushing the button has no effect; this is Requirement R3 and expressed by Rule `CB2` and its preference over `CB1`. Because we already have defined that a push of a button adds time to the timer (except for the conditions already captured), if the button is not pushed, then we do not add time. This is a strict rule that in DPL is expressed by a disjunction.

The final requirement to model is the bell, which is armed while cooking, and rings when there is no time left. This is the transition `~timeLeft` from BELL_ARMED to BELL_RINGING in Fig. 5. After ringing, the bell is off, and when cooking time is added to the timer, it becomes armed. The logical model is extremely simple, because the condition that departs from BELL_ARMED to BELL_RINGING is the negation of the one that moves from to BELL_OFF to BELL_ARMED. Moreover, we always move from BELL_RINGING to BELL_OFF. The most important aspect of this approach is that this is *all* the software analysis required in order to obtain the working program.

4.2 Translation into Code

Once the high level model has been established in DPL, translation into code is straightforward. A Haskell proof engine implementation of DPL allows the interpretation and formal verification of the developed rule sets [5]. This implementation was extended to include a translator that generates code that can be used in C, C++, Objective-C, C# and

```
#define dontCook ( \
      doorOpen \
   || !timeLeft \
)

#define cook ( \
   !doorOpen && timeLeft \
)
```

(a) Tube, fan and plate

```
#define lightOff ( \
   !doorOpen && !timeLeft \
)

#define lightOn ( \
      doorOpen \
   || timeLeft \
)
```

(b) The light

```
#define add ( \
 buttonPushed && !doorOpen \
)


#define wait ( \
 !buttonPushed \
)
```

(c) Button and timer

```
#define noTimeLeft ( \
   !timeLeft \
)
```

(d) The bell

Fig. 6. Translated C expressions for transitions

Java. The Haskell translator creates optimised Boolean expressions through the truth tables generated from the DPL rules. These Boolean expressions are then written out as C code that can directly be compiled and linked with libraries and application code. Incidentially, the syntax for Boolean expressions is not only the same in supersets of C (such as C++ and Objective-C), but also in modern, related programming languages such as Java (at this stage, expressions are generated using the `#define` preprocessor syntax, that is not supported directly in Java, but the actual expression can easily be extracted using a script or even copy and paste). This code can then directly be used as a header file for a generic embedded system state machine to test the transition conditions. Subsequent refinements of the rules do not require any changes to the generic state machine code. A simple recompilation against the updated, generated header files is sufficient to update the behaviour of the state machine. Moreover, the compiled rules table can be dumped into a simple text file that can be loaded into the running proof engine via a dynamic CDL whiteboard bridge. This way, the logic or any of its rules can be changed at run time, without interrupting the current behaviour. Figure 6 shows the DPL theories for the state diagram transitions translated into C by the Haskell parser.

5 Evaluation

The original approaches to modelling behaviour with finite state diagrams [25,28] had little expectation that the models would directly translate to implementations without the involvement of programmers using imperative object-oriented programming languages. However, the software development V-model [35] has moved the focus to requirements modelling, and then directly obtaining a working implementation, because this collapses the requirements analysis phase with the verification phase. There are typically two approaches. First, emulating or simulating the model, which has the advantage that software analysts can validate the model and implementation by running as many scenarios as possible. The disadvantage is the overhead incurred through the interpretation of the model, rather than its compilation. The second approach consists of generating code directly from the model [31,30]. This removes the overhead of interpreting at run-time the modelling constructs. Approaches to the automatic execution

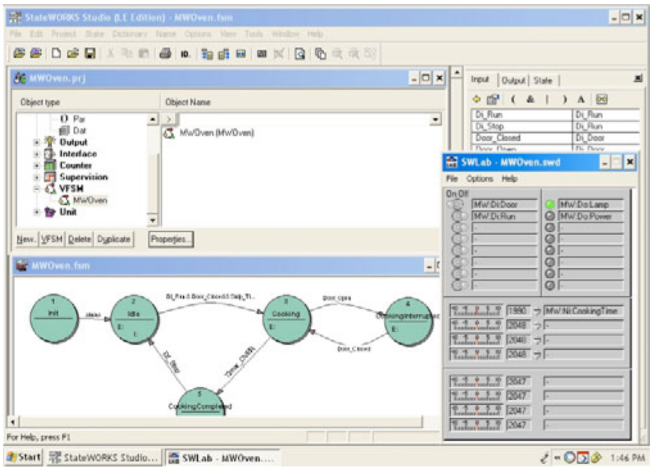


Fig. 7. The execution of the example model provided in the demo version of *stateWorks* for a microwave oven

or translation of models for the behaviour of software have included the use of UML state diagrams for generating code [20], the automatic emulation or code generation from Petri nets [13], and the automatic emulation or code generation from *Behavior Trees* [33,34,32]. A more recent trend is *models@run.time*, where “there is a clear pressure arising for mirroring the problem space for more declarative models” [6].

5.1 Contrast with State Diagrams

Simulation and direct generation of code from a state diagram is clearly possible, since one only needs to produce generic code that reads the transition table (encoded in some standard form), then deploy and interpret that repeatedly by analysing the events received as well as the current state, and moving to the proper subsequent state. This has been suggested for UML [20] and is the basis of the design pattern `state` [18, p. 406]. However, while Finite State Machines continue to enjoy tremendous success [31], “there is no authoritative source for the formal semantics of dynamic behavior in UML” [36]. The best example for the success of Finite State Machines is *stateWorks* (www.stateworks.com) and its methodology [30]. We have downloaded the 60-day free license of *stateWorks Studio* and the *SWLab* simulator (Fig. 7). Note that this finite state machine has only 5 states (the documentation of this example with *stateWorks* admits the model has issues, e.g. “to reset the system for the next start, we have to open and then close the door” and “the control system always starts even when the timeout value is 0”). These issues can be fixed but additional infrastructure is necessary, including ‘counters’ and ‘switch points’ as well as usage of the ‘real time database (RTDN)’. The *stateWorks* example describes generic microwave behaviour and needs some more polishing to capture more detailed requirements e.g. those outlined in Table 1. Since we argue in favour of using finite state machines for modelling behaviour, this tool

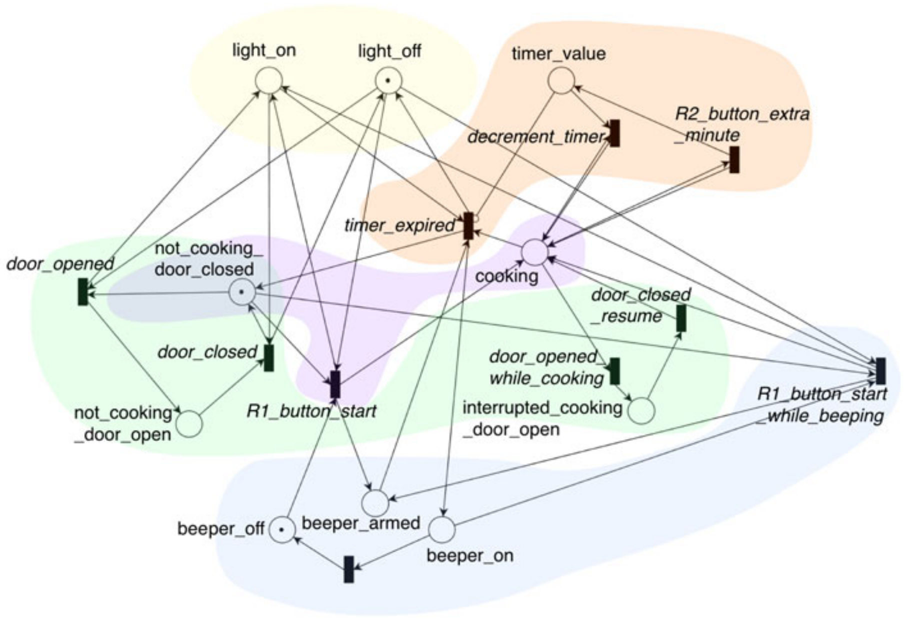


Fig. 8. Capturing the requirements in Table 1 as a Petri Net, grouped by the various components of the microwave

concurs with that approach. However, our evaluation confirms that using PL is more powerful and closer to the original specification than the "positive-logic" transitions in *stateWorks*.

5.2 Contrast with Petri Nets

Petri Nets [24,15] provide a formal model for concurrency and synchronisation that is not readily available in state diagrams. Thus, they offer the possibility of modelling multi-threaded systems that support requirements for concurrency. Early in the modelling effort Petri Nets were dismissed: "Although they succeed well as an abstract conceptual model, they are too low level and inexpressive to be useful to specify large systems" [25, p. 144]. However, because it is quite possible to simulate or interpret a Petri Net (or to generate code directly from it), they continue to be suggested as an approach to directly obtain implementations from the coding of the requirements [14,19,17,27].

We used PIPE 2.5 (`pipe2.sf.net`) to construct a model of the microwave as per the requirements in Table 1 (see Fig. 8). It becomes rapidly apparent that the synchronisation of states between components of the system forces the display of connectors among many parts in the layout, making the model hard to grasp. Even if we consider incremental development, each new requirement adds at least one place and several transitions from/to existing places. Thus, we tend to agree that even for this small case of the one-minute microwave, the Petri Net approach seems too low level, and the models do not provide a level of abstraction to assist in the behaviour engineering of the system. One advantage of Petri Net models is that there are many tools and algorithms for different aspects of their validation. For example, once a network is built with PIPE, this software has algorithms to perform GSPN Analysis, FSM analysis, and Invariant Analysis. However, even for this simple, illustrative example, the model is not a suitable input for any of the verification analysis in PIPE. On a positive note, some first-order logics have been included in transitions to create *Predicate/Transition Nets* [12,11]. Hence, we believe our approach to use a non-monotonic logic can be applied to Petri Nets.

5.3 Contrast with Behavior Trees

Behavior Trees [32] is another powerful visual approach for *Behaviour Engineering* (the systematic progression from requirements to the software of embedded systems). The approach provides a modelling tool that constructs acyclic graphs (usually displayed as rooted tree diagrams) as well as a Behavior Modelling Process [9,22] to transform natural language requirements into a formal set of requirements. The *Embedded Behavior Runtime Environment (eBRE)* [22] executes Behavior Tree models by applying transformations and generating C source code. The tool *Behavior Engineering Component Integration Environment (BECIE)* allows Behavior Trees to be drawn and simulated. Proponents of Behavior Trees argue that these diagrams enable requirements to be developed incrementally and that specifications of requirements can be captured incrementally [34,33]. The classic example of the one-minute microwave has also been extensively used by the Behavior Tree community [33,9,22]. Unfortunately, for this example, Behavior Trees by comparison are disappointing. In the initial phase of the method, requirements R1, R2, R5, R6 in Table 1 use five boxes [33,9], while R3 and R4 demand four. Six boxes are needed for requirement R7 [33,9]. Then, the *Integration Design Behavior Tree (DBT)* demands 30 nodes (see [9, p. 9] and [33, Fig. 5]). By the time it becomes a model for *eBRE* the microwave has 60 nodes and 59 links! [22, Fig. 6] and the *Design Behavior Tree* [22, Fig. 8] does not fit legibly on an A4 page. Sadly, the approach seems to defeat its purpose, because on consideration of the system boundaries [22, Fig. 7], outputs to the alarm are overlooked. Moreover, the language for logic tests in the tools for Behavior Trees is far more limited than even the ‘positive-logic’ of *stateWORKS*. The equivalence between finite state machines and Behavior Trees [4] is based on the observation that Behavior Trees correspond to the depth-first search through the sequence of the states of the system behaviour control. It is not surprising that the notion needs far more nodes and connections than the corresponding finite state machine.

6 Final Remarks

We stress two more aspects of the comparison. First, the approaches above attempt, in one way or another, to construct the control unit of the embedded system, and from it the behaviour of all of its components. This implies that the control unit has a state space that is a subset of the Cartesian product of the states of the components. Our approach is more succinct not only because of a more powerful logic to describe state transition, but

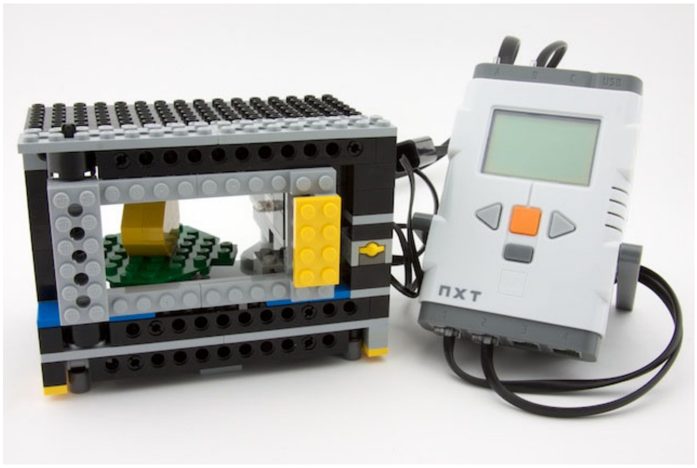


Fig. 9. Hardware running Java generated code

because our software architecture decouples control into descriptions for the behaviour of components. Second, our experience with this approach and the development of non-monotonic models show that capturing requirements is structured and incremental, enabling iterative refinement. That is, one can proceed from the most general case, and produce rules and conditions for more special cases. We ensured that our method delivers executable embedded systems directly from the modelling by implementing an oven where the hardware is constructed from LEGO Mindstorm pieces, sensors, and actuators (see Fig. 9). As with *eBRE*, we output Java source code but execute a finite state machine. The execution then is verified because of the clear connection between the model and the source code (as well as testing it on the hardware)[4].

References

1. Antoniou, G.: Nonmonotonic Reasoning. MIT Press, Cambridge (1997)
2. Billington, D.: Propositional clausal defeasible logic. In: Hölldobler, S., Lutz, C., Wansing, H. (eds.) JELIA 2008. LNCS (LNAI), vol. 5293, pp. 34–47. Springer, Heidelberg (2008)
3. Billington, D., Estivill-Castro, V., Hexel, R., Rock, A.: Architecture for hybrid robotic behavior. In: Corchado, E., Wu, X., Oja, E., Herrero, Á., Baruque, B. (eds.) HAIS 2009. LNCS, vol. 5572, pp. 145–156. Springer, Heidelberg (2009)
4. Billington, D., Estivill-Castro, V., Hexel, R., Rock, A.: Plausible logic facilitates engineering the behaviour of autonomous robots. In: Proceedings of the IASTED International Conference on Software Engineering, Innsbruck, Austria, The International Association of Science and Technology for Development (February 2010)
5. Billington, D., Rock, A.: Propositional plausible logic: Introduction and implementation. Studia Logica 67, 243–269 (2001)

[4] See www.youtube.com/watch?v=iEkCHqSfMco for the system in operation. The corresponding Java sources and incremental Petri net stages for Fig. 8 are at vladestivill-castro.net/ additions.tar.gz as well as material from [4].

6. Blair, G., Bencomo, N., Frnce, R.B.: Models@run.time. IEEE Computer 42(10), 22–27 (2009)
7. Brooks, R.A.: Intelligence without reason. In: Myopoulos, R., Reiter, R. (eds.) Proceedings of the 12th International Joint Conference on Artificial Intelligence, ICJAI 1991, San Mateo, CA, pp. 569–595. Morgan Kaufmann Publishers, Sydney (1991) ISBN 1-55860-160-0
8. Compton, P.J., Jansen, R.: A philosophical basis for knowledge acquisition. Knowledge Acquisition 2(3), 241–257 (1990)
9. Dromey, R.G., Powell, D.: Early requirements defect detection. TickIT Journal 4Q05, 3–13 (2005)
10. Ellis, C.: Team automata for groupware systems. In: GROUP 1997: Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work, pp. 415–424. ACM, New York (1997)
11. Genrich, H.J.: Predicate/transition nets. In: Jensen, K., Rozenberg, G. (eds.) High-level Petri Nets, Theory and Applications, pp. 3–43. Springer, Heidelberg (1991)
12. Genrich, H.J., Lautenbach, K.: The analysis of distributed systems by means of predicate/transition-nets. In: Kahn, G. (ed.) Semantics of Concurrent Computation. LNCS, vol. 70, pp. 123–147. Springer, Heidelberg (1979)
13. Girault, C., Valk, R.: Petri Nets for System Engineering: A Guide to Modeling, Verification, and Applications. Springer-Verlag New York, Inc., Secaucus (2001)
14. Gold, R.: Petri nets in software engineering. Arbeitsberichte Working Papers, Fachhochschule Ingolstadt, University of Applied Sciences (June 2004)
15. Holloway, L.E., Kroch, B.H., Giua, A.: A survey of Petri net methods for controlled discrete event systems. Discrete Event Dynamic Systems: Theory and Applications 7, 151–190 (1997)
16. Hull, E., Jackson, K., Dick, J.: Requirements Engineering, 2nd edn. Springer, USA (2005)
17. Lakos, C.: Object oriented modelling with object petri nets. In: Agha, G., De Cindio, F., Rozenberg, G. (eds.) APN 2001. LNCS, vol. 2001, pp. 1–37. Springer, Heidelberg (2001)
18. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice-Hall, Inc., Englewood Cliffs (1995)
19. Lian, J., Hu, Z., Shatz, S.M.: Simulation-based analysis of UML statechart diagrams: Methods and case studies. The Software Quality Journal 16(1), 45–78 (2008)
20. Mellor, S.J.: Embedded systems in UML. OMG White paper (2007), http://www.omg.org/news/whitepapers/, label: We can generate Systems Today
21. Mellor, S.J., Balcer, M.: Executable UML: A foundation for model-driven architecture. Addison-Wesley Publishing Co., Reading (2002)
22. Myers, T., Dromey, R.G.: From requirements to embedded software - formalising the key steps. In: 20th Australian Software Engineering Conference (ASWEC), Gold Cost, Australia, April 14-17, pp. 23–33. IEEE Computer Society, Los Alamitos (2009)
23. Nuseibeh, B., Easterbrook, S.M.: Requirements engineering: a roadmap. In: ICSE - Future of SE Track, pp. 35–46 (2000)
24. Peterson, J.L.: Petri nets. Computer Surveys 9(3), 223–252 (1977)
25. Rumbaugh, J., Blaha, M.R., Lorensen, W., Eddy, F., Premerlani, W.: Object-Oriented Modelling and Design. Prentice-Hall, Inc., Englewood Cliffs (1991)
26. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice-Hall, Inc., Englewood Cliffs (2002)
27. Saldhana, J.A., Shatz, S.M.: Uml diagrams to object petri net models: An approach for modeling and analysis. In: International Conference on Software Engineering and Knowledge Engineering (SEKE), Chicago, pp. 103–110 (July 2000)
28. Shlaer, S., Mellor, S.J.: Object lifecycles: modeling the world in states. Yourdon Press, Englewood Cliffs (1992)

29. ter Beek, M.H., Ellis, C.A., Kleijn, J., Rozenberg, G.: Synchronizations in team automata for groupware systems. Computer Supported Cooperative Work (CSCW) 12(1), 21–69 (2003)
30. Wagner, F., Schmuki, R., Wagner, T., Wolstenholme, P.: Modeling Software with Finite State Machines: A Practical Approach. CRC Press, NY (2006)
31. Wagner, F., Wolstenholme, P.: Modeling and building reliable, re-useable software. In: IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2003), pp. 277–286. IEEE Computer Society, Los Alamitos (2003)
32. Wen, L., Colvin, R., Lin, K., Seagrott, J., Yatapanage, N., Dromey, R.G.: "Integrare", a collaborative environment for behavior-oriented design. In: Luo, Y. (ed.) CDVE 2007. LNCS, vol. 4674, pp. 122–131. Springer, Heidelberg (2007)
33. Wen, L., Dromey, R.G.: From requirements change to design change: A formal path. In: 2nd International Conference on Software Engineering and Formal Methods (SEFM 2004), Beijing, China, September 28-30, pp. 104–113. IEEE Computer Society, Los Alamitos (2004)
34. Wen, L., Kirk, D., Dromey, R.G.: A tool to visualize behavior and design evolution. In: Di Penta, M., Lanza, M. (eds.) 9th International Workshop on Principles of Software Evolution (IWPSE 2007), in conjunction with the 6th ESEC/FSE joint meeting, Dubrovnik, Croatia, September 3-4, pp. 114–115. ACM, New York (2007)
35. Wiegers, K.E.: Software Requirements, 2nd edn. Microsoft Press, Redmond (2003)
36. Winter, K., Colvin, R., Dromey, R.G.: Dynamic relational behaviour for large-scale systems. In: 20th Australian Software Engineering Conference (ASWEC 2009), Gold Cost, Australia, April 14-17, pp. 173–182. IEEE Computer Society, Los Alamitos (2009)
37. Wooldridge, M.: An Introduction to Multiagent Systems. John Wiley & Sons, NY (2002)

Towards a Better Change Impact Analysis in Architecture Description Languages

Mohamed Oussama Hassan, Laurent Deruelle, Adeel Ahmad, and Henri Basson

Université Lille Nord de France, Laboratoire d'Informatique Signal et Image de la Côte d'Opale
50, rue Ferdinand Buisson BP 719, 62228 Calais Cedex, France
{ahmad,deruelle,hassan,basson}@lisic.univ-littoral.fr
http://www-lisic.univ-littoral.fr/

Abstract. This chapter proposes a multi-modeling approach destined to better control the software evolution. The presented approach follows formal models on software architecture and source code level. It formalizes the elements of software architecture, their interdependent relationships and their source codes to analyze the impact propagation of an intended change. The constituents of these models are evaluated with a reasoning based expert system. The expert system is validated as a platform based on eclipse plug-ins to analyze the architecture description languages. The software architecture and source codes are parsed to generate the facts of the distributed knowledge-based system, which executes change propagation rules to evaluate the impact of a change performed on distributed components.

Keywords: Distributed software Architecture analysis, Change propagation, Knowledge-based system, Software evolution, Change impact analysis.

1 Introduction

The practices of software engineering are expected to respond the development or evolution of distributed applications on heterogeneous platforms with less delay, lower cost, and better quality. The applications are continuously growing in size and complexity making the change more difficult to control and to manage. Moreover, the software development nowadays is distributed using heterogenous multiple languages. Hence, it makes software more vulnerable to changes and evolve. It is largely admitted that the quality of large applications can be improved using formalized architectural models at the earlier phases of requirements specifications and design. Therefore, over the past decade software architecture has received increasing attention as an important subfield of software engineering aiming to face the growing size and complexity of software [12].

It is inevitable that a software undergoes some changes in its lifetime. A change introduced to one component of an application has often effects on several others parts. If this process is uncontrolled, changes may have unexpected side effects or complex implications on the behavior of the whole system. The cost to fix an error or an incoherence, resulting from changes during requirements, or early design phases, is largely lower than the cost of correcting the same error found during system testing or in production [4]. Use of software tools is increasing estimate the effort required to incorporate a change in developed software. The change impact analysis refers to track the effects of a change by providing visibility of the potential effects of the proposed change before it is implemented. The traditional change impact analysis tools trace the changes in a limited part of distributed software. The targeted part is often developed in one programming language. The software change impact analysis and tracing its propagation in distributed software are difficult activities of successful change incorporation process. These two activities have become most difficult with the emergence of new processes (including iterative development, egile development, and software evolution) in classical software development and maintenance process.

We focus mainly on the incremental changes to support impact analysis and trace its propagation in distributed software applications. It can greatly help maintainers to determine appropriate actions to take with respect to change in decisions, schedule plans, cost and resource estimations. Our approach, primarily, describes the architectural organization of software constituents on high level. Later, this description becomes primordial to trace the change impact propagation in the whole software and support evolution.

Software research community is doing substantial study of the evolution mechanism in the software. We note that the specification of the evolution can be realized by the semi-automatic tools that serve for the provision of meta-information of related software artifacts concerning a change. In this work, we propose a model, called Architecture Software Components Model (ASCM), to represent and to unify the major concepts defined in a large number of existing architecture description languages (ADL). The ASCM is coupled to another model called Source Code Structural Model (SCSM) [7,6,1] in order to represent relationships between components from the architecture level and those of the source code level. The two models represent the architecture and the source codes as graphs, in which the nodes are components and the edges are relationships. The graphs are distributed according to the location of the architecture specification and the relevant source codes. The main purpose of our models is to provide a basis to track the change propagation process using adequate graphs on which we define formal propagation rules.

The rest of the chapter is organized as follows: Section 2 presents related works for distributed software architecture evolution. Section 3 describes our formal model ASCM to represent the common concepts defined in the major ADLs. Section 4 proposes a typology of formalized change operations applied on software architecture, which permits the change impact analysis. Section 5 discusses the formal process to deal with the change propagation in distributed software. Section 6 presents our integrated platform that implements the formal models and a knowledge-based system to support the change propagation process. Section 7 gives a scenario of a change impact analysis performed on a distributed software architecture. Finally, section 8 provides some conclusions and perspectives.

2 Related Works

Software architecture has emerged as an area of intense research over the last decade. Many approaches have been proposed to deal with architectural specification and analysis [10,3,5]. In these approaches, a large number of ADLs have been developed to represent different aspects of distributed software architectures based on the middlewares. The ADLs do not provide features to deal with the evolution management of architecture description in a distributed environment.

Models are cost-effective mediums for the representation of actual applications. Use of models for change impact analysis is a widely applied approach in the domain of software evolution. A lot of work has been done for studying the evolution mechanism in the ADLs. We note that the specification of the evolution is realized by the ADL that serves for the architecture description. Therefore, the evolution management is included in the architectural specification, and so it will be difficult to be distinguished. Moreover, it is almost impossible to identify all possible evolution operations that may occur when specifying architecture.

It is hard to deal with the non planned evolution, considering the difficulty to study and analyze automatically the changes impact without incoherence. The number of propositions dealing with architecture change impact analysis is very restricted.

3 Distributed Software Architecture Modeling: ASCM

A lot of research work has been focussing to provide architectural modeling [8], in which the following common high-level elements are prominent:

1. *The Components* are units of computation or data stores. For example, a component can be a Thread, a Procedure or a whole application. An instance of a component may be distributed on many sites and may interoperate with other distributed components.
2. *The Connectors* are architectural building blocks used to represent the interactions between local and distributed components. For example, a $Thread$ deployed on one site can be connected to another $Thread$ deployed on another site to send messages. The second component may treat messages using local components.
3. *The Configurations* are connected graphs of local and distributed components and connectors that describe the structure of the distributed architecture.

Although these concepts define a high-level common structure for the distributed software architecture, we need to refine them in a more accurate model to describe more precisely the structure of a distributed architecture.

The model ASCM leads to represent the distributed software architecture, based on architecture specifications described using different ADLs. We attempt to represent in a model the common high-level concepts defined in the major ADLs. This allows us to analyze various architectural description documents, which specify distributed software architectures, in order to extract the elements and to represent them using ASCM.

By comparing in more detail, the definition of the ADLs, we can refine these elements to provide a more precise model. Let us describe the ASCM model presented in figure 1:

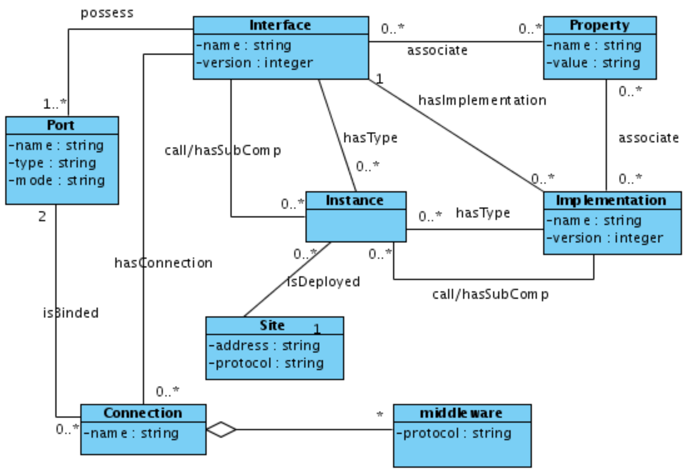


Fig. 1. ASCM: UML-based representation of Architecture Description Languages

– The *interface* defines the services that other components wants to invoke. An *interface* may define *ports* used by a *connection*. A *port* can be used by several connectors. In the ADLs, an interface is mandatory to describe a distributed component in the architecture.
– The *implementation* reflects the source codes of the services defined in the *interface*. For example, the *implementation* can describe how a component calls subprograms to realize the services.
– The *connection* provides a link mechanism between distributed components to exchange information. The link is usually managed using a *middleware* that allows a component deployed on one system to access programs and data on another one.
– The *middleware* is a set of services that allows interactions between multiple components running on many sites. It helps to resolve the complex situations created by heterogeneity, interoperability and distributed computed problems.
– The *instance* represents a sub-component belonging to a component. The subcomponent is instantiated from the interface or the implementation of another component.
– The *site* is the element on which the instances of a component will be deployed. It defines the hardware and software requirements for the instances running.
– The *properties* can specify constraints on the interface or the implementation of a component. The constraints may be functional or not, such as related source code of a component, performance, reliability, and availability.

We coupled ASCM with SCSM, by formalizing a generic relationship that may exist between their respective elements. The relationship will be used for propagating the impact of a change done on the architectural description to its corresponding source code and reciprocally. The models SCSM and ASCM are coupled by a projection relationship. When a *projection relationship* exists, the software modeling is more precise and the change impact analysis is performed on the architectural level and the source codes.

Table 1. Typology of change operations applied on Software Architecture

Name	Operation	Pre-condition	Post-condition	Invariant
Component interface adding	$add\_int\_comp(c, t)$	$-c$ $+t$	$+c$	$-edge(*, c)$
Component implementation adding	$add\_imp\_comp(cimp, c)$	$-cimp$	$+cimp$ $+edge(*, c, cimp)$	$+c$
Port adding	$add\_port(a, p)$	$-p$	$+p$ $+type(p, port)$ $+edge(hasPort*, a, p)$	$+a$
Subcomponent adding	$add\_sub\_comp(a, s, t)$	$-s \vee$ $(+s \wedge -edge(e, a, s))$	$+s$ $+edge(e, a, s)$ $+type(e, hasSubComp)$	$+a$ $+t$
Component interface deletion	$del\_comp\_int(a)$	$+a$	$-a$ $-impl(*, a)$ $-edge(*, a)$	$-subComp(*, a)$ $(-subComp(*, ai),$ $+ impl(ai, a))$
Component implementation deletion	$del\_comp\_imp(c, cimp)$	$+cimp$	$-cimp$ $-edge(*, cimp)$	$+c$ $-subComp(*, cimp)$ $-source(hasSubComp*, cimp)$
Connector deletion	$del\_conn(cn, c, a, b)$	$+cn$ $+edge(hasConn*, c, r)$ $+edge(usePortSrc*, cn, a)$ $+edge(usePortDst*, cn, b)$	$-cn$ $-edge(*, cn)$	$+a$ $+b$ $+c$
Port deletion	$del\_port(a, p)$	$+p$ $+type(p, port)$ $+edge(e, a, p)$	$-p$ $-edge(*, p)$	$+a$ $-dest(usePortSrc*, p)$ $-dest(usePortDst*, p)$
Subcomponent deletion	$del\_subcomp(a, s)$	$+s$ $+edge(hasSubComp*, a, s)$	$-s$ $-edge(*, s)$	$+a$ $-connSubComp(*, s)$

Hereafter, we propose a typology of formalized change operations applied on the distributed architecture, which are represented by ASCM model instances.

4 Typology of Change Operations

A typology of change operations is presented in the table 1, using the assertion formalism. Assertions are widely used in the software community to formalize the programs behavior [11]. For each operation, we specify a pre-condition to be checked in order to allow the execution of the operation. The post-condition indicates the operation results. The invariants represent the propositions to be verified before and after the operation execution.

We describe here, a change operation to illustrate the assertion formalism. The change operation propagates its impact in the distributed architecture and in the source codes.

Let us consider the operation $del\_comp\_imp(c, cimp)$, which intends to delete the implementation $cimp$ associated to the component interface c. To allow the operation, the implementation of the component had to exist in the architecture and therefore a corresponding node must be present in the ASCM graph. The invariant stipulates that the existence of the interface component c must be verified before and after the operation is performed. No instance of $cimp$ should be defined in the architecture $(-subComp(*, cimp))$. $cimp$ should not contain any subcomponents $(-source(hasSubComp*, cimp))$. The result of the operation is the deletion of the component implementation and all of its input and output edges. In the case of at least one invariant is not respected, an impact is propagated locally to the linked nodes (components) in the ASCM graph.

In the case of the component implementation is not instantiated and does not contain any subcomponents, its deletion has no effect on the architecture. Regarding the source code, the statements associated to this implementation have to be deleted. But when the component implementation contains one or more subcomponents, this leads to propagate the impact by marking them and all connectors that use their ports. If there is a subcomponent of type *cimp*, this also leads to an *a priori* marking of these subcomponents. In fact, when one of the operation invariants is not satisfied, this launch the change impact propagation process and consequently the marking of the nodes and edges that could be affected by the operation.

Considering the distributed environment, and during software analysis, the graphs are constructed on each site and are interconnected. Then, the implementation deletion may have an impact on other nodes belonging to an ASCM graph which is distributed on a remote site. This is done by the distributed knowledge-based system.

5 Change Impact Propagation in Architectural Description Languages

The change impact propagation process refers to the process of actually carrying out a set of initial modifications to the software components, and to re-establish the system consistency, by making a set of estimated consequent changes. This process would involve advising the user the software components to be changed and the types of the changes.

Our approach is based on the ECA formalism (Event - Condition - Action) to describe the change impact propagation rules [9]. It consists of analyzing the impact on the distributed architecture by defining generic rules. These could be applied, independent of ADLs, to propagate the impact locally from the architecture to its corresponding source codes and then to the distributed ones.

5.1 Knowledge-Based System for Software Evolution

The change impact analysis is based on a knowledge-based system to manage evolution rules for distributed architecture and source codes. These rules estimate the impact of a change performed on any component belonging to the architecture or to the source codes.

The knowledge-based system consists of three main components : the facts base, the rules base and the inference engine. The facts base constitutes the working memory and the dynamic part of the knowledge-based system (KBS). It contains the set of facts that allows firing change propagation rules. A Fact represents a graph node or edge representing the elements of ASCM or SCSM model. These are added to the facts base during the analysis phase of the architecture specification and related source codes. Applying a change on a node is reflected in the facts base. It may cause the inference engine to fire rules to perform change impact analysis [2].

The knowledge-based system is distributed according the repartition of the architecture specifications and the source codes on multiple sites. Distributed knowledge-based

Table 2. Assertions list for the graph marking

Assertion	**Signification**
$mark(el)$	$\forall el \in E \vee el \in N,\ state(el) = affected$
$mark(v, op)$	$\forall e \in E, v_1 \in N : ((+edge(e, v, v_1) \vee +edge(e, v_1 v)) \wedge +conductivity(op, e, v, v_1)) \longrightarrow mark(e) \wedge mark(v_1)$
$mark(source(t*, v), op)$	$\forall e \in E, v_1 \in N : (type(e) = t \wedge +edge(e, v, v_1) \wedge +conductivity(op, e, v, v_1)) \longrightarrow mark(e) \wedge mark(v_1)$
$mark(target(t*, v), op)$	$\forall e \in E, v_1 \in N : (type(e) = t \wedge +edge(e, v_1, v) \wedge +conductivity(op, e, v, v_1)) \longrightarrow mark(e) \wedge mark(v_1)$

systems are interconnected in order to accumulate the results of the change impact process. This means that a change, which is performed on some architecture part, will induce a change impact analysis locally by identifying the affected set of nodes and edges by the change. The result set will be then propagated among interconnected knowledge-based systems. These will perform a local change impact analysis to further insert the result set in their local fact base. The distributed propagation process is repeated until all affected nodes and edges are identified by firing all the evolution rules.

5.2 Change Propagation Rules Definition

The change propagation rules describe formally the impact of the change operations performed on the elements of the ASCM model. The fact corresponding to the changed node or edge is updated in the knowledge-based system. This starts the change propagation process by selecting the set of rules having the updated fact in their pre-condition.

The table 2 shows the formalization of the marking assertions, which consists of identifying the components and the relationships affected by a change operation. The impact propagation to the neighborhood of a marked component is defined following the nature of incoming and outgoing relationships.

This refers to the impact conductivity of each relationship. The assertion $+conductivity(op, e, A, B)$ formalizes the impact conductivity of a relationship e: $A \longrightarrow B$, with A is the source component of e, B the destination component of e and op is the change operation applied on A. The marking assertions are defined as follows:

- The assertion $mark(el)$ consists to change the state of components and relationships as $affected$, in the facts base.
- The assertion $mark(v, op)$ indicates for an operation op applied on v, that the edges of v are marked as affected, according to the conductivity of the relationship.
- The assertions $mark(source(t*, v, op))$ and $mark(target(t*, v, op))$ consists to mark the relationships e of type t having v as a source node or target node and to mark the related target node or source node of each relationship e following its conductivity for the operation op.

The process of marking nodes and propagating the impact stops when all rules are fired and there is no more candidate facts.

We define three generic rules to deal with the change impact propagation:

```
$Rule\_Execute\_Operation(Op, el)\{$
    $TRUE(Op.Pre - Condition) \wedge TRUE(Op.Invariant)$
     $\longrightarrow Op.Post - Condition$
$\}$
```

The rule $Rule\_Execute\_Operation(Op, el)$ consists to perform the change operation *op* on an architectural element el when the pre-conditions and invariants of *op* are satisfied. There is no impact on the distributed architecture.

```
$Rule\_Impact\_Operation(Op, el)\{$
    $TRUE(Op.Pre - Condition) \wedge FALSE(Op.Invariant)$
     $\longrightarrow mark(el)$
$\}$
```

The rule $Rule\_Impact\_Operation(Op, el)$ changes the state of el to $affected$ in order to identify the impact of *op* when at least one invariant is not respected for *op*.

```
$Rule\_Propagate\_Impact(Op, el)\{$
    $+ state(el, 'affected')$
     $\longrightarrow mark(el, op)\}$
$\}$
```

The rule $Rule\_Propagate\_Impact(Op, el)$ propagates the impact to the neighborhood of an affected element el. The propagation is based on the marking assertions in order to change the state of the edges and of the linked nodes.

6 Prototype of Validation

For the purpose of demonstration, the presented approach is implemented in context of *Eclipse* integrated development environment. Our platform, named as *Architect* is a change impact analysis tool for distributed software applications. An Eclipse project manages a set of resources that can be source code files, libraries and architecture description files (which are described using an ADL). *Architect* analyzes these heterogenous sources and parses their elements to represent as the instantiations of ASCM and SCSM models. The elements of each model are presented as a graph on which change operations can be performed. The figure 2 presents the global architecture of our platform based on Eclipse which is extended for the change impact analysis. *Architect* as a prototype to validate the presented models is divided into four main components:

The *multi-languages analyzer* allows analyzing the source codes and the architectural description of a software. The analysis is based on the grammar of each language used to write a project file. The multi-languages analyzer is based on parsers which are generated through the Java Compiler Compiler (JavaCC). The multi-languages analysis result consists of producing XML descriptions that are used for the graph construction by the second plugin.

The *software modeler* is the second Eclipse plug-in that contains the implementation of both SCSM and ASCM models. It provides an XML flow analyzer that matches the components or relationships defined in our model with the XML tags provided by the *multi-language analyzer*. The two models are instantiated as graph where nodes represent the components and edges their relationships. The software modeler performs

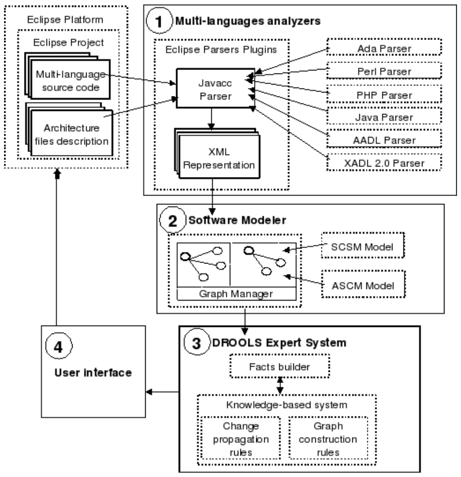


Fig. 2. The architecture of the eclipse-based platform

fact assertions in the knowledge-based system, which is the central component of our implementation and provides the change impact propagation mechanism.

The *expert system engine* is the third plugin and represents the implementation of the knowledge-based system. It allows implementing the change impact propagation process and manages the change propagation rules.

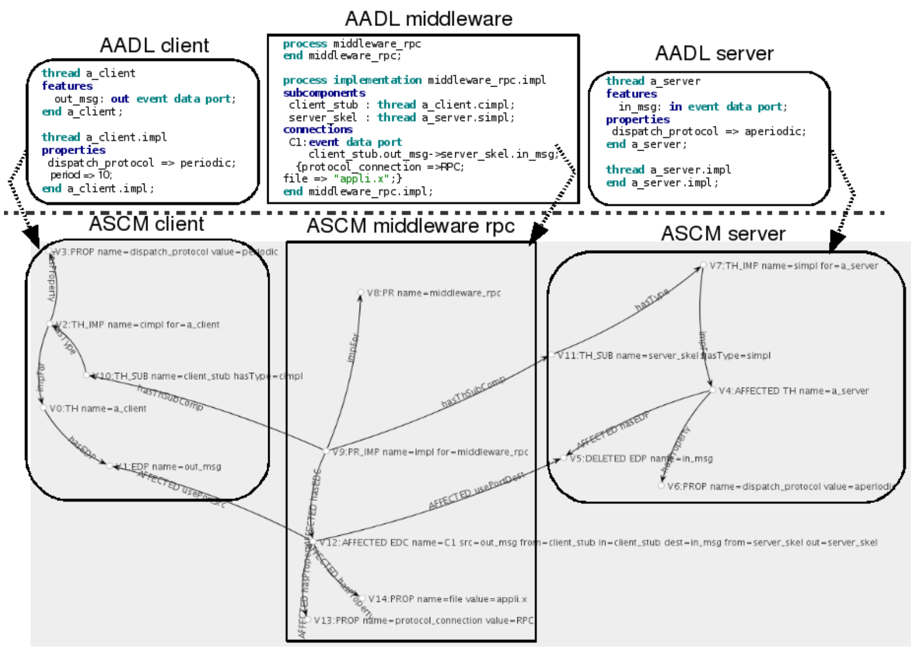
The *user interface* is the forth Eclipse plugin that allows the graph visualization and the execution of change operations on it. It allows maintainer to manage interactively the evolution and the maintenance of a software system.

Hereafter, we propose a scenario to illustrate the graph representation of a software architecture and the change propagation process.

7 Change Propagation Scenario

The proposed scenario illustrates the change impact analysis on a distributed and multi-threaded application. The application defines client threads that send messages using Remote Procedure Call protocol (RPC). The messages are consumed by threads deployed on a server. This application architecture is specified using the AADL language, presented in the listings on Figure 3. Following are the major elements are involved in this scenario:

– *AADL Client* is the client thread type definition, noted a_client defines an output port out_msg. The port allows to send data using Remote Procedure Call. The implementation $a\_client.impl$ defines a property for the periodical sending of data.
– *AADL Server* consists of the server thread type definition, noted a_server provides an input port in_msg for receiving data sent by the client thread. The Server implementation $a\_server.impl$ may define a subprogram to consume data.


AADL client
AADL middleware
AADL server
ASCM client
ASCM middleware rpc
ASCM server
V3:PROP name=dispatch_protocol value=periodic
hasProperty
V2:TH_IMP name=cimpl for=a_client
hasType
V10:TH_SUB name=client_stub hasType=cimpl
impFor
V0:TH name=a_client
hasThSubComp
hasEDP
V1:EDP name=out_msg
AFFECTED usePortSrc
V8:PR name=middleware_rpc
impFor
V9:PR_IMP name=impl for=middleware_rpc
hasThSubComp
AFFECTED hasEDC
V12:AFFECTED EDC name=C1 src=out_msg from=client_stub in=client_stub dest=in_msg from=server_skel out=server_skel
AFFECTED usePortDest
AFFECTED hasProperty
V14:PROP name=file value=appli.x
V13:PROP name=protocol_connection value=RPC
V7:TH_IMP name=simpl for=a_server
hasType
V11:TH_SUB name=server_skel hasType=simpl
impFor
V4:AFFECTED TH name=a_server
AFFECTED hasEDP
V5:DELETED EDP name=in_msg
hasProperty
V6:PROP name=dispatch_protocol value=aperiodic

```
thread a_client
features
  out_msg: out event data port;
end a_client;

thread a_client.impl
properties
 dispatch_protocol => periodic;
 period => 10;
end a_client.impl;
```

```
process middleware_rpc
end middleware_rpc;

process implementation middleware_rpc.impl
subcomponents
 client_stub : thread a_client.cimpl;
 server_skel : thread a_server.simpl;
connections
 C1: event data port
     client_stub.out_msg->server_skel.in_msg;
  {protocol_connection =>RPC;
file => "appli.x";}
end middleware_rpc.impl;
```

```
thread a_server
features
  in_msg: in event data port;
properties
 dispatch_protocol => aperiodic;
end a_server;

thread a_server.impl
end a_server.impl;
```

Fig. 3. Distributed Architecture evolution and related graphs resulting from the port deletion operation

– *AADL middleware* is the middleware process definition, noted $middleware\_rpc$ is implemented by $middleware\_rpc.impl$. The implementation defines two subcomponents: $client\_\ stub$ of type $a\_client.impl$ which represents the RPC calls performed by the client, and $server\_skel$ of type $a\_server.impl$ which represents the server skeleton to manage the received RPC calls. These subcomponents communicate using a RPC connection $C1$ which allows sending message from client to server. The RPC connection is specified in RPC Language in file $appli.x$.

The architecture specification files and related source codes are distributed over two Eclipse environments, which are interconnected to exchange facts in the distributed knowledge-based system. For instance, one of the Eclipse environment creates graph nodes (facts) and triggers the second one to update the graph and the fact base.

The presented scenario consists to delete the server port in_msg using the change operations called $del\_port(a, p)$. Regarding the operation invariants and the ASCM graph, an edge exists between the node $EDPname = in\_msg$ and the node $EDCname = C1$. This edge provides conductivity of the change impact on the port component. The impact of the operation is calculated and propagated by firing the propagation rules introduced into the knowledge-based system. The rule $ImpactDeletePort$, described below, changes the state of the node (port) as $affected$. This fires the second rule, called $PropagateImpactDeletePort$ to mark the edges entering in the port and the related target nodes. These marked nodes are sent to the Eclipse client to be inserted in the distributed fact base, which will fire the rules for the local propagation of the impact. The propagation is done through relationships according to theirs conductivity.

The result of the change propagation process can be shown in figure 3. The nodes and the edges, which have been marked by our expert system, are shown with the *affected* label. The result shows the projection relationship which leads the impact to the source codes.

```
rule "ImpactDeletePort"
when
 n : ArchitectNode(
    stateNode==ArchitectNode.STATE_DELETED,
    typeNode=="EDP"
     )
 n1: ArchitectNode()
 e : ArchitectEdge()
 eval((e.getNodeDest()==n && e.getNodeSrc()==n1)
      ||
      (e.getNodeSrc()==n && e.getNodeDest()==n1)
     )
then
 n.setLabel("AFFECTED "+n.getLabel());
 n.setStateNode(ArchitectNode.STATE_IMPACTED);
end

rule "PropagateImpactDeletePort"
when
 n : ArchitectNode (
    stateNode==ArchitectNode.STATE_IMPACTED,
    typeNode=="EDP"
     )
 n1: ArchitectNode()
 e : ArchitectEdge()
 eval((e.getNodeDest()==n && e.getNodeSrc()==n1)
      ||
      (e.getNodeSrc()==n && e.getNodeDest()==n1)
     )
then
 e.setState(ArchitectEdge.STATE_IMPACTED);
 n1.setStateNode(ArchitectNode.STATE_IMPACTED);
end
```

8 Conclusions and Future Works

In this chapter, we have presented an ASCM model and a knowledge-based system approach to deal with the change impact analysis on distributed software architecture. Our model represents the information extracted from distributed architectural descriptions, independent of the ADLs. It helps a progressive and detailed elaboration of change operation. The change operations are specified by the invariants, pre and post conditions. The extracted facts from software applications are stored in a knowledge-based operation. These are used to fire the change propagation rules to identify the impact of a change and to propagate it to linked nodes. The validation of this modeling approach is implemented as plug-ins in Eclipse Environment.

Perspectives of this work are to further enhance the change impact analysis approach to deal with structural and qualitative aspects of software. We are currently specifying a profiling model and an instrumentation mechanism, to allow us, to identify the change impact on the quality of software services.

References

1. Ahmad, A., Basson, H., Deruelle, L., Bouneffa, M.: Towards a better control of change impact propagation. In: INMIC 2008: 12th IEEE International Multitopic Conference, pp. 398–404. IEEE Computer Society, Los Alamitos (December 2008)
2. Ahmad, A., Basson, H., Deruelle, L., Bouneffa, M.: A knowledge-based framework for software evolution control. In: INFORSID 2009: Actes du XXVIIème Congrès Informatique des organisation et systèmes d'information et de décision, pp. 111–126. IRIT Press, Toulouse (May 2009), www.irit.fr
3. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison Wesley, Reading (1998)
4. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison Wesley, Reading (2002)
5. Clements, P., Shaw, M.: "The golden age of software architecture" revisited. IEEE Software 26, 70–72 (2009)
6. Deruelle, L., Basson, H., Bouneffa, M., Hattat, J.: An eclipse platform extension for analysis and manipulation of multi-language software code, pp. 174–179 (2007)
7. Deruelle, L., Bouneffa, M., Melab, N., Basson, H.: A change propagation model and platform for multi-database applications. In: IEEE International Conference on Software Maintenance, pp. 42–51 (2001)
8. Garlan, D., Monroe, R., Wile, D.: Acme: An architecture description interchange language. In: Proceedings of CASCON 1997, pp. 169–183 (1997)
9. Hassan, M.O., Deruelle, L., Basson, H.: Towards a change propagation process in software architecture. In: 18th International Conference on Software Engineering and Data Engineering (SEDE 2009), Las Vegas, Nevada, USA, pp. 85–90 (June 2009)
10. Medvidovic, N., Taylor, R.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering 26, 70–93 (2000)
11. Mens, T.: Transformational software evolution by assertions. In: Workshop on Formal Foundations of Software Evolution, CSRM 2001 (2001)
12. Taylor, R.N., Medvidovic, N., Dashofy, E.M.: Software Architecture: Foundations, Theory, and Practice. Wiley Publishing, Chichester (2009)

Common Languages for Web Semantics

Seiji Koide[1] and Hideaki Takeda[1,2]

[1] Information Technology, The Graduate University for Advanced Studies (SOKENDAI)
[2] National Institute of Informatics
2-1-2, Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
{koide,takeda}@nii.ac.jp
http://www-kasm.nii.ac.jp/

Abstract. RDF is a language to express propositions on the WWW and OWL is a language for defining Web ontologies. It seems that RDF and OWL have established themselves as a standard in Semantic Webs. However, endeavors to describe ontology in OWL are revealing the extent of the language capability in practical views. In this paper, firstly we give an overview of basic assumptions as knowledge representation languages for Semantic Webs, and then point out several basic and problematic issues of OWL mainly arose from the difference of the foundation among languages. They are captured by our own experience of developing an object oriented language for Semantic Webs and its applications. They are solved in our language by means of i) explicit descriptions of role concepts, ii) auto-epistemic local closed world assumption, iii) ternary truth values, and iv) unique name assumption for atomic objects. Finally, we envision the direction of language development for web semantics with reviewing Common Logic.

Keywords: Semantic Web, RDF, OWL, KIF, Common logic, Common lisp, SWCLOS.

1 Introduction

Resource Description Framework (RDF) is an assertional language intended to be used to express propositions on the WWW[1]. The OWL Web Ontology Language is a language for defining and instantiating Web ontologies[2]. Today, it seems that RDF and OWL have successfully established themselves as a *de facto* standard of ontology description language not only in the Semantic Web community. Especially, OWL has been spread over diverse disciplines and engineering fields, e.g., ontology, linguistics, UML modeling in software engineering, enterprise business patterns, etc. However, along with spreading of OWL, the extent of the modeling capability of OWL has become well known.

We had also developed an object oriented semantic language called SWCLOS[3][1] on top of Common Lisp Object System (CLOS), and attempted to apply it in several applications. SWCLOS is an amalgamation of object oriented language in Lisp and OWL, and then we saw how an ontology description language that is firmly underpinned

[1] Source codes and documents are available on the web site http://www-kasm.nii.ac.jp/∼koide/SWCLOS2-en.files/Page408.htm

by a formal logic and denotational semantics is useful to software engineering so as to assure formal descriptions of system specification.

In the process of developing SWCLOS we, however, encountered a few basic and nice problems of semantic disparity to be coped with that arose from the difference of language foundation on RDF, OWL, logics, and object oriented language. For example, ordinary programming languages and predicate calculus stand on Unique Name Assumption (UNA), but OWL is not regarded to stand on UNA. Ordinary software models and predicate calculus are based on Closed World Assumption (CWA), but OWL and Description Logics (DLs) are regarded as on Open World Assumption (OWA). In case that we had set such full setting as non-UNA and OWA for Semantic Webs into SWCLOS, it amounted to the result of either very few viable interpretations with less common knowledge or excessive need of common knowledge for models on class disjointness and individual differentiation in several Semantic Web applications. Hence, we performed refactoring SWCLOS with introducing new moderate settings in order to unify RDF and OWL on top of CLOS. They are i) *context dependent role and disjointness of substance classes*, ii) *auto-epistemic local closed world assumption*, iii) *ternary truth values* that allow unknown value, and iv) *UNA for atomic objects in non-UNA environment*. Such experience of refactoring SWCLOS and the subsequent applications brought us deeper understanding on the theory and relations of RDF(S)/OWL, logics, and object oriented language semantics. In this paper, we describe the basic and nice semantic gap among those languages and present our solutions.

Table 1 summarize basic computational foundations underlying Common Lisp, Description Logic, RDF, OWL, and Common Logic.

Table 1. Basic Computational Foundations of Languages

	UNA/nonUNA	CWA/OWA	Truth Value	arity
Common Lisp	UNA	CWA	Ternary	n
Description Logic	UNA	OWA	Binery	2
RDF	UNA	(CWA)	Binery	2
OWL	nonUNA	OWA	Binery	2
Common Logic	nonUNA	OWA	Ternary?	n

This paper is structured as follows. In Section 2, we give an overview of RDF, RDF Schema, OWL, Common Lisp, and Common Logic. Section 3 describes problematic non-UNA and OWA, and also discusses the equality of entities in universe of discourse. In Section 4, we propose a new framework that involves role concepts based on discussion upon several top ontologies. The auto-epistemic local closed world assumption and ternary truth value are explained at Section 5. Finally, we envision the direction of languages for web semantics at Section 6.

2 Semantics in RDF, OWL, Common Lisp, and Common Logic

2.1 Denotational Semantics in RDF

Every scientific theory is a system of sentences which are accepted as true and which may be called *asserted statements* [4]. To deduce the truth value of asserted sentences, words in sentences are distinguished from things denoted by words (denotations), and then the relations between words and denotations and among denotations themselves are interpreted according to axioms and rules in a given formal way. Such systematic denotational semantics is called "Tarskian" [5]. A set of all entities as denotation is called a *domain* or *universe of discourse* [4]. Axioms and rules for interpretation must enable the structure of the universe of discourse so as to reflect the structure of the real world. Entailment in logic with axioms must follow rules in the real world, see Figure 7.6 in Russell and Norvig [6].

RDF is applicable to WWWs with the components such as URI references, literals (with/without a language tag), and XML schema typed literals [1]. RDF captures WWWs as labeled directed graphs. The semantics of RDF graph is specified and formalized as follows by set-theoretical denotational semantics based on the Tarskian model theory.

In the RDF simple interpretation $\mathcal{I}$ of vocabulary $\mathcal{V}$,

1. A non-empty set $\boldsymbol{R}^{\mathcal{I}}$ of entities, called the domain or universe of $\mathcal{I}$.
2. A set $\boldsymbol{P}^{\mathcal{I}}$, called the set of properties of $\mathcal{I}$.
3. $EXT^{\mathcal{I}} : \boldsymbol{P}^{\mathcal{I}} \to \mathcal{P}(\boldsymbol{R}^{\mathcal{I}} \times \boldsymbol{R}^{\mathcal{I}})$, namely a mapping from $\boldsymbol{P}^{\mathcal{I}}$ into the powerset of the set $\boldsymbol{R}^{\mathcal{I}} \times \boldsymbol{R}^{\mathcal{I}}$, i.e., the set of sets of pairs $\langle x^{\mathcal{I}}, y^{\mathcal{I}} \rangle$ with $x^{\mathcal{I}}$ and $y^{\mathcal{I}}$ in $\boldsymbol{R}^{\mathcal{I}}$.
4. A mapping $S^{\mathcal{I}}$ from URI references in $\mathcal{V}$ into $\boldsymbol{R}^{\mathcal{I}} \cup \boldsymbol{P}^{\mathcal{I}}$.
5. A mapping $L^{\mathcal{I}}$ from typed literals in $\mathcal{V}$ into $\boldsymbol{R}^{\mathcal{I}}$.
6. A distinguished subset LV of $\boldsymbol{R}^{\mathcal{I}}$, called the set of literal values, which contains all the plain literals in $\mathcal{V}$.

Here, $EXT^{\mathcal{I}}(p^{\mathcal{I}})$ is called the property extension of $p^{\mathcal{I}}$, and it represents a set of pairs which identify the arguments for which the property is true, that is, a binary relational extension $\langle x^{\mathcal{I}}, y^{\mathcal{I}} \rangle$, where $x^{\mathcal{I}}$ and $y^{\mathcal{I}}$ are entities in the universe of discourse. In other words, a property makes a set of the binary relation between entities in the universe of discourse.

A particular pair of $\langle x^{\mathcal{I}}, y^{\mathcal{I}} \rangle$ for property $p^{\mathcal{I}}$ is called a triple in infix notation in text or $x\ p\ y$. In this expression, x is called *subject*, y is called *object*, and p is called *predicate*. A set of triples composes and represents an RDF graph in Semantic Webs. If all triples denotes true, then the RDF graph denotes true. An RDF graph may include blank nodes. A blank node has no URI reference and may be designated by a *nodeID* in triples instead of a URI reference. An RDF graph that does not include blank nodes is called a ground graph. The denotation of a ground RDF graph (truth value) in $\mathcal{I}$ is given by recursively applying the above interpretation and axioms for ground triples. The semantics of ungrounded graphs is extended from the ground graphs, see Section 1.4 and 1.5 in [1] for the details.

The notion of property is redefined with two terms, rdf:Property and rdf:type, in rdf vocabulary as follows.

Axiom 1. If an entity is a member of the set of properties of $\mathcal{I}$, then the entity makes a pair with $\mathit{rdf}:\mathit{Property}^{\mathcal{I}} = S^{\mathcal{I}}(\mathrm{rdf}:\mathit{Property})$[2] and then the pair is a member of property extension of $\mathit{rdf}:\mathit{type}^{\mathcal{I}} = S^{\mathcal{I}}(\mathrm{rdf}:\mathrm{type})$, and vice versa:

[2] rdf:Property is a QName in XML Namespace for URI reference http://www.w3.org/1999/02/22-rdf-syntax-ns#Property.

$$x \in \boldsymbol{P}^{\mathcal{I}} \Leftrightarrow \langle x, \mathit{rdf}{:}\mathit{Property}^{\mathcal{I}} \rangle \in \mathit{EXT}^{\mathcal{I}}(\mathit{rdf}{:}\mathit{type}^{\mathcal{I}})$$

Next two subsections describe the semantic extensions of interpretation for RDF Schema (RDFS) and OWL.

2.2 Semantics of Class in RDF Schema

RDF Schema (RDFS) is a semantic extension of RDF that provides a device using rdfs vocabulary for describing ontology as the minimal type system. The notion of class-instance is introduced as an rdfs-extension of the universe of discourse using the notion of class extension, see below.

Axiom 2. If an entity is a member of class extension of another entity, then a pair of both becomes a member of property extension of $\mathit{rdf}{:}\mathit{type}^{\mathcal{I}}$, and vice versa.

$$x^{\mathcal{I}} \in \mathit{CEXT}^{\mathcal{I}}(y^{\mathcal{I}}) \Leftrightarrow \langle x^{\mathcal{I}}, y^{\mathcal{I}} \rangle \in \mathit{EXT}^{\mathcal{I}}(\mathit{rdf}{:}\mathit{type}^{\mathcal{I}})$$

Here, $\mathit{CEXT}^{\mathcal{I}} : \boldsymbol{C}^{\mathcal{I}} \rightarrow \mathcal{P}(\boldsymbol{R}^{\mathcal{I}})$ represents $\boldsymbol{C}^{\mathcal{I}}$ a mapping from a set of all classes in the universe to the set of subsets of $\boldsymbol{R}^{\mathcal{I}}$. $\mathit{CEXT}^{\mathcal{I}}(y^{\mathcal{I}})$ is called the class extension of $y^{\mathcal{I}}$, and it denotes a set of instances of $y^{\mathcal{I}}$, and $y^{\mathcal{I}}$ is called a class.

It is obvious that every property in the universe turns out to be an instance of $\mathit{rdf}: \mathit{Property}^{\mathcal{I}}$. Here, $\boldsymbol{R}^{\mathcal{I}}$, which is initially defined as the universe itself in the rdf simple interpretation, is named with a term rdfs:Resource as a class extension of $\mathit{rdfs}: \mathit{Resource}^{\mathcal{I}}$. In addition, using the notion of class extension, the set of all classes in the universe $\boldsymbol{C}$ is also named with a new term rdfs:Class. Datatypes and literals are also named as follows.

$$\begin{aligned}
\boldsymbol{P}^{\mathcal{I}} &= \mathit{CEXT}^{\mathcal{I}}(\mathit{rdf}:\mathit{Property}^{\mathcal{I}}) \\
\boldsymbol{R}^{\mathcal{I}} &= \mathit{CEXT}^{\mathcal{I}}(\mathit{rdfs}:\mathit{Resource}^{\mathcal{I}}) \\
\boldsymbol{C}^{\mathcal{I}} &= \mathit{CEXT}^{\mathcal{I}}(\mathit{rdfs}:\mathit{Class}^{\mathcal{I}}) \\
\boldsymbol{DC}^{\mathcal{I}} &= \mathit{CEXT}^{\mathcal{I}}(\mathit{rdfs}:\mathit{Datatype}^{\mathcal{I}}) \\
LV &= \mathit{CEXT}^{\mathcal{I}}(\mathit{rdfs}:\mathit{Literal}^{\mathcal{I}})
\end{aligned}$$

For these rdfs vocabulary, rdfs-interpretation satisfies the extra conditions for RDFS, see [1].

The class-superclass relation in the universe is specified with a term rdfs:subClassOf as the inclusiveness of the class extensions of class/superclass as follows.

Axiom 3. If a pair of two entities is a member of property extensions of $\mathit{rdfs}{:}\mathit{subClassOf}^{\mathcal{I}}$, then the both entities are instances of $\mathit{rdfs}{:}\mathit{Class}^{\mathcal{I}}$ and the class extension of the predecessor in the pair is included by the class extension of the successor.

$$\begin{aligned}
&\langle x^{\mathcal{I}}, y^{\mathcal{I}} \rangle \in \mathit{EXT}^{\mathcal{I}}(\mathit{rdfs}:\mathit{subClassOf}^{\mathcal{I}}) \Rightarrow \\
&\qquad\qquad x, y \in \boldsymbol{C}^{\mathcal{I}} \wedge \mathit{CEXT}^{\mathcal{I}}(x) \subseteq \mathit{CEXT}^{\mathcal{I}}(y)
\end{aligned}$$

Note that this axiom in RDFS is not a *if and only if* construct. Thus, this condition is called *weak subsumption*.

2.3 Semantics in OWL

The OWL Web Ontology Language is a language for defining and instantiating Web ontologies. The OWL language provides three increasingly expressive sublanguages designed for use by specific communities of implementers and users. OWL Lite is the simplest sublanguage and it supports those users primarily needing a classification hierarchy and simple constraint features. OWL DL supports those users who want the maximum expressiveness without losing computational completeness (all entailments are guaranteed to be computed) and decidability (all computations will finish in finite time) of reasoning systems. OWL DL was designed to support the existing Description Logic business segment and has desirable computational properties for reasoning systems [2]. However, OWL Lite and OWL DL do not support RDF semantics. OWL Full is meant for users who want maximum expressiveness and the syntactic freedom of RDF. In OWL Full a class can be treated simultaneously as a collection of individuals[3] and as an individual in its own right (metamodeling).

The OWL specifications include many features and capabilities that are useful to describe Web ontologies, and OWL Lite and DL specifications and its semantics are described in the bunch of specifications [2,7,8]. OWL Full specification is, however, removed from them, and not yet developed as well.

OWL Compatibility to RDF. The compatibility between OWL and RDF is discussed in [9]. Although OWL is regarded to be constructed on top of RDF, it was actually not so easy by the semantic disparity between Description Logics and RDF and the historical reasons. The specification [9] that specifies the universe of OWL is included in the universe of RDF on one hand, see below.

$$\begin{aligned} \boldsymbol{OC}^{\mathcal{I}} &= CEXT^{\mathcal{I}}(owl\!:\!Class^{\mathcal{I}}) \subset \boldsymbol{C}^{\mathcal{I}} \\ \boldsymbol{OT}^{\mathcal{I}} &= CEXT^{\mathcal{I}}(owl\!:\!Thing^{\mathcal{I}}) \subset \boldsymbol{R}^{\mathcal{I}} \end{aligned}$$

On the other hand, it simultaneously states in the document that $\boldsymbol{OC}^{\mathcal{I}} = \boldsymbol{C}^{\mathcal{I}}$ and $\boldsymbol{OT}^{\mathcal{I}} = \boldsymbol{R}^{\mathcal{I}}$ for OWL Full. As a matter of fact, the OWL definition file[4] contains the definition of the first inclusiveness between OWL classes and RDF classes, and the second one can be entailed within the RDF universe by the rdfs entailment rule **rdfs4a**[5]. Thus, SWCLOS realized the OWL universe in the RDF universe naturally by loading the file in RDF subsystem of SWCLOS.

In addition, we introduced the following axiom in order to include OWL classes in the OWL universe, and to make the OWL universe OWL Full. See details in [3] and SWCLOS documents[6].

Axiom 4. The extension of the denotation of URI reference of owl:Class is included by the extension of the denotation of URI reference of owl:Thing.

$$\boldsymbol{OC}^{\mathcal{I}} \subset \boldsymbol{OT}^{\mathcal{I}} \tag{1}$$

[3] "individual" is a term in Description Logic and synonymous with "instance".

[4] http://www.w3.org/2002/07/owl.rdf

[5] http://www.w3.org/TR/rdf-mt/#rulerdfs4

[6] SWCLOS documents are available at http://www-kasm.nii.ac.jp/~koide/SWCLOS2.files/Page408.htm

Note that this inclusiveness between $\boldsymbol{OC}$ and $\boldsymbol{OT}$ is similar to the inclusiveness between $\boldsymbol{C}$ and $\boldsymbol{R}$ in RDF universe. $\boldsymbol{OC} \subset \boldsymbol{OT}$ implies the system is 'higher-order' as well as $\boldsymbol{C} \subset \boldsymbol{R}$.

This axiom enables every class in the OWL universe to have roles (properties) for individuals such as owl:sameAs, owl:differentFrom, etc.

Metamodeling Capability in OWL Full. SWCLOS is the first full-fledged language as OWL Full processor, in which the capability of metamodeling objects is borrowed from the dynamic and reflective features of Lisp and metaclassing capability of CLOS. We had implemented many OWL axioms into CLOS using Meta-Object Protocol (MOP) [10] of CLOS. Whereas unrestricted freedom of metamodeling certainly results in undecidability, examples demonstrated by the OWL DL camp in Semantic Web community as OWL Full undecidability are always unreasonably extreme and make no sense from the view of engineering. We had showed several metamodeling examples of SWCLOS in [3] within the understandable rationale of engineering from our practical experience, and in [11], we addressed a set of metamodeling criteria that enables SWCLOS to perform ontology metamodeling.

2.4 Semantics in Common Lisp

Common Lisp as a dialect of lisp is produced by the activity of ANSI standardization on lisps during 1981 through 1997 in U.S., and many systems are running on Common Lisp today in academia and industry. However, the semantics embraces some ambiguities specifically from the viewpoint of denotational and extensional semantics like RDF semantics. The problem of subclassing and metaclassing in Common Lisp is discussed in [11]. In this section, we attempt to categorize computer languages with emphasizing the specialty of lisp as computer language, and discuss the relation among them according to the semantics that is addressed by [12] on the reflective language 3-Lisp.

In a lisp system like early lisp system Lisp 1.5, which is equipped with symbol, function, and list, and without any other structural devices like object in object oriented programming, a syntactic lisp expression (S-expression) is reduced to a nominal form that is equivalent to the original one in the sense of λ-calculus. For example, an expression "`(+ 1 2)`" is reduced to "`3`". However, we can quote an expression to inhibit the reduction, and then we can change expressions and construct another form, such as "`(cons (quote *) (cdr (quote (+ 1 2))))`" turns out "`(* 1 2)`". This specific feature of lisp family languages is recently called *homoiconic*. In this first computational model, lexical expressions are not discriminated from the denotations.

Second computational model in this section discriminates symbols from the denotations. A lexical expression "`3`" denotes number 3, and a lexical expression "`t`" and "`nil`", which are reduced to "`t`" and "`nil`", denote true and falsity, respectively, in the universe of discourse. However, no symbol but "`t`" and "`nil`" denotes anything until it is defined so. This mapping from a lexical expression to the denotation is analogous to that in RDF semantics.

In the third computational model, a symbol refers a complex structure as internal realization in a modern computer language. In this case, a symbol can be used to refer a referent that denotes an entity in the universe. Figure 1 shows the framework in this third computational semantic model [12].

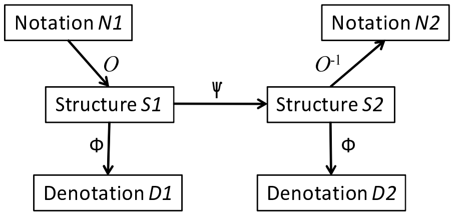


Fig. 1. The Framework for Computational Semantics by Smith

Smith called the mapping O *internalization*, and the inverse operation O^{-1} *externalization*. He also noted that O (and O^{-1}) is usually ignored in logic. The ϕ is the interpretation function, which is analogous to the interpretation in denotational semantics, and the reduction ψ, which is, Smith says, the relationship among symbols, corresponds entailment rules and rule application in logic. Smith pointed out that the lisp evaluator crosses semantical levels, and therefore obscures the difference between the simplification ψ and the interpretation ϕ. Smith called this lisp specific nature *de-reference* ($\phi = \psi$). It has become the theoretic base of his work on the reflective language 3-Lisp.

Assumptions and axioms in domain knowledge can be syntactically represented by a set of symbols and structures expressed among the symbols. Those expressions of assumptions are reduced to entailed assertions by a prover ψ.

2.5 Semantics in Common Logic

Common Logic[7] is an abstract language of ISO standard of a logic framework intended for information exchange and transmission. The framework allows a variety of different syntactic forms, called dialects. The dialects, which have different syntax but interchangeable from one to another, include Common Logic Interchange Format (CLIF)[8], Conceptual Graph Interchange Format (CGIF)[9], XML Common Logic (XCL)[10], and Common Logic Controlled English (CLCE)[11]. CLIF may be conceived to be a modernized version of Knowledge Interchange Format (KIF)[12][13].

Common Logic has some novel features syntactically and semantically. It allows a syntax which is signature-free. The abstract syntax model is analogous to polymorphism in object oriented programming languages and no fixed arity like Common Lisp (*polyadic*). As shown in Table 1, the arity of RDF and OWL is strictly constrained to 2. Not only Common Logic allows n-ary, but also the arity is not fixed for a predicate or property. Guha and Hayes initially proposed such features as the RDF syntax for a common base language of Semantic Web languages [15]. In their proposal for the candidate of RDF, they expected it would be a base language for Semantic Web languages, and claimed the basic language, called L_{base}, that would support basic inference and

[7] http://common-logic.org/
[8] http://www.ihmc.us/users/phayes/CLIF.html
[9] http://conceptualgraphs.org/
[10] http://www.altheim.com/specs/xcl/1.0/
[11] http://www.jfsowa.com/clce/specs.htm
[12] http://www-ksl.stanford.edu/knowledge-sharing/kif/

semantics, and then would allow RDF and extending different semantics at the upper layers in the Semantic Web stack. They imagined that L_{base} would provide L_i language in i-th layer of Semantic Web language stack.

Common Logic permits ‘higher-order’ constructions such as quantification over classes or relations while preserving a first-order model theory. The semantics allows theories to describe intensional entities such as classes and properties. The first solution of this ‘higher-order’ constructions will be *metamodeling* in Common Logic.

It seems that the modernized features of Common Logic is a reflection of the progress of modern computer languages. For example, the semantics of Common Logic introduced a new term, *universe of reference*[13], in addition to the universe of discourse in denotational semantics. The universe of reference include the universe of discourse. In case that there are names such that they denote entities in the universe of reference but not in the universe of discourse in an interpretation, the language is called segregated dialect. All names in a non-segregated dialect are discourse names. “Segregated dialects are commonly described to have a universe of discourse, without mentioning the universe of reference; and for non-segregated dialects the universes of discourse and of reference are identical. The distinction makes it possible to provide a single semantics which can cover both styles of dialect”. The motivation of introducing the universe of reference and non-discourse names is likely to be for the provision against people who do not want to concern some terminologies out of concerning ontologies.[14] However, this notion is very akin to Smith’s framework in the previous section. This language model will support to develop logic systems using objects in imperative computer languages which may include variable names for individual objects, class objects, property objects, function objects, etc.

3 Non-unique Name Assumption and Equality

3.1 Equality of Individuals

Unique Name Assumption, that is, different names always denote different entities, which is usually adopted into computer languages, is not adopted in Semantic Webs. In RDF, different URI references denotes different graph nodes. However, in OWL language, owl:sameAs property may be applied to different URIs to indicate that two different URI references denote the same entity as individual in the OWL universe. Oppositely, the owl:differentFrom property (and the combination of owl:AllDifferent and owl:distinctMembers, too) may be used to indicate two different URI references denote different entities. Thus, in case of no information on the individual equality in OWL, the equality of two entities is not determined[15], then, the decision of the equality of entity must be performed in the RDF universe. To discuss the equality of entities in RDF semantics, it is appropriate to discuss the equality of two subgraphs that the two entities are in position of *subject*.

[13] http://standards.iso.org/ittf/licence.html

[14] from the discussion on [14] at the conference.

[15] Instance properties of owl:FunctionalProperty and owl:InverseFunctionalProperty also affect the equality as individual.

The algorithm for the equality computation in the RDF universe is explained as follows[16].

Two RDF graphs G and G' are equivalent if there is a bijection $\mathcal{M}$ between the sets of triples for the two graphs, such that:

1. $\mathcal{M}$ maps blank nodes to blank nodes.
2. $\mathcal{M}(lit) = lit$ for all RDF literals lit which are nodes of G.
3. $\mathcal{M}(uri) = uri$ for all RDF URI references uri which are nodes of G.
4. The triple $s/p/o$ is in G if and only if the triple $\mathcal{M}(s)/p/\mathcal{M}(o)$ is in G'.

Note that these are not described in denotational semantics. RDF is property-centric but OWL is object-centric (it means a subject node and linked nodes with one hop predicates are regarded as an object like object-oriented language). Then, we modify the above algorithm to meet OWL object-centric paradigm.

For each subgraph composed of a subject node and one-hop linked nodes in G and G',

1. $\mathcal{M}$ maps blank nodes to blank nodes.
2. $\mathcal{M}(lit) = lit$ for all RDF literals lit which are nodes of G.
3. $\mathcal{M}(uri^{\mathcal{I}}) = uri^{\mathcal{I}}$ for all RDF URI references uri which denote nodes of G.
4. For every s of triple $s/p/o$ for G, $\langle s^{\mathcal{I}}, o^{\mathcal{I}}\rangle \in EXT^{\mathcal{I}}(p^{\mathcal{I}})$ if and only if $\langle \mathcal{M}(s^{\mathcal{I}})$, $\mathcal{M}(o^{\mathcal{I}})\rangle \in EXT^{\mathcal{I}}(\mathcal{M}(p^{\mathcal{I}}))$ is in G'.

For the discussion of equality under the non-UNA condition, we superimpose owl:sameAs and owl:differentFrom properties onto the above algorithm. In case that $s^{\mathcal{I}}$ in G and $s'^{\mathcal{I}}$ in G' are blank nodes in a bijection $\mathcal{M}$, $\langle s^{\mathcal{I}}, o^{\mathcal{I}}\rangle \in EXT^{\mathcal{I}}(p^{\mathcal{I}})$ is equivalent to $\langle \mathcal{M}(s^{\mathcal{I}}), \mathcal{M}(o^{\mathcal{I}})\rangle \in EXT^{\mathcal{I}}(\mathcal{M}(p^{\mathcal{I}}))$, if $o^{\mathcal{I}} = \mathcal{M}(o'^{\mathcal{I}})$ in OWL semantics. Namely two blank nodes $s^{\mathcal{I}}$ and $s'^{\mathcal{I}}$ are equivalent in OWL. We apply the same algorithm for non-blank node in non-UNA condition. In case that $s^{\mathcal{I}}$ and $s'^{\mathcal{I}}$ are named with different names and we cannot determine the equality by the names, our approach determines the equality between $s^{\mathcal{I}}$ and $s'^{\mathcal{I}}$ through the subgraphs of the both. Namely, we check $p^{\mathcal{I}}$ and the equality of $o^{\mathcal{I}}$ and $o'^{\mathcal{I}}$. This algorithm traverses two graphs, until the decision is obtained. Note that RDF graph is a directed graph. In this graph equality checking, if two nodes have sub-trees, the corresponding sub-trees on both graphs are recursively checked for the equality. Thus, if we reach at terminal nodes (atomic nodes that do not have edges any more) but no information is obtained, we fall into a troublesome situation. For example, in comparison of ex:Y/ex:p/ex:A and ex:Z/ex:p/ex:B, if ex:A and ex:B are both atomic, the non-UNA computation cannot conclude whether or not ex:Y is equivalent to ex:Z. In such condition, in order to derive useful computational results, we must define the equality or difference among every atomic individuals. It is very laborious work to describe common knowledge such as Bill is different from George, Barack, Al, and so on.

Therefore, we devised a flag for non-UNA and set up falsity to the flag as default. Note that the equality of two blank nodes is checked both in UNA and in non-UNA. In the default condition, we stand in UNA as well as for ordinary computer languages, then two nodes that have different URI references are different, and then two blank

[16] http://www.w3.org/TR/rdf-concepts/#section-graph-equality

node trees are distinct if we cannot find the corresponding edges of graphs or we find the lexically different URI references at the corresponding positions in the trees. In non-UNA condition with the flag setting, the graph equality checking is performed even though two URIs at the corresponding positions are different, until we find either the difference of graph structures or the difference of nodes that are explicitly stated in OWL statements. In our approach, two atomic nodes with different names are regarded as different in the equality checking, even though the flag indicates non-UNA. Thus, this algorithm is paraphrased *UNA for atomic objects in the non-UNA condition*.

3.2 Equivalency and Disjointness of Classes

Complete Relation for Class Equivalency. In OWL, owl:equivalentClass is applicable to indicate the equivalency of two objects as class. For example, food:Wine in Food Ontology[17] is equivalent to vin:Wine in Wine Ontology with the statement of owl:equivalentClass. In addition, the other three complete relations,[18] i.e., owl:intersectionOf, owl:unionOf, and owl:oneOf also decide the equivalency of classes. If two concepts (classes) have equivalent values for these complete relational properties, the two concepts must be conceived to be equivalent. For example, vin:DryWine and vin:TableWine in Wine Ontology are equivalent as class in OWL semantics (they share the same class extensions), because the both have the same value for owl:intersectionOf property.

Explicit and Implicit Disjointness of Classes. Meanwhile, owl:disjointWith can be applied to classes to state disjoint classes. This and owl:complementOf property explicitly state that two concepts are definitely different as class. Thus, in case of no declaration of equivalency and disjointness of classes, we cannot conclude the equality as classes immediately. However, the complete relations except owl:equivalentClass decide not only the equality but also the difference of classes. For example, even though we have no direct statement of disjointness for vin:RedWine and vin:WhiteWine, the disjointness is deduced through property owl:intersectionOf and owl:hasValue restriction vin:Red of vin:RedWine and vin:White of vin:WhiteWine, because it is explicitly stated that vin:Red is different from vin:White. Furthermore, we can also conclude some useful results by resorting to the rdf graph checking mentioned above. For example, we can find that vin:CaliforniaWine is not equal to vin:ItalianWine in spite of no explicit information of disjointness, because the graph equality checking deduces that vin:CaliforniaRegion, in which vin:CaliforniaWine is located, is different from vin:ItalianRegion, in which vin:ItalianWine is located, even if we are in non-UNA.

However, for atomic concepts that have no edges except being pointed as superclass, we cannot conclude that Man is disjoint to Woman, if those concepts are atomic in non-UNA. Thus, we are forced to do very laborious work to describe common knowledge such as Man and Woman are disjoint, Plant and Animal are disjoint, Ape and Monkey are disjoint, Virus and Bacteria are disjoint, and so on[19].

[17] http://www.w3.org/TR/2004/REC-owl-guide-20040210/food.rdf

[18] http://www.w3.org/TR/owl-ref/#DescriptionAxiom

[19] Actually, 58% is for class disjointness in lines of pizza.owl for only 23 pizza and 29 pizza toppings. The number of lines for disjointness will explode with the number of classes.

ANSI Common Lisp specifies that CLOS classes are pairwise disjoint if they have no common subclass and one class is not a subclass of the other. Namely, each class is disjoint to the others as default until we connect them in superclass relation or set a common subclass. This agreement is supported by the premise that an object in CLOS is typed to only one class. In the RDF universe, an entity may be typed to more than one class. So, the nature of disjointness in CLOS is not applicable in the RDF universe in theory. However, in SWCLOS, the pseudo multiple-classing machinery is implemented using the CLOS class and multiple-inheritance mechanism. Therefore, from the viewpoint of CLOS, the algorithm of disjointness for CLOS is still valid in the RDF universe in virtue of CLOS. In the next section, we introduce an idea of *role concept* that is divided from *substantial concept* with the premise of pairwise disjointness.

4 Substantial Concepts and Role Concepts

4.1 Ontological Categories and Disjointness

OWL provided the description of class disjointness and forced us labor-intensive work as described above. W3C new recommendation for OWL, OWL 2 specification [16,17], attempts to solve the disjointness problems without ontological consideration in depth. Person may be described as owl:disjointUnionOf Man and Woman in OWL 2. However, we are still forced to describe explicitly disjointness for all disjoint classes, or basic atomic concepts. We strongly claim that the approach to describe disjointness must be more well-founded on ontological consideration.

Sowa [18,19] showed a lattice of the top-level ontological categories of things. Each of the twelve elemental concepts in the top ontology has different characteristics and those combinations, i.e., *independent*, *physical*, *relative*, *abstract*, and *mediating*. The concepts of the independent exist itself and they show the firstness. The concepts of the relative or *role* only live with the firstness and they show the secondness. The mediating describes concepts that mediate the firstness and the secondness.

Guarino [20] parted ontology into two catagories, i.e., *particular* that represents substantial entities and *universal* that is the category of entities required to describe the particulars. Physical objects, abstract processes, phenomena, quality, and materials fall into the particular, and attributes, relations are categorized into the universal.

Mizoguchi, et al., developed an ontology building tool called Hozo [21,22] based on ontological deep discussion and have utilized Hozo for many application field of ontology building. Using Hozo, ontology builders can easily construct complex concepts that are composed of substantial sorts and non-substantial roles. For example, Wife is a part of Family and composed of Woman and Wife-role. The concept Woman is a substantial and may have slots of gender, age, etc. The role concept Wife-role is not a substantial, in other words, it always requires substantial concepts to work, but may have its own slots such as married-year, spouse, etc. In a sense, it is regarded that the concept Family represents the context in which the concept Wife is activated from Woman with Wife-role.

We also proposed *Aspect Theory* of ontology in the study of *Knowledgeable Community* [23], which is a framework of knowledge sharing and reuse based on a multi-agent architecture. In this framework, while ontologies are the minimum requirement for each agent to join the community, each of heterogeneous ontologies describes an aspect of an entity and knowledge. A mediator agent that embodied knowledge for mediation helps other agents to communicate each other. In this theory, the aspect may be rephrased as a context on which an agent focused for discourse. For example, a concept Temple is an aggregation of concepts in aspect of religion, cultural asset, building architecture, corporate body, and so on. In most case without communication, we usually focus on one aspect of entity and do not need to take care of the other aspects in a particular context. However, for agents in a particular discourse, the mediator translates heterogeneous ontologies from one to another and mediates agent's speech acts that are broad-casted in the community.

4.2 Introduction of Role Concepts

In order to solve the labor-intensive disjointness problem, we propose two ontological categories according to [22], i.e., substantial concepts and role concepts, and realize them on top of RDF and extended OWL semantics. The substantial concepts are described in OWL, but we adopt the assumption of implicit disjointness for substantial concepts in the same way as CLOS described above. On the other hand, a role concept is an extension of owl:Restriction. Neither owl:disjointUnionOf nor owl:AllDisjointClasses in OWL 2 are introduced. Instead, we extend owl:Restriction, which has property-value restrictions but usually no name and no super restrictions in OWL, to the role concept that is able to have a name and supers. The instance of role is attached to an instance of substantial classes in the same way as owl:Restriction provides the definition of *predicate/object* at *subject* or an instance of substantial class. A complex concept is composed of a substantial class and role concepts. For example, a complex concept Husband is composed of Man and Husband-Role that has spouse and marriage-date properties, and Teacher is composed of Person and Teacher-Role that has subject and classInCharge property. The discussion of disjointness on role concept is meaningless, because the role concept cannot have any instance by itself as well as owl:Restriction. Husband and Teacher can share individuals, but those individuals should be interpreted as instances of Man in Husband and Person in Teacher.

5 Open World Assumption and Ternary Truth Value

5.1 Auto-epistemic Local Closed World Assumption

Negation as Failure (NaF) is a well-known convention for inference in Closed World Assumption (CWA). This convention is, however, not applicable in World Wide Webs. Therefore, two queries are usually issued as "P?" and "not P?" for query-answer systems. In case that we cannot obtained any results with two queries, it may be called unknown. As we see so far, rigorous non-UNA and full Open World Assumption in Semantic Webs turn out to be shortcomings in practical ontology building. The implicit disjointness principle adopted in SWCLOS is very useful all over the life-cycle of ontology engineering. This principle can be rephrased such that we assume classes are pairwise disjoint until the equality or disjointness are explicitly axiomatized. In some sense, it is a kind of default reasoning. A reasoner replies that the concept A is disjoint with the concept B because of no evidence that supports it. After the statement that the concept A and B has a subclass C, the same agent replies the concept A and B are not disjoint. However, even so, note that SWCLOS signals an alarm if a user attempts to make an instance of class A and B that is explicitly stated as disjoint, and also note that SWCLOS implicitly makes a *shadowed*-class as a subclass in CLOS of class A and B, if A and B are not explicitly disjoint and it is required by users [3].

Concerned with the existential restriction of property or owl:someValuesFrom, the full OWA is also meaningless from the viewpoint of ontology building, since the existential restriction under the OWA means the possibility that a satisfiable value may be defined somewhere in WWW or someone in the team members may add a proper constraint tomorrow or after. The full OWA implies that ontology builders cannot know all for target ontologies. However, this assumption is not enjoyable in actual fact in personal and collaborative ontology building process. It is natural to distinguish the local world for target ontologies and the given general WWW. Hence, we have introduced the notion of *auto-epistemic local closed world assumption*. In this idea, agents can introspectively check their knowledge within their extent of capabilities. An agent sits in locally closed world as environments around it. The flag for auto-epistemic local closed world assumption is set true as default in SWCLOS, and the satisfiability for slot value is aggressively checked even in case of the existential restriction. Namely, if an existential restriction is not satisfied, then the interpretation is not satisfied. Setting the flag false means the completely full OWA. In this case, no alarm is signaled for the existential restrictions.

5.2 cl:Subtypep

`cl:subtypep` in ANSI Common Lisp returns two values, say, *value1* and *value2*. Table 2 is taken from ANSI Common Lisp specs[20].

Table 2. Two Return Values on cl:subtypep(type1, type2)

$value1$	$value2$	meaning
true	true	$type1$ is definitely a subtype of $type2$.
false	true	$type1$ is definitely not a subtype of $type2$.
false	false	subtypep could not determine the relation ship, so $type1$ might or might not be a subtype of $type2$.

If *value1* is true, then *value2* is definitely true. So, a return value of pair ⟨t, nil⟩ never happens in ANSI Common Lisp.

We extended this semantics and applied it for subtype (subclass) predicates in RDFS `gx:subtypep` and OWL `gx:subsumed-p` in addition to type predicate `gx:typep`. Namely, we see that ⟨t, t⟩ is true value, ⟨nil, t⟩ is false value, and ⟨nil, nil⟩ is unknown

[20] http://www.franz.com/support/documentation/8.1/ansicl/dictentr/subtypep.htm

value in RDF(S) and OWL semantics. The ternary truth table are used for the subsumption computation and elsewhere in SWCLOS, see the details in [3].

6 Conclusions and Future Work

In this paper, we described an overview of several knowledge representation languages around World Wide Webs, focusing on semantics of languages. We claimed that today's OWL, including OWL 2, embraces some drawbacks for the practical usage. It seems to lead people into a blind alley without thinking what ontology is and how it should be reflected in the OWL specification. Common Logic intends to be a common framework of concrete knowledge representation languages including RDF and OWL. Although actual dialect implementation of Common Logic is not emerging and no one can foresee the future of Common Logic, we are sure that our experience for SWCLOS suggests the future of something else than today's OWL.

References

1. Hayes, McBride, P.B.: RDF Semantics. W3C Recommendation (2004), http://www.w3.org/TR/rdf-mt/
2. Smith, M.K., Welty, C., McGuinness, D.L.: OWL Web Ontology Language Guide. W3C Recommendation (2004), http://www.w3.org/TR/owl-guide/
3. Koide, S., Takeda, H.: OWL-Full reasoning from an object oriented perspective. In: Mizoguchi, R., Shi, Z.-Z., Giunchiglia, F. (eds.) ASWC 2006. LNCS, vol. 4185, pp. 263–277. Springer, Heidelberg (2006)
4. Tarski, A.: Introduction to Logic. Dover, New York (1946/1995); This book is an extended edition of the book of title "On Mathematical Logic and Deductive Method," which appeared at 1936 in Polish
5. McDermott, D.: Tarskian semantics, or no notation without denotation! Cognitive Science 2, 277–282 (1978)
6. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice Hall, Englewood Cliffs (2003)
7. McGuinness, D.L., van Harmelen. F.: OWL Web Ontology Language Overview, W3C Recommendation (2004), http://www.w3.org/TR/owl-features/
8. Patel-Schneider, P.F., Hayes, P., Horrocks, I.: OWL Web Ontology Language Semantics and Abstract Syntax, W3C Recommendation (2004), http://www.w3.org/TR/owl-semantics/
9. Patel-Schneider, P.F., Hayes, P., Horrocks, I.: OWL Web Ontology Language Semantics and Abstract Syntax section 5 rdf-compatible model-theoretic semantics, W3C Recommendation (2004), http://www.w3.org/TR/owl-semantics/rdfs.html
10. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press, Cambridge (1991)
11. Koide, S., Takeda, H.: Meta-circularity and mop in common lisp for owl full. In: The 6th European Lisp Workshop ELW 2009, pp. 28–34. ACM, New York (2009)
12. Smith, B.C.: Reflection and Semantics in Lisp. In: 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1984, pp. 23–35. ACM, New York (1984)
13. Hayes, P., Menzel, C.: A Semantics for the Knowledge Interchange Format. In: IJCAI 2001 Workshop on the IEEE Standard Upper Ontology (2001)

14. Neuhaus, F.: The Semantics of Modules in Common Logic. In: Smith, B., Mizoguchi, R., Nakagawa, S. (eds.) Interdisciplinary Ontology, Open Research Centre for Logic and Formal Ontology, Keio University, vol. 3, pp. 107–117 (2010)
15. Guha, R.V., Hayes, P.: Lbase: Semantics for Languages of the Semantic Web, Note, W3C (2003)
16. Motik, B., Patel-Schneider, P.F., Grau, B.C.: OWL 2 Web Ontology Language Direct Semantics, W3C Recommendation (2009), http://www.w3.org/TR/owl2-direct-semantics/
17. Carroll, J., Herman, I., Patel-Schneider, P.F.: OWL 2 Web Ontology Language RDF-based Semantics, W3C Recommendation (2009), http://www.w3.org/TR/owl2-rdf-based-semantics/
18. Sowa, J.F.: Top-level Ontological Categories. Int. J. Hum.-Comput. Stud. 43(5-6), 669–685 (1995)
19. Sowa, J.F.: Knowledge Representation: Logical, Philosophical, and Computational Foundations. Brooks Cole Publishing, Monterey (1999)
20. Guarino, N.: Some Ontological Principles for Designing Upper Level Lexical Resources. In: Rubio, N., Castro, T. (eds.) 1st Int. Conf. Lexical Resources and Evaluation, pp. 527–534 (1998)
21. Mizoguchi, R., Sunagawa, E., Kozaki, K., Kitamura, Y.: The Model of Roles within an Ontology Development Tool: Hozo. Appl. Ontol. 2(2), 159–179 (2007)
22. Kozaki, K., Sunagawa, E., Kitamura, Y., Mizoguchi, R.: Role Representation Model Using OWL and SWRL. In: 2nd Workshop on Roles and Relationships in Object Oriented Programming, Multiagent Systems, and Ontologies (2007)
23. Takeda, H., Iino, K., Nishida, T.: Agent Organization and Communication with Multiple Ontologies. Int. J. Cooperative Inf. Syst. 4(4), 321–338 (1995)

Generating Code for Associations Supporting Operations on Multiple Instances

Mayer Goldberg and Guy Wiener

Ben-Gurion University, Beersheba, Israel
{gmayer,gwiener}@cs.bgu.ac.il

Abstract. Associations between objects are one of the most fundamental concepts in object-oriented design. The choices of how to implement associations determine how operations on the associated instances are performed: Sequentially or in parallel, with or without cached results, and with a transient or persistent effect. In this work, we propose a scheme that allows for generated code to support different methods of operating on associated instances, without requiring changes to the client code. These methods include using indices, traversing the association in parallel, or using a database. Instead of the sequential iterator interface, we propose to use an interface that include operations over multiple instances: *Foreach*, *Filter*, *Map* and *Fold*. This interface allows for realizing designs that involve sending messages to multiple associated instances, such as UML sequence and communication diagrams. The realization does not depend on the implementation details of the associations.

1 Introduction

Generating code from UML diagrams requires translating the high-level UML specifications into concrete statements in some programming language. UML specifications differ from statements in Object-Oriented Programming Languages (OOPL) in the following aspects. First, UML statements have a richer semantics then their OOPL counterparts: Not every element in a UML diagram can be expressed as a single statement in an OOPL. Second, UML is more abstract then OOP code: It specifies the expected structure and behavior of software components, not how to implement it.

Associations between classes and between objects are a basic part of UML class and object diagrams. They also affect sequence and communication[1] diagrams, since that object can send messages to other associated objects. The gap between the semantics of associations in these diagrams and OOPL has been discussed extensively in the UML literature. The full semantics of associations, as described in UML reference manual [1], includes bi-directionality and multiplicity constraints, as well as advanced features that have no direct equivalence in OOPLs: Relations between associations (subset, re-definition, etc.), association classes, and general OCL constraints[2]. In their seminal work on Object-Oriented Analysis and Design (OOAD), Martin and Odell discuss this problem [4]. They suggest to implement bi-directional associations

[1] UML 2 *Communication diagrams* were called *collaboration diagrams* in UML 1.

[2] For details on OCL, see [2,3].

by using either pairs of references or association objects. Craig Larman suggests refining bi-directional associations to uni-directional ones [5]. Several works [6,7,8] offer different implementations to associations that preserve more of their original semantics. It is clear from these works that implementing associations requires several code statements. This fact suggests that code generation would be useful.

The second aspect, namely that UML is more abstract then OOP code, is commonly overlooked in the literature. The gap between the specification and the code is intentional. Harrison et al., in their detailed work on mapping UML specifications to Java, explain that a model expressed independently of a specific implementation provides greater flexibility [9]. This flexibility is required: According to [10], the cost of maintenance is 90% from its development cost, and [11] show that 80% of the maintenance cost are dedicated to *perfective activities*[3] and *adaptive activities*[4]. Therefore, any re-use of the *model* that is independent of a platform or performance constraints can lower the cost of maintenance.

The semantics of UML associations only specifies the relations that the system should maintain. The details of how to implement and use them are left out. Specifically, the UML standard does not specify how to loop over multiple instances, or how to send a message to associated instances. All of the works mentioned above focus on providing a single implementation to associations, based on standard collections frameworks — For example, Java collections[5]. This approach provides the richer semantics of associations in code, but does not allow for replacing the implementation without re-writing the code. For example, most standard collections frameworks provide only sequential traversal over an entire collection. This limitation neither appears nor is implied by the UML standard. We would like to allow the developer to choose between sequential and parallel traversal by changing a property of the association, without re-writing the code. Fast access to elements in a collection with specific values requires in most frameworks adding a supporting data structure explicitly (E.g., a map from integers to persons in a collection that are of a given age). We would like to allow for adding such indices as a property of the association, or even let a smart code-generator decides which indices should be used.

In this work we present a code generation scheme that allows for replacing the generated implementation of associations without requiring changes to the rest of the code. Section 2 describes the interfaces for accessing, updating, and traversing over associations. Section 3 describes different schemes for generating implementations for those interfaces. Section 4 describes implementing dynamic designs (sequence and communication diagrams) using these interfaces. Section 5 provides an example of implementing and using this scheme to improve the implementation of associated classes. Section 6 concludes.

2 Interfaces

To replace the implementation of associations independently of the client code[6], we must provide uniform interfaces for the following operations over associations:

[3] Activities that improve the software.

[4] Activities that adapt the software to new platforms.

[5] http://java.sun.com/docs/books/tutorial/collections

[6] “Client code” here refers to the code that uses the associations, as opposed to the code that implements the associations.

1. Adding and removing pairs of associated instances from the relation.
2. Traversing over the instances associated with a given object.

We ignore the case of changing the set of instances that are associated with a given object, since it can be implemented by adding and removing pairs from the relation. Operating on pairs instead of unrelated collections allows the implementation to enforce bi-directionality and multiplicity constraints. This approach is common to many of the works mentioned above.

The second kind of operations, traversing associated instances, requires special attention. Most work on associations propose to use a variation of the Iterator pattern [12]. This pattern provides a way to perform a sequential traversal over a data structure without exposing its implementation. This approach, however, has several drawbacks:

1. It forces the traversal to be sequential and does not enable parallel traversal, even when there are no data dependencies between the iterations.
2. It does not encapsulate the selection of elements. Therefore, optimizations like using hash-tables or caching must be a part of the client code.
3. Iterators, together with loop commands (`for`, `while`, etc.) often serve for implementing operations over a range of elements. The body of the loop represents an operation over a single element and is dependent on its internal structure. The iterator pattern encapsulates the traversal, but exposes the structure of the traversed elements. Moving this implicit operation to a method in the class of the element and explicitly applying this method on all associated instances would provide better encapsulation.

To overcome these drawbacks, we propose to use different set of interfaces, similar to the one outlined in [4]:

- The *Association* interface represents the relation between two types.
- The *AssocEnd* interface represents the collection of instances that are associated with a given object.
- The *Foreach*, *Map*, *Filter*, and *Fold* interfaces represent operations over a collection of instances. They are called *aggregator interfaces*.

The aggregator interfaces are *generated* and include sub-sets of the messages from the associated type. The implementations of these interfaces are also generated and include the concrete operations over the collection. The *Association* and *AssocEnd* interfaces are not generated, since that their operations signature are constant. However, the implementations of *AssocEnd* are parametrized by the concrete type of aggregators that they returns. Therefore the implementations of *AssocEnd* are either generated, or take *AssocEnd* factories as parameters (See the *factory* design pattern at [12]). Similarly, the implementation of *Association* is also parametrized by the concrete type of *AssocEnd*s that it returns. Figure 1 shows these interfaces. Figure 2 shows a loop that breaks encapsulation. Figure 3 shows our approach: Accessing attributes is moved to methods and the loop is replaced with using the above interfaces. The following sections describe these interfaces in greater detail.

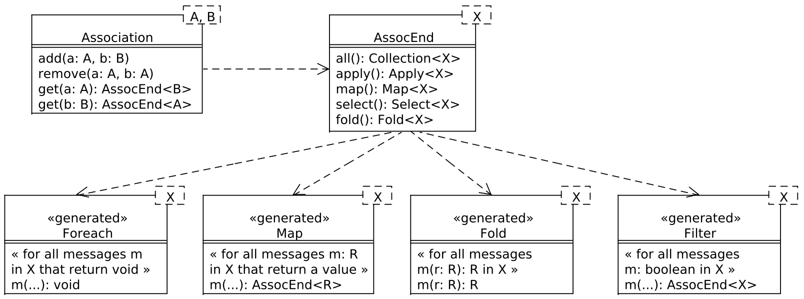


Fig. 1. The interfaces for associations

```
for (Person p: getEmployees()) {
  if (p.geName().getSalary() > limit) p.setSalary(p.getSalary()*1.2);
}
```

Fig. 2. An example of the body of the loop that breaks encapsulation

```
public class Person {
  public boolean earnsMoreThan(int x) { return getSalary() > x; }
  public void giveRaise(double d) { setSalary(getSalary() * d); }
}
getEmployees().filter().earnsMoreThan(limit).foreach().giveRaise(1.2);
```

Fig. 3. An encapsulation-preserving approach to associations

2.1 The Association Interface

The Association⟨A,B⟩ interface represents the relation itself. It contains the operations for adding and removing pair from the association. The get operations return an AssocEnd. The getter methods of the participating classes, A and B, delegate to the get operations in the association. Figure 4 shows a simple association. Figure 5 shows the Association interface and its relations with the participating classes, where `get` operations are implemented as `return f.get(this)`. As mentioned above, the implementation of *Association* requires a decision which concrete *AssocEnd*s to return. This decision can be taken either at design time, by hard-coding it or generating code for it, or at run-time, by using a factory.

2.2 The Association End Interface

The AssocEnd⟨X⟩ interface represent one end of the association. The get operation in Association returns an instance of this interface. This instance represents all the instances of the opposite type that are associated with the given object. The interface has two kinds of operations:

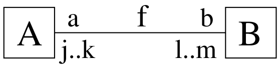


Fig. 4. A minimalistic example of an association

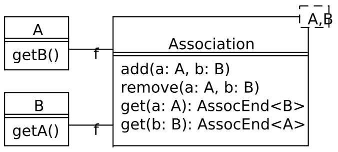


Fig. 5. The association interface and participating classes

- The operation all returns the associated objects as a platform-specific collection. For example, in our Python implementation from Sect. 5, it returns a list of all associated objects.
- The operations foreach, map, filter, and fold return an *aggregator*. The aggregator represents operations over the collection of associated instances, without exposing the implementation details. The aggregation interfaces are described below.

As mentioned above, the implementation of *AssocEnd* requires a decision which concrete instances of *foreach*, *map*, *filter* and *fold* to return. These decisions can be taken either at design time, by hard-coding it or generating code for it, or at run-time, by using a factory per aggregator type.

2.3 Aggregation Interfaces

Foreach, *Map*, *Filter*, and *Fold* are *aggregation interfaces*. They are named after higher-order functions from functional programming languages. They represent operations over a set of associated instances. The operations in each aggregation interfaces derive from the type of the instances that it wraps. Therefore, these interfaces are *generated* for each type that participates in an association. Figure 6 shows an example of a type and the aggregation interfaces that are derived from it. The details for each interface follow.

Foreach. The *Foreach* aggregator represents sending a message m_i to all the associated instances, where m_i returns no value. The implementation is not required to wait for the return value, so it can be parallel or asynchronous. Figure 7 shows a sequential generated code of a single method in an Foreach aggregator. Applying a message m on the associated instances of a is coded as `a.getB().foreach().m()`.

Map. The *Map* aggregator is similar to *Foreach*, but for messages that has a return value. The map higher-order function is described in "Anatomy of LISP" [13], and was even a part of the APL programming language [14]. It is similar to the OCL collection operation `collect`, see [2] for details. The collection of return values is returned as an aggregator interface itself. Again, the decision which concrete AssocEnd to use can be taken at design-time or run-time. Figure 8 shows a sequential generated code of a single method in a Map aggregator. Since that Map requires the return values, it can run in parallel or send asynchronous messages, but must wait for all the responses to arrive. Getting the mapped values of a message m on the associated instances of a is coded as `a.getB().map().m()`. Aggregated operations can be concatenated. For example, `a.getB().map().m().foreach().f()`.

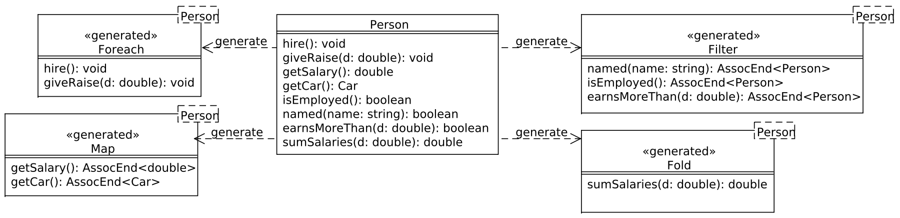

Person
«generated» Foreach
hire(): void
giveRaise(d: double): void
generate
Person
hire(): void
giveRaise(d: double): void
getSalary(): double
getCar(): Car
isEmployed(): boolean
named(name: string): boolean
earnsMoreThan(d: double): boolean
sumSalaries(d: double): double
generate
Person
«generated» Filter
named(name: string): AssocEnd<Person>
isEmployed(): AssocEnd<Person>
earnsMoreThan(d: double): AssocEnd<Person>
Person
«generated» Map
getSalary(): AssocEnd<double>
getCar(): AssocEnd<Car>
generate
generate
Person
«generated» Fold
sumSalaries(d: double): double

Fig. 6. The type Person and generated aggregation interfaces

```
public class ForeachPerson implements Foreach<Person> {
  public void giveRaise(double d) {
    for (Person p: associatedPersons)
      p.giveRaise(d);
  }
}
```

Fig. 7. A sequential implementation of the method giveRaise in Foreach⟨Person⟩

```
public class MapPerson implements Map<Person> {
  public AssocEnd<Car> getCar() {
    AssocEnd<Car> ret = (select a concrete AssocEnd);
    for (Person p: associatedPersons)
      ret.add(p.getCar());
    return ret;
  }
}
```

Fig. 8. A sequential implementation of the method getCar in Map⟨Person⟩

Filter. The *Filter* aggregator filters associated instances that return `true` in response to a boolean method m_i. It is similar to the OCL collection operation `select`, see [2] for details. Similarly to *Map*, the selected instances are returned as an instance of AssocEnd, whose concrete type is chosen either at design- or run-time. The parametric type of the returned association end is the same as the one of the *Filter* aggregator. Figure 9 shows a sequential generated code of a single method in a *Filter* aggregator. Like *Map*, *Filter* can have a parallel or asynchronous implementation, but it returns only after receiving all the responses. Getting the selected associated instances of a by a boolean message m is coded as `a.getB().filter().m()`.

Fold. The *Fold* aggregator performs an operation over the associated instances where each step depends on the result of the previous one[7] It is similar to the OCL collection operation `iterate`, see [2] for details. The signature of a method that performs such operation is such that it takes as a parameter a value of the same type as it returns:

[7] The *fold* operation here is from the first elements to the last. The opposite *fold* can be implemented in the same way.

```
public class FilterPerson implements Filter<Person> {
  public AssocEnd<Person> earnsMoreThan(double d) {
    AssocEnd<Person> ret = (select a concrete AssocEnd);
    for (Person p: associatedPersons)
      if (p.earnsMoreThan(d))
        ret.add(p);
    return ret;
  }
}
```

Fig. 9. A sequential implementation of the method earnsMoreThan in Filter⟨Person⟩

```
public class FoldPerson implements Fold<Person> {
  public double sumSalaries(double d) {
    double ret = d;
    for (Person p: associatedPersons)
      ret = p.sumSalaries(ret);
    return ret;
  }
}
```

Fig. 10. A sequential implementation of the method sumSalaries in Fold⟨Person⟩

$m_i(r : R) : R$. The folded operation returns the result of the last operation in this chain. The implementation of *Fold* must be sequential. Figure 10 shows an example of a generated code of a single method in a *Fold* aggregator. Getting the final folded result r2 of message m over the associated instances of a with initial value r1 is coded as `R r2 = a.getB().fold().m(r1)`.

3 Implementation

The Association, AssocEnd, and the aggregator interfaces only specify *what* are the operations over the association, and not *how* to implement them. The concrete implementation may vary in the following aspects:

1. Where to store the relation itself.
2. How to implement the traversal operations of the Foreach and Map aggregators: Sequentially or in parallel.
3. How to implement the selection operations of the Filter aggregator: By traversing the association or using auxiliary data structures.

3.1 Storing the Relation

There are two options for storing the data of the relation:

Internal. The relation is stored in the main memory.
External. The relation is handled by an external component, such as a database or a storage service.

Internal Storage. Storing the relation in main memory, as a part of an OOPL class hierarchy, has been discussed thoroughly in the works mentioned above. Figure 11 shows a design example for implementing an association using classes (in an OOPL). Figure 12 outlines the code for this implementation.

External Storage. The interfaces discussed in Sect. 2 can act as a uniform façade for persistency components, such as databases or files. As is, the proposed interfaces can not bridge the gap between the object-oriented design and relational databases or sequential files. However, it can provide a common interface for external persistency solutions. For example:

- Object-to-Relational Mapping (ORM) systems, e.g. Hibernate[8]
- Object-Oriented Databases (OODB), e.g. DB4O[9]
- Files — See [15] for Object-to-File serialization techniques

Having a common interface to different solutions allows the developer to switch between solutions seamlessly, thus removing a potential vendor lock. Figure 13 shows an example of hiding the persistency details by using the Association interface: It includes extra code to update a DB4O database.

3.2 Traversal Operations

The iteration over an association does not have to be sequential, as in the examples in Sect. 2.3. Unlike *Fold* operations, *Foreach*, *Map*, and *Filter* operations do not imply a specific order in which the aggregated message is sent to the associated instances. *Foreach* operations can be sent asynchronously, without waiting for a reply. Figure 14(a) shows a sequence diagram of performing an *Foreach* operation asynchronously. *Map* and *Filter* operations must wait for returned values, but can still be performed in parallel. Figure 14(b) shows a sequence diagram of performing a *Map* operation using several threads.

The parallel implementation of traversal operations in similar to the implementation of *parallel iterators*, as described in [16]. However, parallel iterators and parallel operations differ in several points. First, parallel iterators require changes to the client code. They implements the standard iterator interface, but the programmer still has to add threads to the code. Second, in the cases of *map*, *filter* and *fold*, the programmer has to add a *reduce* method to collect the results. Finally, parallel iterators encourage exposing the structure of the traversed elements, just as sequential ones (see Sect. 2).

3.3 Selecting Instances

A common scenario of traversing over associated instances is finding an instance, or a set of instances, that satisfies a condition — E.g., has a given name. The operations in the *Filter* interface (Sect. 2.3) represent this scenario. Figure 9 shows the most naïve implementation of this scenario.

A common optimization of this scenario is to keep a cache — E.g., a hash-table mapping names to instances with that name. This optimization reduces the run-time

[8] http://www.hibernate.org
[9] http://www.db4o.com

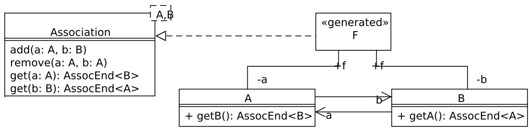


Fig. 11. Storing a relation in main memory

```
public class F
implements Association<A, B> {
  public void add(A a, B b) {
    a.b.add(b);
    b.a.add(a);
  }
  public void remove(A a, B b) {
    a.b.remove(b);
    b.a.remove(a);
  }
  public AssociationEnd<B> get(A a) {
    // return an AssocEnd of a.b
  }
  public AssociationEnd<A> get(B b) {
    // return an Assocnd of b.a
  }
}
```

Fig. 12. Implementing an association using the fields approach

```
public class F
implements Association<A, B> {
  public void add(A a, B b) {
    a.b.add(b);
    b.a.add(a);
  ⇒ db.set(a);
  ⇒ db.set(b);
  ⇒ db.commit();
  }
  public void remove(A a, B b) {
    a.b.remove(b);
    b.a.remove(a);
  ⇒ db.set(a);
  ⇒ db.set(b);
  ⇒ db.commit();
  }
}
```

Fig. 13. Implementing add and remove using DB4O. Marked lines are DB4O-specific.

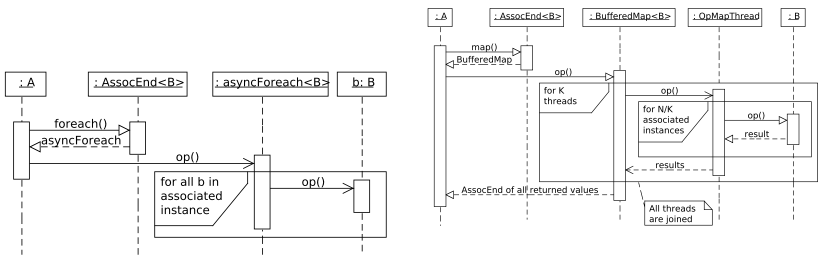
in an order of magnitude. Another possible form of cache is to query a database, as discussed in Sect. 3.1.

Caching requires adding code to the add and remove operation and change specific Filter operations. Implementing it as a part of the client code makes the client code dependent of the specific association implementation. A better solution is to generate this code as a part of the implementations of the Association and Filter interfaces, thus de-coupling the client code from the optimization code. Figures 15 and 16 shows the modifications to Association and Filter when caching a property. Figure 17 outlines a similar method using an external database.

4 Implementing Behavioral Designs

Associations are declared in UML class diagrams, and may appear in all diagrams that specify the interaction between instances, including sequence and communication diagrams. In behavioral designs, an object can send a message to other visible objects: Either associated objects or references returned by a previous message. When the association is between two objects, sending a message is implemented as a method call[10].

[10] Assuming that both objects are within a single process.



(a) Asynchronous *Foreach* (b) Buffered *Map* using K threads over N instances

Fig. 14. Sequence diagrams for parallel variations of traversal operations. Asynchronous operations are marked with a non-filled arrow-head.

```
public class F
implements Association<A, B> {
  public void add(A a, B b) {
    a.b.add(b);
    b.a.add(a);
    a.cache.put(b.key, b);
  }
  public void remove(A a, B b) {
    a.b.remove(b);
    b.a.remove(a);
    a.cache.remove(b.key, b);
  }
}
```

Fig. 15. Caching the property B.key

```
public class FilterB
implements Filter<B> {
  public AssocEnd<B> isKey(Key k) {
    AssocEnd<B> ret =
      (select a concrete AssocEnd);
    ret.add(a.cache.get(key));
    return ret;
  }
}
```

Fig. 16. Selecting an instance of B by B.key using the cache

The same solution does not apply when the message is sent to multiple associated instances. The textbook solution, appearing in all the works cited above, is to use a sequential iterator or a for loop. In some cases this choice is hard-coded into the design itself as a *loop* block in a sequence diagram. This approach may appear to be straightforward, but has two drawbacks: It is inherently sequential, and makes the client code dependent on the exact implementation of the association.

Our work suggests a different approach. Instead of translating the design specification of sending a message to multiple instances into a loop, the developer can code it directly using a *Foreach*, *Map*, *Filter* or *Fold* aggregator. This approach allows for optimizations, such as parallelism and caching, and removes the dependency between the association and client code.

Figure 18 demonstrates the possible mappings from specifications in communication diagrams to implementations based on association aggregators. Figure 18(a) shows the simple case when the message does not return a value. Figure 18(b) show the case when the returned value is assigned to a local variable for future use. Figure 18(c) shows the

```
public class FilterB implements Filter<B> {
  public AssocEnd<B> keyed(Key k) {
    AssocEnd<B> ret = (select a concrete AssocEnd);
  ⇒Query q = objectContainer.query();
  ⇒q.constrain(Class.B);
  ⇒q.descend("key").constrain(key);
  ⇒ret.add(q.execute());
    return ret;
  }
}
```

Fig. 17. Selecting an instance of B by B.key using DB4O. Marked lines are DB4O-specific.

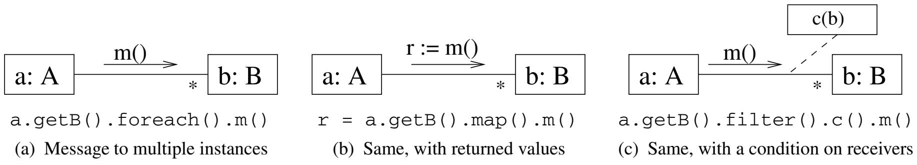

a: A
m()
*
b: B
a.getB().foreach().m()
(a) Message to multiple instances
a: A
r := m()
*
b: B
r = a.getB().map().m()
(b) Same, with returned values
c(b)
a: A
m()
*
b: B
a.getB().filter().c().m()
(c) Same, with a condition on receivers

Fig. 18. Mapping communication diagrams to code

case when there receivers must fulfill the condition c. The *Fold* aggregators, not shown here, is the default option for more general cases, such as loops.

4.1 Sample Design and Implementation

This example demonstrates how behavioral designs are mapped to code. Figure 19 specifies how a request handles several resources. First, the request asks its associated resource managers to obtain missing locks, if any. The resource managers lock any unlocked lock. Each call to a resource manager returns a resource, and the request starts all the resources. Figures 20 and 21 outlines the relevant methods. Notice that handling the returned values from obtain is done by appending the foreach operation.

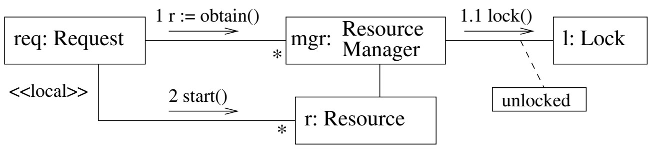


Fig. 19. A communication diagram for Request.run()

5 Implementation Example

To demonstrate our scheme and its usage, we provide the Python module assoc and code examples. The module includes functions that generates code for associations, using the classes described above.

```
getMgr().map().obtain()
        .foreach().start();
```

Fig. 20. `Request.run()`

```
r = getLocks().filter().unlocked();
r.foreach().lock();
return r;
```

Fig. 21. `ResourceManager.obtain()`

We decided to use Python in order to avoid the need to generate source code as text. We make use of Python classes being first-class objects that can be created and manipulated to create classes for the association and its ends, and add methods to the participating classes.

5.1 Implementation Details

The `assoc` module works as follows:

1. The class `Assoc_n_n` represent an association. It constructor arguments are the participating classes, the names of the association ends, and factory functions for *AssocEnd* objects. Factory function may take additional factory functions for aggregators as arguments.
2. When created, the association class adds code to the participating classes that implements the association as a pair of list fields. The added code in each class includes: (a) A wrapper to the constructor that initialize the relation data and the *AssocEnd* object. (b) A getter for the association end.
3. The association object uses the factory functions to return association ends and aggregators.
4. The *AssocEnd* and aggregator objects are initialized with a reference to a list of associated instances.
5. Each *AssocEnd* object has `onAdd` and `onRemove` methods, to handle special cases, such as caching (see Sect. 3.3).

The module supports the following aggregators: Serial traversal, parallel traversal with K threads, and a cache for filtering instances.

```
class Dept(object): pass

class Employee(object):
  def __init__(self, id, name):
    self.id = id
    self.name = name
  def getId(self): return self.id
  def hasId(self, id):
    return self.id == id
  def getName(self): return self.name
```

Fig. 22. Associated classes (Python)

```
Emps = Assoc_n_n(
 cls1 = Dept,
 cls2 = Employee,
 name1 = 'workers',
 name2 = 'worksIn',
 factory1 =
  ListAssocEndFactory(Employee),
 factory2 =
  ListAssocEndFactory(Dept))
```

Fig. 23. Association object (Python)

5.2 Usage Example

Figure 22 shows Python classes representing workers in a department. We would like to implement an association between the department and its employees, so that each department will hold a set of its workers. Figure 23 shows the code for creating the association object with default factory function. The command `Emps.add(dept, emp)` will associate the given department and employee.

When the association object is created, it creates getter methods for the association ends — E.g., the method `getWorkers()` to the class `Dept`, and `getWorksIn()` to `Employee`. The *AssocEnd* objects provide the aggregator operations. Figure 24 shows how to find a worker in a department by id.

```
def deptWorkerById(dept, id):
  return dept.getWorkers().filter().hasId(id)[0]
```

Fig. 24. Getting a worker in a department by id

The association in Fig. 22 uses the default settings: Aggregator operations are performed by iterating over the list of associated objects sequentially. Finding an employee with a given id by scanning the entire list takes a long time if a department has many employees. We solve this problem by replacing the default filter factory function. The association shown in Fig. 25 uses a *Filter* operation with caching. As expected, our measurements shows that the function from Fig. 24 runs faster by an order of magnitude with these settings. Note that no change in this function or in the above classes is required to achieve this boost in performance. Similarly, Fig. 26 shows an association that uses multiple threads for some aggregator operations. These settings can speed up batch I/O operations (for example, if employees writes something to a file), without changing any client code. This module is available from http://www.cs.bgu.ac.il/˜gwiener/software/associations.

```
Emps = Assoc_n_n(
 cls1=Dept, cls2=Employee,
 name1='workers', name2='worksIn',
 factory1=ListAssocEndFactory(
  Employee,
  filterImpl=makeListQualFilter(
   Employee.getId, Employee.hasId)),
 factory2=ListAssocEndFactory(Dept))
```

Fig. 25. Association with caching for employee ids

```
Emps = Assoc_n_n(
 cls1=Dept, cls2=Employee,
 name1='workers', name2='worksIn',
 factory1=ListAssocEndFactory(
  Employee,
  foreachImpl=
   makeListSpawnForeach(5),
  mapImpl=makeListSpawnMap(5)),
 factory2=ListAssocEndFactory(Dept))
```

Fig. 26. Association with 5 threads for *Foreach* and *Map* operations

6 Conclusions and Future Work

In this work we present a novel approach for generating code for associations. Rather than focusing on a single, sequential, in-memory implementation, we presented a flexible and open approach. Our approach is based on a set of interfaces that provide operations over the entire relation. These operations include managing the relation through the *Association* interface, and traversing it using *Foreach*, *Map*, *Filter*, and *Fold*. Traversals, excluding *Fold*, can be implemented by a multi-threaded code. The *Association* and *Filter* interfaces encapsulate optional optimizations, such as caching or using a database. This encapsulation de-couples the client code from the implementation details of the association.

The use of a closed set of interfaces for operations on associations might seem as a limitation. However, these interfaces and their combinations cover a wide range of common scenarios. For an object associated with many peers, they cover the range of operations on the whole relation, similarly to sequential access methods (BSAM, ISAM, QSAM, etc) used in record-based data processing on IBM mainframes [17]. When composed together, they provide an implementation scheme for behavioral designs in UML, as demonstrated in Sect. 4.1. In the cases that are not covered by this scheme, the *Fold* aggregator provides a fallback to sequential processing, still without exposing the implementation details.

Our current work covers the basic functionality of associations, i.e., to operate on the set of associated objects. It does not handle properties of associations, such as association classes, and *set*, *ordered*, *subset* and *override* annotations. We plan to include these topics our future work.

References

1. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Pearson Higher Education, London (2004)
2. Object Management Group: OCL 2.0 Specification. Formal specification (2005)
3. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
4. Martin, J., Odell, J.J.: Object-oriented analysis and design. Prentice-Hall, Inc., Upper Saddle River (1992)
5. Larman, C.: Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development. Prentice Hall PTR, Upper Saddle River (2004)
6. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
7. Gessenharter, D.: Mapping the UML2 semantics of associations to a Java code generation model. In: Czarnecki, K., Ober, I., Bruel, J.M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 813–827. Springer, Heidelberg (2008)
8. Akehurst, D., Howells, G., McDonald-Maier, K.: Implementing associations: UML 2.0 to Java 5. Software and Systems Modeling 6(1), 3–35 (2007)
9. Harrison, W., Barton, C., Raghavachari, M.: Mapping UML designs to Java. ACM SIGPLAN Notices 35(10), 178–187 (2000)
10. Pigoski, T.M.: Practical Software Maintenance: Best Practices for Managing Your Software Investment. John Wiley & Sons, Inc., New York (1996)

11. Martin, J., McClure, C.: Software Maintenance: The Problems and Its Solutions. Prentice Hall Professional Technical Reference (1983)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of reusable object-oriented software. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
13. Allen, J.: Anatomy of LISP. McGraw-Hill Book Company, New York (1978)
14. Iverson, K.E.: A Programming Language. John Wiley & Sons, Inc., Chichester (1962)
15. Blaha, M., Premerlani, W.: Object-oriented modeling and design for database applications. Prentice-Hall, Inc., Upper Saddle River (1997)
16. Giacaman, N., Sinnen, O.: Parallel iterator for parallelising object-oriented applications. Technical Report 669, University of Auckland (November 2008)
17. Rogers, P., Janssen, R., Otto, A., Pleus, R., Salla, A., Sokal, V.: Storage management software and hardware. ABCs of z/OS System Programming, vol. 3. IBM Redbooks (2010)

Know How and Know What for Software Processes

Jan Kožusznik, Svatopluk Štolfa, Marie Duží, Michal Košinár and Martina Číhalová

Department of Computer Science, VŠB - Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
708 33, Ostrava - Poruba, Czech Republic
{jan.kozusznik,svatopluk.stolfa,marie.duzi,
michal.kosinar,martina.cihalova}@vsb.cz

Abstract. Formal specification of a software process, as well as its optimal design, is a fundamental landmark and tenet that any successful software company must follow. Recent trends can be characterized as a knowledge-base support of the software-process development, standardization and improvement. To this end we create semantic annotations (ontologies) of processes which should serve as a stable unifying core of the software-process development. However, when doing so, we meet the problem how to transform various forms of tacit, implicit knowledge into an explicit knowledge specification that is logically tractable and machine readable. In this paper we focus on the transformation of informal tacit knowledge about a software process (or any part of the process) to the formal knowledge specification that can be used for building machine readable knowledge bases. In particular, we aim at optimizing and improving software-process development using knowledge bases which are created to the purpose of a formal description of the software-process development.

Keywords: Software process improvement, Knowledge, Rules, Facts, Knowledge base, Software process.

1 Introduction

The main goal of every software company is to develop a high-quality software, and at the same time to minimalize expenses and resources needed for the development. One way to meet such a goal is to follow good practices as they are described in software-processes standards. These software development standards are currently broadly used by numerous software companies. Every software company uses some type of software process. Even if this process is undocumented and/or unknown, it is still there[1,2]. Describing and maturing software processes is a key element of a company's strategy, because a more mature software process means higher quality and less expenses [3,4].

In order to be able to improve the software-process development, it is desirable to specify common practices and software company culture so that each member of the company knows what to do. True, almost every company has some type of a *human readable knowledge base (HRKB)* that describes a variety of practices in the company. There are even such human readable knowledge bases that describe reference software processes and/or good company practices [1]. Yet, such a HRKB is usually not computationally tractable and cannot serve as a generator for computer-aided software design and development. We need to upgrade a HRKB to a *machine readable knowledge base (MRKB)* in order to achieve an automated knowledge management.

This paper is a proposal how to fill this gap. We are going to describe the method of creating a MRKB that provides a facility to support basic assessment of software processes. We also discuss the problem of exploiting such a MRKB in the current approach to the assessment and enhancement of software processes.

The paper is organized as follows: Section 2 discusses the pros and cons of the various approaches including knowledge-based approach; Section 3 describes the fundamental tenets of the software-process development; Section 4 introduces the concept of creating, sharing and comparing knowledge bases. In Section 5 we propose a method of semi-automated assessments and evaluation of the similarity between a software process and a reference process; the focus is on the creation of a MRKB. Finally, concluding Section 6 provides a summary and discusses future research.

2 Knowledge-Based Approach to Software Engineering

Knowledge-based approach to software engineering has been dealt with already in [5-7]. The authors describe a well-defined meta-model of a software process. Benefits of the knowledge-based approach to modelling and simulation compared to a Discrete-Event Simulation (DES - [8]) and System Dynamics (SD - [9]) are discussed in [10]. Each kind of an activity as defined by the process life-cycle is examined here from the knowledge-base point of view.

Now we provide a brief summary of the comparison between Knowledge-Based Systems (KBS) in contrast to the System Dynamics (SD) and Discrete-event Simulation (DES) models.[1]

Meta-modelling: A knowledge-based software process meta-model was developed for the purpose of constructing and refining process ontology and behaviour logic (rules and computational methods). Thus the KBS meta-model and modelling ontology can be extended as needed, whereas prevailing modelling approaches (DES, SD) use a fixed vocabulary.

Modelling: The knowledge-based approaches support an incremental model development. On the other hand, DES and SD approaches require that a significant portion of the model must be fully specified prior to its evaluation.

Analysis: Knowledge-based approach supports the evaluation of both static and dynamic properties of the model such as consistency, completeness, internal correctness, traceability and other semantic checking. This concerns also the feasibility assessment and optimization. Some DES approaches provide similar facilities like Petri nets, see [11]), but in the SD model these features are not of a primary interest.

Simulation: Knowledge-based approach supports an incremental simulation, persistence of simulation execution traces, reversible simulation computation, query-

[1] Portions of this section draw on material presented in [10].

driven simulation and reconfigurable simulation space in the run time. In contrast, common DES and SD models support only a monolithic simulation and do not provide these desirable facilities.

Redesign: Knowledge-based process redesign can support automated reorganization and optimization of the process structure. No explicit support is provided by DES or SD models for automated process redesign; thus within DES or SD models these activities must be performed manually.

Visualization: DES and SD approaches provide mature facilities for creating and displaying graphic or animated views of software processes. Knowledge-based approach is rather restricted in this respect; only relations between the pre-defined knowledge rules are supported by the visualization interface.

Prototyping, Walk-through and Performance Support: The knowledge-based process engineering model provides this facility due to its open environment. None of the DES or SD packages that have been used for software processes readily support prototyping.

Administration: Assigning and scheduling specified resources to the modelled user roles is (to some degree) supported by all the above mentioned systems.

Integration: Knowledge-based approach used in [10] provides a computational workspace that makes it possible to link together particular components of the system like roles, tasks and resources. None of the DES or SD packages readily support process-based integration.

Environment Generation: KBS supports automatic transformations of a process model into a process-based computing environment in order to present a prototype of system to end-users. None of the DES or SD packages readily support this capability.

Instantiation and enactment: KBS model supports performing the modelled process using a process instance interpreter. None of the DES or SD packages that have been used for software processes readily support prototype execution.

Monitoring, Recording and Auditing: All the above mentioned systems support monitoring and similar facilities to a certain extent.

History Capture and Replay: Knowledge-based model used in [10] supports these facilities. The SD and DES environments can most probably be easily extended in order to support the history records as well.

Articulation: Here we deal with the problem of a process failure due to an unexpected event like missing resources. In such a situation the process-management should be able to diagnose the failure and cure the malfunctioning process. While this problem-solving capability has been developed and demonstrated within a knowledge-based approach in [12], it appears that it is beyond the meta-model underlying DES and SD capacities to support this function in a systematic way.

Evolution: A well-designed process should be flexible in order to meet newly emerging user requirements. Thus the process model should be easy to tune and open to reengineering and restructuring in accordance with the evolution of users demands. KBS, DES and SD models support these capabilities to a certain extent; yet the knowledge-based approach seems to be more promising in this respect due to its support of automated transformations.

Process Asset Management: Since a software process development comprises many different activities and tools, the problem of organisation these resources arises. We need to deal with collections of meta-models, models, simulation tools, instances of models, documents, human and material resources, etc. etc. Thus the process management becomes crucial for a successful process development. While the knowledge-based approach has proved to be apt for managing these activities, these capabilities appear to be beyond of the capacities of popular DES and SD packages. KBS makes it possible to easily support a tight integration of modelling, analysis and simulation [13,6]. To this end a knowledge-based process meta-model has been developed, see [6,12].

2.1 Benefits of the Knowledge-Based Approach

Summarising, we now list the activities supported by particular models. Yet when comparing different approaches, we must distinguish between their capabilities in general and those that were really supported during their testing and checking. Thus we identify those activities that are readily supported by KBS, DES or SD, or that can be easily supplemented. They are *Administration*, *Monitoring*, *recording and auditing*, *History capture and replay.*

The following activities are supported by all the addressed models, but the knowledge-based approach appears to be more appropriate: *Meta-modeling* (the meta-model and model ontology can be easily managed by KBS); *Modeling* (KBS supports an incremental development of a model); *Analysis* (this is supported only by few implementations of DES like Petri nets [11]); *Simulation* (DES and SD support only the monolithic simulation); *Redesign, Evolution* (the KBS approach supports an automatic transformation).

The following activities are supported only by the KBS approach: *Prototyping*, *walk-through and performance support*; *Integration*; *Environment generation*; *Instantiation and enactment*; *Process asset management.* Yet, in our opinion, this restriction is due to the framework used for the model development rather than due to the SED or SD methodology in general.

Concerning the *Articulation* activity it seems that the DES and SD models will most probably never support this facility; or, if they will, then only in an *ad hoc* way. Thus it appears that only the *Visualization* facility is supported in a much better way by the DES and SD models than by the KBS approach.

3 Software Process

Software process is a business process of a company specialised in the Information Technologies (IT). For a definition of a business process, we refer to[14], and quote:

"Business process is a set of one or more linked procedures or activities which collectively realize a business objective or policy goal, normally within the structure defining functional roles and relationships." Thus we can speak about a software process using the same notions as applied in the business process technology.

3.1 Basic Definitions

Referring again to [14], we recapitulate: Process definition is the representation of a business process in a form which supports automatic manipulation: modelling, or enactment by a workflow management system. Activity is a description of a piece of work that forms one logical step within a process. Activity instance (task) is the representation of an activity within a (single) enactment of a process. Actor (work performer) performs the work represented by a workflow activity instance. Work item is the representation of the work to be processed (by an actor) in the context of an activity within a process instance.

4 Knowledge Base

Knowledge base serves as an artificial memory [15]. The content of a knowledge base should comprise general rules and facts (the state of the system and its environment). As stated at the outset of this paper, we aim at using and exploiting knowledge bases and ontologies as basic building blocks of a software-process development and evaluation. In this section we introduce our basic tenets and summarise our approach to knowledge management. For details see, e.g., [16,17].

4.1 Human Readable Knowledge Base

Human readable knowledge bases (HRDB) can be found in every organization. They are used to document, share and evaluate knowledge on the company behaviour, goals and culture. However, as the acronym HRDB reveals, there are some shortcomings connected with their application.

First, such a knowledge base comprises tacit knowledge created and used by humans. Though it is easy to understand and also easily sharable by the humans who create it, its content is often vague and contains much implicitly hidden information so that its computer-aided application is almost impossible.

Documentation. The most commonly used tools to document knowledge in organizations are Wikis, project portals and also FAQs. Typical examples are applications like web documentation of RUP, portals like MSDN and Wikipedia.

Sharing. Sharing human readable knowledge typically consists in a personal communication (meetings, brainstorming, conferences etc.), possibly both in the personal and electronic form (video calls, conference calls etc.), or in a purely electronic form (e-mail client, Exchange, Lotus, intranet etc.).

Evaluation. The assessment of a HRKB is problematic, indeed. The HRKB-based evaluation of a software process can be executed only manually, which means a tough comparison of particular documents and tacit knowledge with a real process.

4.2 Machine Readable Knowledge Base

The main difference between a HRKB and a *machine readable knowledge base* (MRKB) is this. While HRKB contains an implicit or tacit knowledge, MRKB makes this knowledge *explicit* and *logically tractable*. Moreover, a MRKB is, or should usually be, equipped with an inference machine that makes it possible to infer consequences of the explicitly recorded facts and thus to increase our knowledge. Hence we can use smart algorithms instead of a manual comparison.

However, in order to be able to make use of a MRKB, we must specify a domain ontology prior to MRKB using. Such an ontology consists of two parts; base ontology describing a general software process (that is concepts like request, use case, change, sequence diagram, class etc. and basic relations between them) and a specialized ontology which is an extension of the base ontology. The specialized ontology serves as a domain ontology of a particular software process; it is a formal semantic description of the process that makes it possible to build a knowledge base for the process assessment. We will deal with this problem in more details in Section 5.

Creation and Storage. Unlike the case of using a HRDB, when we only spontaneously record company know-how, in case of creating MRKB we must carefully fill the base with exact rules and facts in a systematic way. To this end we use a domain ontology that should be specified in a formal language (for instance Prolog, OWL DL, TIL-Script).

Yet similarly as in case of HRKB, the role of a domain expert is indispensable in building up a MRKB as well. More details on formal languages used in knowledge bases can be found in [16-18] .

Sharing. There are many ways of using and sharing a MRKB. First, such a knowledge base is an indispensable part of a multi-agent system of autonomous, intelligent agents who communicate with each other *via* a MRKB. Second, it is often useful to transform the content of a MRKB into natural language so that the humans can easily understand the facts and rules contained in the base. To this end we must use an expressive specification language such as *TIL-Script* or a semantic web like *WordNet*. Third, there is also the possibility to read the content of the base in its plain form; however, this is possible only if the specification language is close to natural language and thus comprehensible to the users.

Evaluation. Once a MRKB is at our disposal, we can benefit from it in particular in the process of software evaluation. Thus we can execute a machine-aided comparison of two (or more) knowledge bases, which in turn is much easier than a manual comparison of two human readable knowledge bases.

4.3 Knowledge Base Comparison

Comparing the content of two or more machine readable knowledge bases is easy if every piece of knowledge is specified in the same formal language and thus comparable to the others. In other words, we aim at using a homogenous data model to store knowledge. Thus each piece of knowledge can be stored as a text information comparable to the other pieces of knowledge by simple set-theoretical operations like union, intersection and difference of sets.

If the above condition on homogeneous formal specification were not satisfied, we would need to build up translation and transformation routines. This is possible, of course, but such a transformation comes down with many problems which are out of the scope of this paper.

Thus for now we suppose that one and the same formal language has been used to specify particular knowledge bases. Then a simple relation to compare two knowledge bases can be defined. Let A and B be two knowledge bases viewed as sets of character strings, and let $|A \cap B|$, $|A \cup B|$ be cardinality of their intersection and union, respectively. Then the *degree of similarity* of A and B is defined as the ratio of the cardinality of their intersection and union, respectively:

$$Similarity(A, B) = |A \cap B| / |A \cup B| \tag{1}$$

This function returns a number within the interval <0, 1>. We can deal with this number as with the fuzzy value of knowledge bases similarity.

5 Knowledge Support for Software Processes

The idea of knowledge support for the assessment and evaluation of software processes is based on the fact, that in our opinion the assessment, enhancement and monitoring can be supported by the creation and usage of machine readable knowledge bases. A lot of manual procedures can be automated. The goal is to create a more effective environment for the assessment, enhancement and monitoring of software processes.

One of the many tasks of this domain area is the comparison between the reference software process and real software process that is used in the company. The issue is to find the similarity between the real software process and the reference software process and to provide an evaluation of the current state. Typically, the real process is assessed and human readable knowledge bases are searched for similarities. Everything is performed manually. Our proposal is to improve these procedures by using a machine readable knowledge base for the automatic evaluation of the similarity between the real and reference processes.

Our approach can be basically described as follows:

1. The first step is the creation of the particular reference knowledge base – *Knowledge base transformation*.

Documented software processes that are described in the human readable knowledge bases are analysed and ontology for each software process is created. Next, the ontology and knowledge obtained from the HRKB are transformed into the machine readable knowledge base. A new MRKB is created for every type of the reference software process.

2. The second step is the study and creation of the real knowledge base and the comparison of the reference and real knowledge base that is enhancing software processes with knowledge management.

A real software process is modelled and the ontology for this process is created. The obtained ontology is then mapped into the reference software process ontology. The mapped ontology set and the knowledge base obtained from the model are then transformed into the machine readable knowledge base.

Afterwards, both knowledge bases are automatically compared and the result is a number that shows the degree of similarity of real and reference software processes.

It is obvious that the first step can be performed only once for every reference software process. The result is then applied for every real software process that we want to evaluate.

5.1 Knowledge Base Transformation

We have already sketched a basic idea of how to transform a human readable knowledge base into its machine readable clone. Now, we are ready to describe the details.

At first, we should explain the basic reason for transforming the human readable KB into the machine readable KB. The answer is simple. If we want to evaluate a process that is based on some reference software process we would have to manually compare it with the reference model process described in its human readable knowledge base. With knowledge enhancements and basic approaches described above, we can sort this problem automatically, that is by transforming the reference knowledge base into the machine readable knowledge base (see Fig.1).

This can be done by building an ontology for a concrete domain (i.e. a software process) using the human readable reference documentation. Then, the knowledge base is defined by such ontology's content. We can see particular transformation in greater detail in Fig. 2.

An ontology builder has to describe the domain in the ontology data file (using tools like Protégé as shows[19] etc.) and then fill the reference knowledge base with rules and facts (using some content language like OWL-DL, TIL-Script or PROLOG) describing the reference domain (using terms and relations from domain ontology).

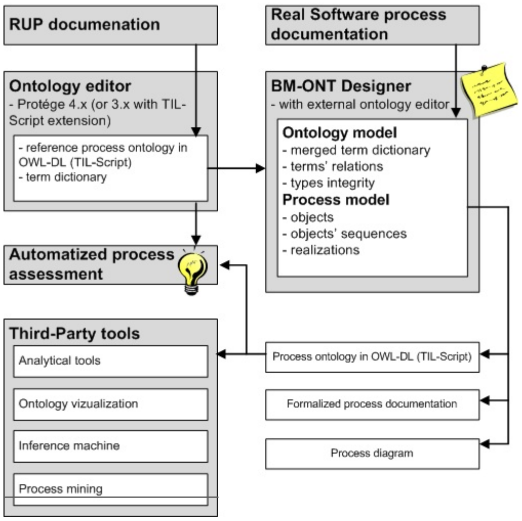


Fig. 1. Scheme of the machine readable knowledge base creation - RUP example

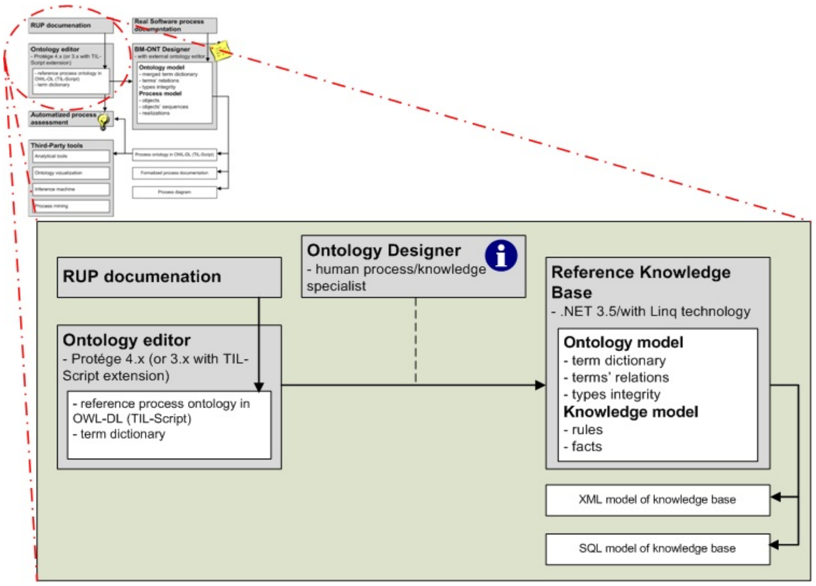

RUP documenation
Ontology editor
- Protége 4.x (or 3.x with TIL-Script extension)
- reference process ontology in OWL-DL (TIL-Script)
- term dictionary
Ontology Designer
- human process/knowledge specialist
Reference Knowledge Base
- .NET 3.5/with Linq technology
Ontology model
- term dictionary
- terms' relations
- types integrity
Knowledge model
- rules
- facts
XML model of knowledge base
SQL model of knowledge base
Real Software process documentation
BM-ONT Designer
Automatized process assessment
Third-Party tools

Fig. 2. Knowledge base transformation

5.2 Building Process Definition with Ontology Background - Example

The following example defines "Change request management" process that is part of RUP – Rational Unified Process. Simplified version of this process is shown in figure 3. Detailed description can be found in [20].

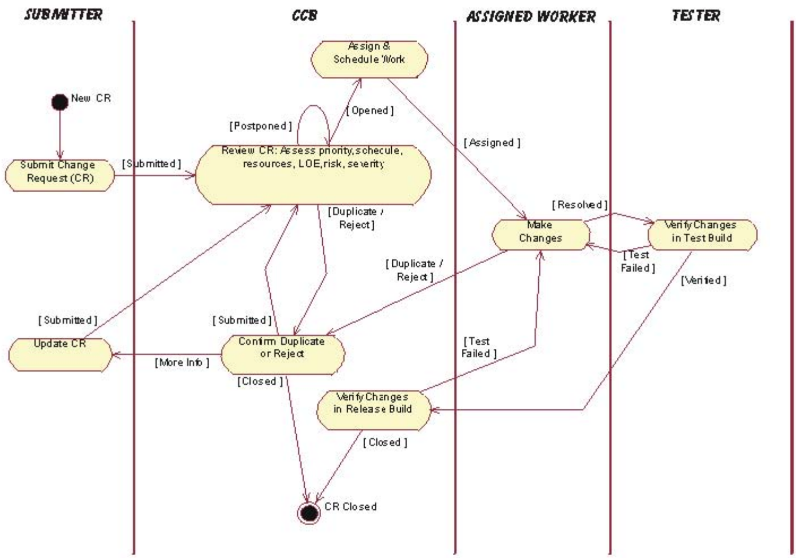


Fig. 3. Simplified version of the "Change request management" process

Ontology Background of Process Definition. To create semantic annotations for process, we have to find right and exact terms for ontology interpretations.

Software process ontology design proceeds in three levels of abstraction – Process itself, Activities (tasks) of process and Steps of activities. Therefore we define following base terms for software process domain:

Table 1. Mapping process elements

Term	*Description*	*Examples*	*Level*
Process	Designed Process itself	Change request management	0
Activities/Tasks	Most simple activities in process	Submit Change Request activity in Change Request Management process	1
Artefacts	One of many kinds of tangible byproduct produced during the development of software	Change Request of Change Request Management	1
Roles	A definition of the behaviour and responsibilities of an individual	Programmer in Change Request Management	1
Steps	Steps are simplest items of Activities.	Turn on PC. Rethink problem.	2

All other terms used in RUP Software Process documentation are specializations of the base terms defined in Table 1.

Now we can create an example ontology of Change Request Management from RUP. All defined terms are inheriting from their base classes in the above mentioned table.

Process: Change Management
Artefact: Change Request (CR)

Roles

- Submitter
- Change Control Manager (CCB)
- CCB Delegate
- Test Manager
- Project Manager
- Team member
- Tester

Activities

- Submit CR
- Update CR
- Review CR
- Confirm CR reject
- Assign CR
- Make Changes
- Verify Changes

To this end we are using TIL-Script and OWL content languages. We are able to transform any OWL ontology to the TIL-Script ontology [21]. However, only a selected subset of the TIL-Script ontologies can be transformed into the OWL ontologies. This restriction arises from the fact that the TIL-Script language is based on a higher-order logic than the OWL language.

TIL-Script Ontology Example: Ontology rules are in a uniform form:

ProcessElement/(*who* (role), Input (artifacts) → output (artifact))

This rule specifies who is responsible for a given Process element (process, activity or steps) and what input or output artifacts flow in and out. We can say that the set of these rules represents a formal functional view of the process.

Process ontology design defines three levels of abstraction:

- level 0 – the specification of a process
- level 1 – activities for individual process are specified
- level 2 – steps for elementary activities are described

Example of the defined rules follows:

Level 0 (**Process: Change Management**)
Change / (*CCB*, CR submitted → CR Closed)

Level 1 (**Process: Change Management:** Activities)

- ***Submit CR* /** (*Submitter*, CR New → CR submitted)
- ***Update CR* /** (*Submitter*, CR more info → CR submitted)
- ***Review CR*** (*CCB*, CR submitted → CR opened; CR postponed; CR rejected-duplicate)
- ***Assign CR*** (*Project Manager*, CR opened → CR assigned)
- ***Review CR PP*** / (*CCB*, CR postponed → CR opened; CR postponed; CR rejected-dupl)
- ***Confirm CR reject*** / (*CCB Delegate*, CR rejected-duplicate → CR closed, CR more info)
- ***Make Changes*** / (*Team member*, CR assigned → CR resolved)
- ***Activity steps as in normal development process***
- ***Make Changes Failed*** / (*Team member*, CR test failed → CR resolved)
- ***Verify Changes Test*** / (*Tester*, CR resolved → CR verified; CR test failed)
- ***Verify Changes Release*** / (*CCB delegate*, CR verified → CR closed; CR test failed)

Example of Level 2:

Level 2 (**Process: Change Management:** Activity: ***Make Changes***: Steps)
- ***Thinking***
  - ***Conduct review***
  - ***Create Integration Workspaces***
  - ***Develop Development Case***
  - ***Develop Iteration Plan***
- ***Performing***
  - ***Launch development Process***
- ***Reviewing***
  - ***Organize*** review and testing

As shown above, rules define also the states for input and output artefacts. This fact can be used for an automatic generation of a process behaviour [22].

OWL Example: To analyse, design and build the semantic annotation of a software process we follow the W3C recommendations on OWL/DLW3C, see [23]. From the point of view of using OWL language for the description of a software process domain we are interested in the following aspects.

1. ISA relations among classes (inheritance)
2. Whole–part relations
3. Attributes (properties)
4. Integrity constraints

Applying the above described advanced procedure we can build an effective but simple methodology for ontology designers and process modellers.

This example is only a selected part of the OWL ontology description of our example to prove the mapping of TIL-Script ontologies to OWL ontologies (and vice versa).

Change request class is defined as a subclass of Artefact class. It has properties described by restrictions of the class. These restrictions declare that Change request artefact has its State property (ChangeRequestState data type) and some other properties (i.e. pertinence to Change Request management Process – higher abstraction level).

```
<owl:Class rdf:about="#Change_Request">
  <rdfs:subClassOf rdf:resource="#Artifact" />
  <owl:equivalentClass>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#has_State"/>
      <owl:onClass rdf:resource="#Change_Request_State"/>
           <owl:qualifiedCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:qualifiedCardin
ality>

  </owl:Restriction>
  </owl:equivalentClass>
  <owl:equivalentClass>
      <owl:Restriction>
           <owl:onProperty rdf:resource="#has"/>
      <owl:onClass rdf:resource="#Change_Request_Management"/>
           <owl:qualifiedCardinality
rdf:datatype="&xsd;nonNegativeInteger">1</owl:qualifiedCardin
ality>
</owl:Restriction>
      </owl:equivalentClass>
</owl:Class>
```

This statement declares the ChangeRequestState – class for Change Request State interpretation.

```
<owl:Class rdf:about="#Change_Request_State"/>
```

Examples of interested individuals (roles):
Submitter is an element of Role class.

```
<owl:Thing rdf:about="#Submitter">
   <rdf:type rdf:resource="#Role"/>
</owl:Thing>
```

Submitted is a state of Change request artifact and is declared as an element of ChangerequestState class.

```
<Change_Request_State rdf:about="#Submitted">
    <rdf:type rdf:resource="&owl;Thing"/>
</Change_Request_State>
```

To define the function New CR between Change Request and Submitter, we would have to define new class that will describe such function.

5.3 Enhancing Software Processes with Knowledge Management

The ability of transforming human readable knowledge bases of documented software processes into machine readable knowledge bases means that we have a tool for the automatic assessment of software processes.

To this end we create two knowledge bases.

1. Knowledge base that contains data on the optimal software process transformed from human readable base containing the process' documentation (RUP documentation on the Web). Let us call this knowledge base a "*Template*".
2. The second knowledge base that contains data on the actual software process that is used in an organization. This knowledge base must be filled by an ontology expert in processing and consulting services using the same ontology dictionary as is contained in the "Template". The name of this set will be the "*Actual*".

Now, when we have two knowledge bases whose contents are documented and the real software processes we can use the comparison function as defined by (1) to evaluate the similarity of these knowledge bases.

The content comparison can be performed as an add-on to the assessment. We can directly search for differences between individual members of both sets. However, this means basically a brute-force comparison of the two sets, which is not optimal. Thus the main task of our future research is an optimization of the comparison in order to find the gaps in the "Actual" knowledge base of a process compared to the "Template".

6 Conclusions and Future Work

This paper described an approach to knowledge transformation in the domain of software (business) processes. Our methodology is driven by the fact that there are many benefits of knowledge-based support that can be used for the software process improvement. However, this is only the first step to the comprehensive approach to the assessment, evaluation and improvement of software processes and practices in software companies. The next step will consists in the creation of a complex modelling tool for formalized software processes (a simple modelling tool is already implemented and used) and the implementation of a knowledge base capable to store complex organizational "know-how" and "know what". Only then can we apply such sources for the automated search, evaluation and improvement suggestions that make use of the template software process models. We also rely on the experience that is gained from practical experimental applications of this approach in real software companies.

Acknowledgements. This research has been supported by the internal grant agency of VSB-TU of Ostrava - SP/2010214 Modeling, simulation and verification of software processes II and by the Grant Agency of the Czech Republic, project No 401/10/0792, "Temporal aspects of knowledge and information". Michal Košinár is also grand aided student of Ostrava City Authority, Czech Republic.

References

1. Humphrey, W.S.: A Discipline for Software Engineering. Addison-Wesley Professional, Reading (1995)
2. Thayer, R.H.: Software System Engineering: A Tutorial. Computer 35(4), 68–73 (2002), doi:10.1109/mc.2002.993773
3. Makinen, T., Varkoi, T.: Assessment driven process modeling for software process improvement. In: International Conference on Management of Engineering & Technology, PICMET 2008, Portland, July 27-31, pp. 1570–1575 (2008)
4. Software Engineering Institute: CMMI staged-version 1.1. (2002)
5. Garg, P.K., Scacchi, W.: ISHYS: Designing an Intelligent Software Hypertext System. IEEE Expert: Intelligent Systems and Their Applications 4(3), 52–63 (1989)
6. Mi, P., Scacchi, W.: A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes. IEEE Trans. on Knowl. and Data Eng. 2(3), 283–294 (1990), doi:10.1109/69.60792
7. Mi, P., Scacchi, W.: A meta-model for formulating knowledge-based models of software development. Decis. Support. Syst. 17(4), 313–330 (1996), `http://dx.doi.org/`, doi:10.1016/0167-9236(96)00007-3
8. Raffo, D.M.: Modeling software processes quantitatively and assessing the impact of potential process changes on process performance. Ph.D. thesis, Carnegie Mellon University (1996)
9. Madachy, R.J.: Software Process Dynamics, 2nd edn. Wiley-IEEE Press (2008)
10. Scacchi, W.: Experience with software process simulation and modeling. J. Syst. Softw. 46(2-3), 183–192 (1999)
11. Peterson, J.L.: Petri Net Theory and the Modeling of Systems. Prentice Hall, Englewood Cliffs (1981)
12. Mi, P., Scacchi, W.: Articulation: an integrated approach to the diagnosis, replanning, and rescheduling of software process failures. In: Proceedings of Eighth Knowledge-Based Software Engineering Conference, pp. 77–84 (1993)
13. Garg, P.K., Mi, P.W., Thuan, P., Scacchi, W., Thunquest, G.: The Smart Approach for Software Process Engineering. In: Proc. Int. Conf. Softw., pp. 341–350 (1994)
14. Workflow Management Coalition: Terminology & Glossary (1999)
15. Brachman, R., Levesque, H.: Knowledge Representation and Reasoning. Morgan Kaufmann, San Francisco (2004)
16. Frydrych, T., Kohut, O., Košinár, M.: Transparent Intensional Logic in Knowledge Based Multiagent Systems. In: Sojka, P., Horák, A. (eds.) RASLAN 2008. Masaryk University, Brno (2008)
17. Ciprich, N., Duží, M., Košinár, M.: The TIL-Script Language. In: Kiyoki, Y., Tokuda, T., Jaakola, H., Chen, X., Yoshida, N. (eds.) Information Modelling and Knowledge Bases XX, pp. 166–179. IOS Press, Amsterdam (2009)

18. Ciprich, N., Duží, M., Košinár, M.: TIL-Script: Functional Programming Based on Transparent Intensional Logic. In: Sojka, P., Horák, A. (eds.) RASLAN 2007, pp. 37–42. Masaryk University, Brno (2007)
19. Noy, N.F., Sintek, M., Decker, S., Crubézy, M., Fergerson R.W., Musen M.A.: Creating Semantic Web Contents with Protégé-2000, vol. 16 (2001)
20. Kruchten, P.: The Rational Unified Process: An Introduction. Addison-Wesley Professional, Reading (2003)
21. Karkoška, T.: Vytvoření nástroje pro podporu tvorby ontologií v multi-agentním prostředí. Master thesis, VŠB-TUO, Ostrava, Czech Republic (2008)
22. Jezek, D., Vondrak, I.: HDA and resources modeling in business process. In: Baake, U.F., Herbst, J., VanLandeghem, R. (eds.) 11th European Concurrent Engineering Conference 2004 - Worldwide Partnerships and Mergers, pp. 27–29 (2004)
23. W3C: OWL 2 Web Ontology Language (2009), http://www.w3.org/TR/owl2-overview/

A Model Based Testing Approach for Model-Driven Development and Software Product Lines

Beatriz Pérez Lamancha[1], Macario Polo Usaola[2], and Mario Piattini Velthius[2]

[1] Software Testing Center, Republic University, Montevideo, Uruguay
[2] ALARCOS Research Group, Castilla – La Mancha University, Ciudad Real, Spain
bperez@fing.edu.uy, {macario.polo,mario.piattini}@uclm.es

Abstract. This work presents a model based testing approach to be used in Model Driven Development and Software Product Lines projects. The approach uses OMG standards and defines model transformations from design models to test models. The approach was implemented as a framework using existing modelling tools in the market and QVT transformations.

Keywords: Software product lines, Model driven engineering, Model driven testing, UML Testing profile, QVT, MOFScript.

1 Introduction

Model-based Testing (MBT) provides a technique for the automatic generation of test cases using models extracted from software artefacts [1]. Model-Driven Engineering (MDE) and Software Product Lines (SPL) are new software development paradigms. In MDE, models are transformed to obtain the product code, while in SPL, several products share the same base structure. In both approaches, automation is one of the main characteristics; in MDE the code generation is automated from models while in SPL each product is automatically generated from a base structure, also typically described with models. In addition, there are many works that merge MDE and SPL [2-4].

The aim is to maximize reuse and minimize time to market, without losing the final product quality. In SPL, the products in the line share common functionalities and differ in a set of variabilities. If a defect is present in one of the common parts, that defect is propagated to each product in the SPL. The final product quality directly depends on the quality of each of the parts.

In this context, the goal is to reduce the test time without affecting the product quality. In the case of MDE, a change in one model involves rebuilding the models and code automatically, and it takes little time to generate the new code. However, from the testing point of view, ensuring that this change does not introduce defects entails to retest almost everything again. If the tests are manually executed, the cost of testing increases. The same applies to SPL. Testing the common functionalities is not sufficient; the integration of each product must also be tested. For this reason, the automation of tests from models in these two paradigms is essential.

This work presents an automated framework for model-driven testing that can be applied in MDE and SPL development. The main characteristics of the framework are:

- **Standardized:** The framework is based on Object Management Group (OMG) standards, where possible. The standards used are UML, UML Testing Profile as metamodels, and Query/View/Transformation (QVT) and MOF2Text as transformation languages.
- **Model-driven Test Case Scenarios Generation:** The framework generates the test cases at the functional testing level (which can be extended to other testing levels); the test case scenarios are automatically generated from design models and evolve with the product until test code generation. Design models represent the system behaviour using UML sequence diagrams.
- **Framework Implementation using Existing Tools:** No tools have been developed to support the framework: existing market tools that conform to the standards can be used. The requisite is that the modelling tool can be integrated with the tools that produce the transformations.

This paper is organized as follows: Section 2 introduces the main concepts used in the article and outlines a Lottery SPL, which is used as the running example. Section 3 outlines the entire model-driven testing framework. Section 4 describes the activities for the framework in MDE development and, once these ones have been presented, the corresponding activities for SPL are described in Section 5. Section 6 summarizes related works. Finally, Section 7 draws some conclusions and presents future lines of work.

2 Background

Model-Driven Engineering (MDE) considers models as first-order citizens for software development, maintenance and evolution through model transformation [5]. In addition to independence between models, **Model-Driven Architecture** (MDA,[6]) clearly separates business complexity from implementation details by defining several software models at different abstraction levels. MDA defines three viewpoints of a system: (i) the Computation Independent Model (CIM), which focuses on the context and requirements of the system without considering its structure or processing, (ii) the Platform Independent Model (PIM), which focuses on the operational capabilities of a system outside the context of a specific platform, and (iii) the Platform Specific Model (PSM), which includes details related to the system for a specific platform.

The **UML 2.0 Testing Profile** (UML-TP) defines a language for designing, visualizing, specifying, analyzing, constructing and documenting test artefacts. It extends UML 2.0 with test specific concepts for testing, grouping them into test architecture, test data, test behaviour and test time. As a profile, UML-TP seamlessly integrates into UML. It is based on the UML 2.0 specification and is defined using the metamodeling approach of UML [7]. The test architecture in UML-TP is the set of concepts to specify the structural aspects of a test situation [8]. It includes TestContext, which contains the test cases (as operations) and whose composite structure defines the test configuration. The test behaviour specifies the actions and evaluations necessary to evaluate the test objective, which describes what should be tested. The test case behaviour is described using the Behavior concept and can be shown using UML interaction diagrams, state machines and activity diagrams. The TestCase specifies one case to test the system, including what to test it with, the required input, result and initial conditions. It is a complete technical specification of how a set of TestComponents interacts with an SUT to realize a TestObjective and return a Verdict value [7]. This work focuses on test cases, whose behavior is represented by UML sequence diagrams.

Software Product Lines (SPL) are suitable for development with Model Driven principles: an SPL is a set of software-intensive systems sharing a common, managed set of features which satisfy the specific needs of a particular market segment or mission and which are developed from a common set of core assets in a prescribed way [9]. Therefore, products in a line share a set of characteristics (commonalities) and differ in a number of variation points, which represent the variabilities of the products. Software construction in SPL contexts involves two levels: (1) Domain Engineering, referred to the development of the common features and to the identification of the variation points; (2) Product Engineering, where each concrete product is built, what leads to the inclusion of the commonalities in the products, and the corresponding adaptation of the variation points. Thus, the preservation of traceability among software artefacts is an essential task, both from Domain to Product engineering, as well as among the different abstraction levels of each engineering level.

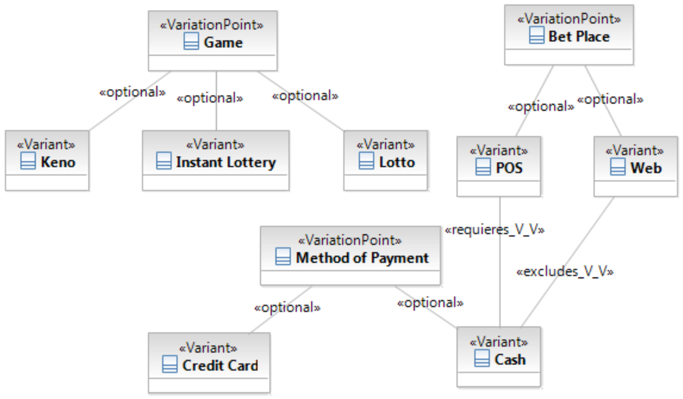


Fig. 1. OVM model for Lottery SPL

The way in which variability is managed in SPL is critical in SPL development. In this work, the proposal by Pohl et al. [10] is used to manage the variability, defined in their **Orthogonal Variability Model** (OVM). In OVM, variability information is saved in a separate model containing data about variation points and variants (a variation point may involve several variants in, for example, several products). OVM allows the representation of dependencies between variation points and variable elements, as well as associations among variation point and variants with other software development models (i.e., design artefacts, components, etc.). Associations between variants may be *requires_V_V* and *excludes_V_V*, depending on whether they denote that a variation *requires* or *excludes* another variation. In the same way, associations between a variation and a variation point may be *requires_V_VP* or *excludes_V_VP*, also to denote whether a variation requires or excludes the corresponding variation point. The variants may be related to artefacts of an arbitrary granularity. Since variants may be related to any type of software artefact (and in the proposal the software artefacts are described using a UML metamodel), to obtain the best fit in this integration, OVM was translated into an UML Profile. With this solution, OVM is managed and manipulated as a part (actually, an extension) of UML 2.0. More details about the defined OVM profile can be found in [11]. Fig. **1** shows the OVM model for the Lottery SPL used as an example in this paper. Lottery SPL manages the bets and payments for different lottery-type games. The types of games considered are: Lotto: played by selecting a predetermined quantity of numbers in a range: depending on the right numbers, the prize is greater or lower. For example, one chooses six numbers from 1 to 49; Keno: basically played in the same manner, although it differs from "Lotto" games in that (i) the population of playing game pieces is even larger, e. g., integers from 1 to 80; (ii) participants can choose the quantity of numbers that they want to match; and (iii) the number of winning game numbers, e. g., twenty, is larger than the number of a participant's playing numbers, e. g. two to ten. One example of the Keno type is Bingo.

This SPL has several variation points, but for purposes of illustration, this paper only analyzes the variation points in Fig. 1 that includes Game (Instant Lottery, Lotto or Keno), Bet Place (a Point of Sale, POS, or a web page) and Method of Payment (Cash or Credit card).

3 Model Based Testing Framework

Fig. 2 shows a global overview of the framework, which is divided horizontally into Domain Engineering and Application Engineering. In Domain Engineering, the SPL core assets are modelled. In Application Engineering, each product is modelled; it can be derived from the SPL or can be one single product developed following MDE software development.

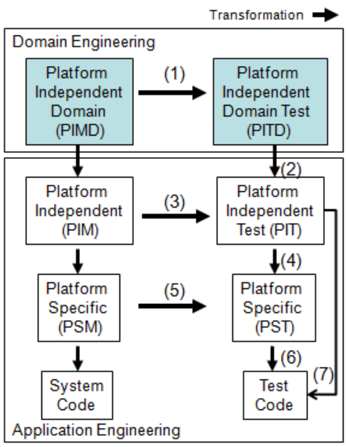


Fig. 2. Testing framework

The framework is also divided vertically into Design models (left) and Testing models (right). In Domain Engineering, a test model is generated for SPL core assets. In Application Engineering, the models follow the MDA levels, and are based on the idea from Dai [12].

The arrows in Fig. 2 represent transformations between models. The objective is to automate the generation of test models from design models using model transformation. To develop the entire framework, the following decisions were taken:

- **Tool to support the framework:** This decision is crucial for the development of the framework. We could develop a tool to support the framework or could use tools already available on the market. The aim of our proposal is to automate model based testing in MDE and SPL. Therefore, a tool built by us must consider modelling elements for both development paradigms. In this case, developers and testers using our approach must use our tool to model the line or product to obtain the test cases. However, these models must also be used for code generation (due to the fact that the development follows an MDE approach), for which specific tools already exist. It seems unrealistic to think that developers will do the double job of modelling, with its associated maintenance cost. Thus, we decided to develop the framework using existing tools on the market, which brought about another problem: the integration of existing tools to achieve the complete implementation of the framework.

- **Design Metamodel:** We can develop our own metamodels or use existing ones. We decided to use existing metamodels and specifically use UML 2.0 [13] due to its being the most widely used metamodel to design software products and the fact that there are several tools to support it in the MDE environment.

- **Testing Metamodel:** Again, we could develop our metamodel or use an existing one. We decided to use the UML 2.0 Testing Profile.

- **Standardized approach:** Since UML 2.0 is used as the design metamodel and the UML Testing Profile as the testing metamodel, both OMG standards and those using commercial tools are more likely to be compatible with standardized approaches. We decided to use standards whenever possible for the construction of the framework.

- **Variability metamodel:** Unfortunately, there is no defined standard for defining metamodel variability in product line development. Several metamodels to represent variability exist. This work uses the Orthogonal variability model (OVM, [10]), represented as a UML Profile (see Section 2).

- **Model to Model Transformation language:** A model transformation is the process of converting one model to another model in the same system [6]. The most important elements in a transformation are: (1) source model and target model, (2) source metamodel and target metamodel and (3) the definition of the transformation. A model transformation language is a language that takes a model as input and, according to a set of rules, produces an output model. Using transformations between models, arrows 1,2,3,4 and 5 in Fig. 2 can be solved. The OMG standard for model transformation in the MDA context is the Query-View-Transformation language (QVT, [14]), which depends on MOF (Meta-Object Facility, [15]) and OCL 2.0 [16] specifications.

- **Model to Text Transformation Language:** Arrows 6 and 7 in Fig. 2 require the transformation from model to test code (for example, this can be the JUnit test code). The OMG standard to translate a model to text is the Model to Text standard (MOF2Text, [17]).

3.1 Models in Domain Engineering

As discussed above, the framework uses UML as it's metamodel. UML has several diagrams to represent the static and dynamic aspects in software development. Fig. 3 shows the UML diagrams used in the framework. This can be extended to other UML diagrams, but for the moment, the framework supports the models defined in Fig. 3.

In Domain Engineering, product line design models are automatically transformed into test models following UML-TP (arrow 1). The variability is traced from the design to the test models.

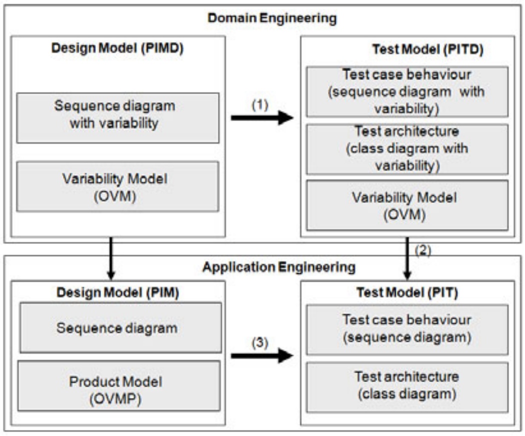

Domain Engineering
Design Model (PIMD)
Sequence diagram with variability
Variability Model (OVM)
(1)
Test Model (PITD)
Test case behaviour (sequence diagram with variability)
Test architecture (class diagram with variability)
Variability Model (OVM)
(2)
Application Engineering
Design Model (PIM)
Sequence diagram
Product Model (OVMP)
(3)
Test Model (PIT)
Test case behaviour (sequence diagram)
Test architecture (class diagram)

Fig. 3. Models at PIM level

In the transformation, the following models are used as source models:

- **Sequence Diagrams with Variability**, which describe use case scenarios. As a metamodel, this kind of model uses extended UML interactions with stereotypes to represent variability. The extension represents each variation point as a CombinedFragment stereotyped with a Variation Point. Each variant is an InteractionOperand stereotyped as a Variant (see Section 5).

- **Variability Model:** this model represents the variability in the SPL. The definition of a UML profile to integrate OVM into UML is required.

These models are transformed, using the QVT language, into the following target UML-TP elements (arrow 1 in Fig. 3):

- **Test Case Behaviour:** describes the test case behaviour that tests the source sequence diagram. As a metamodel, this model uses the same variability extension for UML interactions as the source sequence diagram (see Section 5).

- **Variability Model:** this is the source variability model, but in the transformation, the variability model is augmented by traces to the test artefacts.

- **Test Architecture:** this model is a class diagram that uses an extension for the UML Testing Profile as it's metamodel. This extension applies the stereotypes Variation Point and Variant to the variable elements in the test architecture (see Section 5).

3.2 Models in Application Engineering

Application Engineering takes into account both the MDE and SPL development. In the case of SPL, at this level the variability must be resolved. Thus, this level contains both the test cases refined from the domain engineering for a product (which involves resolving the variability corresponding to arrow 2), as well as the test cases for the functionalities added only for that product. For the new functionalities, the test cases are automatically generated using QVT from sequence diagrams (arrow 3). The transformation generates the test case behaviour as another sequence diagram and a class diagram representing the test architecture. Both models conform to the UML Testing Profile.

3.3 Framework Implementation

The implementation of the framework requires the selection of a modelling tool from those on the market and defining the tools that perform the transformations between the models and from model to code. For the modelling, the selected tool was IBM Rational Software Modeler (IRSM). This tool graphically represents the sequence diagrams and exports them to UML2 through XMI.

Transformations between the models use QVT language, which requires having a tool that implements the standard. *medini QVT* implements OMG's QVT Relations specification in a QVT engine. We used it to develop and execute the QVT transformations. The XMI exported by IRSM is the input for the QVT transformation, which returns the XMI corresponding to the Test Model. This output XMI is imported to the IRSM, which shows the graphical representation for the test cases. The models shown in this paper were obtained using this tool.

The Eclipse IDE makes it possible to use modelling tools in an integrated way, using extensions in the form of plug-ins. There exists a *medini QVT* plug-in for Eclipse, which we use for the model transformation. Other Eclipse plug-ins are used to perform the modelling tasks. Eclipse Modelling Framework (EMF) is a modelling framework that allows the development of metamodels and models, from a model specification described in XMI, provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. UML2 is an EMF-based implementation of the UML 2.0 OMG metamodel for the Eclipse platform and UML2 Tools is a Graphical Modelling Framework editor for manipulating UML models. Using these integrated tools for transformations between models requires the input models for transformations to be XMI (which is the default serialized form of EMF) in Eclipse UML2 format. Therefore, the selected tool for the graphical modeling must support the import and export of models in the UML2 format through XMI. There are many tools available that export UML models to the UML2 format through XMI, but few import the UM2 format. In our case, since the behavior of the test case is automatically generated as a sequence diagram, it is crucial that the modelling tool be able to import the transformed models and visualize them.

4 Model Based Testing Activities for MDE

The preceding sections have presented the decisions taken in the process of defining the framework, and the metamodels and models defined for it. This section describes the necessary activities to implement the testing framework in MDE development. Fig. 4 shows the process for generating the test model for MDE development.

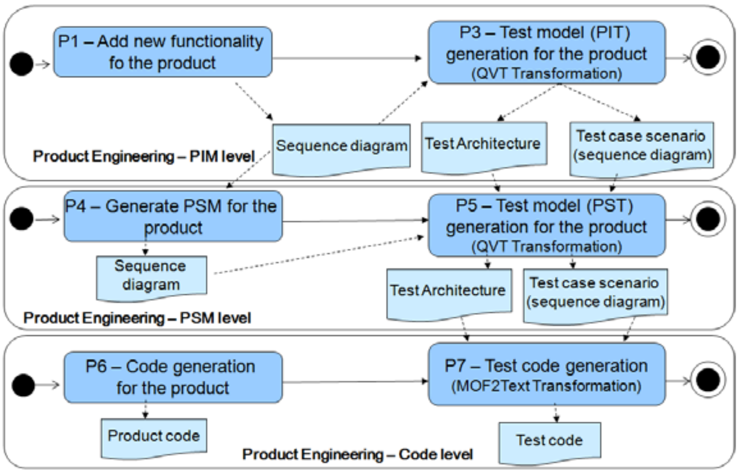

P1 – Add new functionality fo the product
P3 – Test model (PIT) generation for the product (QVT Transformation)
Sequence diagram
Test Architecture
Test case scenario (sequence diagram)
Product Engineering – PIM level
P4 – Generate PSM for the product
P5 – Test model (PST) generation for the product (QVT Transformation)
Sequence diagram
Test Architecture
Test case scenario (sequence diagram)
Product Engineering – PSM level
P6 – Code generation for the product
P7 – Test code generation (MOF2Text Transformation)
Product code
Test code
Product Engineering – Code level

Fig. 4. Framework activities in MDE development

The activities at the PIM level are:

- **P1-Add New Functionality for the Product:** in this activity, the functionality for the product is described. The result is a sequence diagram representing a use case scenario. Fig. 5 shows the Interaction diagram for the functionality to check the results for a bet in the Lotto game.

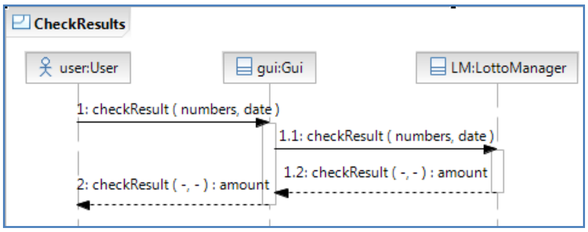

CheckResults
user:User
gui:Gui
LM:LottoManager
1: checkResult (numbers, date)
1.1: checkResult (numbers, date)
1.2: checkResult (-, -) : amount
2: checkResult (-, -) : amount

Fig. 5. Check results functionality

- **P3-Test Model Generation for the Product:** this activity consists of running the QVT scripts which automatically generate the test models for the product. The inputs for the transformation are the sequence diagram generated in activity P1, which were exported to the XMI format. The outputs are the test architecture and the test case scenario, both of which follow the UML-TP. These models are imported to the modelling tool. Using the UML-TP, actors are represented by TestComponents, whilst the System is represented by the SUT. In our proposal, each message between the actor and the SUT must be tested (functional testing).

Fig. 6 shows the test case generated for the Check Results functionality in Fig. 5. Fig. 7 summarizes the semantic of the QVT transformation to generate the test case scenario, the following steps are necessary (more details can be found in [18]):

- **Obtaining the test data:** To execute the test case, according to UML-TP, the test data are stored in a DataPool. The TestComponent asks for the test data using the DataSelector operation in the DataPool. Fig. 6 shows the *dp_checkResult()* stereotyped as DataSelector, which returns the values data1, data2 and expected. The first two are the values to test the parameters in the operation and the third is the expected result for the test case.

- **Executing the test case in the SUT:** The TestComponent simulates the actor and stimulates the system under test (SUT). The TestComponent calls the message to test in the SUT. For the example in Fig. **6**, the operation to test is *checkResult*. It is tested with the data returned by the DataPool. The operation is called in the SUT and returns the *result* data.

- **Obtaining the test case verdict:** The TestComponent is responsible for checking whether the value returned for the SUT is correct, and informs the Arbiter of the test result. For the example in Fig. **6**, the validation action checks if the *result* (actual value) is equal to the *expected* (expected value) to return a verdict for the test case.

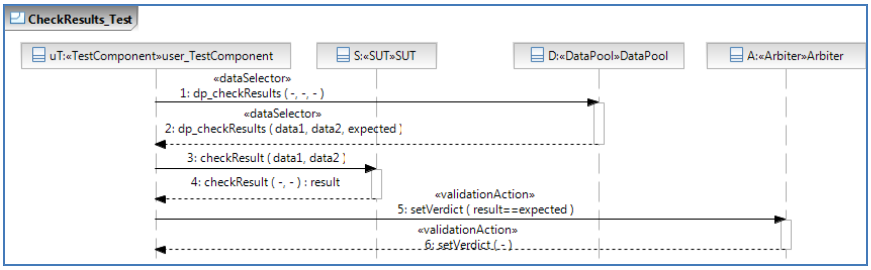

CheckResults_Test
uT:«TestComponent»user_TestComponent
S:«SUT»SUT
D:«DataPool»DataPool
A:«Arbiter»Arbiter
«dataSelector»
1: dp_checkResults (-, -, -)
«dataSelector»
2: dp_checkResults (data1, data2, expected)
3: checkResult (data1, data2)
4: checkResult (-, -) : result
«validationAction»
5: setVerdict (result==expected)
«validationAction»
6: setVerdict (-)

Fig. 6. Test case for Check Results

The activities at the PSM level are similar, but in this case the models are refined with platform specific aspects. The activities at code level are:

- **P6 –Code Generation for the Product:** in this activity the product code is generated following specific MDE tools. Once the executable product is obtained, it can be tested.

- **P7 – Test Code Generation for the Product:** this activity consists of running the MOF2Text scripts which automatically generate the test code from the PIT or PST models. The inputs for the transformation are the sequence diagram that represents the test cases generated in activities P3 or P5. The output is the test case code following the same development language used at the PSM level. For example, if Java is used, the test cases can be developed using JUnit.

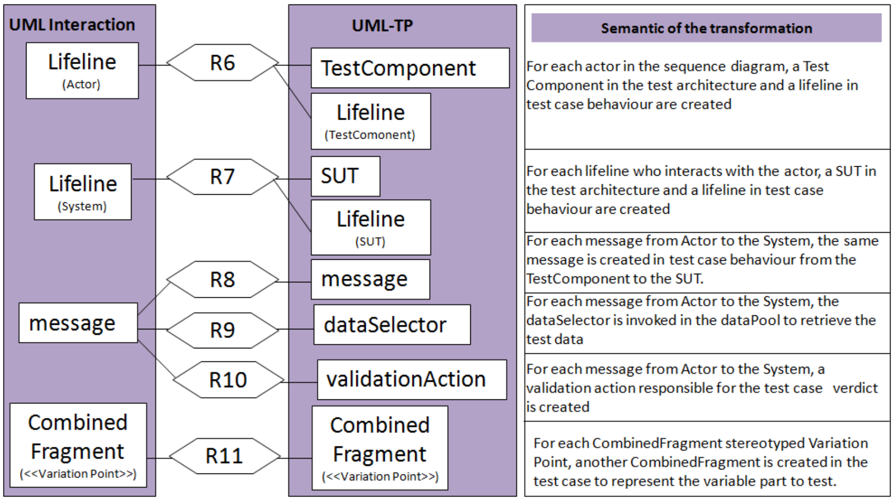

UML Interaction
Lifeline (Actor)
Lifeline (System)
message
Combined Fragment (<<Variation Point>>)
R6
R7
R8
R9
R10
R11
UML-TP
TestComponent
Lifeline (TestComonent)
SUT
Lifeline (SUT)
message
dataSelector
validationAction
Combined Fragment (<<Variation Point>>)
Semantic of the transformation
For each actor in the sequence diagram, a Test Component in the test architecture and a lifeline in test case behaviour are created
For each lifeline who interacts with the actor, a SUT in the test architecture and a lifeline in test case behaviour are created
For each message from Actor to the System, the same message is created in test case behaviour from the TestComponent to the SUT.
For each message from Actor to the System, the dataSelector is invoked in the dataPool to retrieve the test data
For each message from Actor to the System, a validation action responsible for the test case verdict is created
For each CombinedFragment stereotyped Variation Point, another CombinedFragment is created in the test case to represent the variable part to test.

Fig. 7. Semantic of QVT transformation for test case generation

5 Model Based Testing Activities for SPL

This section describes the activities necessary to implement the testing framework in SPL development. The activities required for SPL are added to those existing for MDE development. Fig. 8 shows the activities added to Fig. 4.

For Domain Engineering, the activities added are:

- **D1 – Design the Variability Model:** in this activity, the OVM model for the SPL is developed. This model follows the UML Profile defined for OVM.

- **D2 – Design the Functionality:** in this activity, the common functionalities for the SPL are described, including the variabilities. The result is a sequence diagram with the extension defined to deal with variability. The defined profile for SPL represents each variation point as a CombinedFragment stereotyped as <<Variation Point>> and each variant is an InteractionOperand in the CombinedFragment. Fig. 9 shows the Bet Payment functionality for the Lottery SPL. In it, the player calculates the amount of the bet and then makes the payment. As can be seen in Fig. 1, the method of payment is a Variation Point, and the Combined Fragment is thus stereotyped as <<Variation Point>> representing that the behaviour differs if the payment is by credit card or with cash.

- **D3 – Test Model Generation:** this activity consists of running the QVT scripts which automatically generate the test models for the SPL. The inputs for the transformation are the OVM model and the sequence diagram with variability. The outputs are the test architecture and the test case scenario, both of which follow the UML Testing Profile and the extension defined for variability. Fig. 10 shows the test case for Bet Payment, the same steps that the P3-Test model generation for the product activity is doing, but in this case, the CombinedFragment is translated to the test case. More about test case generation in the SPL context can be found in [19].

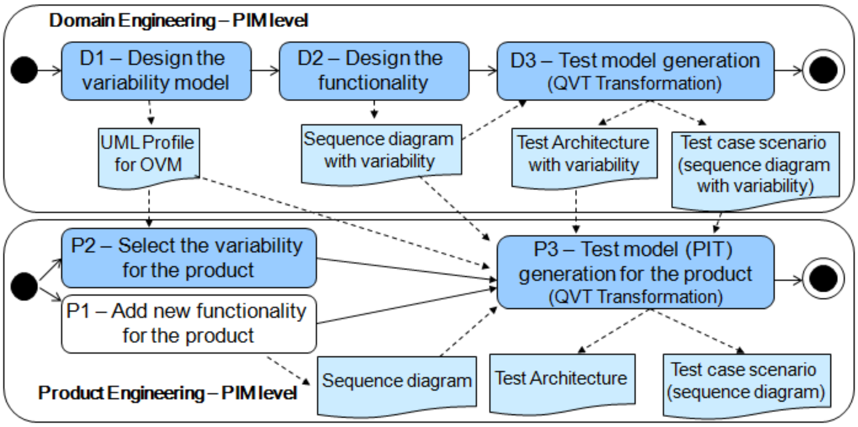

Domain Engineering – PIM level
D1 – Design the variability model
D2 – Design the functionality
D3 – Test model generation (QVT Transformation)
UML Profile for OVM
Sequence diagram with variability
Test Architecture with variability
Test case scenario (sequence diagram with variability)
P2 – Select the variability for the product
P1 – Add new functionality for the product
P3 – Test model (PIT) generation for the product (QVT Transformation)
Sequence diagram
Test Architecture
Test case scenario (sequence diagram)
Product Engineering – PIM level

Fig. 8. Framework activities in SPL development

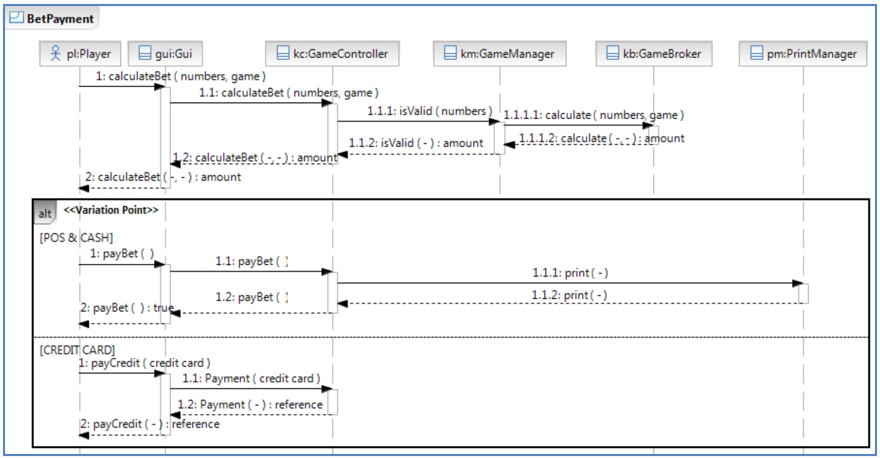

BetPayment
pl:Player
gui:Gui
kc:GameController
km:GameManager
kb:GameBroker
pm:PrintManager
1: calculateBet (numbers, game)
1.1: calculateBet (numbers, game)
1.1.1: isValid (numbers)
1.1.1.1: calculate (numbers, game)
1.1.1.2: calculate (-, -) : amount
1.1.2: isValid (-) : amount
1.2: calculateBet (-, -) : amount
2: calculateBet (-, -) : amount
alt
<<Variation Point>>
[POS & CASH]
1: payBet ( )
1.1: payBet ( )
1.1.1: print (-)
1.1.2: print (-)
1.2: payBet ( )
2: payBet ( ) : true
[CREDIT CARD]
1: payCredit (credit card)
1.1: Payment (credit card)
1.2: Payment (-) : reference
2: payCredit (-) : reference

Fig. 9. Bet Payment Functionality

At Application Engineering, the variability is resolved and then the test model for each product is generated; the activities added are:

- **P1 – Add New Functionality for the Product:** in this activity, the functionality (which is specific to the product) is described. The result is a sequence diagram representing the functionalities present only in this product. It is the same activity as for MDE development**.**

- **P2 – Select the Variability for the Product:** To determine the test cases corresponding to each product, it is necessary to know which variation points and variants are included in each product.

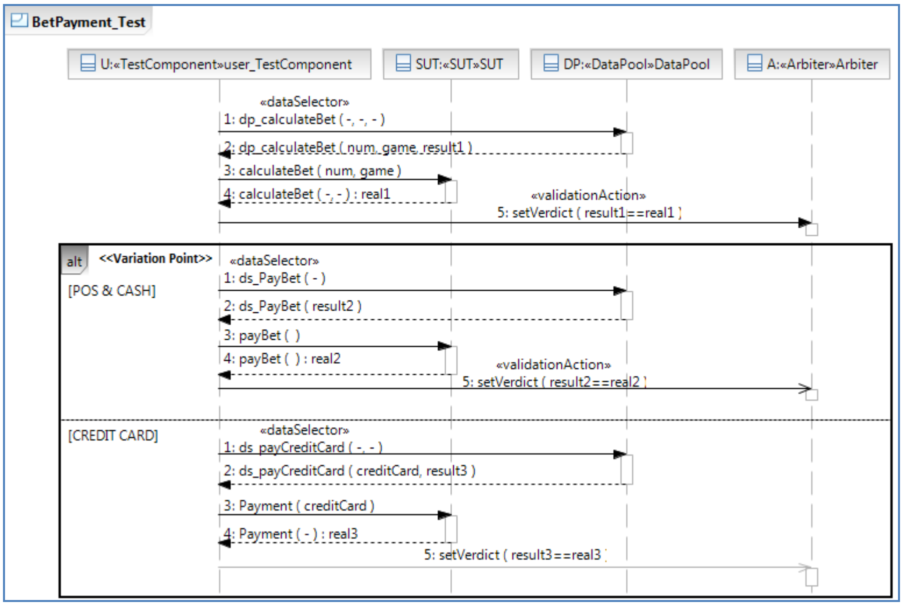

BetPayment_Test
U:«TestComponent»user_TestComponent
SUT:«SUT»SUT
DP:«DataPool»DataPool
A:«Arbiter»Arbiter
«dataSelector»
1: dp_calculateBet (-, -, -)
2: dp_calculateBet (num, game, result1)
3: calculateBet (num, game)
4: calculateBet (-, -) : real1
«validationAction»
5: setVerdict (result1==real1)
alt
<<Variation Point>>
[POS & CASH]
«dataSelector»
1: ds_PayBet (-)
2: ds_PayBet (result2)
3: payBet ( )
4: payBet ( ) : real2
«validationAction»
5: setVerdict (result2==real2)
[CREDIT CARD]
«dataSelector»
1: ds_payCreditCard (-, -)
2: ds_payCreditCard (creditCard, result3)
3: Payment (creditCard)
4: Payment (-) : real3
5: setVerdict (result3==real3)

Fig. 10. Test case for Bet Payment

In this activity, the valid variants selected for the product are stored in the Orthogonal Variability Model of each product (OVMP). Fig. 11 shows the variants selected for the Lotto Web product.

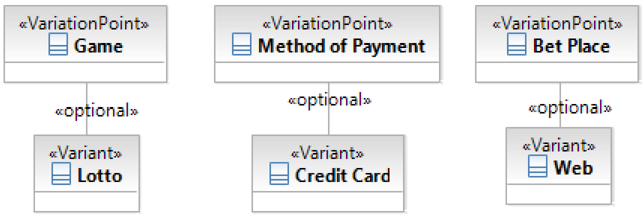


Fig. 11. Var. model for Lotto Web product

- **P3 – Test Model Generation for the Product:** Taking the test case for *Bet Payment* (Fig. 9) as an example, to generate the test cases for the product Lotto Web, the variability in the CombinedFragment must be resolved. The inputs are: (1) the variability model for the Lotto Web (Fig. 11), and (2) the *Bet Payment*_Test test case (Fig. 10). The output is the *Bet Payment_Test* test case for the Lotto Web product (Fig. 12). The entire CombinedFragment is deleted in the final test case, i.e. the variability is resolved at the product level.

The activities described in this section are added in SPL development to the existents for MDE development (see Fig. 4). Furthermore, the activities defined for the PSM and code level also apply to SPL development.

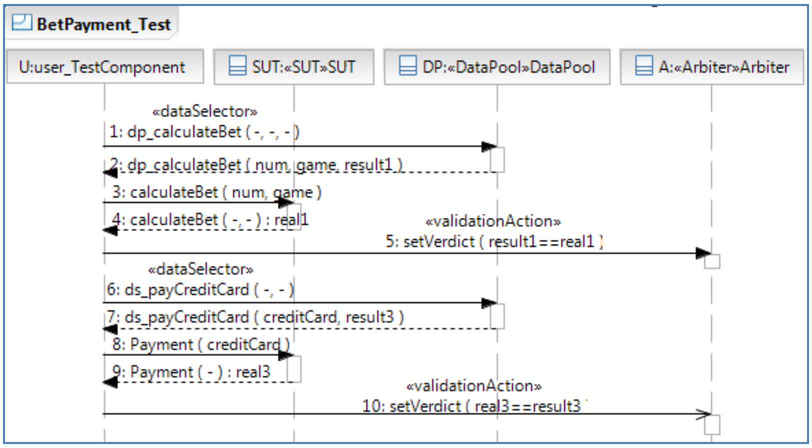

BetPayment_Test
U:user_TestComponent
SUT:«SUT»SUT
DP:«DataPool»DataPool
A:«Arbiter»Arbiter
«dataSelector»
1: dp_calculateBet (-, -, -)
2: dp_calculateBet (num, game, result1)
3: calculateBet (num, game)
4: calculateBet (-, -) : real1
«validationAction»
5: setVerdict (result1==real1)
«dataSelector»
6: ds_payCreditCard (-, -)
7: ds_payCreditCard (creditCard, result3)
8: Payment (creditCard)
9: Payment (-) : real3
«validationAction»
10: setVerdict (real3==result3)

Fig. 12. Bet Payment test case for Lotto Web Product

Table 1 shows the pseudocode of the transformation QVT to obtain the test case for a specific product. The input models are the Variability Model for the Line (VML), the Variability Model for the Product (VMP) and the test model for the line (TML). The output is the test model for the product (TMP).

Table 1. Pseudocode for the product test case generation

```
1.  Create TMP
2.  For each test case tc (interaction diagram) in the TML
3.      If tc does not has variability
4.          Copy tc into TMP
5.      Else
6.          Create a new test case in TMP
7.          Add the lifelines that without the <<Variant>> stereotype
8.          Add the messages outside of a Combined Fragment stereotyped <<Variation Point>>
9.          For each CombinedFragment (cf) stereotyped <<Variation Point>>
10.             Find the Variation Point (vp) associated in the VML
11.             If vp is included in VMP
12.                 For each InteractionOperand io in cf
13.                     Find the Variant (v) associated with io in VML
14.                     If v is included in VMP
15.                         Add the lifelines stereotyped <<Variant>> associated with v
16.                         Add the messages inside io
17.                     EndIf
18.                 EndFor
19.             EndIf
20.         EndFor
21.     EndElse
22. EndFor
```

The *CombinedFragment* and the *InteractionOperand* disappears in the product test case as was shown in Figures 8 and 9.

Table 2 shows the QVT transformation code for the messages included in an *IteractionOperand* when the *CombinedFragment* is stereotyped *Variation Point*. This transformation resolves the lines 12, 13, 14 and 16 in the pseudocode in Table 1.

Table 2. QVT transformation for the messages inside the InteractionOperand

```
top relation translateInteractionOperandForProduct{
-- For each interaction
  checkonly domain source i:uml::Interaction{
    -- For each CombinedFragment
      fragment = cf:uml::CombinedFragment{
      --For each InteractionOperand
      operand = io : uml::InteractionOperand {
      --For each message in the InteractionOperand,
       fragment = mes : uml::MessageOccurrenceSpecification{}
   }}};
-- Find the association between the InteractionOperand and the Variant in
VML
      checkonly domain source asoc:uml::Association{
        name = 'A_CF_' + io.name,
        ownedEnd = p1 : uml::Property {type = i },
        ownedEnd = p2:uml::Property{type = vp:uml::Class{} }};
-- Find the Variant in VMP
      checkonly domain ovmp var:uml::Class{name = vp.name };
-- Create the message in TMP
      enforce domain target it:uml::Interaction{
      fragment = mes:uml::MessageOccurrenceSpecification{} };
when {  -- Check that the stereotype is applied in the CombinedFragment
         st  = getStereotype ( 'Varition Point' );
         cf.isStereotypeApplied(st);      }
}
```

6 Related Works

This section reviews the most significant works in this field. Several proposals for test case generation in SPL use UML artefacts as a basis. All of them provide traceability between Domain and Application Engineering in SPL. However, none of them take into account the capabilities of standard test models, such as UML-TP. Moreover, since model-based approaches are quite suitable for SPL, using a standard transformation language to automate the model generation is quite appropriate. A full description of existing works on SPL can be found in a recently published systematic review [20]. Nebut et al. [21] propose a strategy in which test cases for each of the different products of an SPL are generated from the same SPL functional requirements. Test cases are obtained from high level sequence diagrams. Bertolino et al. [22] propose an abstract methodology, PLUTO (Product Line Use Case Test Optimization), for planning and managing abstract descriptions of test scenarios, which are described in PLUCs (Product Line Use Cases). A PLUC is a traditional use case where scenarios are described in natural language, but which also contain additional elements to describe variability. Each PLUC includes a set of categories (input parameters and environment description) and test data. Then, and according to the variability labels, categories are annotated with restrictions, to finally obtain the test cases. Kang et al. [23] use an extended sequence diagram notation to represent use case scenarios and variability. The sequence diagram is used as the basis for the formal derivation of the test scenario given a test architecture. Reuys et al. [24] present ScenTED (Scenario-based Test case Derivation) where the test model is represented as an activity diagram from which test case scenarios are derived. Test case scenarios are specified in sequence diagrams without providing concrete test data. Test case scenarios can be generated automatically, but test case specifications are developed manually.

Olimpiew and Gomma [25] describe a parametric method, PLUS (Product Line UML-based Software engineering). Here, customizable test models are created during software product line engineering in three phases: creation of activity diagrams from the use cases, creation of decision tables from the activity diagrams, and creation of test templates from the decision tables. Test data would then be generated to satisfy the execution conditions of the test template.

There exist many proposals about model-based testing, but few of them focus on automated test model generation using model transformation. Dai [12] describes a series of ideas and concepts to derive UML-TP models from UML models, which are the basis for a future model-based testing methodology. Test models can be transformed either directly to test code or to a platform specific test design model (PST). After each transformation step, the test design model can be refined and enriched with specific test properties. However, to the best of our knowledge, this interesting proposal has no practical implementation for any tool. These transformations are carried out with Java algorithms, which results in a mixed proposal between the two approaches described in this paper.

Baker et al. [8] define test models using UML-TP. Transformations are done manually instead of with a transformation language. Naslavsky et al. [26] use model transformation traceability techniques to create relationships among model-based testing artefacts during the test generation process. They adapt a model-based control flow model, which they use to generate test cases from sequence diagrams. They adapt a test hierarchy model and use it to describe a hierarchy of test support creation and persistence of relationships among these models. Although they use a sequence diagram (as does this proposal) to derive the test cases, they do not use it to describe test case behaviour.

Javed et al. [27] generate unit test cases based on sequence diagrams. The sequence diagram is automatically transformed into a unit test case model, using a prototype tool based on the Tefkat transformation tool and MOFScript for model transformation.

7 Conclusions

A framework for model-driven testing that can be applied in MDE and SPL development was presented. The proposal includes a methodological approach to automate the generation of test models from design models. The paper also describes a way to handle SPL variability in test models, which is based on OVM. Currently, the proposal is implemented for PIM models in domain and application engineering. Future work includes extending the proposal for PSM and code.

Acknowledgements. This research was financed by the projects: PRALIN (PAC08-0121-1374) and MECCA (PII2I09-00758394) from the "Consejería de Ciencia y Tecnología, JCCM" and the project PEGASO/MAGO (TIN2009-13718-C02-01) from MICINN and FEDER. Beatriz Pérez has a grant from JCCM Orden de 13-11-2008.

References

1. Dalai, S., et al.: Model-based testing in practice. In: ICSE (1999)
2. Trujillo, S., Batory, D., Diaz, O.: Feature oriented model driven development: A case study for portlets. In: ICSE (2007)
3. Deelstra, S., et al.: Model driven architecture as approach to manage variability in software product families. In: MDAFA (2003)
4. Czarnecki, K., et al.: Model-driven software product lines. In: OOPLSLA (2005)
5. Mens, T., Van Corp, P.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Sciences (2006)
6. Miller, J., Mukerji, J.: MDA Guide Version 1.0. 1. In: OMG (ed.) (2003)
7. OMG, UML testing profile Version 1.0. In: OMG (ed.) (2005)
8. Baker, P., et al.: Model-Driven Testing: Using the UML Testing Profile. Springer, Heidelberg (2007)
9. Clements, P., Northrop, L.: Software Product Lines - Practices and Patterns. Addison Wesley, Reading (2001)
10. Pohl, K., Böckle, G., Van Der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
11. Pérez Lamancha, B., Polo Usaola, M., Piattini, M.: Towards an Automated Testing Framework to Manage Variability Using the UML Testing Profile. In: AST, Canada (2009)
12. Dai, Z.: Model-Driven Testing with UML 2.0. In: EWMDA, Canterbury, England (2004)
13. OMG, Unified Modeling Language, Superestructure specification. In: OMG (ed.) (2007)
14. OMG, MOF Query/View/Transformation Specification. In: OMG (ed.) (2007)
15. OMG, Meta Object Facility Specification. In: OMG (ed.) (2002)
16. OMG, Object Constraint Language, Version 2.0. In: OMG (ed.) (2006)
17. OMG, MOF Model to Text Transformation Language. In: OMG (ed.). OMG (2008)
18. Pérez Lamancha, B., et al.: Automated Model-based Testing using the UML Testing Profile and QVT. In: MODEVVA, USA (2009)
19. Pérez Lamancha, B., Polo Usaola, M., García Rodriguez de Guzmán, I.: Model-Driven Testing in Software Product Lines. In: ICSM, Canadá (2009)
20. Pérez Lamancha, B., Polo Usaola, M., Piattini, M.: Software Product Line Testing, A systematic review. In: ICSOFT, Bulgaria (2009)
21. Nebut, C., et al.: Automated requirements-based generation of test cases for product families. In: ASE (2003)
22. Bertolino, A., Gnesi, S., di Pisa, A.: PLUTO: A Test Methodology for Product Families. In: PFE (2004)
23. Kang, S., et al.: Towards a Formal Framework for Product Line Test Development. In: CIT (2007)
24. Reuys, A., et al.: Model-based System Testing of Software Product Families. In: Pastor, Ó., Falcão e Cunha, J., et al. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 519–534. Springer, Heidelberg (2005)
25. Olimpiew, E., Gomaa, H.: Customizable Requirements-based Test Models for Software Product Lines. In: SPLiT (2006)
26. Naslavsky, L., Ziv, H., Richardson, D.J.: Towards traceability of model-based testing artifacts. In: A-MOST, United Kingdom (2007)
27. Javed, A., Strooper, P., Watson, G.: Automated generation of test cases using model-driven architecture. In: AST (2007)

Large-Scale Agile Software Development at SAP AG

Joachim Schnitter and Olaf Mackert

SAP AG, Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany
{j.schnitter,olaf.mackert}@sap.com

Abstract. In an ongoing change process SAP AG has managed to move the software development processes from a waterfall-like approach to agile methodologies. We outline how Scrum was introduced to implement a lean development style as well as the model chosen to scale Scrum up to large product development projects. The change affected about 18.000 developers in 12 global locations. This paper is an extended and revised version of an earlier publication [15]. It includes recent findings.

1 Introduction

Agile methodologies have proven to offer many benefits when developing user-centric software. Early findings about how to optimize product development led to iterative processes stressing prototyping and early feedback [19]. Agile project management methods, e. g. Scrum [16,17], take into account that product development is essentially a learning process. Long-term planning based on product requirements, which appear rock-solid but are in reality merely educated guesses, is substituted by an iterative approach which focuses on early delivery of partial functionality. In combination with regular feedback from users and stakeholders a software development project converges into something useful without the exact goal being known beforehand. Strict prioritizing of all requirements ensures that risks remain low. Even a project that was cancelled because of time and budget constraints may nevertheless have delivered a useful result if the essential requirements were implemented by the time the project was cancelled.

Scrum is known to work well in teams of up to around 10 people programming for a customer on the basis of a project contract. How well can Scrum perform in a global software company with thousands of developers in various locations working on a few dozen highly complex software products? This paper describes some of the experiences gained during the change process at SAP AG, the world's leading producer of enterprise software, transitioning from a waterfall-like overall process model to an agile development model that incorporates the values of lean development and production.

In the following section we will outline the situation at SAP before the introducton of agile development. The third section describes the activities during and results of the pilot phase. The fourth section gives a brief overview of process tailoring before introducing Scrum throughout the development organizations. In the fifth section we describe the steps taken to introduce lean development. In the sixth section the model chosen to scale Scrum up to very large development projects is described. The final section summarizes the results and observations.

2 SAP Before Lean Development and Scrum

SAP, founded in 1972, offers a portfolio of enterprise applications around its flagship, the SAP Business Suite. Many of these products are aimed at large companies, although products for small and medium businesses have been available for some time or are in the making. With the acquisition of Business Objects SA in 2008, SAP gained access to user-centric analytics products which have since been fully integrated into SAP's traditional product line. In 2010 SAP started shipping new applications and platforms based on in-memory database technology which is also to be included in the products of another recent acquisition of SAP, Sybase Inc.

In its first two decades SAP developed software for and together with its customers. This close co-operation helped the company focus on business essentials, keep pace with business and technology trends. It eventually led to the fully developed client-server System R/3 which supported all critical business functions for every major industry. Developers had regular contact with customers and users in different roles, not only as programmers but also as consultants, trainers, and support engineers. The success of R/3 reshaped the market for business applications. Newer customers, though, expected that R/3 was a finished, standardized product they could easily install, configure and run. At the same time many of them expected full support of their individual business processes. Demand shifted from more business functionality to a higher degree of configurability, easier lifecycle management, and solutions covering only parts of the R/3 application suite with more detail and specificity.

Within a few years SAP's development organization grew from about 2,000 to over 18,000 employees in various functions. In earlier days developers were expected to be generalists, but by the time growth started slowing most developers had become experts on a particular technology or application area. Specialization increased in parallel with division of labour. By now there were experts working on requirements roll-in, other experts working on detailed requirements specifications, yet others laying out the basic functions and algorithms, programming, testing, and so on. By 2001 it had become clear that issues with communication among the specialized teams, departments, organizations were becoming a major cause of reduced product quality on many levels. To overcome these problems SAP introduced a development process framework to co-ordinate development efforts throughout all involved organizations. Involvement and responsibilities were assigned to departments according to their primary activities and skills overall leading to a waterfall-like process model with handover procedures among the involved parties. This process model was traversed once per year by most development departments which resulted in new products or major product versions each year.

When agile methodologies gained broader public interest around 2004 some teams at SAP started to experiment with Extreme Programming and Scrum. This approach was not feasible for regular product development teams because they had no longer easy access to customers and users. Therefore agile project management practices were first used in prototype development with strong customer involvement.

3 Pilot Phase

In 2006 SAP formed a small expert team of people experienced in agile development practices. This team began to educate and support development teams interested in Scrum. Within two and a half years it was possible to run a substantial number of projects according to the Scrum methodology. During this time the expert team gained important insight into applicability, limiting factors, and resistance within the organization.

Pilot teams were chosen by organizations according to their own interest. Since some of the early teams displayed scepticism due to personal attitude, each project team had to approve its participation in the pilot program unanimously.

3.1 Education

Scrum education for pilot teams consisted of the following elements:

1. A two-hour introduction into motivation and practices of Scrum for development managers.
2. A six-hour training for the development team, Scrum master, and product owner. This included an in-depth analysis of project setup, risks, and interdependencies with other teams.
3. A four-hour in-depth training for Scrum masters and product owners.

Each person in the expert team supported a number of pilot teams for some months as a Scrum mentor. This support included participation in all sprint planning and review meetings, retrospectives, help with tools, and advocating Scrum before development management. Additionally mentors tried to track and resolve all issues the teams had either internally or with other teams, organizations, and management.

After eight months of direct team support the Scrum expert team started offering trainings for future Scrum mentors. Mentor training took one week and included in-depth Scrum knowledge, team dynamics, and selected areas of software engineering, e. g. requirements management. A prerequisite for the mentor training was experience with Scrum either as a Scrum master or product owner. Training included Scrum master certification by the Scrum Alliance. The teaching team consisted of the Scrum expert team, a psychologist, and a practitioner from the Scrum Alliance.

After one year there were 25 mentors available. By September 2008 these mentors had supported about 120 projects in 10 global development locations.

3.2 Project Management Tools

When the Scrum expert team started its work, few project management software systems were available which supported Scrum. They all lacked sub-project support, hierarchical backlog management, and interfaces for inclusion into SAP's toolchain, e. g. to the requirements database, issue tracker, and test management systems. Therefore SAP decided to extend its own product for project management, SAP cProjects, to support Scrum projects. This extension project was also a showcase for Scrum with direct involvement of users because the system was to be used internally. A member of the Scrum expert team took on the role of the product owner.

3.3 Lessons Learned

A wiki was used to manage basic information and observations. For each pilot team the following data were collected by the Scrum mentors: scope of the project, duration, product owner, scrum master, dependencies with other teams, locality, critical decisions and impediments, and other observations. These data were used to select and communicate best practices, create a network of Scrum masters and product owners, and set up a round table of experienced practicioners. In-depth interviews with Scrum masters, product owners, and team members provided valuable information and suggestions for improvement.

Scrum gives teams more power to make decisions concerning development speed and quality. These aspects led to wide acceptance among teams. On the other hand many individuals felt that they were being monitored exessively as a result of the high degree of interaction both within the team and with external individuals and organizations. The predominant arguments against Scrum were:

1. Team member: "Scrum forces me to report to the team every failure and impediment. This puts me under pressure."
2. Team member: "I work more effectively when working alone. Daily Scrum meetings force me to interrupt my work."
3. Team member: "My manager might get a wrong impression of my work in case someone reports how slow my work progresses."
4. Development manager: "I have no insight into the team's work. Appearently nobody monitors what is going on."
5. Other teams: "We cannot rely on teams which do not deliver according to an agreed-upon schedule."

Most resistence of team members was removed by pointing out that the transparency of activity within the team was necessary because it enabled the team to manage its work independently. In order to be released from management's scrutinity the team has to monitor itself. Sceptics from management were invited to sprint planning and review meetings. They were regularly impressed by the teams' skills in planning all activities with great detail, focusing on risk management and high product quality.

What could not be mitigated were issues with teams dependent on "unreliable" Scrum teams, at least if the Scrum teams were isolated in a non-Scrum environment. Scrum teams working jointly on a particular software product rarely had complaints about unreliable delivery because they were familiar with the Scrum methodology and figured out how to get their requirements added to other teams' backlogs and with the right priorities.

Globally distributed teams often successfully applied Scrum although we were able to identify a number of problem areas which increased the reluctance to set up new distributed project teams:

1. Project kick-off meetings with all team members brought together in one location were rarely performed but were considered to be essential.
2. Neither telephone nor video conferencing systems could produce the feeling of proximity which was necessary to maintain team spirit.

3. Oral communication was burdensome because in every location English was spoken with a local accent.
4. Timezone differences of more than three hours made it difficult to agree on a common time for the daily Scrum meeting.
5. Multicultural teams typically broke up within two months mainly because the role of the Scrum master is interpreted very differently among European, Asian, and American people. A Scrum master from one culture found it hard to meet the expections of team members from another culture.

Many issues were not directly related to Scrum usage but became visible in this context. Teams tended to relate problems to Scrum usage, but deeper analysis showed that these issues had existed in the same teams before or could be observed in non-Scrum teams as well. This showed the high potential of Scrum to expose problem areas which otherwise would have gone unidentified or had been hidden intentionally.

3.4 Other Issues

Around the same time some problems surfaced which were not connected to Scrum usage but needed to be resolved before moving to Scrum on a large scale:

1. Communication among product management, software architects, development teams, and quality assurance was not strong enough to ensure a common understanding of development goals. Product management often was unable to communicate product requirements to software engineers. Languages and notations differed significantly.
2. The role of software architecture (high-level design) was not yet well understood. This led to increased dependencies among products and components, e. g. because componentization was weak and code reuse was highly valued. Another result was the lack of conceptual preparation for the development teams for whom there were too many obstacles to harmonize on interfaces and adhere to delivery timelines.

To overcome both problems a team was formed to provide education on software architecture for developers, software architects, and product management. A simple modeling language was created by combining selected Unified Modeling Language (UML) elements with diagram types from the Fundamental Modeling Concepts (FMC) notation [6,7]. FMC block diagrams in particular have proven to be an immense help to communicate technical concepts to customers, product managers, and developers. In parallel SAP revised its internal quality standards and added rules for component separation and usage.

4 Development Process Reform, Round One

A major revision of the standard development process took place while the Scrum expert team was still running pilot projects. Early experiences with Scrum came to the attention of management who ordered a reform of the development process framework to involve customers and users more directly. The “reform” team which included the members of the Scrum expert team defined three process models for different product development types.

Improvement. This process model is used to improve existing products and for minor extensions. In the context of business software this can mean e. g. adaptions to changed legal requirements or newer technical standards. An important criteria for this process model is that the project requirements include few changes to implemented business processes, only minor changes to the user interface, and that no newer technologies need to be added.

New Product. This model is chosen for new products or major product extensions. It includes Scrum as the project management method. Customer and user involvement is secured by appropriate contracts, and a minimum number of customers has to be involved. A calculated business case must exist as a basis for a development decision.

Research. This process model is used basically for research and prototyping projects. These projects are mostly free to chose whatever tools and methods they wish.

An important result of this approach was that it increased awareness of the need to tailor the process model according to certain factors, e. g. software type (infrastructure, UI, business logic etc.), project size, product development or customer-specific development, architecture constraints (from-scratch product development, product extensions, major refactoring, and renovation).

5 Introducing Lean Development

From industry customers who had long-standing experience with lean production and the Toyota production system (TPS) and from talks and publications by Mary and Tom Poppendieck [12,13,14] SAP management learnt about the lean development approach. This knowledge provided the business background for research into how lean development could be applied at SAP. It turned out that Scrum is particularily well suited to implementing lean software development:

1. Scrum's iterative cycles, called sprints, implement *takt*, i. e. the cyclic, repetitive work approach.
2. The *pull* principle of TPS (often discussed in conjunction with *Kanban*) can be found in Scrum in two places: (i) When planning a sprint the team "pulls" only so many requirements from the product backlog as can be fulfilled during the next sprint. Scrum does not allow for asking for more than the team promises. (ii) Each team member picks tasks from the sprint backlog when he or she sees fit. No manager assigns tasks to a team member.
3. The *Genchi Genbutsu* (go and see for yourself) principle is reflected by the product owner's participation in sprint reviews at regular intervals.
4. The *Kaizen* principle of continuous improvement is put into practice by regular retrospectives and the Scrum master's role to collect and communicate impediments and ideas for improvement to stakeholders.

In every major SAP development area (2000 to 5000 developers per area) a lean implementation team was set up to explore what had to be done to implement lean development processes. One team took over the pilot role by implementing their process model six months earlier thereby gaining experiences to share with the other teams. These teams applied Scrum to their own change management projects.

It is worth to note that the pull principle had been in use at SAP for software configuration management for years. Change (delta) propagation from a development codeline to a test or consolidation codeline was usually done by a responsible person pulling the changes into the target codeline. This process was implemented for ABAP (SAP's proprietary application programming language) around 1990. For source code in other programming languages implementation took place in 2007.

5.1 Academic Experiments

Before applying Scrum on a large scale some experiments were carried out by the Hasso Plattner Institute, Potsdam (HPI). The HPI provides education for master and bachelor degrees in software systems engineering. As part of their practical education students have to carry out a half-year software development project with about 50 other students. These development projects were used to experiment with various approaches to upscaling Scrum, e. g. Scrum of Scrums, hierarchical backlog, hierarchy of product owners. Many insights were gained in these projects which proved beneficial to applying Scrum to multi-team projects at SAP [8]. Research on this topic is ongoing.

5.2 Continuous Improvement

In each development area a team was formed to collect and resolve problems the individual teams could not resolve themselves. These teams have to deal with both technical development problems and issues resulting from the new lean development approach. Therefore no limits are imposed on the types of problems communicated. These continuous improvement teams are also responsible for idea management in each area and serve as escalation contacts for Scrum masters.

5.3 Education

A one-day "lean awareness" training provided the background of the lean development approach. This training described the basic principles of lean production and development. To scale Scrum up fast to an entire organization Scrum mentors were trained by the existing Scrum mentors who had at least 1.5 years experience with Scrum in various contexts. This training was similar to the mentor trainings given earlier.

Scrum mentors held training courses for development teams, product owners, and Scrum masters. In order to provide training for all teams in a short time, the time allocated for training was reduced. Instead of training each team individually for one day as done during the pilot phase, two teams were trained together for half a day. Focus was put on in-depth training of Scrum masters and product owners.

People who attended the lean awareness training frequently commented that the knowledge about lean production helped them understand customer needs better, but the relationship between lean production and software development needed clarification. Many teams who had participated in the shortened training struggled later with communication and motivation issues. It turned out that in contrast to the pilot phase too few Scrum mentors were available to deal with the numerous team problems. To overcome these problems team training sessions were extended to one day per team and the “lean mentor” role was created. Lean mentors underwent in-depth training in Scrum, lean methodologies, and communication. Scrum master and product owner training sessions were extended to two days each.

6 Scaling Scrum to the Max

Scaling agile software development processes is still subject to research [1,5,10,11]. Several approaches are known to co-ordinate the work of several Scrum teams working on one product, e.. Scrum of Scrums [9] or MetaScrum [18]. The size and complexity of SAP’s products makes it necessary to consider these techniques. To co-ordinate the work of multiple Scrum teams and to communicate the requirements via team product backlogs, various approaches were combined. Special product teams were formed for this purpose. These teams can be regarded as both a permanent Scrum of Scrums and a supervisory product owner.

6.1 Product Teams

Product teams were introduced as a second organizational layer above the Scrum teams. A product team is responsible for the work of up to 7 development teams. It consists of the product owners of those teams and the same number of team members who are specialists in particular fields. Depending on the problem area other experts may be included. This allows for full engineering coverage of all problem areas expected, well-organized communication among the teams, and direct communication with the development teams to detect and mitigate risks. Typically the following expert roles can be found in product teams:

1. Chief product owner (responsible product manager),
2. Product team Scrum master (facilitator),
3. Software architect,
4. Delivery manager,
5. Knowledge management and product documentation expert,
6. User interface designer,
7. Stakeholder representative.

These roles may be taken by dedicated people or by development team members. For a product team to function properly, the following prerequisites have to be met:

1. Every member of the product team is also a member of one of the related Scrum teams.
2. The product team has full responsibility for requirements scope, quality, and delivery of the product. Each product team has a budget including external and travel costs.
3. No product team member is a line manager of another team member.
4. All product team members are collocated.

The tasks of the product team include: (a) product and budget management, (b) definition of the skill profiles, size, and numbers of Scrum teams, (c) assigning product backlog items to Scrum teams, (d) collecting project status information, (e) synchronizing the work of the Scrum teams, (f) working closely with the continuous improvement teams, (g) requesting supportive actions in case of impediments.

Three focus questions were dealt with in particular by dedicated roles: The chief product owner who is also the product owner of the product team has to answer the question *what* has to be delivered. The delivery manager decides *when* a particular functionality is to be delivered. The software architects decide *how* this functionality is to be implemented.

6.2 Area Product Teams

Product teams can rarely support more than seven Scrum teams. If the products or components are be too big or too complex to be developed in this organizational setup, an intermediate organizational layer is inserted between the product team and the Scrum teams. These teams are called “area product teams”. Their responsibility is the same as the product teams’ responsibility with the exception of budget and final product decisions. In this three-layer organization a product team works similar to a project management office. The complete budget and product responsibility is with the product team, while the operational day-to-day decisions are handled by the area product teams.

6.3 Observations

The Scrum roll-out happened very fast and was based on the assumption that during the pilot phase enough knowledge had been collected to go for a company-wide adoption. Upscaling methods were mainly driven by theoretical considerations. Implementing them revealed that the skills and knowledge of product managers and software architects were often not sufficient to manage a number of Scrum teams concurrently. Product teams had to communicate with business management, marketing, and field services who were not fully prepared for how Scrum and the product teams’ expectations would change their daily work. The process of software requirements engineering were found to require a major adjustment.

Teams typically needed a full year to completely adopt Scrum. Later we could see signs of erosion especially in teams which had no Scrum mentor to support them. Team members often reported that they felt expected to work harder than was sustainable. Programmers complained about the pressure both from management and from within their teams. The first explanation given by the Scrum mentors was that these teams would pick more from the Scrum backlog than what they could implement. To counteract this they educated the teams about effort estimation and work-life balance. A detailed analysis revealed that the real root cause for demotivation and exhaustion was competition among the team members. They suspected that Scrum had been introduced to strive for higher efficiency and exchangeability of programmers, and a flatter management hierarchy with fewer career oportunities. As a result many programmers worked very hard to be visible and distinguishable from the others in the team.

Teams who had adopted Scrum to their satisfaction started looking into other agile methods. As of this writing many teams have adopted test-driven development [2] and pair-programming [3]. This has been supported by trainings aiming at combining both methods into a team-wide, sustainable practice.

Cutural differences were not only important factors for Scrum adoption in the teams but also in management. We observed that Scrum was very differently introduced and implemented in various global locations, e. g. in Germany, the USA, Canada, France, India, Israel, and China. Work and management culture sometimes distorted Scrum to a means for additional career oportunities, more control, more documentation, more beaurocracy, or pressure for higher development speed, in stark contrast to the agile manifesto [4].

7 Conclusions and Outlook

The Scrum project management methodology has gained broad acceptance at SAP. Most teams consider Scrum their method of choice despite the many serious problems encountered. Employees who know SAP from its early days call it a dej vu although direct customer involvement in current development projects is very limited. Scrum has also been used for internal IT projects, change management, and marketing projects which all have in common that the exact goals could not be defined in the first place.

Scaling Scrum to a multi-team development organization is not as easy. Scrum of Scrums and backlog preparation for many teams can be combined by product teams as outlined above. Looking at all known parameters of agile project management at SAP we tend to say that not much more than 130 development employees may be organized in that fashion. This number sums up developers in 7 teams (max. 70 people), the product team (max. 16), development infrastructure responsibles (about 10), quality assurance and testers (about 25), general management (about 10).

A three-level hierarchy of teams to manage requirements for product backlog generation has not yet proven to work as expected. Currently the high complexity of SAP's products which is reflected in the organizational structure of SAP's development areas makes it impossible to drop one hierarchy level of project management. This observation has triggered significant effort to rigorously break up SAP's product lines into separate components to ensure that no product component needs more resources than the 130 people as calculated above.

To learn about the effects of lean development as implemented at SAP, the company chose to participate in the DIWA-IT study about health protection in the IT industry sponsored by the German Bundesministerium fr Bildung und Forschung and the European Union. An ongoing research project has been set up to examine long-term effects.

We understand lean development merely as a set of values and objectives rather than a framework of processes and best practices. There is still a lot of misunderstanding at SAP and in the world about lean development. Lean *production* processes have a long-standing tradition in industry, while lean *product development* processes have only recently started to be dicussed. It is dangerous to refer to lean production as a blueprint for product development. Both processes of value creation are completely different, and using terms from one process in the context of the other has often led to misinterpretations and confusion. We found that it is not enough to train software developers about Scrum and lean values. Significant effort needs to be put into educating people in business administration and general management about innovation management and the pecularities of product design and development. In this respect software development is a very special case as the scientific management methods of Taylorism are unsuitable for processes with artifacts mostly existing in the minds of programmers.

We expect that after each product release the maintenance effort will increase for a couple of years when a new product gets widely adopted by customers. As discussed in a recent case study this maintenance effort requires additional considerations when creating the product backlog [20]. We consider the "competition" between new-product development and old-product maintenance a challenge to the long-term acceptance of Scrum at SAP. To counterbalance this effect organizational measures might be advisable.

Introducing lean development is a learning process which brings many problem areas to light. Many observations made at SAP would not have been made in the old environment. It is still too early to assess the benefits of the ongoing change in completeness, but the transparency of the development processes and the broad acceptance of agile methods gained make the change a worthwile effort. In addition to Scrum SAP is in the process of applying more and more of the developer-centered methods of Extreme Programming, e. g. test-driven development, pair programming, project metaphor, early code integration.

Looking back at the waterfall model that SAP used for about 10 years, the authors found that the biggest disadvantage of that model is not that it makes it impossible to correct errors at a later stage, but the strong impact the model has on organization and communication. The waterfall model suppresses communication along its path. Experts for one phase only communicate with experts for other phases through highly formalized documents and status data. Informal communication is regarded as undermining the model's simplicity. Unfortunately this attitude destroys the only remaining risk mitigation strategy left: communication.

Acknowledgements. The change described in this paper could not be carried out without a solid fundament of determined colleagues willing to accept delay and hardship while moving forward. In particular we wish to name Martin Fassunge, Alexander Gerber, Bernhard Gröne, Christiane Kuntz-Mayr, Christian Schmidkonz, and Jürgen Staader whose contributions were outstanding. We also would like to thank Jürgen Müller and Thomas Kowark of the Hasso Plattner Institute for their support to run the Scrum scaling experiments.

References

1. Ambler, S.W.: Scaling agile software development through lean governance. In: SDG 2009: Proceedings of the 2009 ICSE Workshop on Software Development Governance, pp. 1–2. IEEE Computer Society, Washington, DC, USA (2009)
2. Beck, K.: Test-driven development: By example. Addison-Wesley Professional, Boston (2002)
3. Beck, K.: Extreme programming explained: Embrace change, 2nd edn. Addison-Wesley Professional, Boston (2004)

4. Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: The agile manifesto (2001), http://agilemanifesto.org/
5. Eckstein, J., Josuttis, N.: Agile Softwareentwicklung im Großen: Ein Eintauchen in die Untiefen erfolgreicher Projekte. dpunkt, Heidelberg (2004)
6. Keller, F., Wendt, S.: FMC: An approach towards architecture-centric system development. In: ECBS, pp. 173–182. IEEE Computer Society, Los Alamitos (2003)
7. Knöpfel, A., Gröne, B., Tabeling, P.: Fundamental Modeling Concepts: Effective Communication of IT Systems. John Wiley & Sons, Chichester (2006)
8. Kowark, T., Müller, J., Müller, S., Zeier, A.: An educational testbed for the computational analysis of collaboration in early stages of software development processes. Accepted for HICSS 2011 (2011)
9. Larman, C., Vodde, B.: Scaling lean & agile development: Thinking and organizational tools for large-scale Scrum. Addison-Wesley, Upper Saddle River (2009)
10. Lee, G., Xia, W.: Towards Agile: An integrated analysis of quantitative and qualitative field data. MIS Quarterly 34(1), 87–114 (2010)
11. Leffingwell, D.: Scaling Software Agility: Best Practices for Large Enterprises. The Agile Software Development Series. Addison-Wesley Professional, Boston (2007)
12. Poppendieck, M., Poppendieck, T.: Introduction to lean software development. In: Baumeister, H., Marchesi, M., Holcombe, M. (eds.) XP 2005. LNCS, vol. 3556, p. 280. Springer, Heidelberg (2005)
13. Poppendieck, M., Poppendieck, T.: Implementing lean software development: From concept to cash. The Addison-Wesley signature series. Addison-Wesley Professional, Boston (2006)
14. Poppendieck, M., Poppendieck, T.: Leading lean software development: Results are not the point. The Addison-Wesley signature series. Addison-Wesley Professional, Boston (2009)
15. Schnitter, J., Mackert, O.: Introducing agile software development at SAP AG — Change procedures and observations in a global software company. In: 5th International Conference on Evaluation of Novel Approaches to Software Engineering, Athens, Greece, July 22-24, pp. 132–138 (2010)
16. Schwaber, K.: Agile project management with Scrum. Microsoft Press, Redmond (2004)
17. Schwaber, K.: The enterprise and Scrum. Microsoft Press, Redmond (2007)
18. Sutherland, J.: Future of scrum: Parallel pipelining of sprints in complex projects. In: ADC 2005: Proceedings of the Agile Development Conference, pp. 90–102. IEEE Computer Society, Washington, DC, USA (2005)
19. Takeuchi, H., Nonaka, I.: The new new product development game. Harvard Business Review 64, 137–146 (1986)
20. Vlaanderen, K., Brinkkemper, S., Jansen, S., Jaspers, E.: The agile requirements refinery: Applying Scrum principles to software product management. In: 3rd International Workshop on Software Product Management, Atlanta, Georgia, USA (September 1, 2009)

Systems Evolution and Software Reuse in OOP and AOP

Adam Przybyłek

University of Gdańsk, Department of Business Informatics
Piaskowa 9, 81-824 Sopot, Poland
adam@univ.gda.pl

Abstract. New programming techniques make claims that software engineers often want to hear. Such is the case with aspect-oriented programming (AOP). This paper describes a quasi-controlled experiment which compares the evolution of two functionally equivalent systems, developed in two different paradigms. The aim of the study is to explore the claims that software developed with aspect-oriented languages is easier to maintain and reuse than this developed with object-oriented languages. We have found no evidence to support these claims.

Keywords: Aspect-oriented programming, Separation of concerns, Software evolvability, Software reusability.

1 Introduction

Object-oriented programming (OOP) aims to support software maintenance and reuse by introducing concepts like abstraction, encapsulation, aggregation, inheritance and polymorphism. However, years of experience have revealed that this support is not enough. Whenever a crosscutting concern needs to be changed, a developer has to make a lot of effort to localize the code that implements it. This may possibly require him to inspect many different modules, since the code may be scattered across several of them.

An essential problem with traditional programming paradigms is the tyranny of the dominant decomposition [36]. No matter how well a software system is decomposed into modules, there will always be concerns (typically non-functional ones) whose code cuts across the chosen decomposition [27]. The implementations of these crosscutting concerns will spread across different modules, which has a negative impact on maintainability and reusability.

The need to achieve a better separation of concerns (SoC) gave rise to aspect-oriented programming (AOP) [22]. The idea behind AOP was to implement crosscutting concerns as separate modules, called aspects. AOP has been proven to be effective in lexically separating different concerns of the system [33]. However, the influence of AOP on other quality attributes is still unclear.

On the one hand, replacing code that is scattered across many modules by a single aspect can potentially reduce the number of changes during maintenance [28]. In addition, core modules may be easier to reuse, since they implement single concerns and do not contain tangled code.

On the other hand, constructs such as pointcuts and advices can make the ripple effects of aspect-oriented (AO) systems far more difficult to control than in OO systems. A change in the method signature captured by the pointcut invalidates this pointcut definition. The reason is that core modules are oblivious of aspects that modify their behavior. Moreover, obliviousness leads to "programs that are unnecessarily hard to develop, understand, and change" [15]. Since not all the dependencies between the modules in AO systems are explicit, an AO maintainer has to perform more effort to get a mental model of the source code [35]. Creating a good mental model is crucial to understand the structure of a system before attempting to modify it [25]. Studies of software maintainers have shown that 30% to 50% of their time is spent in the process of understanding the code that they are to maintain [13], [34], [14].

Moreover, AO systems suffer from an issue called the pointcut fragility problem, which occurs when a pointcut unintentionally captures or misses a given join point as a consequence of seemingly harmless changes to the base code [23], [28]. Kästner et al. [21] observed such silent changes during AO refactoring. Therefore, it appears that the very techniques that AOP provides to solve or limit some of the evolution problems with traditional software, actually introduce a series of new evolution problems. This phenomenon is called the evolution paradox of AOP [37].

Furthermore, incremental modifications and code reuse are not directly supported for the new language features of AspectJ [16]. In particular, concrete aspects cannot be extended, advice cannot be overridden, and concrete pointcuts cannot be overridden. Hanenberg & Unland proposed four rules of thumb [16], which allow to build reusable and incrementally modifiable aspects. However, enormous complexity is the price that has to be paid for it.

2 Background

When development of a software product is complete and it is released to the market, it enters the maintenance phase of its life cycle. Software maintainability is the ease with which a software product can be modified after delivery [19]. ISO/IEC [20] defines four categories of maintenance: perfective, adaptive, corrective, and preventive. As software is used, the user usually requests additional features and capabilities. Perfective maintenance extends the software beyond its original requirements. Over time, the original environment (terminal devices, operating system, laws, regulations, business rules, external product characteristics) for which the software was developed is likely to change. Adaptive maintenance accommodates the software to its external environment. It has been estimated that 80% of the software maintenance effort is devoted to software evolution (adaptive and perfective maintenance) [30].

Even with the best quality assurance activities, it is likely that the delivered software contains some latent defects that were not detected during testing. Corrective maintenance repairs these defects. Computer software deteriorates due to change, and because of this, preventive maintenance, often called software reengineering, must be conducted to enable the software to operate effectively and to make subsequent maintenance easier. In essence, preventive maintenance refers to enhancements to software modularity or understandability. It may also include the study of a system to detect and correct latent faults in the software product before they become effective faults [20].

Software maintenance has been recognized as the most costly and difficult phase in the software life cycle. Studies and surveys over the years have indicated that software changes typically consume 40% to 80% of overall software development costs [14], [17]. Hewlett-Packard estimates that 60% to 80% of its R&D personnel are involved in maintaining existing software, and that 40% to 60% of software budget are directly related to maintenance [11], [26].

Composing systems from existing modules rather than building from scratch has been one of the main goals of the software engineering since its beginning in the 1960s. Reusability is the ease with which existing modules can be used in new context. Using previously written modules as building blocks allows programmers to simplify the construction of software, since the traditional phases of development are replaced with processes of module search and selection [1]. Such approach reduces the development time and costs, downgrades the risk of new projects, and improves the software quality. One of the obstacles to a massive application of software reuse in industrial environments is that creating reusable software modules requires a huge initial investment which is not rapidly amortized.

3 Motivations

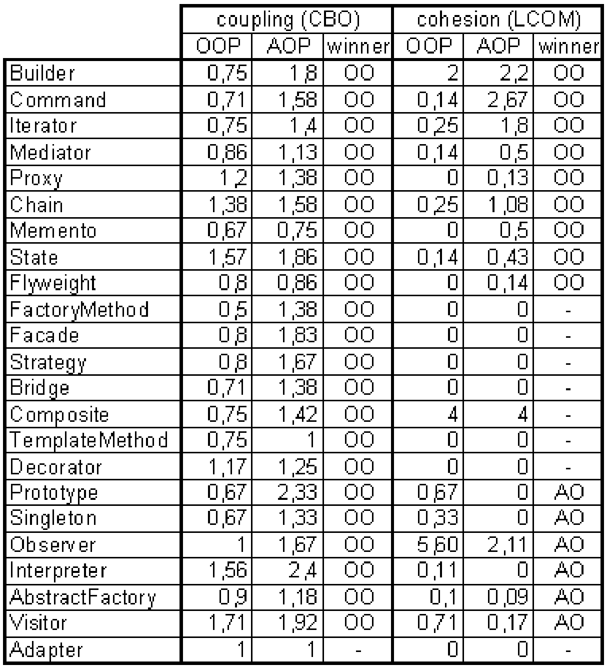
Many unsupported claims have been made about AOP. Here are a few examples:

- AOP "can be seen as a way to overcome many of the problems related to software evolution" [27];
- AOP "produces code that is simpler and more maintainable, as well as increasing the flexibility, extensibility and re-usability of the separated concerns" [4];
- AO software "is supposed to be easy to maintain, reuse, and evolution" [39];
- AOP leads to "the production of software systems that are easier to maintain and reuse" [33];
- AOP "increases understandability and eases the maintenance burden, because modules tend to be more cohesive and less coupled" [24].

However, every new programming technique is overpromised and begins with naive euphoria. In our previous study [31] we compared OO and AO implementations of the 23 GoF design patterns with regard to coupling and cohesion. The evaluation was performed applying the CBO and LCOM metrics from the CK suite, which had been adapted for use on AO systems. Table 1 presents the mean values of the metrics, over all modules per pattern. The lower numbers are better. There is no pattern whose AO implementations exhibits lower coupling. With regard to cohesion the OO implementations were superior in 9 cases, while the AO ones in 6 cases. 8 patterns exhibited the same cohesion in both implementations.

Since it is commonly acknowledged that designs with low coupling and high cohesion lead to software that is both, more reusable and more maintainable (Table 2 enumerates work that documented these relationships), we intend to investigate the claims about the impact of AOP on reusability and evolvability.

Table 1. Modularity metrics computed as arithmetic means



	coupling (CBO)			cohesion (LCOM)		
	OOP	AOP	winner	OOP	AOP	winner
Builder	0,75	1,8	OO	2	2,2	OO
Command	0,71	1,58	OO	0,14	2,67	OO
Iterator	0,75	1,4	OO	0,25	1,8	OO
Mediator	0,86	1,13	OO	0,14	0,5	OO
Proxy	1,2	1,38	OO	0	0,13	OO
Chain	1,38	1,58	OO	0,25	1,08	OO
Memento	0,67	0,75	OO	0	0,5	OO
State	1,57	1,86	OO	0,14	0,43	OO
Flyweight	0,8	0,86	OO	0	0,14	OO
FactoryMethod	0,5	1,38	OO	0	0	-
Facade	0,8	1,83	OO	0	0	-
Strategy	0,8	1,67	OO	0	0	-
Bridge	0,71	1,38	OO	0	0	-
Composite	0,75	1,42	OO	4	4	-
TemplateMethod	0,75	1	OO	0	0	-
Decorator	1,17	1,25	OO	0	0	-
Prototype	0,67	2,33	OO	0,67	0	AO
Singleton	0,67	1,33	OO	0,33	0	AO
Observer	1	1,67	OO	5,60	2,11	AO
Interpreter	1,56	2,4	OO	0,11	0	AO
AbstractFactory	0,9	1,18	OO	0,1	0,09	AO
Visitor	1,71	1,92	OO	0,71	0,17	AO
Adapter	1	1	-	0	0	-

Table 2. Impact of coupling and cohesion on reusability and maintainability

	reusability	**maintainability**
coupling	[6], [18]	[6], [18], [7], [9]
cohesion	[6], [5]	[6], [29]

4 Measurement System

In order to define the metrics to be collected during the study, we used the G-Q-M (Goal-Question-Metric) approach [3]. G-Q-M defines a measurement system on three levels (Fig. 1) starting with a goal. The goal is refined in questions that break down the issue into quantifiable components. Each question is associated with metrics that, when measured, will provide information to answer the question.

Our goal is to compare AO and OO systems with respect to software evolvability and reusability from the viewpoint of the developer. Evolvability and reusability are quality characteristics that we cannot measure directly. Instead, we can perform an experiment that involves maintenance tasks and then we can measure how much effort is required to evolve the system and how much of the existing code can be reused in the consecutive release.

We chose to measure reusability as simply the number of lines of code that were added in order to extend the program's functionality in a prescribed way. The more lines required, the lower the reusability is.

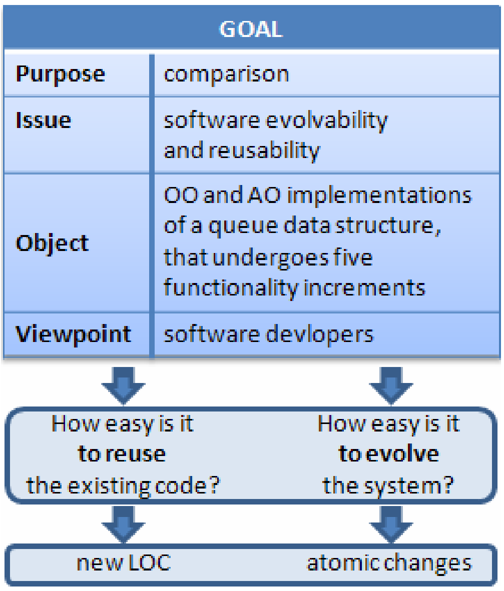

GOAL
Purpose
comparison
Issue
software evolvability and reusability
Object
OO and AO implementations of a queue data structure, that undergoes five functionality increments
Viewpoint
software devlopers
How easy is it to reuse the existing code?
How easy is it to evolve the system?
new LOC
atomic changes

Fig. 1. Goal-Question-Metric

The evolution metric we used is based on previous studies performed by Zhang et al. [38] and Ryder & Tip [32]. In their work, the difficulty of evolvability is defined in terms of atomic changes to the modules in a program. At the core of this approach is the ability to transform source code edits into a set of atomic changes, which captures the semantic differences between two releases of a program. Zhang et al. [38] presented a catalog of atomic changes for AspectJ programs. For the purpose of our study, we slightly modified their catalog. Firstly, we consider deleting a non-empty element as an atomic change. Secondly, we used the term "module" as a generalization of class, interface, and aspect. Our list of atomic changes is follows: add an empty module, delete a module, add a field, delete a field, add an empty method, delete a method, change body of method, add an empty advice, delete an advice, change an advice body, add a new pointcut, change a pointcut body, delete a pointcut, introduce a new field, delete an introduced field, change an introduced field initializer, introduce a new method, delete an introduced method, change an introduced method body, add a hierarchy declaration, delete a hierarchy declaration, add an aspect precedence, delete an aspect precedence, add a soften exception declaration, delete a soften exception declaration.

5 Empirical Evaluation

We compare OOP with AOP on a classical producer-consumer problem. In a producer-consumer dilemma two processes (or threads), one known as the "producer" and the other called the "consumer", run concurrently and share a fixed-size buffer. The producer generates items and places them in the buffer. The consumer removes items from the buffer and consumes them. However, the producer must not place an item into the buffer if the buffer is full, and the consumer cannot retrieve an item from the buffer if the buffer is empty. Nor may the two processes access the buffer at the same time to avoid race conditions. If the consumer needs to consume an item that the producer has not yet produced, then the consumer must wait until it is notified that the item has been produced. If the buffer is full, the producer will need to wait until the consumer consumes any item.

We assume to have an implementation of a cyclic queue as shown in Fig. 2. The put(..) method stores one object in the queue and get() removes the oldest one. The nextToRemove attribute indicates the location of the oldest object. The location of a new object can be computed using nextToRemove, numItems (number of items) and buf.length (queue capacity). We also have an implementation of a producer and a consumer.

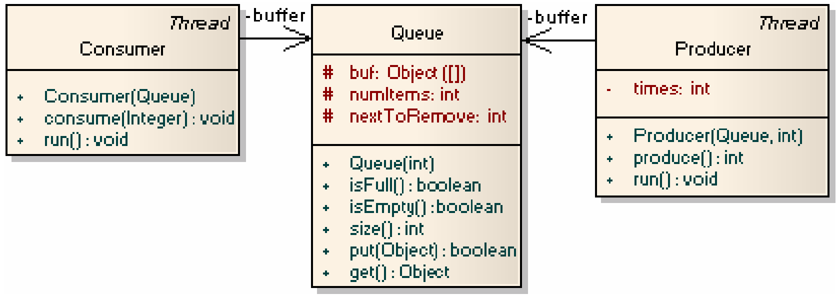


Fig. 2. An initial implementation

The experiment encompasses five maintenance scenarios which deal with the implementation of a new requirement.

5.1 Adding a Synchronization Concern

To use Queue in a consumer-producer system an adaptation to a concurrent environment is required. Both put(..) and get() methods have to be executed in mutual exclusion. In addition, a thread has to be blocked when it tries to put an element into a full buffer or when it tries to get an element from an empty queue. In Java these methods have to be wrapped in synchronization code (Fig. 3).

The above implementation tangles the synchronization concern with the core logic. Moreover, the synchronization code is scattered through the accesors methods. As a result, the put(Object) and get() contain similar fragments of code for cooperating synchronization.

A lexical separation of synchronization concern can be achieved by using AO constructs (Fig. 4). However, benefits of such separation are doubtful. Firstly, the SynchronizedQueue aspect is explicitly tied to the Queue class, and so cannot be reused in other contexts. Secondly, Queue is oblivious to the synchronization aspect.

This makes it difficult to know what changes to Queue will lead to undesired behavior.

```
public class Buffer extends Queue {
  public Buffer(int n) {
    super(n);
  }
  public synchronized boolean put(Object x) {
    while ( isFull() ) try {
      wait();
    } catch (InterruptedException e) {}
    super.put(x);
    notifyAll();
    return true;
  }
  public synchronized Object get() {
    while ( isEmpty() ) try {
      wait();
    } catch (InterruptedException e) {}
    Object tmp = super.get();
    notifyAll();
    return tmp;
  }
}
```

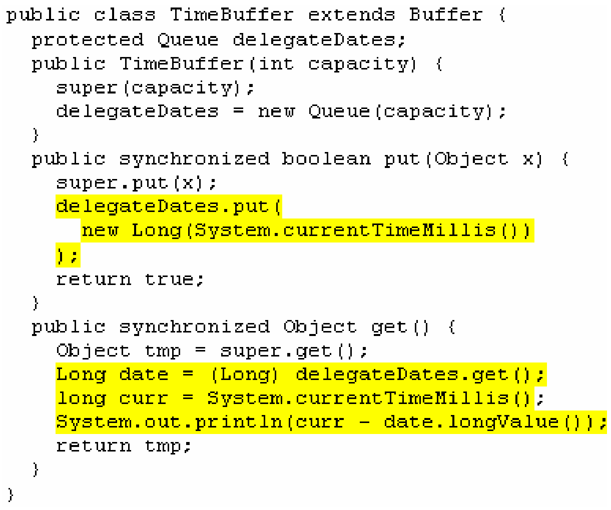
Fig. 3. A blocking queue

```
public aspect SynchronizedQueue
 pertarget(target(Queue)) {
  protected pointcut call_get():
    call( Object Queue.get() );
  protected pointcut call_put(Object x):
    call(boolean Queue.put(Object)) && args(x);
  Object around(Queue q): call_get() && target(q) {
    synchronized(this) {
      while( q.isEmpty() ) try {
        wait();
      } catch (InterruptedException e) {}
      Object tmp = proceed(q);
      notifyAll();
      return tmp;
    }
  }
  boolean around(Queue q, Object x):
   call_put(x) && target(q) {
    synchronized(this) {
      while ( q.isFull() ) try {
        wait();
      } catch (InterruptedException e) {}
      proceed(q,x);
      notifyAll();
      return true;
    }
  }
}
```

Fig. 4. The SynchronizedQueue aspect

5.2 Adding a Timestamp Concern

After implementing the buffer a new requirement has occurred: the buffer has to save current time associated with each stored item. Whenever an item is removed, the time how long it was stored should be printed to standard output. A Java programmer may use inheritance and composition as reuse techniques (Fig. 5). The problem is that three different concerns are intertwined within put/get and so these concerns cannot be composed separately. It means that e.g. if a programmer wants a queue with timing he cannot reuse the timing concern from TimeBuffer; he has to reimplement the timing concern in a new class that extends Queue. A slightly better solution seems to be using AOP and implementing the timing as an aspect (Fig. 6).



```
public class TimeBuffer extends Buffer {
  protected Queue delegateDates;
  public TimeBuffer(int capacity) {
    super(capacity);
    delegateDates = new Queue(capacity);
  }
  public synchronized boolean put(Object x) {
    super.put(x);
    delegateDates.put(
      new Long(System.currentTimeMillis())
    );
    return true;
  }
  public synchronized Object get() {
    Object tmp = super.get();
    Long date = (Long) delegateDates.get();
    long curr = System.currentTimeMillis();
    System.out.println(curr - date.longValue());
    return tmp;
  }
}
```

Fig. 5. The TimeBuffer class

Unless explicitly prevented, an aspect can apply to itself and can therefore change its own behavior. To avoid such situations, the instantiation pointcut is guarded by !cflow(within(Timing). Moreover, the instantiation pointcut in SynchronizedQueue has to be updated. It must be the same as in Timing. This can be done only destructively, because AspectJ does not allow for extending concrete aspects.

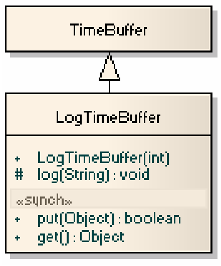
5.3 Adding a Logging Concern

The buffer has to log its size after each transaction. The OO mechanisms like inheritance and overridden allow a programmer for reusing TimeBuffer (Fig. 7a). The only problem is that four concerns are tangled within the LogTimeBuffer class. A module that addresses one concern can generally be used in more contexts than one that combines multiple concerns.

The AO solution is also noninvasive and it reuses the modules from the earlier stages. It just requires defining a new aspect (Fig. 7b). The bufferChange pointcut enumerates all join points that need to captured, by their exact signature. Such pointcut definition is particularly fragile to accidental join point misses. An evolution of the buffer will require revising the pointcut definition to explicitly add all new accessor methods to it.

```
public privileged aspect Timing
 pertarget( instantiation() ) {
  protected Queue delegateDates;

  protected pointcut instantiation():
    target(Queue) && !cflow(within(Timing));
  protected pointcut init(Queue q):
    execution( Queue.new(..) ) && target(q);
  protected pointcut execution_get():
    execution( Object Queue.get() );
  protected pointcut execution_put():
    execution(boolean Queue.put(Object));
  after(Queue q): init(q) {
    delegateDates = new Queue(q.buf.length);
  }
  after(): execution_get() {
    Long date = (Long) delegateDates.get();
    long curr = System.currentTimeMillis();
    System.out.println(curr - date.longValue());
  }
  after(): execution_put() {
    delegateDates.put(
     new Long(System.currentTimeMillis()) );
  }
}
```

Fig. 6. The Timing aspect



```
public aspect Logging {
  pointcut bufferChange(): (
    execution( * Queue.get() ) ||
    execution( * Queue.put(..)
    ) && !cflow(within(Timing));
  after(Queue q) returning:
   bufferChange() && target(q) {
    System.out.println(
      "buf size: "+q.size() );
  }
}
```

Fig. 7. a) A new class for Stage 3; b) The Logging aspect

5.4 Adding a New Getter

The buffer has to provide a method to get “N” next items. There is no efficient solution of this problem neither using Java nor AspectJ. In both cases, the condition for waiting on an item has to be reinforced by a lock flag. A lock flag is set when some thread initiates the “get N” transaction by getting the first item. The flag is unset after getting the last item. In Java (Fig. 8), not only does the synchronization concern has to be reimplemented but also logging. The reason is that in LogTimeBuffer logging is tangled together with synchronization, so it cannot be reused separately. The duplicate implementation might be a nightmare for maintenance.

In AspectJ, although synchronization is implemented in a separate module, it also cannot be reused in any way because an aspect cannot extend another concrete aspect. Thus, all code corresponding to the synchronization concerns has to be reimplemented (Fig. 9). A new method to get N items and locking mechanism are introduced to Queue by means of inter-type declaration. In addition, destructive changes in the Logging::bufferChange() pointcut are required (Fig. 9). Otherwise logs would be reported n times in response to the get(int n) method, instead of just once after completing the transaction. This is due to that get(int n) uses get() for retrieving every single item from the buffer.

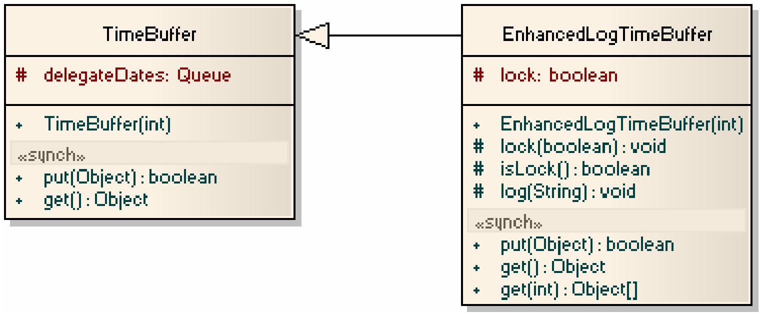


Fig. 8. A new class for Stage 4

5.5 Removing Logging and Timestamp

A programmer needs the enhanced buffer from Stage 4, but without the logging and timing concerns. In Java, he once again has to reimplement the get(int) method and much of the synchronization concerns. All to do in the AO version is to remove Logging and Timing from the compilation list.

6 Lessons Learned

In a AO system, one cannot tell whether an extension to the base code is safe simply by examining the base program in isolation. All pointcuts referring to the base program need to be examined as well. In addition, when writting a pointcut definition a programmer needs global knowledge about the structure of the application. This is due to the fact that pointcuts try to define intended conceptual properties about the base program, based on structural properties of the program. E.g. when implementing the Timing aspect, a programmer has to know that the synchronization concern affects each Queue structure, while the timing concern requires a non-blocking Queue.

In most cases, aspects cannot be made generic, because pointcuts as well as advices encompass information specific to a particular use, such as the classes involved, in the concrete aspect. As a result, aspects are highly dependent on other modules and their reusability is decreased. E.g. at Stage 1, the need to explicitly specify the Queue class and the two synchronization conditions means that no part of the SynchronizedQueue aspect can be made generic.

Futhermore, we have confirmed that the reusability of aspects is also hampered in cases where "join points seem to dynamically jump around", depending on the context certain code is called from [8]. In addition, the variety of pointcut designators makes pointcut expressions cumbersome (see EnhancedSynchronizedQueue::call_get()).

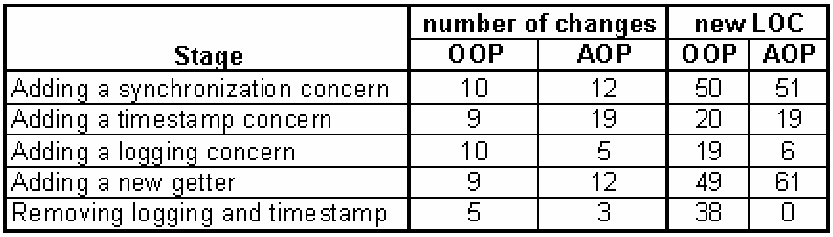
```
public aspect EnhancedSynchronizedQueue pertarget (
 target(Queue) && !cflow(within(Timing)) ) {
  private boolean Queue.lock = false;
  public void Queue.lock(boolean b) { lock=b; }
  public boolean Queue.isLock() { return lock; }
  public synchronized Object[] Queue.get(int n) {
    while ( isEmpty()||isLock() ) try { wait();
    } catch (InterruptedException e) {}
    lock(true);
    Object[] tmp = new Object[n];
    for(int i=0; i<n; i++) {
      while ( isEmpty() ) try { wait();
      } catch (InterruptedException e) {}
      tmp[i] = get();
    }
    lock(false); return tmp;
  }
  protected pointcut call_get():
   call( Object Queue.get() ) &&
   !cflow(withincode(* Queue.get(int) ));
  Object around(Queue q):
   call_get() && target(q) {
    synchronized(this) {
      while( q.isEmpty()||q.isLock() ) try {
        wait();
      } catch (InterruptedException e) {}
      Object tmp=proceed(q);
      notifyAll(); return tmp;
    }
  }
  //...
}

public aspect Logging {
  pointcut bufferChange(): (
    execution( * Queue.get(..) ) ||
    execution( * Queue.put(..) )
    ) && !cflow(within(Timing))
    && !cflow(withincode(* Queue.get(int) ));
  after(Queue q) returning :
   bufferChange() && target(q) {
    System.out.println( "buf size: "+q.size() );
  }
}
```

Fig. 9. The EnhancedSynchronizedQueue and Logging aspect (Stage 4)

7 Empirical Results

The measures for our two metrics were collected manually for each of the 31 modules across the OO implementations and 30 modules across the AO implementations. Table 3 presents the number of atomic changes for each stage. The lower numbers are better. In general, the data collected demonstrated that there is no winner with respect to software evolvability. AOP manifested superiority at Stage 3 and 5, while OOP in the rest of the cases. At Stage 3 we had to implement a logging concern which is one of the flagship examples of AOP usage.

Table 3. Numbers of atomic changes and new LOC per stage



Stage	number of changes		new LOC	
	OOP	AOP	OOP	AOP
Adding a synchronization concern	10	12	50	51
Adding a timestamp concern	9	19	20	19
Adding a logging concern	10	5	19	6
Adding a new getter	9	12	49	61
Removing logging and timestamp	5	3	38	0

Table 3 also shows how many new lines of code each stage requires. Stages 3 and 5 indicate significant differences between the implementations in favor of AOP. Stage 5 shows that lexical separation of concerns allows programmers to plug-out some concerns. The OO solution was substantial better only at Stage 4.

8 Related Work

Coady & Kiczales [10] compared the evolution of two versions (C and AspectC) of four crosscutting concerns in FreeBSD. They refactored the implementations of the following concerns in v2 code: page daemon activation, prefetching for mapped files, quotas for disk usage, and tracing blocked processes in device drivers. These implementations were then rolled forward into their subsequent incarnations in v3 and v4 respectively. In each case they found that, with tool support, the AO implementation better facilitated independent development and localized change. In three cases, configuration changes mapped directly to modifications to pointcuts and makefile options. In one case, redundancy was significantly reduced. Finally, in one case, the implementation of a system-extension aligned with an aspect was itself better modularized.

Bartsch & Harrison conducted an experiment [2] in which 11 students were asked to carry out maintenance tasks on one of two versions (Java and AspectJ) of an online shopping system. The results did seem to suggest a slight advantage for the subjects using the OO version since in general it took the subjects less time to perform maintenance tasks and it averagely required less line of code to implement a new requirement. However, the results did not show a statistically significant influence of AOP at the 5% level.

Sant'Anna et al. [33] conducted a quasi-controlled experiment to compare the use of OOP and AOP to implement Portalware (about 60 modules and over 1 KLOC). Portalware is a multi-agent system (MAS) that supports the development and management of Internet portals. The experiment team (3 PhD candidates and 1 M.Sc. student) developed two versions of the Portalware system: an AO version and an OO version. Next, the same team simulated seven maintenance/reuse scenarios that are recurrent in large-scale MAS. For each scenario, the difficulty of maintainability and reusability was defined in terms of structural changes to the artifacts in the AO and OO systems. The total lines of code, that were added, changed, or copied to perform the maintenance tasks, equaled 540 for the OO approach and 482 for the AO approach.

9 Summary

This paper presents a laboratory experiment comparing OOP and AOP with respect to software evolvability and reusability. Although a general conclusion cannot be drawn from only the one discussed experiment, an important outcome has been achieved in that the advocates of AOP have to take a position on our results. We have found no evidence to confirm the claim that AO software is easier to evolve. The experience gathered during the maintenance tasks points out that understanding the intricate dependencies existing between the modules of an AO system can be an arduous task. The advantage of AOP over OOP is also doubtful from the reusability point of view. In the current version of AspectJ aspects are holding too much information (the crosscutting logic and target module information) to fully take advantage of a lexical SoC. Nevertheless, since the superiority of AOP has been observed in some cases, we suggest that more research is necessary to identify the kind of crosscutting concerns that derive benefits from the aspect-oriented SoC.

References

1. Andrews, A., Ghosh, S., Man Choi, E.: A Model for Understanding Software Components. In: IEEE Inter. Conf. on Software Maintenance (ICSM 2002), Montreal, Canada (2002)
2. Bartsch, M., Harrison, R.: An exploratory study of the effect of aspect-oriented programming on maintainability. Software Quality Journal 16(1), 23–44 (2008)
3. Basili, V.R., Caldiera, G., Rombach, H.D.: Goal Question Metric Approach. In: Encyclopedia of Software Engineering, pp. 528–532. John Wiley & Sons, Inc., Chichester (1994)
4. Beltagui, F.: Features and Aspects: Exploring feature-oriented and aspect-oriented programming interactions. Tech. Report No: COMP-003-2003, Lancaster University (2003)
5. Bieman, J.M., Kang, B.: Cohesion and reuse in an object-oriented system. SIGSOFT Softw. Eng. Notes 20(SI), 259–262 (1995)
6. Bowen, T.P., Post, J.V., Tai, J., Presson, P.E., Schmidt, R.L.: Software Quality Measurement for Distributed Systems. Guidebook for Software Quality Measurement. Technical Report RADC-TR-83-175, vol. 2 (July 1983)

7. Breivold, H.P., Crnkovic, I., Land, R., Larsson, S.: Using Dependency Model to Support Software Architecture Evolution. In: 23rd IEEE/ACM Inter. Conf. on Automated Software Engineering, L'Aquila, Italy (2008)
8. Brichau, J., De Meuter, W., De Volder, K.: Jumping Aspects. In: Workshop on Aspects and Dimensions of Concerns at ECOOP 2000, Sophia Antipolis and Cannes, France (2000)
9. Chaumun, M.A., Kabaili, H., Keller, R.K., Lustman, F., Saint-Denis, G.: Design Properties and Object-Oriented Software Changeability. In: 13th Conf. on Software Maintenance and Reengineering, Kaiserslautern, Germany (2000)
10. Coady, Y., Kiczales, G.: Back to the future: a retroactive study of aspect evolution in operating system code. In: 2nd Inter. Conf. on Aspect-oriented software development (AOSD 2003), Boston, Massachusetts (2003)
11. Coleman, D., Ash, D., Lowther, B., Oman, P.: Using metrics to evaluate software system maintainability. IEEE Computer 27(8), 44–49 (1994)
12. Figueiredo, et al.: Evolving software product lines with aspects: An empirical study on design stability. In: 30th Inter. Conf. on Software Engineering, Leipzig, Germany (2008)
13. Fjeldstad, R., Hamlen, W.: Application program maintenance-report to to our respondents. In: Parikh, G., Zvegintzov, N. (eds.) Tutorial on Software Maintenance, pp. 13–27. IEEE Computer Soc. Press, Los Alamitos (1983)
14. Glass, R.L.: Facts and Fallacies of Software Engineering. Addison Wesley, Reading (2002)
15. Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H.: Modular Software Design with Crosscutting Interfaces. IEEE Software 23(1), 51–60 (2006)
16. Hanenberg, S., Unland, R.: Using and Reusing Aspects in AspectJ. In: Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA 2001, Tampa Bay, Florida (2001)
17. Hatton, L.: Does OO sync with how we think? IEEE Software 15(3), 46–54 (1998)
18. Hitz, M., Montazeri, B.: Measuring Coupling and Cohesion in Object-Oriented Systems. In: 3rd Inter. Symposium on Applied Corporate Computing, Monterrey, Mexico (1995)
19. IEEE Std 610.12-1990 (R2002), IEEE Standard Glossary of Software Engineering Terminology: IEEE (1990)
20. ISO/IEC 14764-1999, Software Engineering-Software Maintenance: ISO and IEC (1999)
21. Kästner, C., Apel, S., Batory, D.: A Case Study Implementing Features using AspectJ. In: 11th Inter. Conf. of Software Product Line Conf. (SPLC 2007), Kyoto, Japan (2007)
22. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Cristina Lopes, C., Loingtier, J., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
23. Koppen, C., Störzer, M.: PCDiff: Attacking the fragile pointcut problem. In: European Interactive Workshop on Aspects in Software, Berlin, Germany (2004)
24. Lemos, O.A., Junqueira, D.C., Silva, M.A., Fortes, R.P., Stamey, J.: Using aspect-oriented PHP to implement crosscutting concerns in a collaborative web system. In: 24th Annual ACM Inter. Conf. on Design of Communication, Myrtle Beach, South Carolina (2006)
25. Mancoridis, S., Mitchell, B.S., Rorres, C., Chen, Y., Gansner, E.R.: Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In: 6th Inter. Workshop on Program Comprehension (IWPC 1998), Ischia, Italy (1998)
26. McKee, J.: Maintenance as a function of design. In: 1984 National Computer Conf. AFIPS, vol. 53, pp. 187–193. AFIPS Press, Reston (1984)

27. Mens, T., Mens, K., Tourwé, T.: Software Evolution and Aspect-Oriented Software Development, a cross-fertilisation. ERCIM special issue on Automated Software Engineering, Vienna, Austria (2004)
28. Mortensen, M.: Improving Software Maintainability through Aspectualization. PhD thesis, Department of Computer Science, Colorado State University (2009)
29. Perepletchikov, M., Ryan, C., Frampton, K.: Cohesion Metrics for Predicting Maintainability of Service-Oriented Software. In: 7th Inter. Conf. on Quality Software (QSIC 2007), Portland, Oregon (2007)
30. Pigoski, T.M.: Practical Software Maintenance. Wiley Computer Publishing, Chichester (1997)
31. Przybylek, A.: An empirical assessment of the impact of AOP on software modularity. In: 5th Inter. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE 2010), Athens, Greece (2010)
32. Ryder, B.G., Tip, F.: Change impact analysis for object-oriented programs. In: 3rd ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Snowbird, Utah (2001)
33. Sant'Anna, C., Garcia, A., Chavez, C. Lucena, C., von Staa, A.: On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: 17th Brazilian Symposium on Software Engineering (SEES 2003), Manaus, Brazil (2003)
34. Standish, T.: An essay on software reuse. IEEE Transactions on Software Engineering 10(5), 494–497 (1984)
35. Storey, M.D., Fracchia, F.D., Müller, H.A.: Cognitive design elements to support the construction of a mental model during software exploration. J. Syst. Softw. 44(3), 171–185 (1999)
36. Tarr, P., Ossher, H., Harrison, W., Sutton, S.M.: N degrees of separation: multi-dimensional separation of concerns. In: 21st Inter. Conf. on Software Engineering (ICSE 2009), Los Angeles, California (1999)
37. Tourwé, T., Brichau, J., Gybels, K.: On the Existence of the AOSD-Evolution Paradox. In: AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, Massachusetts (2003)
38. Zhang, S., Gu, Z., Lin, Y., Zhao, J.: Change impact analysis for AspectJ programs. In: 24th IEEE Inter. Conf. on Software Maintenance, Beijing, China (2008)
39. Zhao, J.: Measuring Coupling in Aspect-Oriented Systems. In: 10th Inter. Software Metrics Symposium, Chicago, Illinois (2004)

Automatic Assignment of Work Items

Jonas Helming, Holger Arndt, Zardosht Hodaie, Maximilian Koegel, and Nitesh Narayan

Institut für Informatik, Technische Universität München, Garching, Germany
{helming,arndt,hodaie,koegel,narayan}@in.tum.de

Abstract. Many software development projects use work items such as tasks or bug reports to describe the work to be done. Some projects allow end-users or clients to enter new work items. New work items have to be triaged. The most important step is to assign new work items to a responsible developer. There are existing approaches to automatically assign bug reports based on the experience of certain developers based on machine learning. We propose a novel model-based approach, which considers relations from work items to the system specification for the assignment. We compare this new approach to existing techniques mining textual content as well as structural information. All techniques are applied to different types of work items, including bug reports and tasks. For our evaluation, we mine the model repository of three different projects. We also included history data to determine how well they work in different states.

Keywords: Machine learning, Task assignment, Bug report, UNICASE, Unified model, UJP.

1 Introduction

Many software development projects make use of repositories, managing different types of work items. This includes bug tracker systems like Bugzilla [1], task repositories like Jira [2] and integrated solutions such as Jazz [3] or the Team Foundation Server [4]. A commonality of all these repositories is the possibility to assign a certain work item to a responsible person or team [5].

It is a trend in current software development to open these repositories to other groups beside the project management allowing them to enter new work items. These groups could be end-users of the system, clients or the developers themselves. This possibility of feedback helps to identity relevant features and improves the quality by allowing more bugs to be identified [6]. But this advantage comes with significant cost ([7]), because every new work item has to be triaged. That means it has to be decided whether the work item is important or maybe a duplicate and further, whom it should be assigned to. As a part of the triage process it would be beneficial to support the assignment of work items and automatically select those developers with experience in the area of this work item. This developer is probably a good candidate to work on the work item, or, if the developer will not complete the work item himself, he probably has the experience to further triage the work item and reassign it.

There are several approaches, which semi-automatically assign work items (mostly bug reports) to developers. They are based on mining existing work items of a repository. We will present an overview of existing approaches in section 2.1.

In this paper we compare different existing techniques of machine learning and propose a new model-based approach to semi-automatically assign work items. All approaches are applied to a unified model, implemented in a tool called UNICASE [8]. The unified model is a repository for all different types of work items. Existing approaches usually focus on one type of work item, for example bug reports. The use of a unified model enables us to apply and evaluate our approach with different types of work items, including bug reports, feature requests, issues and tasks. We will describe UNICASE more in detail in section 3.

UNICASE does not only contain different types of work items, but also artifacts from the system specification, i.e. the system model ([9]). Work items can be linked to these artifacts from the system specification as illustrated in Figure 1. For example a task or a bug report can be linked to a related functional requirement. These links provide additional information about the context of a work item, which can be mined for semi-automatic assignment, as we will show in section 4. Our new approach for semi-automatic task assignment, called model-based approach, processes this information. The results of this approach can be transferred to other systems such as bug trackers where bug reports can be linked to affected components.

We found that existing approaches are usually evaluated in a certain project state (state-based), which means that a snap shot of the project is taken at a certain time and all work items have a fixed state. Then the assigned work items are classified by the approach to be evaluated and the results are compared with the actual assignee at that project state. We use this type of evaluation in a first step. However, state-based evaluation has two shortcomings: (1) The approach usually gets more information than it would have had at the time a certain work item was triaged. For example, additional information could have been attached to a work-item, which was not available for initial triage. (2) No conclusion can be made, how different approaches work in different states of a project, for example depending on the number of work items or on personal fluctuations. Therefore we evaluated our method also "history-based" which means that we mine all states of the project history and make automatic assignment proposals in the exact instance when a new work item was created. We claim that this type of evaluation is more realistic than just using one later state where possible more information is available.

We evaluate our approach by mining data from three different projects, which use UNICASE as a repository for their work items and system model. To evaluate which approach works best in our context as well as for a comparison of the proposed model-based approach we apply different machine learning techniques to assign work items automatically. These include very simple methods such as nearest neighbor, but also more advanced methods such as support vector machines or naive Bayes.

The paper is organized as follows: Section 2 summarizes related work in the field of automated task assignment as well as in the field of classification of software engineering artifacts. Section 3 introduces the prerequisites, i.e. the underlying model of work items and UNICASE, the tool this model is implemented in. Section 4 and 5 describe the model-based and the different machine learning approaches we applied in our evaluation. Section 6 presents the results of our evaluation on the three projects, in both a state-based and a history-based mode. In section 7 we conclude and discuss our results.

2 Related Work

In this section we give an overview over relevant existing approaches. In section 2.1 we describe approaches, which semi-automatically assign different types of work items. In section 2.2 we describe approaches, which classify software engineering artifacts using methods from machine learning and which are therefore also relevant for our approach.

2.1 Task Assignment

In our approach we refer to task assignment as the problem of classifying work items to the right developer. Determining developer expertise is the basis for the first part of our approach. In our case this is done by mining structured project history data saved within the UNICASE repository.

Most of the approaches for determining expertise rely on analyzing the code base of a software project mostly with the help of version control systems. Mockus et al. [10] treat every change from a source code repository as an experience atom and try to determine expertise of developers by counting related changes made in particular parts of source code. Schuler et al. [11] introduce the concept of usage expertise, which describes expertise in the sense of using source code, e.g. a specific API. Based on an empirical study, Fritz et al. [12] showed that these expertise measures acquired from source code analysis effectively represent parts of the code base, which the programmer has knowledge for. Sindhgatta [13] uses linguistic information found in source code elements such as identifiers and comments to determine the domain expertise of developers.

Other task classification approaches use information retrieval techniques such as text categorization to find similar tasks. Canfora et al. [14] demonstrate how information retrieval on software repositories can be used to create an index of developers for the assignment of change requests. Anvik [5] investigate applying different machine learning algorithms to an open bug repository and compare precision of resulting task assignments. Anvik et al. [7] apply SVM text categorization on an open bug repository for classifying new bug reports. They achieve high precision on the Eclipse and Firefox development projects and found their approach promising for further research. Čubranić et al. [15] employ text categorization using a naive Bayes classifier to automatically assign bug reports to developers. They correctly predict 30% of the assignments on a collection of 15,859 bug reports from a large open-source project. Yingbo et al. [16] apply a machine learning algorithm to workflow event log of a workflow system to learn the different activities of each actor and to suggest an appropriate actor to assign new tasks to.

2.2 Artifact Classification

Machine learning provides a number of classification methods, which can be used to categorize different items and which can also be applied to software artifacts. Each item is characterized by a number of attributes, such as name, description or due date, which have to be converted into numerical values to be useable for machine learning algorithms. These algorithms require a set of labeled training data, i.e. items for which the desired class is known (in our case the developer who an item has been assigned to). The labeled examples are used to train a classifier, which is why this method is called "supervised learning". After the training phase, new items can be classified automatically, which can serve as a recommendation for task assignment. A similar method has been employed by Cubranic et. al. [15] who used a naive Bayes classifier to assign bug reports to developers. In contrast to their work, our approach is not limited to bug reports, but can rather handle different types of work items. Moreover, we evaluate and compare different classifiers. Also Bruegge et. al. [17] have taken a unified approach and used a modular recurrent neural network to classify status and activity of work items.

3 Prerequisites

We implemented and evaluated our approach for semi-automated task assignment in a unified model provided by the tool UNICASE [18]. In this section we will describe the artifact types we consider for our approach. Furthermore we describe the features of these artifacts, which will form the input for the different approaches. UNICASE provides a repository, which can handle arbitrary types of software engineering artifacts. These artifacts can either be part of the system model, i.e. the requirements model and the system specification, or the project model, i.e. artifacts from project management such as work items or developers ([9])

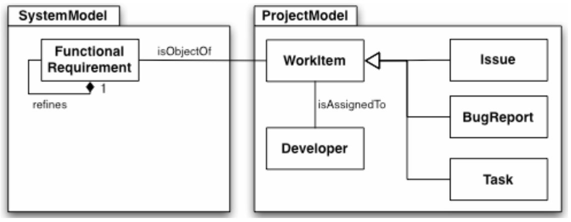


Fig. 1. Excerpt from the unified model of UNICASE (UML class diagram)

Figure 1 shows the relevant artifacts for our approach. The most important part is the association between work item and developer. This association expresses, that a work item is assigned to a certain developer and is therefore the association we semi-automatically want to set. Work items in UNICASE can be issues, tasks or bug reports. As we apply our approach to the generalization work item it is not limited to one of the subtypes as in existing approaches. As we proposed in previous work ([9]), work items in UNICASE can be linked to the related Functional Requirements modeled by the association *isObjectOf*. This expresses that the represented work of the work item is necessary to fulfill the requirement. This association, if already existent adds additional context information to a work item. Modeled by the *Refines* association, Functional requirements are structured in a hierarchy. We navigate this hierarchy in our model-based approach to find the most experienced developer, described in section 4. As a first step in this approach, we have to determine all related functional requirements of the currently inspected work item. As a consequence this approach only works for work items, which are linked to functional requirements.

While the model-based approach of semi-automated task assignment only relies on model links in UNICASE, the machine learning approaches mainly rely on the content of the artifacts. All content is stored in attributes. The following table provides an overview of the relevant features we used to evaluate the different approaches:

Table 1. Relevant features used to evaluate different approaches

Feature	Meaning
Name	A short and unique name for the represented work item.
Description	A detailed description of the work item.
ObjectOf	The object of the work item, usually a Functional Requirement.

We will show in the evaluation section, which features had a significant impact on the accuracy of the approach.

UNICASE provides an operation-based versioning for all artifacts [19]. This means all past project-states can be restored. Further we can retrieve a list of operations for each state, for example when a project manager assigned a work item to a certain developer. We will use this versioning system in the second part of our evaluation to exactly recreate a project state where a work item was created. The goal is to evaluate whether our approach would have chosen the same developer for an assignment as the project manager did. This evaluation method provides a more realistic result than evaluating the approaches only on the latest project state. With this method both approaches, machine learning and model-based, can only mine the information, which was present at the time of the required assignment recommendation.

4 Model-Based Approach

For the model-based assignment of work items we use the structural information available in the unified model of UNICASE. In UNICASE every functional requirement can have a set of linked work items. These are work items that need to be completed in order to fulfil this requirement.

The main idea of our model-based approach is to find the relevant part of the system for the input work item. In a second set we extract a set of existing work items, which are dealing with this part of the system. For a given input work item and based on this set we select a potential assignee. We will describe how this set is created using an example in Figure 2.

The input work item W is linked to the functional requirement B. To create the relevant set of work items (RelevantWorkItems(W)) we first add all work items, which are linked to functional requirement B (none in this example). Furthermore we add all work items linked to the refined functional requirement (A) and all work items linked to the refining requirements (C). In the example the set would consist of the work items 1 and 2. Futhermore, we recursively collect all work items from the refiningRequirements of A, which are neighbors of functional requirement B in the hierarchy (not shown in the example).

Using the set RelevantWorkItems(W) we determine expertise of each developer D regarding W (Expertise$_w$(D)). We defined Expertise$_w$(D) as the number of relevant work items this developer has already completed. After determining Expertise$_w$(D) for all developers, the one with highest expertise value is suggested as the appropriate assignee of the work item W.

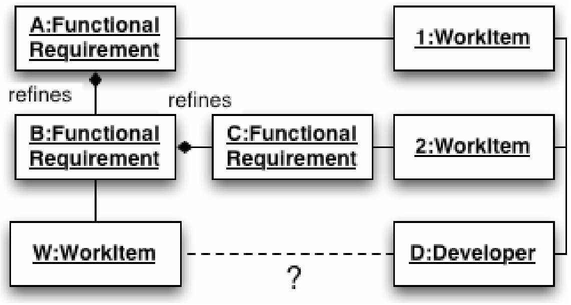


Fig. 2. Example for the model-based approach (UML object diagram)

5 Machine Learning Approaches

We have used the Universal Java Matrix Library (UJMP) [20] to convert data from UNICASE into a format suitable for machine learning algorithms. This matrix library can process numerical as well as textual data and can be easily integrated into other projects. All work items are aggregated into a two-dimensional matrix, where each row represents a single work item and the columns contain the attributes (name, description, ObjectOf association). Punctuation and stop words are removed and all strings are converted to lowercase characters. After that, the data is converted into a document-term matrix, where each row still represents a work item, while the columns contain information about the occurrence of terms in this work item. There are as many columns as different words in the whole text corpus of all work items. For every term, the number of occurrences in this work item is counted. This matrix is normalized using tf-idf (term frequency / inverse document frequency.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}}$$

where $n_{i,j}$ is the number of occurrences of the term t_i in document d_j, and the denominator is the sum of occurrences of all terms in document d_j.

$$idf_i = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

The inverse document frequency is a measure of the general importance of the term: logarithm of total number of documents in the corpus divided by number of documents where the term t_i appears. A deeper introduction to text categorization can be found in [21].

We have not used further preprocessing such as stemming or latent semantic indexing (LSI) as our initial experiments suggested, that it had only a minor effect on performance compared to the selection of algorithm or features. We have used the tf-idf matrix as input data to the Java Data Mining Package (JDMP) [22], which provides a common interface to numerous machine learning algorithms from various libraries. Therefore we were able to give a comparison between different methods:

Constant Classifier. The work items are not assigned to all developers on an equal basis. One developer may have worked on much more work items than another one. By just predicting the developer with the most work items it is possible to make many correct assignments. Therefore we use this classifier as a baseline, as it considers the input features.

Nearest Neighbor Classifier. This classifier is one of the simplest classifiers in machine learning. It uses normalized Euclidean distance to locate the item within the training set which is closest to the given work item, and predicts the same class as the labeled example. We use the implementation IB1 from Weka [23]. We did not use k-nearest neighbors, which usually performs much better, because we found the runtime of this algorithm to be too long for practical application in our scenario.

Decision Trees. Decision trees are also very simple classifiers, which break down the classification problem into a set of simple if-then decisions which lead to the final prediction. Since one decision tree alone is not a very good predictor, it is a common practice to combine a number of decision trees with ensemble methods such as boosting [24]. We use the implementation RandomCommittee from Weka.

Support Vector Machine (SVM). The support vector machine (SVM) calculates a separating hyperplane between data points from different classes and tries to maximize the margin between them. We use the implementation from LIBLINEAR [25], which works extremely fast on large sparse data sets and is therefore well suited for our task.

Naïve Bayes. This classifier is based on 'Bayes' theorem in probability theory. It assumes that all features are independent which is not necessarily the case for a document-term matrix. However, it scales very well to large data sets and usually yields good results even if the independence assumption is violated. We use the implementation NaiveBayesMultinomial from Weka [23] but also considered the implementation in MALLET [26], which showed lower classification accuracy (therefore we only report results from Weka).

Neural Networks. Neural networks can learn non-linear mappings between input and output data, which can be used to classify items into different classes (for an introduction to neural networks see e.g. [27]). We have tried different implementations but found that the time for training took an order of magnitudes longer than for the other approaches considered here. Therefore we were unable to include neural networks into our evaluation.

For the state-bases evaluation, we trained these classifiers using a cross validation scheme: The data has been split randomly into ten subsets. Nine of these sets were selected to train the classifier and one to assess its performance. After that, another set was selected for prediction, and the training has been performed using the remaining nine sets. This procedure has been performed ten times for all sets and has been repeated ten times (10 times 10-fold cross validation). For the history-based evaluation, the classifiers were trained on the data available at a certain project state to predict the assignee for a newly created work item. After the actual assignment through the project leader, the classifiers were re-trained and the next prediction could be made.

Depending on the approach, runtime for the evaluation of one classifier on one project ranged from a couple of minutes for LIBLINEAR SVM to almost two days for the nearest neighbor classifier. Although a thorough comparison of all machine learning methods would certainly have been interesting, we did not include a full evaluation on all projects and performed feature selection using LIBLINEAR, which was the fastest method of all. We argue that an algorithm for automatic task assigment would have to deliver a good accuracy but at the same time the necessary performance in terms of computing time to be useable in a productive environment. Therefore we could also discarded the classifiers nearest neighbour and random committee for the complete evaluation and report results only for UNICASE.

6 Evaluation

In this section we evaluate and compare the different approaches of semi-automated task assignment. We evaluated the approaches using three different projects. All projects have used UNICASE to manage their work items as well as their system documentation. In section 6.1 we introduce the three projects and their specific characteristics. In section 6.2 we evaluate the approaches „state-based“. This means we took the last available project state and tried to classify all assignments post-mortem. This evaluation technique was also used in approaches such as [7]. Based on the results of the state-based evaluation we selected the best-working configurations and approaches and evaluated them history-based. We stepped through the operation-based history of the evaluation projects to the instant before an assignment was done. This state is not necessarily a revision from the history but can be a state in between two revisions. This is why we had to rely on the operation-based versioning of UNICASE for this purpose. On the given state we tried to predict this specific assignment post-mortem and compared the result with the assignment, which was actually done by the user.

We claim this evaluation to be more realistic than the state-based as it measures the accuracy of the approach as if it had been used in practice during the project.

Furthermore it shows how the approaches perform in different states of the project depending on the different size of existing data. As a general measure to assess performance we used the accuracy, i.e. the number of correctly classified developers divided by the total number of classified work items. This measure has the advantage of being very intuitive and easily comparable between different approaches and data sets. Other common measures such as precision or sensitivity are strongly dependent on the number of classes (number of developers) and their distribution and therefore would make it more difficult to interpret the results for our three projects.

6.1 Evaluation Projects

We have used three different projects as datasets for our evaluation. As a first dataset we used the repository of the UNICASE project itself, which has been hosted on UNICASE for nearly one year. The second project, DOLLI 2, was a large student project with an industrial partner and 26 participants over 6 month. The goal of DOLLI was the development of innovative solutions for facility management. The third application is an industrial application of UNICASE for the development of the browser game "Kings Tale" by Beople GmbH, where UNICASE has been used for over 6 months now. The following table shows the number of participants and relevant work items per project.

Table 2. Developer and work items per project

	UNICASE	DOLLI	Kings Tale
Developers	39	26	6
Assigned work items	1191	411	256
Linked work items	290	203	97

6.2 State-Based Evaluation

For the state-based evaluation we used the last existing project state. Based on this state we try to classify all existing work items and compare the result with the actually assigned person. In a first step (section 6.2.1) we evaluate the machine learning approaches. In a second step we evaluate the model-based approach.

6.2.1 Machine Learning Approaches

We have chosen different combinations of features as input of the application and applied the machine learning approaches described in section 4 as well as the model-based approach described in section 5. Our goal was to determine the approaches, configurations and feature-sets, which lead to the best results and re-evaluate those in the history-based evaluation (section 6.3). We started to compare different feature sets. As we expected the name of a work item to contain the most relevant information, we started the evaluation with this feature only. In a second and third run, we added the attribute description and the association ObjectOf. The size of the tf-idf matrix varied depending on the project and the number of selected features, e.g. for the UNICASE project, from 1,408 columns with only name considered to 4,950 columns with all possible features.

Table 3 shows the results of different feature sets for the support vector machine. In all evaluation projects, the addition of the features description and ObjectOf increased accuracy. The combination of all three attributes leads to the best results. As a conclusion we will use the complete feature set for further evaluation and comparison with other approaches.

Table 3. Different sets of features as input data

Name			
	UNICASE	DOLLI	Kings Tale
SVM	36.5% (±0.7%)	26.5% (±0.7)	38.9 (±1.4)
Name and description			
	UNICASE	DOLLI	Kings Tale
SVM	37.1% (±1.0)	26.9% (±1.0)	40.7% (±0.9
Name, description and ObjectOf			
	UNICASE	DOLLI	Kings Tale
SVM	38.0% (±0.5)	28.9% (±0.7)	43.4% (±1.7)

In the next step we applied the described machine-learning approaches using the best working feature set as input (Name, description and ObjectOf). As a base line we started with a constant classifier. This classifier always suggests the developer for assignment who has the most work items assigned. As you can see in table 4, we can confirm the findings of [7], that SVM yields very good results. Random Committee performed quite badly in terms of accuracy and performance so we did not further evaluate them on all projects. The only competitive algorithm in terms of accuracy was Naïve Bayes, which was however worse on the Kings Tale project. As there was no significant difference between SVM and Naïve Bayes we chose SVM for further history-based evaluation due to the much better performance.

Table 4. Different machine learning approaches state-based

	UNICASE	DOLLI	Kings Tale
Constant	19,7%	9,0%	37,4%
SVM (LibLinear)	38.0% (±0.5)	28.9% (±0.7)	43.4% (±1.7)
Naïve Bayes	39.1% (±0.7)	29.7% (±0.9)	37.8% (±1.7)
Random Committee	23.2% (±0.2)		
Nearest Neighbor	6.9% (±0.1)		

6.2.2 Model-Based Approach

In the second step of the state-based evaluation we applied the model-based approach on the same data, which yields in surprisingly good results (see Table 5). The first row shows the accuracy of recommendations, when the model-based approach could be applied. The approach is only applicable to work items, which were linked to functional requirements. The number of work items the approach could be applied to is listed in Table 2. It is worth mentioning that once we also considered the second guess of the model-based approach and only linked work items, we achieved accuracies of 96.2% for the UNICASE, 78.7% for DOLLI and 94.7% for the Kings Tale project. For a fair overall comparison with the machine learning approaches, which are able to classify every work item we calculate the accuracy for all work items, including those without links, which could consequently not be predicted. Table 5 shows that the accuracy classifying all work items is even worse than the constant classifier. Therefore the model-based approach is only applicable for linked work items or in combination with other classifiers.

Table 5. Model-based approach

	UNICASE	DOLLI	Kings Tale
Linked work items	82,6%	58,1%	78,4%
All work items	19,9%	20,7%	29,3%

We have shown that the model-based approach can classify linked work items based on the ObjectOf reference. Therefore the approach basically mines, which developer has worked on which related parts of the system in the past (see section 5). One could claim that the machine learning approaches could also classify based on this information. Therefore we applied the SVM only on linked work items with all features and also only using the ObjectOf feature. The results (see Table 6) show that linked work items are better classified than non-linked. But even a restriction to only the feature ObjectOf did not lead to results as good as the model-based approach. Therefore we conclude to use the model-based approach whenever it is applicable and classify all other elements with SVM.

Table 6. Classification of linked work items

	UNICASE	DOLLI	Kings Tale
Constant	29,3%	18,3%	40,3%
SVM all features	53,9%	33,4%	50,2%
SVM only ObjectOf	49,7%	23,8%	49,2%

6.3 History-Based Evaluation

In the second part of our evaluation we wanted to simulate the actual use case of assignment. The problem with the state-based evaluation is, that the system has actually more information at hand, as it would have had at the time, a work item was assigned. Consequently we simulated the actual assignment situation. Therefore we used the operation-based versioning of UNICASE in combination with an analyzer framework provided by UNICASE. This enables us to iterate over project states through time and exactly recreate the state before a single assignment was done. Note that this state must not necessarily and usually also does not conform to a certain revision from versioning but is rather an intermediate version between to revisions. By using the operation-based versioning of UNICASE we are able to recover these intermediate states and to apply our approaches on exactly that state. For the machine learning approach (SVM) we trained the specific approach based on that state. For the model-based approach we used the state to calculate the assignment recommendation. Then, we compared the result of the recommendation with the assignment, which was actually chosen by the user. For the history-based evaluation we selected the two best working approaches from the state-based evaluation, SVM and the model-based approach. We applied the model-based approach only on linked work items.

We applied SVM and the model-based approach on the UNICASE and the DOLLI project. The Kingsthale project did not capture operation-based history data and was therefore not part of the history-based evaluation. As expected the results for all approaches are worse than in the state-based evaluation (see Table 7). Still all applied approaches are better than the base line, the constant classifier. An exception is the model-based approach applied on the DOLLI project, which shows slightly better results in the history-based evaluation. We believe the reason for this is that the requirements model, i.e. the functional requirements, and the related work items were added continuously over the project runtime. Therefore at the states when the actual assignment was done, the model-based approach could calculate its recommendation based on a smaller, but more precise set of artifacts. Furthermore we can observe, that the results for the UNICASE project differ largely from the state-based evaluation compared to the DOLLI project. A possible explanation for this is the higher personal fluctuation in the UNICASE project. This fluctuation requires the approaches to predict assignments for developers with a sparse history in the project and is therefore much more difficult. In the state-based evaluation the fluctuation is hidden, because the approaches can use all work items of the specific developer no matter when he joined the project.

Table 7. History-based (aggregated accuracy) UC= UNICASE

	UC history	UC state	DOLLI history	DOLLI state
Const.	22%	19,7%	7%	9,0%
SVM	29%	38.0%	27%	28.9%
Model-based	75%	82,6%	61%	58,1%

Figure 3 and 4 show the accuracy over time for the UNICASE project and SVM and model-based approach, respectively. All presented charts show two lines. The first line (black) shows the aggregated accuracy over time. The second line (dotted black) shows the aggregated accuracy for last 50 (DOLLI) and 100 (UNICASE) revisions and therefore reveals short time trends. In the selected time frame, both approaches do not fluctuate significantly. This shows, that both approaches could be applied to a continuous project, were developers join and leave the project.

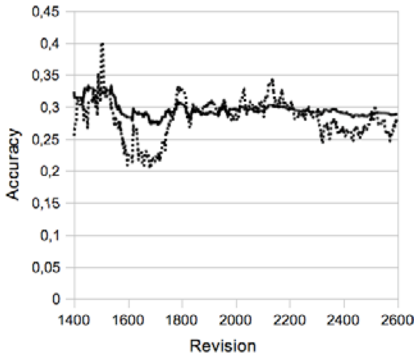


Fig. 3. SVM – UNICASE

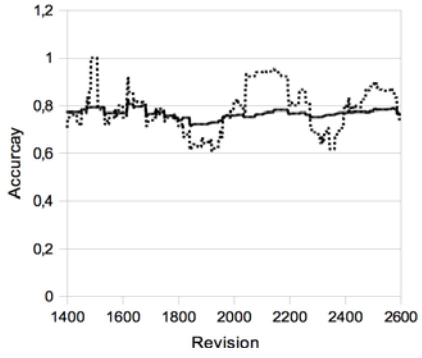


Fig. 4. Model-based – UNICASE

In contrast to the continuous UNICASE, we investigated the DOLLI project from the beginning to the end (Figure 5 and 6) including project start-up and shutdown activities. We observe that SVM lacks in accuracy at the beginning, where new developers start to work on the project. For an efficient classification the SVM approach has to process a reasonable set of work items per developer. Therefore a high accuracy is only reached to the end of the project. A closer look at the accuracy of the model-based approach shows that it decreases at the end of the project. Starting from around revision 430 there has been a process change in the project as well as a reorganization of the functional requirements. This clearly affects the results of the model-based approach as it relies on functional requirements and their hierarchy. In contrast to the model-based approach, SVM seems to be quite stable against this type of change.

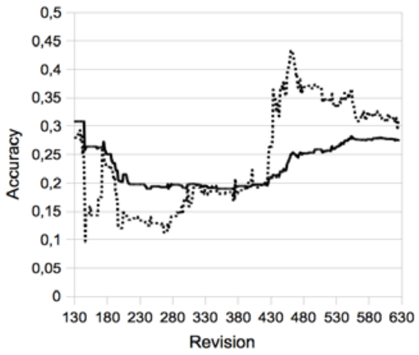


Fig. 5. SVM – DOLLI

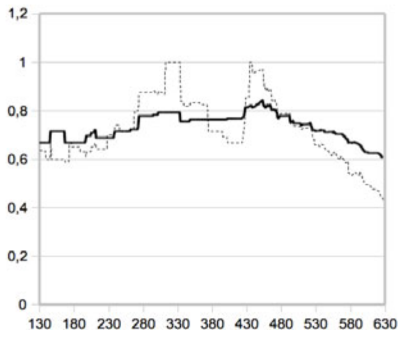


Fig. 6. Model-based – DOLLI

7 Conclusions

We applied machine learning techniques as well as a novel model-based approach to semi-automatically assign different types of work items. We evaluated the different approaches on three existing projects. We could confirm the results from previous authors that the support vector machine (SVM) is an efficient solution to this classification task. The naïve Bayes classifier can lead to similar results, but the implementation we have used showed a worse performance in terms of computing time. The model-based approach is not applicable to all work items as it relies on structural information, which is not always available. However it showed the best results of all approaches whenever it was applicable.

The model-based approach relies on links from work items to functional requirements and is therefore not directly applicable in other scenarios than UNICASE, where these links do not exist. Although we believe that it can be transferred to other systems where similar information is provided. Bug trackers often allow to link bug reports to related components. Components on the other hand have relations to each other, just like the functional requirements in our context. An obvious shortcoming of the model-based approach is that it requires a triage by the affected part of the system no matter which model is used. On the one hand we believe, that it is easier for users to triage a work item by the affected part of the system rather than assign it, especially if they do not know the internal structure of a project. On the other hand if a project decides to use both, links to related system parts and links to assignees, the model-based approach can help with the creation of the latter.

In the second part of our evaluation, we tried to simulate the use case in a realistic assignment scenario. Therefore we applied the two best working approaches over the project history and predicted every assignment at exactly the state, when it was originally done. As a consequence all approaches can process less information than in the first part of the evaluation, which was based on the last project state. As expected the history-based evaluation leads to lower accuracies for all approaches. The model-based approach is less affected by this scenario than the SVM. A possible reason for that is that the model-based approach is not so much depending on the size of the existing data but more on its quality. This assumption is underlined by the behavior of the model-based approach during massive changes in the model, leading to lower results. In contrast to that, the SVM was not so sensible to changes in model, but more to fluctuations in the project staffing.

We conclude that the best solution would be a hybrid approach, i.e. a combination of the model-based approach and SVM. This would lead to high results for linked work items, but would also be able to deal with unlinked items.

References

1. Bugzilla, `http://www.bugzilla.org/`
2. Jira, `http://www.atlassian.com/software/jira`
3. Jazz Community Site, `http://jazz.net/`
4. Team Foundation Server, `http://msdn.microsoft.com/`
5. Anvik, J.: Automating bug report assignment. In: Proceedings of the 28th International Conference on Software Engineering, p. S.940 (2006)
6. Raymond, E.: The cathedral and the bazaar. Knowledge, Technology & Policy 12, S.23–S.49 (1999)
7. Anvik, J., Hiew, L., Murphy, G.C.: Who should fix this bug? In: Proceedings of the 28th International Conference on Software Engineering, pp. S.361–S.370. ACM, Shanghai (2006)

8. Bruegge, B., Creighton, O., Helming, J., Koegel, M.: Unicase – an Ecosystem for Unified Software Engineering Research Tools. In: Workshop Distributed Software Development - Methods and Tools for Risk Management, Bangalore, India, pp. S.12–S.17 (2008)
9. Helming, J., David, J., Koegel, M., Naughton, H.: Integrating System Modeling with Project Management–a Case Study. In: International Computer Software and Applications Conference, COMPSAC 2009 (2009)
10. Mockus, A., Herbsleb, J.D.: Expertise browser: a quantitative approach to identifying expertise. In: Proceedings of the 24th International Conference on Software Engineering, pp. S.503–S.512 (2002)
11. Schuler, D., Zimmermann, T.: Mining usage expertise from version archives. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, pp. S.121–S.124 (2008)
12. Fritz, T., Murphy, G.C., Hill, E.: Does a programmer's activity indicate knowledge of code? In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, p. S.350 (2007)
13. Sindhgatta, R.: Identifying domain expertise of developers from source code. In: Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. S.981–S.989 (2008)
14. Canfora, G., Cerulo, L.: How software repositories can help in resolving a new change request. In: STEP 2005, p. S.99 (2005)
15. Čubranić, D.: Automatic bug triage using text categorization. In: SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering & Knowledge Engineering, pp. S.92–S.97 (2004)
16. Yingbo, L., Jianmin, W., Jiaguang, S.: A machine learning approach to semi-automating workflow staff assignment. In: Proceedings of the 2007 ACM symposium on Applied computing, p. S.345 (2007)
17. Bruegge, B., David, J., Helming, J., Koegel, M.: Classification of tasks using machine learning. In: Proceedings of the 5th International Conference on Predictor Models in Software Engineering (2009)
18. UNICASE, `http://www.unicase.org`
19. Koegel, M.: Towards software configuration management for unified models. In: Proceedings of the 2008 International Workshop on Comparison and Versioning of Software Models, pp. S.19–S.24 (2008)
20. Arndt, H., Bundschus, M., Naegele, A.: Towards a next-generation matrix library for Java. In: COMPSAC: International Computer Software and Applications Conference (2009)
21. Sebastiani, F.: Machine learning in automated text categorization. ACM Computing Surveys (CSUR) 34, S.1–S.47 (2002)
22. Holger Arndt, I.I.: The Java Data Mining Package–A Data Processing Library for Java
23. Witten, I.H., Frank, E.: Data mining: practical machine learning tools and techniques with Java implementations. ACM SIGMOD Record 31, S.76–S.77 (2002)
24. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. Journal of Computer and System Sciences 55, 119–139 (1997)
25. Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: LIBLINEAR: A library for large linear classification. The Journal of Machine Learning Research 9, S.1871–S.1874 (2008)
26. MALLET homepage, `http://mallet.cs.umass.edu`
27. Haykin, S.: Neural networks: a comprehensive foundation. Prentice Hall, Englewood Cliffs (2008)

UDeploy: A Unified Deployment Environment

Mariam Dibo and Noureddine Belkhatir

Laboratoire d'Informatique de Grenoble
681, Rue de la Passerelle, BP 72, 38402, St. Martin d'Hères, France
{Mariam.Dibo,Noureddine.Belkhatir}@imag.fr

Abstract. In the software life cycle we have mainly three activities: (1) the pre-development (requirements, specification and design), (2) the development (implementation, prototyping, testing) and (3) the post-development (deployment). Software deployment encompasses all post-development activities that make an application operational. These activities, identified as deployment life cycle, include: i) software packaging, ii) loading and installation of software on client sites, iii) instance creation, iv) configuration and v) updating. The development of system-based components made it possible in order to highlight this part of the global software lifecycle, as illustrated by numerous industrial and academic studies. However these are generally developed ad hoc, and consequently platform-dependent. Deployment systems, such as supported by middleware environments (CCM, .Net and EJB), specifically develop mechanisms and tools related to pre-specified deployment strategies. Our work, related to the topic of distributed component-based software applications, aims at specifying a generic deployment framework independent of the target environments. Driven by the meta-model approach, we first describe the abstractions used to characterize the deployed software. Then, we specify the deployment infrastructure and processes, highlighting the activities to be carried out and the support for their execution.

Keywords: Deployment, Meta model, Model, Software component, MDA.

1 Introduction

Component-based software approach [25] is intended to improve the reuse of component enabling the development of new applications by assembling pre-existing components. A software component can be deployed independently and may be composed by third parties [25].

Nowadays, the component approach and distribution make deployment a very complex process. Many deployment tools exist, we identified three types of systems: 1) those developed by the industry and integrated into a middleware environment like EJB [8], CCM [21] and .Net [26, 27]; 2) those projected by the OMG (industry) [22] [9] based on more generic models and; 3) the more formal systems projected by academic works in current component models like Open Service Gateway Initiative (OSGI) [1], Web Services [11], SOFA [3], Architecture Description Languages (ADL) [4] and UML 2.0 [24].

Generally, deployment tools are often built in an ad hoc way; i.e. specific to a technology or an architecture and covering partially the deployment life cycle (using generally the installation scripts).

Hence, deployment is seen as the post development activities that make software usable. It covers the description of the application to deploy, the description of the physical infrastructure, the description of the deployment strategies, the planning activities and the plan execution.

The deployment issue deals with aspects as diverse as satisfying software and hardware constraints of the components concerning the resources of the machines that support them, the resolution of inter-component dependency, the installation and "instantiation" of components via the middleware and the container, the interconnection of components, their activation and the management of dynamic updates. Thus, the challenge [5] is to develop a generic framework encompassing a specific approach and supporting the whole deployment process. [6] presents the conceptual framework of this approach and [7] presents the different models based on the MDA approach [23].

This paper focuses on the implementation part fulfilled by UDeploy (models transformation) and the presentation of a case study to illustrate our approach. The rest of this paper is organized as follows: part 2 presents the related works. Part 3 presents the architecture of our deployment tool. Part 4 presents the model transformation. Finally in part 5, we present the perspectives of this work.

2 Related Works

We identified several works on the deployment that have been classified into two broad categories.

In the first category, there are mainly all the more classic works developed for the monolithic software systems and that emphasize on the setup activity.

In the second category, there are all the systems of deployment developed recently for the software based-components. We identified two types of systems in this category:

- those developed by industry on an ad 'hoc way and integrated into a middleware type of environments;
- those of a higher level of abstraction based on explicit model proposed by the OMG on one hand and on the other hand by the academic world.

2.1 Deployment in Middleware

The pros of deployment in application based-component like EJB [8], CCM [21] and .Net [26, 27] relay in the fact that the technologies are effective thus answers specific needs. The cons are that the abstraction level is very low therefore it is necessary to make each activity manually. In such contexts and with these facts, it is easy to deduce that there is a real need to standardize the deployment of distributed applications. The middleware does not support the description of the domain. They contain less semantics to describe applications; for example, the needs of an application may be a specific version of software, and a memory size greater than 10 GB. Since none of these constraints will be checked during installation, this corresponds to a single copy component assembly. The deployment descriptor expresses the same mechanism for each middleware but described them in different ways.

2.2 Deployment in OMG Specification

The industry felt the necessity to join their efforts. They anticipated an approach which capitalizes on their experiences in deployment (OMG's approach). This specification has inspired many academics. OMG's Deployment and Configuration (D&C) [22] specification is based on the use of models, meta-models and their transformation. This specification standardizes many aspects of deployment for component-based distributed systems, including component assembly, component packaging, package configuration, and target domain resource management. These aspects are handled via a data model and a runtime model. The data model can be used to define/generate XML schemas for storing and interchanging metadata that describes component assemblies and their configuration and deployment characteristics. The runtime model defines a set of managers that process the metadata described in the data model during system deployment. An implementation of this specification is DAnCE (Deployment And Configuration Engine) [9].

2.3 Deployment in Academic Approaches

In current component models like, Open Service Gateway Initiative (OSGI) [1], Web Services [11], SOFA [3], Architecture Description Languages (ADL) [4] and UML 2.0 [24], components are defined in the form of architectural units [15]. The ADL [19] such as Acme, AADL, Darwin and Wright allow modeling components, to model connectors and to model architecture configurations; however deployment process in ADL is not specified. UML2.0 allows describing system hardware. But deployment diagram in UML2.0 is a static view of the run-time configuration of processing nodes and the components that run on those nodes. Other approaches such as SOFA do not address the processing part. The plan containing the information on the application is directly executed from a centralized server, assuming that remote sites can instantiate remote components from this server.

Tables 1, 2, 3 and 4 presented in annex, present an assessment related to three main notions occurring in the constitution of a deployment system which are the application, the domain, the deployment strategies and the deployment plan.

- The domain notion covers all machines connected to a network where a software system is deployed. This infrastructure is seen as a set of distributed and interconnected sites. Each site is associated with the meta-information of the site characteristics descriptions.
- The application notion covers all the application components and the meta-information for their descriptions.
- The deployment strategies guide the creation of the deployment plan. The deployment strategies allow expressing the actions to be led to deploy a component by assuring success and safety properties.
- The deployment plan for an application A consists of components C1 to Ci where i>= 1 and for a domain D consisting of Sites S1 to Sj where j> = 1 is all valid

placements (Ci, Sj). It is calculated from a planner engine. This engine operates on a static process which allow visualizing a state of the system and the information remains motionless during the computing plan or following a dynamic process which allows visualizing the forecasts and to supervise their realization; the information used is variable during the computing plan.

3 UDeploy Architecture

Concerning to the assessment obtained from the state of the art practice of the related works, we think that a good solution to automate component based systems deployment owes to [6]:

- cover all deployment activities,
- be independent from technologies,
- be independent from any philosophy of components based approach,
- offer a distributed deployment engine,
- propose specific language strategies in order to make the deployment flexible and to support existing strategies in the deployment environments.

The analysis of a deployment system highlights activities independent from technologies and what we could factor as the:

- modeling of the application to deploy,
- modeling of the components execution environment,
- creation of the deployment plan.

Therefore, we propose a deployment architecture [7] based on MDA (Model-Driven Architecture) approach [23] with the use of models, meta-models and their transformation (MDA approach is described in the section 4.1). MDA approach allows offering a unified framework based on deployment activities using generic descriptors that may subsequently be customized for specific platforms.

Deployment study in enterprise business practices allowed us to understand that the deployment must be flexible according to the needs of the company and according to the technical specifications of the application. Hence, we propose a fourth meta-model related to deployment strategies in addition to the three common meta-models.

Figure 1 illustrates this deployment process comprising the following six main activities:

- The **application modeling** which describes the application to be deployed; in other words, it specifies all the components that compose the application and, the resource constraints of these components.
- The **domain modeling** which describes the deployment environment, meaning which specifies all sites that compose it and the available resources.
- The deployment strategies modeling which allow describing the policies to be implemented in order to make the deployment plan flexible according to specific needs.
- The creation of the deployment plan which from an application model, a domain model and a deployment strategies model produce a deployment plan.

- The transformation covers two main activities:
  - the customization of the deployment plan - the deployment plan produced at the end of the deployment plan creation activity is at a pim level (platform independent model), therefore it is independent from any technology. this deployment plan is seen as a set of placements. this generic plan must be customized to one or several psm level plans (platform specific model); i.e. specific to technologies so that they can be executed by the middleware targets. the deployment plan answers to the question “where to deploy?”.
  - the generation of the deployment descriptor - the deployment descriptor is built from information within the application model and also from other information (application non-functional properties) produced by the *deployer*. the deployment descriptor answers to the question “how the container must manage components to deploy?”.
- The deployment plan execution - some middleware do not offer any support for the implementation of the deployment plan. in that case, the generic plan will be translated into an appropriate description of the target middleware (script). This description will be carried out by our deployment tool by invoking methods of the target middleware.

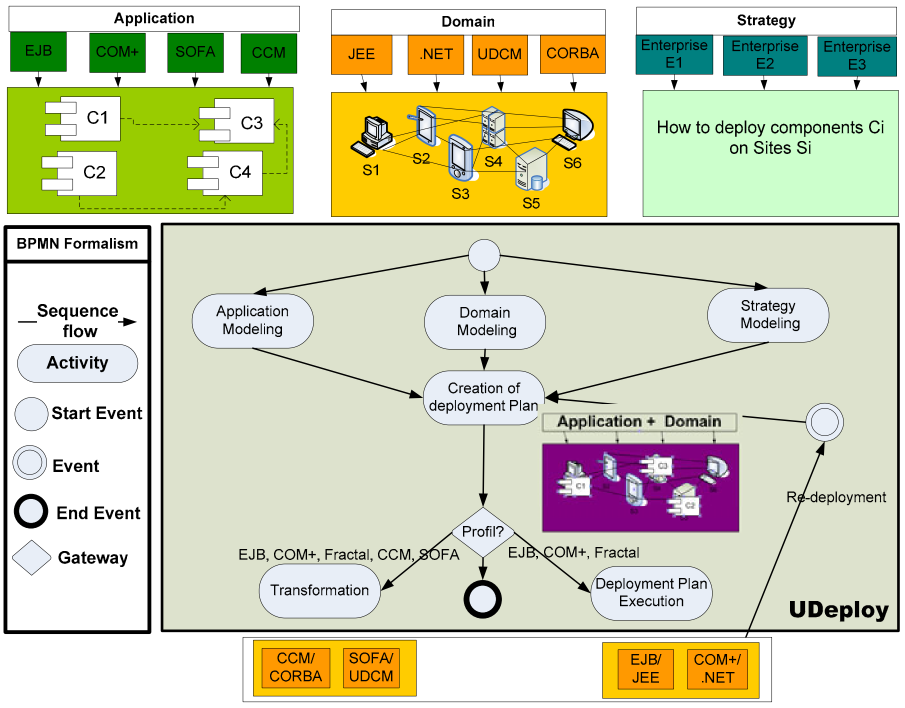


Fig. 1. UDeploy architecture

4 Model Transformation

4.1 MDA Approach

The MDA approach [23] has been proposed by the OMG in response to the problems posed by the multiplicity of systems, languages and technologies. The main idea of the MDA approach is the separation of technical concerns from trades [10]. The key concepts inherent to the MDA approach are:

- The PIM (Platform Independent Model) - these models are independent on the technology platforms such as EJB, CCM, COM + and, provide a high level of abstraction.
- The PSM (Platform Specific Model) - these models are dependent on the technology platforms and correspond to the executable code.
- The transformation - PIM to PSM or PSM to PIM passage occurs by models transformations. A model transformation is defined from a set of rules. These rules can be described using a QVT type transformation tool (Query View Transformation) [20], or by implementing its own processing tool. There are several transformations tools and languages such as QVT-core (MTF [12]), QVT-relations (medini QVT [18]) and QVT-like ATL [14, 13], Tefkat [17] and VIATRA [28]).

4.2 MDA Advantage

The main advantages of the MDA approach are productivity and portability [16]. Productivity is because developers can now focus on the development of the PIM models. They will work at a level where technical details are no longer specified. These technical details will be added to the PSM level at the time of processing. This improves productivity in two ways. First and foremost, PIM developers will omit specific details. Second, several PSM can be obtained for different platforms with less effort. Portability is because a PIM may be automatically transformed into several PSM for various platforms. Thus, everything specified at PIM level will remain portable. The only thing needed is to make sure that the code to be generated is conform to the technology of an execution target platform.

4.3 MDA and Deployment

Conventional deployment tools integrated into the middleware, re-develop in a specific manner the mechanisms and the deployment processes. These tools can be seen to be at the PSM level. So, applying MDA to deploy would define deployment meta-models at PIM level and that can be customized for different platforms.

4.4 Transformation Language

Transformation of models [2] may be operated by a non-formal language, by a specific QVT or by a transformation algorithm that sets the mapping between different models. The transformation language that we propose is mixed, hence based on the QVT ATL and on transformation algorithms (figure 2).

Model transformation is not based on the UDeploy application model, the domain model and the UDeploy strategies model, but covers the UDeploy deployment plan and the UDeploy deployment descriptor model.

Transformation of the deployment plan model consists of the projection of the UDeploy plan model from a PIM level to a PSM level plan models (EJB, CCM, .NET, SOFA). Specific deployment plan models are executed by middleware targets in order to implement the deployment.

Transformation of the deployment descriptor model consists of the transformation of the UDeploy descriptor model from a PSM level to a PIM level descriptor models (EJB, CCM). Specific deployment descriptor models are used by the middleware targets to manage components.

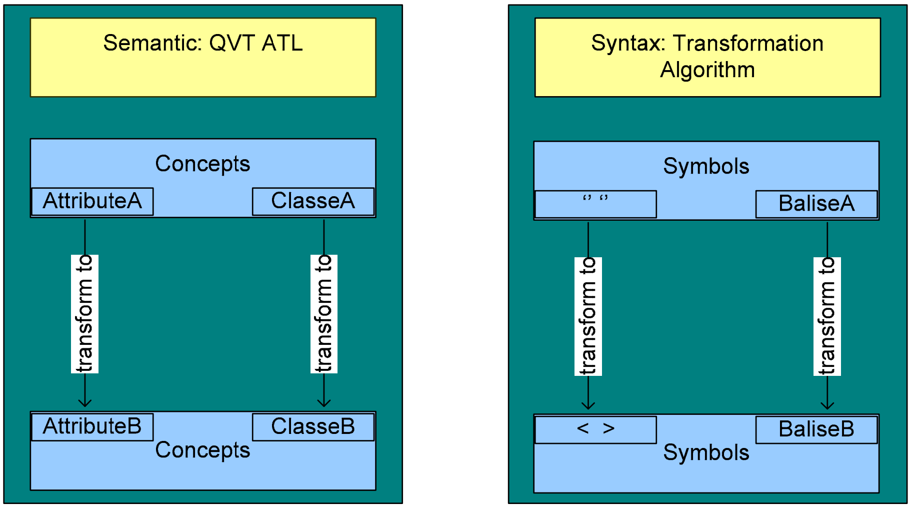


Fig. 2. Transformation language (QVT ATL and algorithm)

4.5 QVT ATL

We use the QVT ATL for semantic transformation (Figure 3). Semantic transformation corresponds to the transformation of the concepts. A concept A in a source model might be called concept B in a target model. ATL is a model transformation language developed on top of the Eclipse platform. It provides ways to generate target models from source models via transformation rules. An ATL transformation rule is written as follow:

```
rule R {
  from e : source-meta-model ! el-e (cond)
  to s : target-meta-model ! el-s
  (-- ex. title<- e.title, name<- e.name+ "new")
}
```

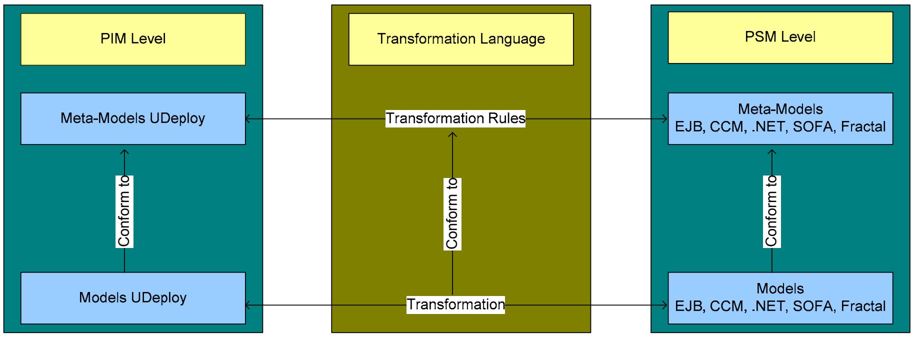


Fig. 3. Transformation QVT ATL

4.6 Transformation Algorithm

We use algorithms for syntactic transformation. An M1 model that meets a source meta-model criteria might be written in Java while an M2 model compliant to a target meta-model might be written in XML. The UDeploy deployment plan meta-models and the UDeploy deployment descriptor meta-models are written in DTD (Document Type Definition). For practical reasons, we have decided to develop our algorithms and to manage the models' persistence with Java. Hence, we needed to operate three basic transformations (figure 4):

- The transformation of the DTD UDeploy meta-models to XSD UDeploy meta-models via the XMLPad tool.
- The transformation of the XSD UDeploy meta-models to Ecore UDeploy meta-models via the EMF tool.
- The transformation of the Ecore UDeploy meta-models to Java UDeploy meta-models via the EMF tool.

The chain of transformation from the DTD meta-model plan and the DTD meta-model descriptor to the Java meta-model plan and the Java meta-model descriptor does occur only once.

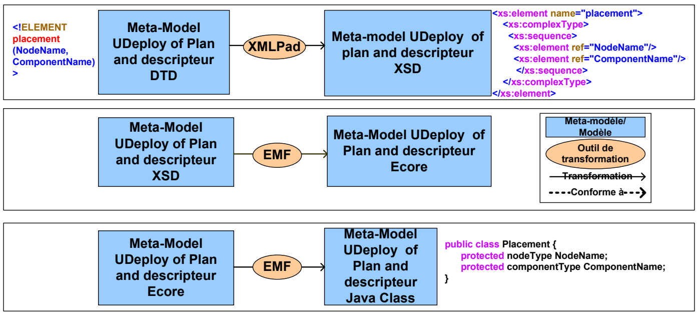
Once the Java classes are created, they will be instantiated by the deployment plan data and the deployment descriptor.

We have syntactic transformation (Figure 4) for each technology such as EJB (AlgoEJBPlan, AlgoEJBDescriptor algorithms CCM AlgoCCMDescriptor (AlgoCCMPlan), .NET (AlgoNETPlan) and SOFA (AlgoSOFAPlan). The algorithm allows producing a target model which will be conformed syntactically to the target meta-model.

4.7 Examples of Model Transformation

4.7.1 EJB, NET and CCM Deployment Plan Personalization (Semantic)

At the end of the planning process, we obtain a PIM level UDeploy deployment plan model. This deployment plan must be customized for execution target platforms.



```
<!ELEMENT
placement
(NodeName,
ComponentName)
>
```

```
<xs:element name="placement">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="NodeName"/>
    <xs:element ref="ComponentName"/>
     </xs:sequence>
  </xs:complexType>
</xs:element>
```

```
public class Placement {
    protected nodeType NodeName;
    protected componentType ComponentName;
}
```

Meta-Model UDeploy of Plan and descripteur DTD
XMLPad
Meta-model UDeploy of plan and descripteur XSD
Meta-Model UDeploy of Plan and descripteur XSD
EMF
Meta-Model UDeploy of Plan and descripteur Ecore
Meta-modèle/ Modèle
Outil de transformation
Transformation
Conforme à
Meta-Model UDeploy of Plan and descripteur Ecore
EMF
Meta-Model UDeploy of Plan and descripteur Java Class

Fig. 4. Transformation algorithme

The example below shows the transformation process of the UDeploy deployment plan meta-model to the EJB, .NET and CCM platforms plan meta-model.

The rule #1 takes as input the UDeploy deployment plan meta-model (source) and as output an EJB, .NET and CCM deployment plan meta-model (target). The transformation concerns the *DeploymentPlan* class of the source meta-model and the *DeploymentPlan class* of the target meta-model. The *PlanId* attribute of the target meta-model will be the *PlanId* attribute of the source meta-model.

```
    rule R1 {
    from in : UDeployDeploymentPlanMetaModel ! DeploymentPlan
    to out : EJB_NET_CCMDeploymentPlanMetaModel ! DeploymentPlan
    PlanId<- in.PlanId }
```

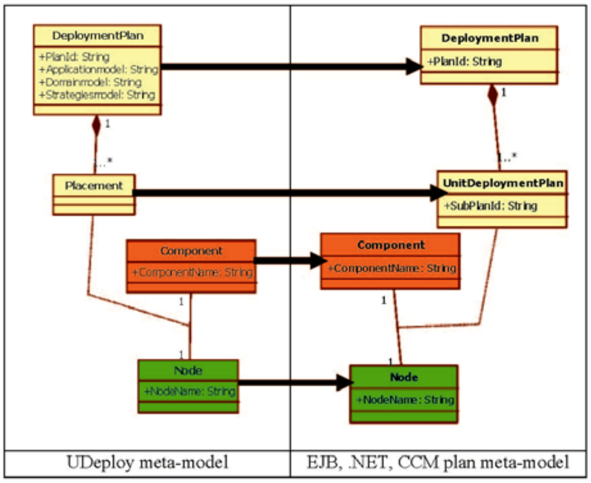


Fig. 5. Semantic transformation

The rule #2 takes as input the UDeploy deployment plan meta-model (source) and as output an EJB, .NET and CCM deployment plan meta-model (target).

The transformation concerns the DeploymentPlan class of the source meta-model and the *UnitDeploymentPlan class* of the target meta-model.

The Sub*PlanId* attribute of the UnitDeploymentPlan class will be a concatenation of the the *PlanId* attribute of the source model and a plan number supplied by the user (getSubPlanNmber () method).

```
    rule R2 {
    from in : UDeployDeploymentPlanMetaModel ! DeploymentPlan
    to out : EJB_NET_CCMDeploymentPlanMetaModel ! UnitDeploymentPlan
    SubPlanId<- in.PlanId + getSubPlanNmber()}
```

The rule #3 takes as input the UDeploy deployment plan meta-model (source) and as output an EJB, .NET and CCM deployment plan meta-model (target).

The transformation concerns the *component* class of the source meta-model and the *component class* of the target meta-model. The *ComponentName* attribute of the target meta-model will be the *ComponentName* attribute of the source meta-model.

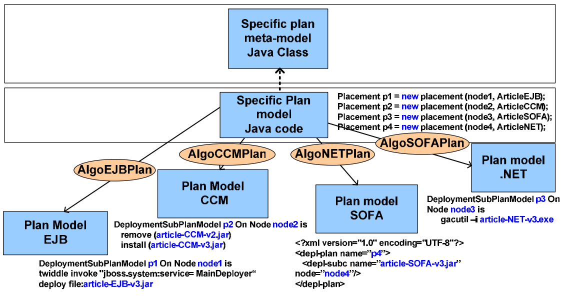
```
    rule R3 {
    from in : UDeployDeploymentPlanMetaModel ! Component
    to out : EJB_NET_CCMDeploymentPlanMetaModel ! Component
    ComponentName<- in.ComponentName }
```

The rule #4 takes as input the UDeploy deployment plan meta-model (source) and as output an EJB, .NET and CCM deployment plan meta-model (target). The transformation concerns the Node class of the source meta-model and the the *Node* class of the target meta-model. The *NodeName* attribute of the target meta-model will be the *NodeName* attribute of the source meta-model.

```
    rule R4 {
    from in : UDeployDeploymentPlanMetaModel ! Node
    to out : EJB_NET_CCMDeploymentPlanMetaModel ! Node
    NodeName<- in.NodeName }
```

4.7.2 EJB, .NET and CCM Deployment Plan Customization (Syntactic)

Below, we will present four examples of syntactic customization (Figure 6). The customization algorithms of the deployment plan for the EJB, CCM, .NET and SOFA platforms are respectively AlgoEJBPlan, AlgoCCMPlan, AlgoNETPlan and AlgoSOFAPlan.



```
Placement p1 = new placement (node1, ArticleEJB);
Placement p2 = new placement (node2, ArticleCCM);
Placement p3 = new placement (node3, ArticleSOFA);
Placement p4 = new placement (node4, ArticleNET);
```

```
DeploymentSubPlanModel p1 On Node node1 is
twiddle invoke "jboss.system:service= MainDeployer“
deploy file:article-EJB-v3.jar
```

```
DeploymentSubPlanModel p2 On Node node2 is
  remove (article-CCM-v2.jar)
  install (article-CCM-v3.jar)
```

```
<?xml version="1.0" encoding="UTF-8"?>
<depl-plan name=”p4”>
  <depl-subc name=”article-SOFA-v3.jar”
node=”node4”/>
</depl-plan>
```

```
DeploymentSubPlanModel p3 On
Node node3 is
    gacutil –i article-NET-v3.exe
```

Fig. 6. Syntactic transformation

```
AlgoEJBPlan
Input: Specific deployment plan EJB mEJB
Ouput: document d
Debprog;
       document d;
       For each placement p in mEJB do
              C=getComponentName(p);
              IC=getImplementation(C);
              N=getNodeName(p);
              NT=getNodeServerType(N);
              if (NT== JBOOS) then d.write('On Node', N , 'is
              twiddle invoke "jboss.system:service= MainDeployer"
              deploy file:',IC);
              endif;

              else if (NT==JONAS) then d.write('On Node', N ,
              'jonas admin -a',IC);
              endelseif ;
       endo;
       Return d;
Finprog;
```

```
AlgoNETPlan
Input: Specific deployment plan .NET mNET
Ouput: document d
Debprog;
       document d;
       For each placement p in mNET do
              C=getComponentName(p);
              IC=getImplementation(C);
              N=getNodeName(p) ;
              d.write('On Node', N, ' is gacutil -i',IC);
       endo;
       Return d;
Finprog;
```

```
AlgoCCMPlan
Input: Specific deployment plan CCM mCCM
Ouput: document d
Debprog;
       document d;
       For each placement p in mCCM do
              C=getComponentName(p);
              IC=getImplementation(C);
              N=getNodeName(p);
              d.write('On Node', N, 'Install(',IC, ')');
       enddo;
       Return d;
Finprog;
```

5 Conclusions and Perspectives

We develop UDeploy, a prototype based on the MDA approach which ensures tree main tasks: (i) it manages the planning process from meta-information related to the application, the infrastructure and the deployment strategies, (ii) it generates specific deployment descriptors related to the application and the environment (i.e. the machines connected to a network where a software system is deployed) and (iii) it executes a deployment plan.

We have positive feedbacks with our case study and its experimentation on EJB, .NET and CCM platforms. Our current projects include carrying out other experiments and evaluations to show the feasibility of the approach, for example its application to industrial systems, .NET and CCM.

References

1. Alliance, O.: OSGi 4.0 release. Specification (October 2005), `http://www.osgi.org/`
2. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model transformations? transformation models! In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
3. Bures, T., Hnetynka, P., Plasil, F.: Sofa 2.0: Balancing advanced features in a hierarchical component model. In: SERA, pp. 40–48. IEEE Computer Society, Los Alamitos (2006)
4. Clements, P.C.: A survey of architecture description languages. In: IWSSD 1996: Proceedings of the 8th International Workshop on Software Specification and Design, p. 16. IEEE Computer Society, Washington, DC, USA (1996)
5. Dibo, M., Belkhatir, N.: Challenges and perspectives in the deployment of distributed components-based software. In: ICEIS (3), pp. 403–406 (2009)
6. Dibo, M., Belkhatir, N.: Defining an unified meta modeling architecture for deployment of distributed components-based software applications. In: 12th International Conference on Enterprise Information Systems (ICEIS), Funchal, Madeira, Portugal (June 2010)
7. Dibo, M., Belkhatir, N.: Model-driven deployment of distributed components-based software. In: 5th International Conference on Software and Data Technologies (ICSOFT), Athens, Greece (July 2010)
8. Dochez, J.: Jsr 88: Java enterprise edition 5 deployment api specification (2009), `http://jcp.org/aboutJava/communityprocess/mrel/jsr088/index.html`
9. Edwards, G.T., Deng, G., Schmidt, D.C., Gokhale, A.S., Natarajan, B.: Model-driven configuration and deployment of component middleware publish/subscribe services. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 337–360. Springer, Heidelberg (2004)
10. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of mda, pp. 90–105. Springer, Heidelberg (2002)
11. Gustavo, A., Fabio, C., Harumi, K., Vijay, M.: Web Services: Concepts, Architecture and Applications (2004)
12. IBM. Mtf: Model transformation framework (2010), `http://www.alphaworks.ibm.com/tech/mtf`
13. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)

14. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., Valduriez, P.: Atl: a qvt-like transformation language. In: OOPSLA Companion, pp. 719–720 (2006)
15. Kaur, K., Singh, H.: Evaluating an evolving software component: case of internal design. SIGSOFT Softw. Eng. Notes 34(4), 1–4 (2009)
16. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston (2003)
17. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In: MoDELS Satellite Events, pp. 139–150 (2005)
18. mediniQVT. medini qvt (2010), `http://projects.ikv.de/qvt`
19. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Trans. Softw. Eng. 26(1), 70–93 (2000)
20. OMG. MOF QVT Final Adopted Specification. Object Modeling Group (June 2005)
21. OMG. Corba component model 4.0. (2006), specification `http://www.omg.org/docs/formal/06-04-01.pdf`
22. OMG. Deployment and configuration of component-based distributed application (2006), specification `http://www.omg.org`
23. T.O.M.G. OMG. Omg model driven architecture (2005), `http://www.omg.org`
24. T.O.M.G. OMG. Unified modeling language (2007), `http://www.omg.org`
25. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley Professional, England (2002)
26. Troelsen, A.: Chapter 1: The Philosophy of .NET, vol. Pro VB 2008 and the .NET 3.5 Platform. APress (2008)
27. Troelsen, A.: Chapter 15: Introducing.NET Assemblies, vol. Pro VB 2008 and the.NET 3.5 Platform. APress (2008)
28. Varró, D., Balogh, A.: The model transformation language of the viatra2 framework. Sci. Comput. Program. 68(3), 214–234 (2007)

Appendix

Table 1. Application meta-model comparison

Approach	Application meta-model			
	Software architecture	Software constraints	Hardware constraints	Descriptor Format
EJB	*	/	/	Conform to DTD ejb-jar
CCM	*	*	*	Conform to DTD SoftwarePackageDescriptor.dtd CORBAComponentDescriptor.dtd
.Net	*	* (only assembly dependencis)	/	Manifest MSI
D&C	*	*	*	ComponentDataModel ComponentManagementModel
Software Dock	*	*	*	Conform to DTD DSD
Orya	*	*	*	Product model
Fractal	*	*	*	Fractal ADL (xml)
SOFA	*	/	/	SOFA component meta-model
UML	*	*	*	Component diagram

* (supported) / (no-supported)

Table 2. Domain meta-model comparison

Approach	Domain meta-model			
	Hardware architecture	Software resources	Hardware resources	Descriptor Format
EJB	/	/	/	/
CCM	/	/	/	/
.Net	/	/	/	
D&C	*	*	*	TargetDataModel TargetManagement Model
Software Dock	*	*	*	Fieldock Releasedock
Orya	*	*	*	Site model
Fractal	/	/	/	/
SOFA	* Docks (remote node)	/	/	Sofanode (centralized node)
UML	*	*	*	Deployment diagram

* (supported) / (no-supported)

Table 3. Deployment strategies meta-model comparison

Approach	Deployment strategies meta-model			
	Technology	Enterprise	Fixed/ Flexible	Language for stratégies specification
EJB	*	/	Fixed	/
CCM	*	/	Fixed	SoftwarePackageDescriptor.dtd CORBAComponentDescriptor.dtd CORBAassemblyDescriptor.dtd
.Net	*	/	Fixed	*(only for application update)
D&C	/	/	/	/
Software Dock	*(configuration)	/	Fixed	/
Orya		* (few semantic)	Flexible	Strategies model
Fractal	*		Fixed	
SOFA	*		Fixed	* (only for dynamic adaptation via DCUP)
UML	/	/	/	/

* (supported) / (no-supported)

Table 4. Deployment plan meta-model comparison

Approach	Deployment plan meta-model			
	Processus de planification supporté	Plan de déploiement complet	Plan de déploiement exécutable	Format du plan de déploiement
EJB	/	/	/	Script
CCM	/	/	/	Script
.Net	/	/	/	Script
D&C	*	*	*	XML document for CCM/Dance
Software Dock	*	*	/	Embedded in the tool (code)
Orya	*	*	/	Embedded in the tool (code)
Fractal	/	/	/	/
SOFA	/	*	*	XML Document
UML	/	/	/	Deployment Diagram

* (supported) / (no-supported)

Author Index