Leszek A. Maciaszek
César González-Pérez
Stefan Jablonski (Eds.)

Communications in Computer and Information Science 69

Evaluation of Novel Approaches to Software Engineering

3rd and 4th International Conferences, ENASE 2008/2009
Funchal, Madeira, Portugal, May 2008 / Milan, Italy, May 2009
Revised Selected Papers

Springer

Communications
in Computer and Information Science 69

Leszek A. Maciaszek César González-Pérez
Stefan Jablonski (Eds.)

Evaluation of Novel Approaches to Software Engineering

3rd and 4th International Conferences
ENASE 2008/2009
Funchal, Madeira, Portugal, May 4-7, 2008
Milan, Italy, May 9-10, 2009
Revised Selected Papers

Springer

Volume Editors

Leszek A. Maciaszek
Macquarie University
Sydney, NSW, Australia
E-mail: leszek@science.mq.edu.au

César González-Pérez
LaPa – CSIC, Spain
E-mail: cesar.gonzalez-perez@iegps.csic.es

Stefan Jablonski
University of Bayreuth, Germany
E-mail: stefan.jablonski@uni-bayreuth.de

Library of Congress Control Number: 2010934784

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

ISSN 1865-0929
ISBN-10 3-642-14818-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-14818-7 Springer Berlin Heidelberg New York

springer.com

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

Software engineering is understood as a broad term linking science, traditional engineering, art and management and is additionally conditioned by social and external factors (conditioned to the point that brilliant engineering solutions based on strong science, showing artistic creativity and skillfully managed can still fail for reasons beyond the control of the development team).

Modern software engineering needs a paradigm shift commensurate with a change of the computing paradigm from:

1. Algorithms to interactions (and from procedural to object-oriented programming)
2. Systems development to systems integration
3. Products to services

Traditional software engineering struggles to address this paradigm shift to interactions, integration, and services. It offers only incomplete and disconnected methods for building information systems with fragmentary ability to dynamically accommodate change and to grow gracefully. The principal objective of contemporary software engineering should therefore be to try to redefine the entire discipline and offer a complete set of methods, tools and techniques to address challenges ahead that will shape the information systems of the future.

This book is a peer-reviewed collection of papers, modified and extended for the purpose of this publication, but originally presented at two successive conferences: ENASE 2008 and ENASE 2009 (ref. http://www.enase.org/). The mission of the ENASE (Evaluation of Novel Approaches to Software Engineering) conference series is to be a prime international forum to discuss and publish research findings and IT industry experiences with relation to the evaluation of novel approaches to software engineering. By comparing novel approaches with established traditional practices and by evaluating them against software quality criteria, the ENASE conference series advances knowledge and research in software engineering, identifies the most hopeful trends and proposes new directions for consideration by researchers and practitioners involved in large-scale software development and integration.

The high quality of this volume is attested twofold. Firstly, all papers submitted to ENASE were subject to stringent reviews that resulted in acceptance rates of 25% or less. Secondly, only selected papers were considered for this volume and only after considering revisions, modifications and extensions.

The book's content is placed within the entire framework of software engineering activities, but with particular emphasis on experience reports and evaluations (qualitative and quantitative) of existing approaches as well as new ideas and proposals for improvements. The book is dedicated to managing one of the most important challenges that society is facing – how to ensure that humans can understand, control and gracefully evolve complex software systems. A related aim of the book is to ensure the uptake of the presented research through further knowledge-transfer activities by researchers, educators, project managers and IT practitioners.

January 2010

Leszek A. Maciaszek
Cesar Gonzalez-Perez
Stefan Jablonski

Organization

Conference Co-chairs

Joaquim Filipe	Polytechnic Institute of Setúbal/INSTICC, Portugal
Leszek A. Maciaszek	Macquarie University, Sydney, Australia

Program Co-chairs

Cesar Gonzalez-Perez	LaPa - CSIC, Spain
Stefan Jablonski	University of Bayreuth, Germany

Organizing Committee

Sérgio Brissos	INSTICC, Portugal
Paulo Brito	INSTICC, Portugal
Marina Carvalho	INSTICC, Portugal
Helder Coelhas	INSTICC, Portugal
Vera Coelho	INSTICC, Portugal
Andreia Costa	INSTICC, Portugal
Bruno Encarnação	INSTICC, Portugal
Bárbara Lima	INSTICC, Portugal
Raquel Martins	INSTICC, Portugal
Carla Mota	INSTICC, Portugal
Vitor Pedrosa	INSTICC, Portugal
Vera Rosário	INSTICC, Portugal
José Varela	INSTICC, Portugal

ENASE Program Committee

Pekka Abrahamsson, Finland
Witold Abramowicz, Poland
Hernán Astudillo, Chile
Colin Atkinson, Germany
Muhammad Ali Babar, Ireland
Giuseppe Berio, France
Robert Biddle, Canada
Maria Bielikova, Slovak Republic
Mokrane Bouzeghoub, France
Dumitru Burdescu, Romania
Ismael Caballero, Spain
Wojciech Cellary, Poland
Sung-Deok Cha, Korea
Panagiotis Chountas, UK
Lawrence Chung, USA
Alex Delis, Greece
Jens Dietrich, New Zealand
Jim Duggan, Ireland
Margaret Dunham, USA
Schahram Dustdar, Austria

Joerg Evermann, Canada
Maria João Ferreira, Portugal
Bogdan Franczyk, Germany
Steven Fraser, USA
Felix Garcia, Spain
Marcela Genero, Spain
Janusz Getta, Australia
Tudor Girba, Switzerland
Cesar Gonzalez-Perez, Spain
Hans-Gerhard Gross, The Netherlands
Jarek Gryz, Canada
Jo Hannay, Norway
Igor Hawryszkiewycz, Australia
Brian Henderson-Sellers, Australia
Zbigniew Huzar, Poland
Stefan Jablonski, Germany
Slinger Jansen, The Netherlands
Stan Jarzabek, Singapore
Wan Kadir, Malaysia
Philippe Kruchten, Canada
Michele Lanza, Switzerland
Xabier Larrucea, Spain
Kecheng Liu, UK
Leszek Maciaszek, Australia
Cristiano Maciel, Brazil
Lech Madeyski, Poland
Radu Marinescu, Romania
Claudia Bauzer Medeiros, Brazil
Leon Moonen, Norway
Johannes Müller, Germany
Sascha Mueller, Germany
Anne Hee Hiong Ngu, USA
Selmin Nurcan, France
James Odell, UK
Antoni Olive, Spain
Maria Orlowska, Poland
Janis Osis, Latvia
Mieczyslaw Owoc, Poland
Marcin Paprzycki, Poland
Jeffrey Parsons, Canada
Mario Piattini, Spain
Klaus Pohl, Germany
Naveen Prakash, India
Lutz Prechelt, Germany
Awais Rashid, UK
Gil Regev, Switzerland
Félix García Rubio, Spain
Francisco Ruiz, Spain
Chris Sacha, Poland
Krzysztof Sacha, Poland
Motoshi Saeki, Japan
Stephen R. Schach, USA
Heiko Schuldt, Switzerland
Manuel Serrano, Spain
Jan Seruga, Australia
Tony Sloane, Australia
Il-Yeol Song, USA
Dan Tamir, USA
Stephanie Teufel, Switzerland
Dave Thomas, Canada
Rainer Unland, Germany
Jean Vanderdonckt, Belgium
Christelle Vangenot, Switzerland
Athanasios Vasilakos, Greece
Olegas Vasilecas, Lithuania
Jari Veijalainen, Finland
Juan D. Velasquez, Chile
Maria Esther Vidal, Venezuela
Maurizio Vincini, Italy
Igor Wojnicki, Poland
Viacheslav Wolfengagen, Russian Federation
Martin Wolpers, Belgium
Huahui Wu, USA
Lu Yan, UK
Sung-Ming Yen, Taiwan
Janette Young, UK
Weihua Zhuang, Canada

Table of Contents

Part I: Evaluation of Novel Approaches to Software Engineering 2008

Part II: Evaluation of Novel Approaches to Software Engineering 2009

Part I
Evaluation of Novel Approaches to Software Engineering 2008

Measuring Characteristics of Models and Model Transformations Using Ontology and Graph Rewriting Techniques

Motoshi Saeki[1] and Haruhiko Kaiya[2]

[1] Tokyo Institute of Technology, Tokyo 152-8552, Japan
saeki@se.cs.titech.ac.jp
[2] Shinshu University, Wakasato 4-17-1, Nagano 380-8553, Japan
kaiya@cs.shinshu-u.ac.jp

Abstract. In this paper, we propose the integrated technique related to metrics in a Model Driven Development context. More concretely, we focus on the following three topics; 1) the application of a meta modeling technique to specify formally model-specific metrics, 2) the definition of metrics dealing with semantic aspects of models (semantic metrics) using domain ontologies, and 3) the specification technique for the metrics of model transformations based on graph rewriting systems.

Keywords: model metrics, model transformation, ontology, graph rewriting, model driven development (MDD).

1 Introduction

The techniques of metrics are to quantify characteristics of software products and development processes, e.g. quality, complexity, stability, development efforts, etc., and are significant to predict these characteristics at earlier steps of the development processes, as well as to know the current status of the products and the processes.

Model Driven Development (MDD) is one of the promising approaches to develop software of high quality with less developers' efforts. There are wide varieties of models such as object-oriented models, data flow models, activity models etc. that can be produced in the MDD processes. For example, object oriented modeling mainly adopts class diagrams consisting of classes and their associations, while in data flow modeling data flow diagrams having processes (data transformation), data flows and data stores, etc. are used. In this situation, according to models, we should use different metrics to quantify their characteristics, and it is necessary to define the metrics according to the models. For example, in the object-oriented models, we can use the CK metrics [5] to quantify the structural complexity of a produced class diagram, while we use another metrics such as Cyclomatic number [15] for an activity diagram of UML (Unified Modeling Language). These examples show that effective metrics vary on a model, and first of all, we need a technique to define model-specific metrics in MDD context.

The existing metrics such as CK metrics and Cyclomatic number are for expressing the structural, i.e. syntactical characteristics of artifacts only, but do not reflect their semantic aspects. Suppose that we have two class diagrams of Lift Control System, which

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 3–16, 2010.

are the same except for the existence of class “Emergency Button”; one includes it, while the other does not. It can be considered that the diagram having “Emergency Button” is structurally more complex rather than the other, because the number of the classes in it is larger. However, it has higher quality in the semantics of Lift Control System because Emergency Button is mandatory for the safety of passengers in a practical Lift Control Systems. This example shows that the metrics expressing semantic aspects is necessary to measure the quality of artifacts more correctly and precisely. In particular, these semantic aspects are from the properties specific to problem and application domains.

In MDD, model transformation is one of the key technologies [17,12,16] for development processes, and the techniques of quantifying the characteristics of model transformations and their processes are necessary. Although we have several metrics for quantifying traditional and conventional software development processes such as staff-hours, function points, defect density during software testing etc., they are not sufficient to apply to model transformation processes of MDD. In other words, we need the metrics specific to model transformations in addition to model-specific metrics. Suppose that a metric value can express the structural complexity of a model, like CK metrics, a change or a difference between the metric values before and after a model transformation can be considered as the improvement or declination of model complexity. For example, the model transformation where the resulting model becomes more complex is not better and has lower quality from the viewpoint of model complexity. This example suggests that we can define a metric of a model transformation with a degree of changing the values of model-specific metrics by its application. Following this idea, the formal definition of a transformation should include the definition of model-specific metrics so that the metrics can be calculated during the transformation.

In this paper, we propose a technique to solve the above three problems; 1) specifying metrics according to modeling methods, 2) semantic metrics, and 3) formal definition of model transformation with metrics. More concretely, we take the following three approaches;

1. Using a meta modeling technique to specify model-specific metrics
   Since a meta model defines the logical structure of models, we can specify the definition of metrics, including its calculation technique, as a part of the meta model. Thus we can define model-specific metrics formally. We use Class Diagram plus predicate logic to represent meta models with metrics definitions.
2. Using a domain ontology
   We use a domain ontology to provide for the model the semantics specific to a problem and application domain. As mentioned in [14], we consider an ontology as a thesaurus of words and inference rules on it, where the words in the thesaurus represent concepts and the inference rules operate on the relationships on the words. Each concept of an ontology can be considered as a semantically atomic element that anyone can have the unique meaning in the domain. The inference rules can automate the detection of inconsistent parts and of the lacks of mandatory model elements [10]. Thus we can calculate semantic metrics such as the degree on how many inconsistent parts are included in the model, considering the mapping from a model to a domain ontology. Intuitively speaking, the semantic metrics value is based on the degree of how faithfully the model reflects the structure of the domain

ontology. In other words, we consider a domain ontology as an *ideal and ultimate* model in the domain, and we calculate semantic characteristics of a model by its deviation from the domain ontology.

3. Using a graph rewriting system to measure model transformations
Since we adopt Class Diagram to represent a meta model, a model following the meta model is mathematically considered as a graph. Model transformation rules can be defined as graph rewriting rules and the rewriting system can execute the transformation automatically in some cases. The metric values to be calculated are attached to graph rewriting rules, and can be evaluated and propagated between the models during the transformation. This evaluation and propagation mechanism is similar to Attribute Grammar, and the evaluation and propagation methods can be defined within the graph rewriting rules. We define the metrics of a model transformation using the model-specific metric values of the models, which are attached to the rules.

The usages of the meta modeling technique for defining model-specific metrics [20] and of graph rewriting for formalizing model transformations [6,19] are not new. In fact, OMG is currently developing a meta model that can specify software metrics [18] and the workshop [4] to evaluate practical usability of model transformation techniques including graph rewriting systems was held. However, but the contribution of this paper is the integrated application technique of meta modeling and graph rewriting to solve the new problems mentioned above, with unified framework.

The rest of the paper is organized as follows. In the next section, we introduce our meta modeling technique so as to define model-specific metrics. Section 3 presents the usage of domain ontologies to provide semantic metrics and the way to embed them into the meta models. In section 4, we define model transformation with graph rewriting and illustrate the metrics being calculated on the transformation. Section 5 is a concluding remark and discusses the future research agenda.

2 Meta Modeling and Defining Metrics

A meta model specifies the structure or data type of the models and in this sense, it can be considered as an abstract syntax of the models. In addition to meta models, we should consider constraints on the models. Suppose that we define the meta model of the models which are described with class diagrams, i.e. object-oriented models. In any class diagram, we cannot have different classes having the same name, and we should specify this constraint to keep consistency of the models on their meta model.

In our technique, we adopt a class diagram of UML for specifying meta models and predicate logic for constraints on models. The example of the meta model of the simplified version of class diagrams is shown in Figure 1 (a). As shown in the figure, it has the concepts "Class", "Operation" and "Attribute" and all of them are defined as classes and these concepts have associations representing logical relationships among them. For instance, the concept "Class" has "Attribute", so the association "has_Attribute" between "Class" and "Attribute" denotes this relationship.

Metrics is defined as a class having the attribute "value" in the meta model as shown in the Figure 1 (b). The "value" has the metrics value and its calculation is defined

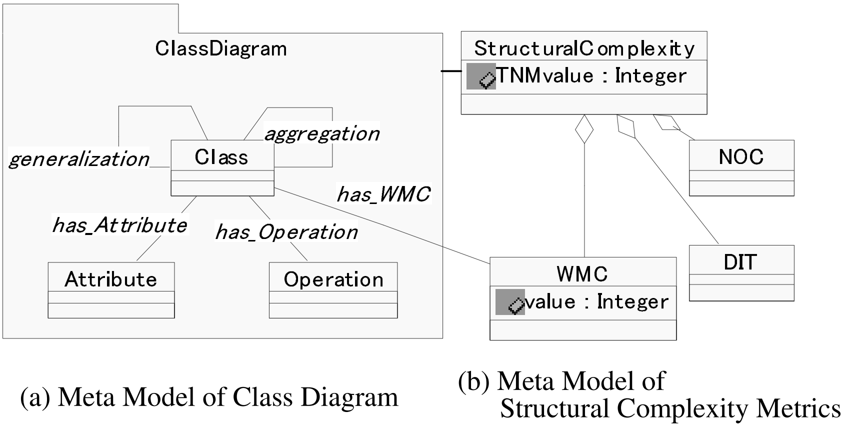


Fig. 1. Meta Model with Metrics Definitions

as a constraint written with a predicate logic formula. For example, WMC (Weighted Method per Class) of CK metrics is associated with each class of a class diagram. Intuitively speaking, the value of WMC is the number of the methods in a class when we let all weighted factors be 1. It can be defined as follows;

$$\begin{aligned}&WMC\_value = \\ &\quad \#\{m : ClassDiagram\_Operation \mid \\ &\qquad \exists c : ClassDiagram\_Class \cdot (has\_WMC(c, self) \wedge has\_Operation(c, m))\}\end{aligned} \tag{1}$$

where predicates $has\_WMC(x, y)$ and $has\_Operation(u, v)$ express that the class x has y of the class WMC and that the class u has the operation v. These predicates are from the associations on the meta model of Class Diagram as shown in Figure 1 (a). ClassDiagram_Operation refers to the class Operation included in the meta model ClassDiagram and denotes a set of instances of the Operation. # P denotes the cardinality of the set P, and *self* represents an instance of the class WMC in a context of this formula, i.e. the instance holding the *WMC_value* in the formula. WMC and the other CK metrics are not for a class diagram but for a class. For simplicity, we omit cardinality information from Figure 1 (a), and the cardinality of *has_WMC* is one-to-one. Thus we use the maximum number of WMC values in the class diagram or the average value to represent the WMC for the class diagram. In this example, which is used throughout the paper, we take the sum total of WMCs for the class diagram, and the attribute TNMvalue of StructuralComplexity holds it as shown in Figure 1 (b). We can define it as follows;

$$StructuralComplexity\_TNMvalue = \sum_{x:WMC} x.value \tag{2}$$

3 Using Domain Ontologies

Ontology technologies are frequently applied to many problem domains nowadays [9,23]. As mentioned in section 1, an ontology plays a role of a semantic domain.

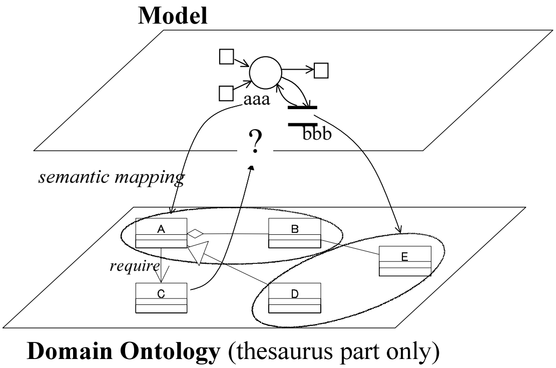


Fig. 2. Mapping from a Model to an Ontology

Basically, our ontology is represented in a directed typed graph where a node and an arc represent a concept and a relationship (precisely, an instance of a relationship) between two concepts, respectively.

Let's consider how a model engineer uses a domain ontology to measure the semantic characteristics of his or her models. During developing the model, the engineer should map its model element into atomic concepts of the ontology as shown in Figure 2. In the figure, the engineer develops a data flow diagram, while the domain ontology is written in the form of class diagrams. For example, the element "aaa" in the data flow diagram is mapped into the concepts A, B and the relationship between them. Formally, the engineer specifies a semantic mapping where $semantic\_mapping$(aaa) = {A, B, a relationship between A and B}. In the figure, although the model includes the concept A, it does not have the concept C, which is required by A on the domain ontology. Thus we can conclude that this model is incomplete because a necessary element, i.e. the concept C is lacking, and we can have the metrics of completeness (COMPL) by calculating the ratio of the lacking elements to the model elements, i.e.

$$COMPL\_value = \\ 1 - \#Lacking\_elements/(\#ModelElement + \#Lacking\_elements) \qquad (3)$$

where

$$Lacking\_elements = \\ \{u : Thesaurus \mid \exists c1 : ModelElement\ \exists e : Thesaurus \cdot \\ (semantic\_mapping(c1, e) \wedge require(e, u) \wedge \\ \neg\exists c2 : ModelElement \cdot semantic\_mapping(c2, u)\}$$

The meta model combined with this semantic metrics definition can be illustrated in Figure 3 in the same way as the syntactical metrics of Figure 1. The right part of the figure is the meta model of the thesaurus part of domain ontologies, i.e. logical structure of domain specific thesauruses. A thesaurus consists of concepts and relationships among the concepts, and it has a variety of subclasses of the "concept" class and "relationship". In the figure, "object" is a subclass of a concept class and a relationship

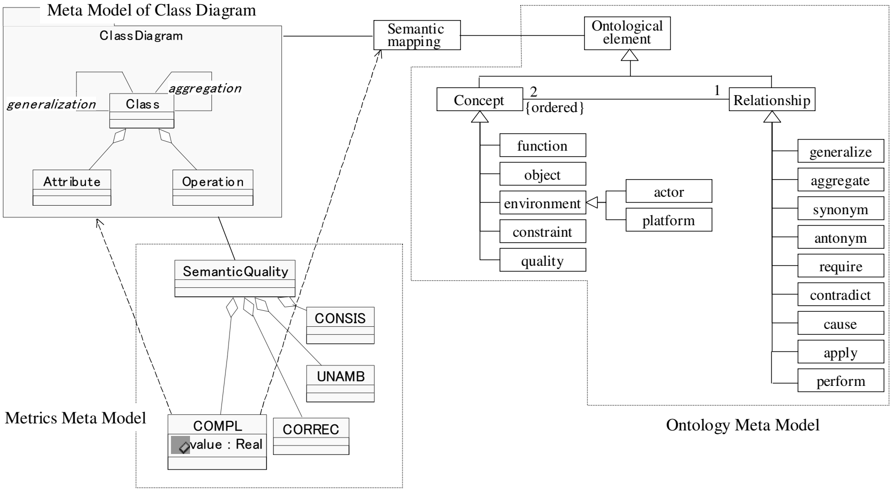


Fig. 3. Combining an Ontology Meta Model to a Meta Model

“apply” can connect two concepts. Concepts and relationships in Figure 3 are introduced so as to easily represent the semantics in models of the information systems to be developed. Intuitively speaking, the concepts “object”, “function”, “environment” and their subclasses are used to represent functional aspects of the models. On the other hand, the concepts “constraint” and “quality” are used to represent non-functional aspects. Semantic mapping plays a role of the bridges between the models written in class diagram and a domain thesaurus, and the model engineer provides the semantic mapping during his or her development of the model. In the figure, as the examples of semantic metrics, there are four metrics completeness (COMPL), consistency (CONSIS), correctness (CORREC) and unambiguity (UNAMB), which resulted from [2]. Their values are calculated from the model, the thesaurus and the semantic mapping, and the attribute “value” of the metrics holds the calculation result. Similar to Figure 1 and the formula (1), the calculation formulas are defined as constraints and the example of $COMPL\_value$ (the attribute “value” in COMPL) was shown in the formula (3).

Figure 4 shows a part of an ontology of Lift Control Systems, and we use class diagram notation to represent the ontology. Stereo types attached to class boxes and associations show their types. For example, “Open” belongs to a “function” concept of Figure 3. An association between “Open” and “Door” is an “apply” relationship, which presents that an object “Door” participates in the function “Open”. In this example, we have 11 model elements in the class diagram of Lift Control System and 2 elements (Close and Stop) to be required are lacking there. Because, in the thesaurus, the functions Open and Move requires Close and Stop respectively. As a result, we can get the completeness metrics (COMPL) $1 - (2/(11 + 2)) = 0.85$. As for the other semantic metrics such as consistency, correctness and unambiguity, their calculation methods were discussed in [10].

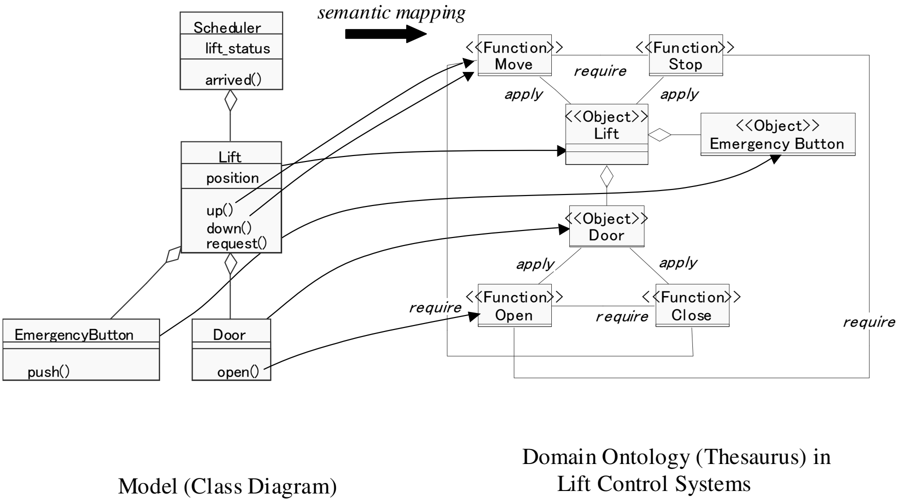


Fig. 4. An Example of a Lift Control System

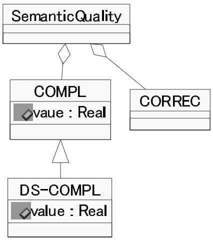


Fig. 5. Domain Specific Metrics

The calculation formula of this example is general because we calculate the ratio on how many required concepts are really included in the model. On the other hand, we sometimes need to define the metrics whose calculation formulas have the domain-specific properties, and these metrics can be defined as sub classes of the general metrics class. In the example of Lift Control System domain, we can consider that the quality of the model having no emergency buttons is low from the viewpoint of completeness. As shown in Figure 5, we set the sub class DS-COMPL of COMPL and specify a new calculation formula for the domain-specific completeness value as follows.

$$\begin{aligned} &DSCOMPL\_value = super.value \times \\ &\quad (1 + \exists c : ModelElement \cdot semantic\_mapping(c, EmergencyButton))/2 \end{aligned} \tag{4}$$

It uses the completeness value of the super class COMPL ($super.value$). If no emergency buttons are included, the completeness value is $super.value \times (1 + 0)/2$, i.e. a half of the previous definition shown in the formula (3).

4 Metrics of Model Transformation

4.1 Graph Rewriting System

In Model Driven Development, one of the technically essential points is model transformation. Since we use a class diagram to represent a meta model, a model, i.e. an instance of the meta model can be considered as a graph, whose nodes have types and attributes, and whose edges have types, so called attributed typed graph. Thus in this paper, model transformation is defined as a graph rewriting system, and graph rewriting rules dominate allowable transformations. A graph rewriting system converts a graph into another graph or a set of graphs following pre-defined rewriting rules. There are several graph rewriting systems such as PROGRESS [21] and AGG [22]. Since we should deal with the attribute values attached to nodes in a graph, we adopt the definition of the AGG system in this paper.

A graph consists of nodes and edges, and type names can be associated with them. Nodes can have attribute values depending on their type. The upper part of Figure 6 is a simple example of rewriting rules. A rule consists of a left-hand and a right-hand side which are separated with "::=". The attribute values should be able to be propagated in any direction, i.e. from the left-hand side of "::=" to the right-hand side, the opposite direction, as well as within the same side, and this mechanism is similar to synthesized and inherited attributes of Attribute Grammar. In this sense, the graph rewriting system that we use is an extended version of AGG.

In figure 6, a rectangle box stands for a node of a graph and it is separated into two parts with a horizontal bar. The type name of a node appears in the upper part of the horizontal bar, while the lower part contains its attribute values. In the figure, the node of "TypeA" in the left-hand graph has the attribute "val" and its value is represented with the variable "x". A graph labeled with NAC (Negative Application Condition) appearing in the left-hand controls the application of the rule. If a graph includes the NAC graph, the rule cannot be applied to it. In addition, we add the conditions that are to be satisfied when the rule is applied. In this example, we have two conditions, one of which says that "val" of the node "1:TypeA" has to be greater than 4 to apply this rewriting rule.

The lower part of Figure 6 illustrates graph rewriting. The part encircled with a dotted rectangular box in the left-hand is replaced with the sub graph that is derived from the right-hand of the rule. The attribute values 5 and 2 are assigned to x and y respectively, and those of the two instance nodes of "TypeD" result in 7 (x+y) and 3 (x-y). The attribute "val" of "TypeD" node looks like an inherited attribute of Attribute Grammar because its value is calculated from the attribute values of the left-hand side of the rule, while "sval" of "TypeC" can be considered as a synthesized attribute. The value of "sval" of the node "TypeC" in the left-hand side is calculated from the values in the right-hand side, and we get 8 (3:TypeD_val + x = 3+5). Note that the value of "val" of "TypeA" is 5, greater than 4, the value of "sval" is less than 10, and none of nodes typed with "TypeD" appear, so the rule is applicable. The other parts of the left-hand side graph are not changed in this rewriting process.

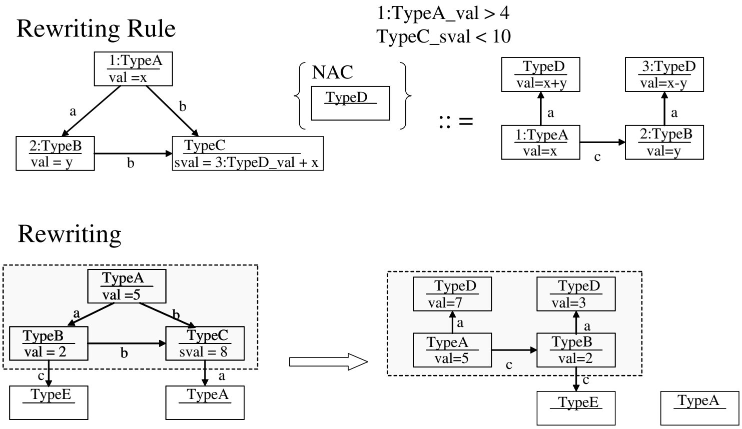
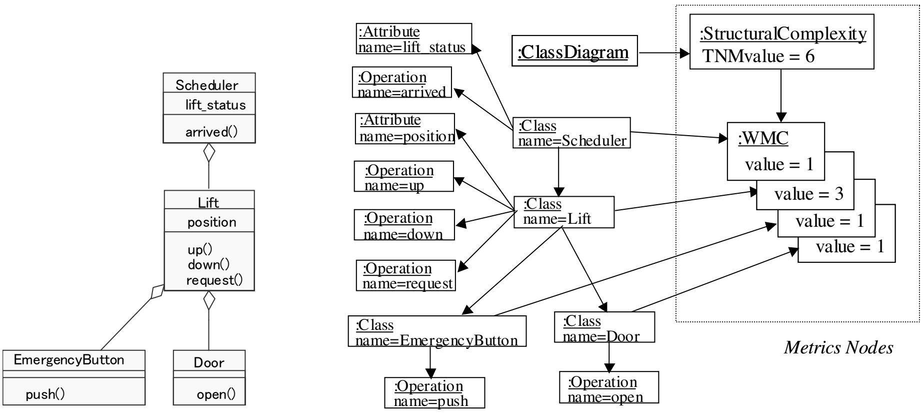


Fig. 6. Graph Rewriting Rules and Rewriting Process

4.2 Attaching Calculation Rules

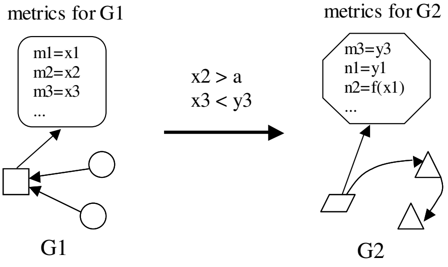
The model following its meta model is represented with an attributed typed graph and it can be transformed by applying the rewriting rules. We call this graph *instance graph* in the sense that the graph is an instance of the meta model. Figure 7 shows the example of a class diagram of Lift Control System and its instance graph following the meta model of Figure 1. The types of nodes result from the elements of the meta model such as Class, Attribute and Operation, while the names of classes, attributes and operations are specified as the values of the attribute "name". In the figure, the class Lift in the class diagram corresponds to the node typed with Class and whose attribute "name" is Lift. Some nodes in the instance graph have metric values as their attribute values. For example, a node typed with WMC has the attribute "value" and its value is the number of the operations of the class, which is calculated using the formula (1). The WMC value of class Lift is 3 as shown in the figure.

We can design graph rewriting rules considering the nodes of the metrics and their values. See an example of a transformation rule shown in Figure 8. Two conditions $x2 > a$ and $x3 < y3$ are attached to the rule for rewriting the graph G1 with G2 and these conditions should be satisfied before the rule is applied. This transformation rule includes two nodes named "metrics for G1" and "metrics for G2", each of which holds the metric values of the model. The first condition $x2 > a$ expresses that the rule cannot be applied until the value of the metric m2 before the rewriting is greater than a certain value, i.e. "a". It means that this model transformation is possible when the model has a metric value higher than a certain standard. The second condition $x3 < y3$ specifies monotonic increasing of the metric m3 in this transformation. This formula has both metric values before and after the transformation as parameters and it can specify the characteristics of the transformation, e.g. a specific metric value is increasing by the transformation. As shown in the figure, the calculation of the metric n2 uses the metric m1 of the model before the transformation, and this calculation formula of n2 shows that



(a) Class Diagram (b) Instance Graph with Metrics Nodes

Fig. 7. Class Diagram and Its Instance Graph



a metric of the transformation: g(x1,x2,x3,..., y1,f(x1),y3,...)

Fig. 8. Metrics and Model Transformation

the metric value of G1 is propagated to G2. The metrics of a transformation process can be formally specified by using this approach. In Figure 8, we can calculate how much a metric value could be improved with the transformation by using the metric values of the model before the transformation and those after the transformation. The function g in the figure calculates the improvement degree of the metric value. This is a basic idea of the metrics of model transformations.

Let's consider the example of a model transformation using graph rewriting rules. The model of Lift Control System in Figure 7 (a) can be considered as a platform independent model (PIM) because of no consideration of implementation situation, and we illustrate its transformation into a platform dependent model (PSM). We have a scheduler to decide which lift should be made to come to the passengers by the information of the current status of the lifts (the position and the moving direction of the lift), but we don't explicitly specify the concrete technique to implement the function of getting the status information from the lifts. If the platform that we will use has an interrupt-handling mechanism to detect the arrival of a lift at a floor, we put a new operation "notify" to catch the interruption signal in the Lift module. The notify operation calls the operation "arrived" of Scheduler and the "arrived" updates the lift_status attribute according to the data carried by the interrupt signal. As a result, we can get a PSM that can operate under the platform having interrupt-handling functions. In Figure 9, Rule #1 is for this transformation and PSM#1 is the result of applying this rule to the PIM of Lift Control System.

Another alternative is for the platform without any interrupt-handling mechanism, and in this platform, we use multiple instances of a polling routine to get the current lift status from each lift. The class Thread is an implementation of the polling routine and its instances are concurrently executed so as to monitor the status of their assigned lifts. To execute a thread object, we add the operations "start" for starting the execution of the thread and "run" for defining the body of the thread. The operation "attach" in Scheduler is for combining a scheduler object to the thread objects. Rule #2 and PSM#2 in Figure 9 specifies this transformation and its result respectively. The TNMvalue, the total sum of the operations, can be calculated following the definition of Figure 1 for PIM, PSM#1 and PSM#2. It can be considered that the TNM value expresses the efforts to implement the PSM because it reflects the volume of the source codes to be implemented. Thus the difference of the TNMvalues ($\Delta TNMvalue$) between the PIM to the PSM represents the increase of implementation efforts. In this example, PSM#1 is easier to implement because $\Delta TNMvalue$ of PSM#1 is smaller than that of PSM#2, as shown in Figure 9. So we can conclude that the transformation Rule #1 is better rather than Rule #2, only from the viewpoint of less implementation efforts. This example suggests that our framework can specify formally the metrics of model transformations by using the metric values before and after the transformations.

5 Conclusion and Future Work

In this paper, we propose three techniques to define metrics in MDD context; 1) the application of a meta modeling technique to specify model-specific metrics, 2) the definition of metrics dealing with semantic aspects of models (semantic metrics) using domain ontologies and 3) the specification technique for metrics of model transformations based on graph rewriting systems.

In addition, the future research agenda can be listed up as follows.

1) **Metrics on graph rewriting.** The metrics mentioned in the previous section was based on model-specific metrics and value changes during model transformations. We can consider another type of metrics based on characteristics of graph rewriting rules. For example, the fewer graph rewriting rules that implement the model transformation may lead the more understandable transformation, and the complexity of the rules can be used as the measure of understandability of the transformation. This kind of complexity such as the number of rules and the number of nodes and edges included in the rules can be calculated directly from the rules. Another example is related to the process of executing graph rewriting. During a transformation process, we can calculate the number of the application of rules to get a final model, and this measure can express

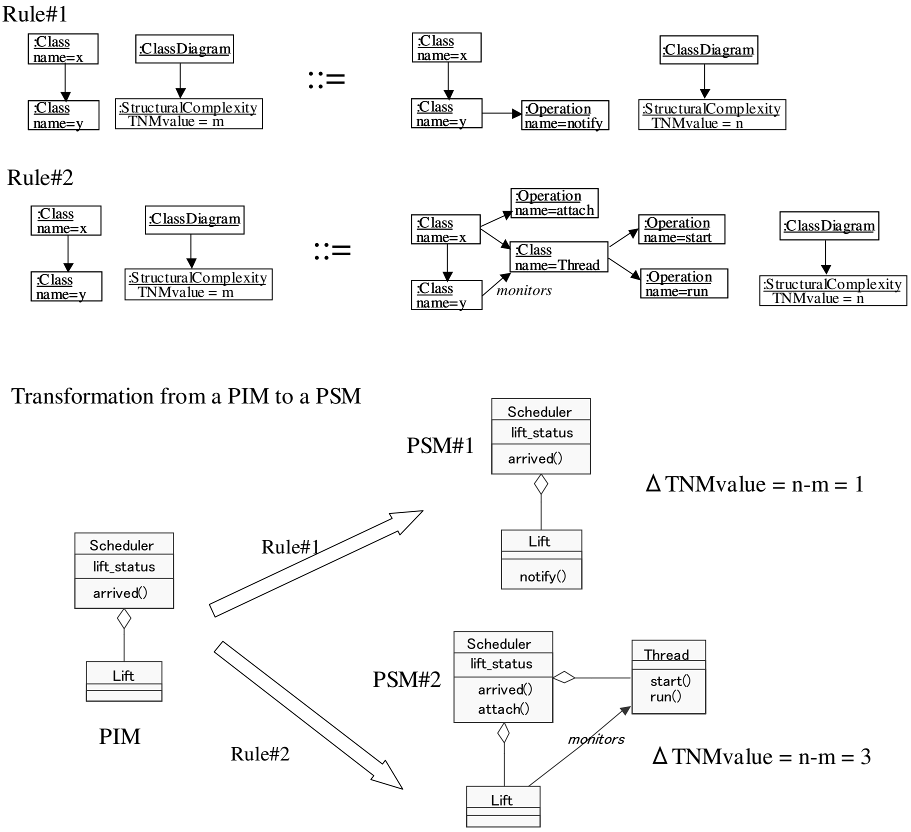


Fig. 9. Model Transformation Example

efficiency of the model transformation. The smaller the number is the more efficient. This type of metrics is apparently different from the above metrics on the complexity of rules, in the sense that it is the metrics related to the actual execution processes of transformation. We call the former type of the metrics *static metrics* and the latter *dynamic metrics*. As mentioned above, our approach has potentials for defining wide varieties of model transformation metrics.

2) Development of supporting tools. We consider the extension of the existing AGG system, but to support the calculation of the metrics of transformations and the selection of suitable transformations, we need more powerful evaluation mechanisms of attribute values. The mechanisms for version control of models and re-doing transformations are also necessary to make the tool practical.

3) Usage of standards. For simplicity, we used class diagrams to represent meta models and predicate logic to define metrics. To increase the portability of meta models and metrics definitions, we will adapt our technique to standard techniques that OMG proposed or is proposing, i.e. MOF, XMI, OCL (Object Constraint Language), QVT and Software Metrics Metamodel [18]. The technique of using OCL on a meta model to specify metrics was also discussed in [3,20], and the various metrics for UML class diagrams can be found in [8].

4) Collecting useful definitions of metrics. In this paper, we illustrated very simple metrics for explanation of our approach. Although the aim of this research project is not to find and collect useful and effective metrics as many as possible, making a kind of catalogue of metric definitions and specifications like [7,13] is important in the next step of the supporting tool. The assessment of the collected metrics is also a research agenda.

5) Constructing domain ontologies. The quality of a domain ontology greatly depends on the preciseness of the semantic metrics, and we should get a domain ontology for each problem and application domain. In fact, developing various kind of domain ontologies of high quality by hand is a time-consuming and difficult task. Adopting text mining approaches are one of the promising ones to support the development of domain ontologies [1,11].

References

1. KAON Tool Suite, `http://kaon.semanticweb.org/`
2. IEEE Recommended Practice for Software Requirements Specifications. Technical report, IEEE Std. 830-1998 (1998)
3. Abreu, F.B.: Using OCL to Formalize Object Oriented Metrics Definitions. In: Tutorial in 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, QAOOSE 2001 (2001)
4. Bézivin, J., Rumpe, B., Schur, A., Tratt, L.: Model Transformations in Practice Workshop. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 120–127. Springer, Heidelberg (2006)
5. Chidamber, S., Kemerer, C.: A Metrics Suite for Object-Oriented Design. IEEE Trans. on Software Engineering 20(6), 476–492 (1994)
6. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture (2003)
7. Ebert, C., Dumke, R., Bundschuh, M., Schmietendorf, A.: Best Practices in Software Measurement. Springer, Heidelberg (2005)
8. Genero, M., Piattini, M., Calero, C.: A Survey of Metrics for UML Class Diagrams. Journal of Object Technology 4(9), 59–92 (2005)
9. Gruninger, M., Lee, J.: Ontology: Applications and Design. Commun. ACM 45(2) (2002)
10. Kaiya, H., Saeki, M.: Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach. In: Proc. of QSIC, pp. 223–230 (2005)
11. Kitamura, M., Hasegawa, R., Kaiya, H., Saeki, M.: An Integrated Tool for Supporting Ontology Driven Requirements Elicitation. In: Proc. of 2nd International Conference on Software and Data Technologies (ICSOFT 2007), pp. 73–80 (2007)
12. Kleppe, A., Warmer, J., Bast, W.: MDA Explained. Addison-Wesley, Reading (2003)
13. Lorenz, M., Kidd, J.: Object-Oriented Software Metrics. Prentice-Hall, Englewood Cliffs (1994)

14. Maedche, A.: Ontology Learning for the Semantic Web. Kluwer Academic Publishers, Dordrecht (2002)
15. McCabe, T., Butler, C.: Design Complexity Measurement and Testing. CACM 32(12), 1415–1425 (1989)
16. Mellor, S., Balcer, M.: Executable UML. Addison-Wesley, Reading (2003)
17. OMG. MDA Guide Version 1.0.1. (2003), `http://www.omg.org/mda/`
18. OMG. ADM Software Metrics Metamodel RFP (2006), `http://www.omg.org/docs/admtf/06-09-03.doc`
19. Saeki, M.: Role of Model Transformation in Method Engineering. In: Pidduck, A.B., Mylopoulos, J., Woo, C.C., Ozsu, M.T. (eds.) CAiSE 2002. LNCS, vol. 2348, pp. 626–642. Springer, Heidelberg (2002)
20. Saeki, M.: Embedding Metrics into Information Systems Development Methods: An Application of Method Engineering Technique. In: Eder, J., Missikoff, M. (eds.) CAiSE 2003. LNCS, vol. 2681, pp. 374–389. Springer, Heidelberg (2003)
21. Schurr, A.: Developing Graphical (Software Engineering) Tools with PROGRES. In: Proc. of 19th International Conference on Software Engineering (ICSE 1997), pp. 618–619 (1997)
22. Taentzer, G., Runge, O., Melamed, B., Rudorf, M., Schultzke, T., Gruner, S.: AGG: The Attributed Graph Grammar System (2001), `http://tfs.cs.tu-berlin.de/agg/`
23. Wand, Y.: Ontology as a Foundation for Meta-Modelling and Method Engineering. Information and Software Technology 38(4), 281–288 (1996)

On-the-Fly Testing by Using an Executable TTCN-3 Markov Chain Usage Model

Winfried Dulz

Department of Computer Science, University of Erlangen-Nuremberg
Martensstr. 3, D-91058 Erlangen, Germany
dulz@cs.fau.de
http://www7.informatik.uni-erlangen.de/~dulz/

Abstract. The TestUS framework offers a range of techniques to obtain a TTCN-3 test suite starting from UML 2.0 requirement definitions. Use case diagrams that contain functional and non-functional requirements are first transformed to a Markov Chain usage model (MCUM). Probability annotations of MCUM state transitions enable the generation of TTCN-3 test cases that reflect the expected usage patterns of system users. Because compiling the associated TTCN-3 test suite can take quite a long time for a realistic SUT (System under Test) we decided to map the MCUM directly into the executable test suite without generating test cases in advance. Test cases and the evaluation of test verdicts are interpreted on-the-fly during executing the test suite. We proved the concept by testing an existing DECT communication system. The compilation time for deriving an executable TTCN-3 test suite was reduced to only 15 minutes and one can interpret as many test cases as one likes on-the-fly.

Keywords: Statistical testing, Automatic test suite generation, Markov Chain usage model, UML 2.0, TTCN-3.

1 Introduction

Model-based development techniques are getting more and more attractive in order to master the inherent complexity of real-world applications. Different models are used for all kind of purposes during the system development cycle and handle static and dynamic aspects of the future system. The latest UML standard [10] will strongly influence more and more areas of software engineering, covering application domains that are also vulnerable for non-functional QoS (quality of service) errors, e.g. real-time or performance errors.

Model-based testing in general is a widespread research topic since many years, [4] give a good review concerning current activities. Examples covering automation tools are contained in [13]. There exist papers on usage models [12] and [14], which mainly focus on model generation and evaluation based on textual descriptions of the usage behavior. In [1] statistical test case generation based on a MCUM that is derived from UML sequence diagram scenarios is discussed. [1] and [2] also explain how to integrate QoS and performance issues in the test process.

In the next section, we will first discuss testing techniques in general that have influenced our method, i.e. black- box testing with TTCN-3 and the statistical usage testing

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 17–30, 2010.

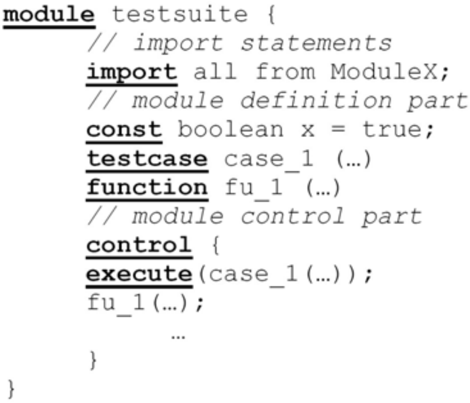
technique. In section 3, our model-based test case generation approach is described in detail. Next, we present the main results of a case study for testing DECT modules and finally we summarize with a conclusion and some final remarks.

2 Testing Concepts

2.1 TTCN-3

TTCN-3 is the most recent version of the well established test notation language TTCN, standardized by ETSI ([5]). It is a universal language for test management and test specification, valid for any application domain, such as protocol, service or module testing. TTCN-3 is suitable for different kinds of testing approaches, e.g. conformance, robustness, interoperability, regression, system or acceptance tests.

Modules are the top-level elements for structuring elements and consist of an optional *import* section, an optional *definition* part and the *control* part. The main functionality of the *test suite* is defined within the *test case* definition statements, where specific responses of the SUT are related to TTCN-3 test verdicts. Inside the control part section the sequential order of the execute statements and the function calls represents the precise test runs of an executable test suite. An example for the definition of a TTCN-3 test suite is given on the left hand side.



```
module testsuite {
     // import statements
     import all from ModuleX;
     // module definition part
     const boolean x = true;
     testcase case_1 (…)
     function fu_1 (…)
     // module control part
     control {
     execute(case_1(…));
     fu_1(…);
          …
     }
}
```

After compiling the TTCN-3 modules an executable or interpretable test suite is provided by the TE (TTCN-3 Executable) element in Fig. 1. Further entities have to be supplied, which are necessary to make the abstract concepts concrete and executable. By means of the TCI (TTCN-3 Control Interface) the test execution can be influenced with respect to test management and test logging (TM). Test component handling for distributed testing (CH) and encoder/decoder functions for different representations of TTCN-3 data types (CD) may also be provided. The TRI (TTCN-3 Runtime Interface) was defined to enable the interactions between the SUT and the test system via a standardized interface.

In Fig. 1 two parts of the TRI are visible: the description of the communication system is specified in the SA (SUT Adapter) and the PA (Platform Adapter) implements timers and external functions based on the underlying operating system.

2.2 Statistical Usage Testing

MBT (Model-based Testing) techniques make use of formal descriptions (models) of either the *SUT* (System under Test) or the expected usage of the users of the SUT. In the former case a behavioral specification is derived from the requirement definitions and

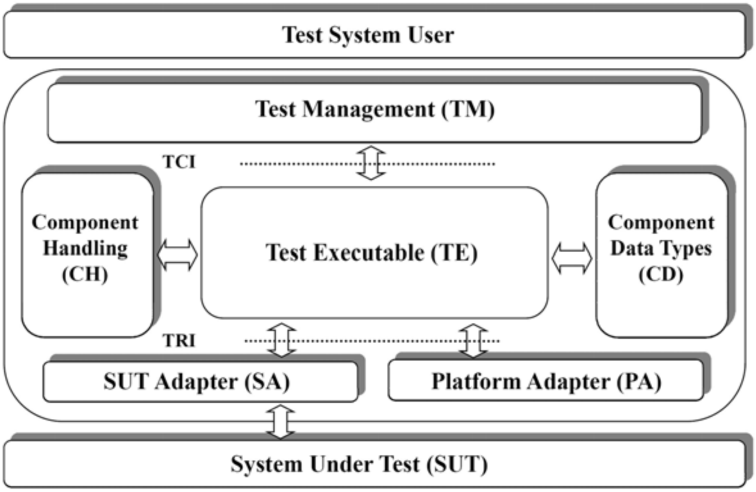


Fig. 1. Building blocks of a TTCN-3 test system

serves as a starting basis to automatically generate test cases in order to test the SUT [11]. In the latter case *usage models* are deduced from the requirements and may be considered as independent of the specification. Because exhaustive testing of real systems is infeasible in practice an appropriate set of test cases is derived for accomplishing a given test goal. At this point Markovian statistics come into play.

Statistical testing relies on MCUMs [14] that characterize the estimated distribution of possible uses of a given software in its intended environment. A Markov chain consists of states and transitions between states [16]. This directed graph structure describes all possible *usage scenarios* for a given SUT derived by randomly traversing the arcs between dedicated start and end states. Transitions are augmented by probabilities from a *usage profile* that reflects usage choices users will have when they interact with the SUT.

Providing more than one usage profile for a given Markov chain structure is also possible. Hereby, the statistical selection of test cases reflects distinct properties of different roles or user classes, e.g. experts or normal users, in interaction with the SUT.

How to build and to integrate the MCUM approach into a UML based development process is explained in the next section.

3 Model-Based Testing

3.1 The TestUS Framework

The test case generation process, as shown in Fig. 2, starts with a UML use case diagram at the top of the diagram. Ovals inside the use cases characterize the usage behavior that is refined by scenario descriptions in form of sequence diagrams. In combination with a usage profile the MCUM is automatically derived by the procedure explained in section 3.3. This model is the base for the automatic generating of the TTCN-3 test suite, as

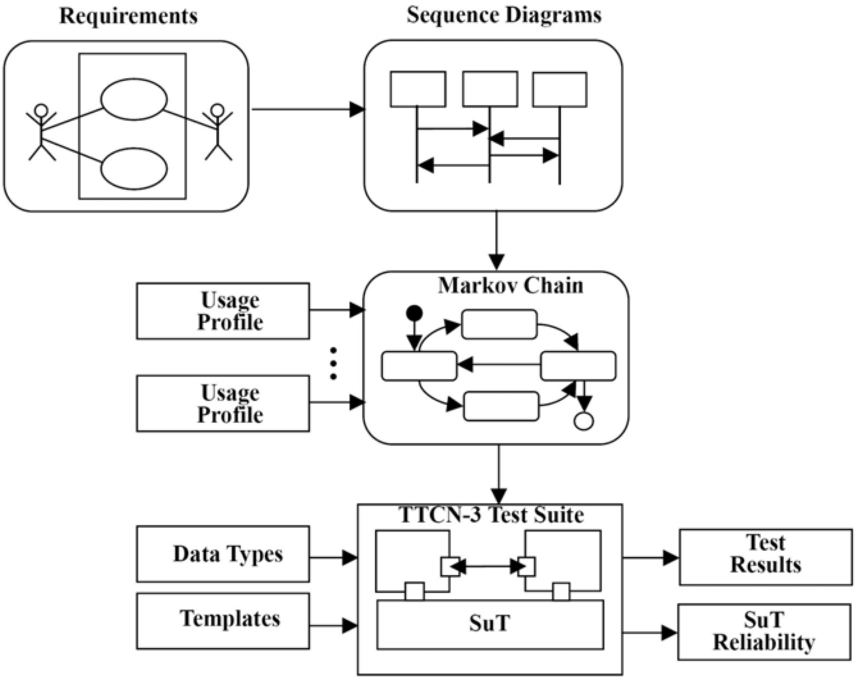


Fig. 2. TestUS framework for a model-based TTCN-3 test suite generation starting from use case scenarios

explained in more details in section 4. After adding additional data types and template definitions for the TTCN-3 test suite compilation, an executable test suite is generated. The evaluation of test verdicts during the test enables the calculation of test results, e.g. coverage of states and transitions and the reliability metric at the end.

3.2 Scenario-Based Requirements

A development process starts with the *requirements* phase. The task is to identify possible use cases and to illustrate the sequence of desired operations in some way. This is covered in the UML by *static* use case diagrams and by *dynamic* diagrams such as activity, state chart and interaction diagrams.

Most common are requirement descriptions in form of *sequence diagrams*. In addition to the characterization by means of simple message interactions, *state invariants* are included to distinguish certain special situations during a user interaction with the system.

For instance after receiving a `Connection_Setup_Confirm` message the user knows that he has a valid connection to the system, which may be reflected in a *connected* state invariant, as shown in Fig. 3. To denote QoS (quality of service) requirements special annotations may be attached to sequence diagrams that are conform to the *UML SPT Profile* (schedulability, performance and time).

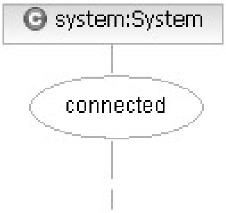


Fig. 3. User provided state invariant

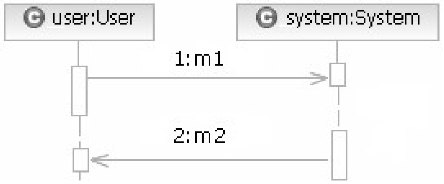


Fig. 4. Trigger message and the system response represented by a sequence diagram

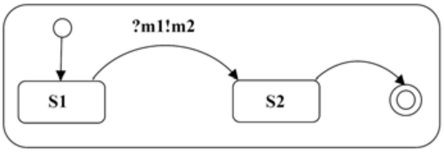


Fig. 5. Trigger message and the system response represented by a MCUM

3.3 Deriving the MCUM Test Model

Providing a set of scenario descriptions as output from the requirement definitions the test model, i.e. the MCUM can be automatically generated. UML protocol state machines are adequate for representing this kind of model. Each sequence diagram contains one lifeline for the SUT; each additional *lifeline* corresponds to a possible user of the system.

Combined fragments in the sequence diagrams are used to specify special situations during the user interactions and state information can be added to define state invariants in the diagrams. The following diagrams ([7] will illustrate the main transformation rules to obtain the protocol state machine from a given set of sequence diagrams:

* In a first step *supplementary* state invariants are added. Apart from user provided state invariants, additional state information is needed to have at most two messages in invariants, additional state information is needed to have at most two messages in between any two states, i.e. a *sending* message m1 and its corresponding *receiving* message m2 reflecting the system response as shown in Fig. 4. We denote by ?m1, respectively by !m2 the trigger message, respectively the system response in the corresponding transition 's1 ?m1!m2 s2' of the generated MCUM as shown in Fig. 5.
* In any other case, each single message, i.e. a trigger message without a direct system response or a spontaneous system response without a previous trigger message, should be enclosed by two states. Whereas sequence diagrams represent a *partial order semantic* by default, the exchange of messages is now *strictly* ordered.
* If M(s) is a MCUM for the sequence $s = s_{11} \cdots s_{1n}$, M(t) is a MCUM for the sequence $t = s_{21} \cdots s_{2m}$ and $s_{1n} = s_{21}$, we generate for the concatenation expression 's t' as shown in Fig. 6.

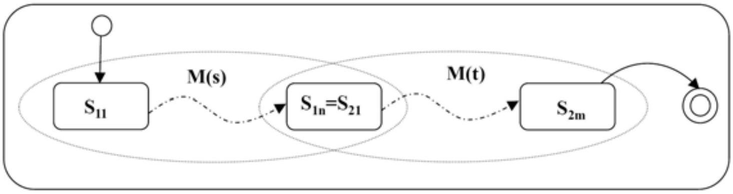


Fig. 6. MCUM resulting from concatenating two message sequences

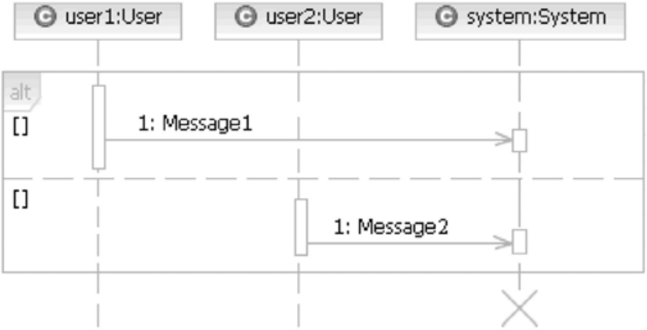


Fig. 7. Sequence diagram containing an alt fragment

For all combined fragments, a *composite state* is generated and a new state machine is added to the MCUM for the included sequence. In more detail the following transformations are considered:

* For the *conditional* fragment (Fig. 7) that represents two alternative user interactions with the system we generate the corresponding MCUM composite state in Fig. 8. In addition, three new supplementary state invariants are automatically generated inside the composite state in order to separate trigger messages and the system's response.
* In general, if M(s) is a MCUM derived from the sequence $s = s_{11} \cdots s_{1n}$ and M(t) is a MCUM derived from the sequence $t = s_{21} \cdots s_{2m}$ we will generate a MCUM composite state for the conditional fragment as shown in Fig. 9.

  This transformation enables the generation of test cases that either contain trigger messages and corresponding system responses from s or from t. If we add transition probabilities from the usage profile to the outgoing transitions of state s_0 it is possible to test alternative user behavior that also reflects the expected usage statistics and not only the correct order of possible user interactions with the SUT.
* For the *loop* fragment that iterates over the sub chain M(s) containing the sequence $s = s_{11} \cdots s_{1n}$ we generate the composite state represented in Fig. 10.

  In this situation, we can generate test cases that contain the sequence s arbitrarily often (including also the Zero case). In general, we are also able to create a MCUM

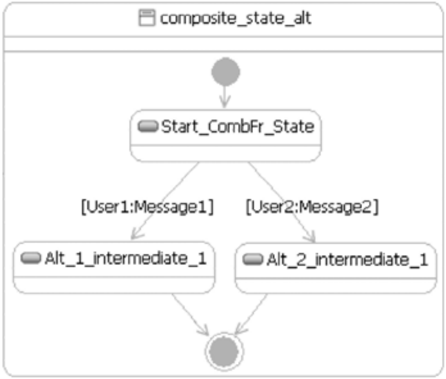


Fig. 8. TMCUM composite state for an alt fragment

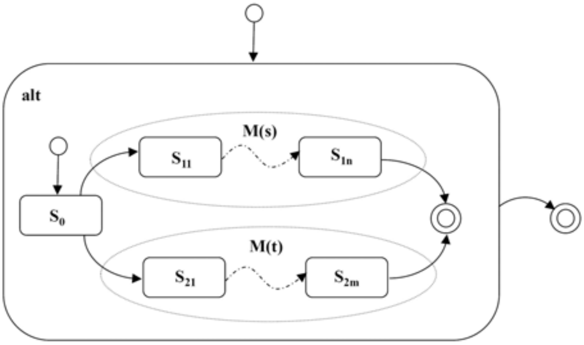


Fig. 9. MCUM composite state resulting from an alt fragment concatenating two message sequences

composite state from loop fragments that contain upper and lower boundaries to express finite loop conditions.

* If M(s) is a MCUM derived from the sequence $s = s_{11} \cdots s_{1n}$ and M(t) is a MCUM derived from the sequence $t = s_{21} \cdots s_{2m}$ we generate the MCUM composite state shown in Fig. 11 for the *parallel* fragment `s par t`.

Here, events of M(s) and M(t) may be arbitrarily interleaved. The main condition is that the test case has to reflect the correct order of events inside the parallel executable sequences s and t.

* Beside the presented combined fragments `alt`, `loop` and `par` we have also considered `opt` for options, `neg` for invalid behavior, `assert` for assertions, `break` for break conditions, `strict` for strict sequencing, `critical` for critical sections and included the necessary MCUM transformation rules in the TestUS framework.

* After having generated the structure of the MCUM it is necessary to attach usage profile probability information to the MCUM transitions in order to model a test characteristic that is as close as possible to the future usage behavior of the SUT.

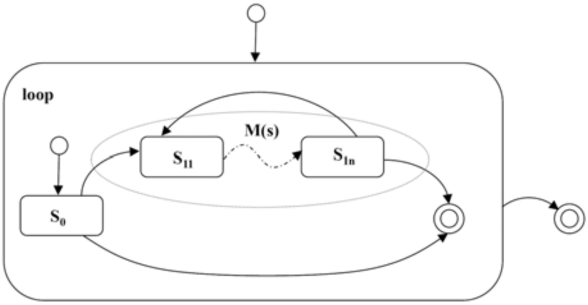


Fig. 10. MCUM composite state resulting from a loop fragment concatenating a message sequence

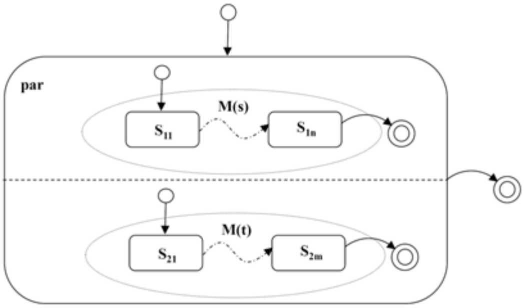


Fig. 11. MCUM composite state resulting from a par fragment concatenating two message sequences

In [15], [9] and [6] proper strategies to derive valid probabilities for the usage profiles are discussed.

* In the last step of the transformation process all final states of the generated MCUM segments are merged to *one* final state. In addition, a new initial state is included and connected to the initial states of otherwise isolated MCUM segments. Finally, equally named user provided state invariants are combined and corresponding incoming (outgoing) transitions are united. The result is an automatic generated MCUM as starting point for generating the TTCN-3 test suite.

* As an example, the MCUM for testing the DECT system in the case study of section 5 resulted from about 230 usage scenarios and consists of about 900 states with over 3400 transitions.

4 Test Suite Generation

4.1 Arguments for Avoiding the Explicit Generation of Test Cases

A test case is any valid path in the MCUM consisting of single test steps that starts from the initial state and reaches the final state resulting either in a `PASS` or a `FAIL` test verdict.

In the previous approach discussed in [2], abstract test cases were generated from the derived MCUM in an intermediate step. To achieve this objective the XMI representation of the UML protocol state machine for the MCUM was processed by means of *XSLT* (Extensible Stylesheet Language Transformation) technology. We have chosen TTCN-3 from ETSI ([5]) because we determined a good tool support by a broad applicability and standardized interfaces both to the SUT as well as to the test management part. The transformation of *abstract* UML test cases to *concrete* TTCN-3 test cases is done automatically. The only manual part was to add missing data definitions and to provide an interface implementation for handling the communication with the SUT.

The main disadvantage of the previous approach is the need to generate test cases first in order to derive an executable test suite, as shown in fig. 12. The duration for generating and transforming a test case from a given MCUM is in the order of six minutes related to realistic applications. At the first glance this generation overhead seems not to be very serious.

On closer examination and especially looking at the big variance between one up to 24 hours for compiling an executable test suite for the DECT system we identified two major reasons for the inefficient TTCN-3 compilation:

- *unfolding* finite loop fragments with upper and/or lower boundaries for not violating the Markovian assumptions of the MCUM theory
- *serialization* of interleaved events inside composite states that are generate from parallel fragments will lead to a factorial growth of the length of test cases.

We also checked the intermediate code of the Telelogic Tau G2 TTCN-3 compiler used in our project with respect to the TTCN-3 interleave construct and noticed that the compiler maps it to a sequence of alt (choice) statements.

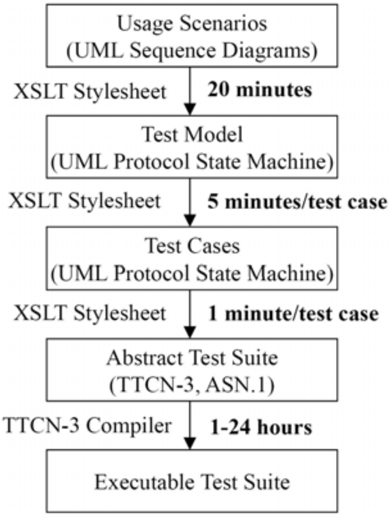


Fig. 12. Duration of the transformation and compilation steps to generate a TTCN-3 test suite for a DECT system

In addition, another drawback arises from the static definition of the test behavior after having compiled the executable TTCN-3 test suite. After finishing the test and estimating the quality of the SUT additional tests may be performed to further improve the reliability estimation. This is due to the fact that the accuracy of the confidence interval for the reliability depends on the number of executed test cases. In this situation new test cases have to be generated and another compilation phase has to be performed. The duration for this task may be in the order of hours, depending on the size of the randomly generated test cases and the resulting TTCN-3 test suite definition.

To avoid these disadvantages and to be more flexible concerning the test execution we decided to cancel the test case generation step and immediately mapped the MCUM protocol state machine into the TTCN-3 ([7]).

4.2 The Executable Markov Chain Usage Model Is the Test Suite

An executable TTCN-3 test suite consists of a set of *concurrent* test components, which perform the test run. There exists always one *MTC* (Master Test Component), created implicitly when a test suite starts. `PTCs` (Parallel Test Components) are generated dynamically on demand. In the TestUS framework the generated test configuration consists of the following parts:

* Every *actor* in the sequence diagrams is represented by one PTC that executes the specific behavior. PTCs are generated and started by the MTC at the beginning of an interpreted test case.
* Synchronization is a major task of the MTC. Synchronization messages are inserted in each of the following situations:
  - at the beginning of a test case right after the creation of a PTCs, a `sync` message is sent from every PTC to the MTC, signaling to be ready for start
  - after gathering these messages, the test component that is responsible for doing the next *test step* is sent a `sync_ack` message by the MTC
  - `syncall` messages are used to inform the MTC that a PTC has received a response from the SUT which is *piggy-back* encoded inside a `sync` message.
* Eventually, the MTC is responsible for logging every test step's verdict, i.e. the positive or negative result of a test step by using the piggy-back information from the PTCs.

Let us explain the main concept in a small example that illustrates the TTCN-3 interpretation of the simple MCUM in Fig. 13.

As discussed in the previous subsection no explicit test case generation is needed. Instead, each transition of the MCUM is considered to be a single test step. Parameters

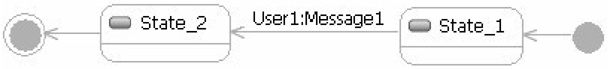
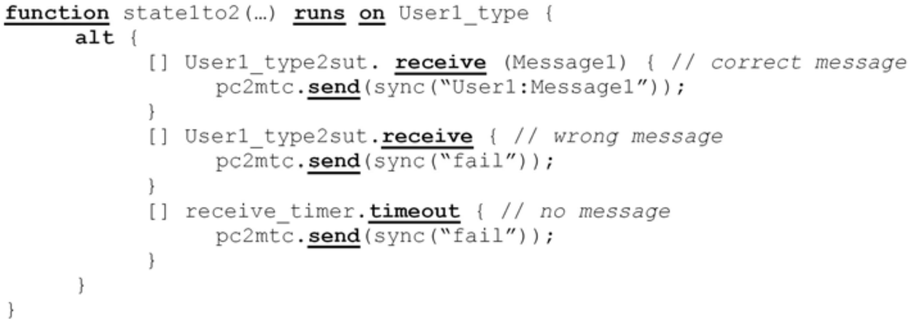


Fig. 13. Simple MCUM to demonstrate the communication between a PTC and the MTC in a TTCN-3 test executable

of the transition, i.e. trigger message, expected result and the associated probability are automatically mapped into a *behavior defining* function that allows the interpretation of the test step on-the-fly during the test execution.

The function name is directly derived from the names of the source and target states. The TTCN-3 keyword `runs on` is used to denote which PTC has to executed the function. Alternative reactions of the SUT are defined in the `alt` statement right after the pair of brackets []. The resulting sync message that is sent from the PTC to the MTC either contains the expected result or a *fail* information. Function `state1to2` below represents the MCUM transition in Fig. 13 from the PTC's point of view that has to handle the `User1` interactions with the SUT.



```
function state1to2(…) runs on User1_type {
      alt {
            [] User1_type2sut. receive (Message1) { // correct message
                  pc2mtc.send(sync("User1:Message1"));
            }
            [] User1_type2sut.receive { // wrong message
                  pc2mtc.send(sync("fail"));
            }
            [] receive_timer.timeout { // no message
                  pc2mtc.send(sync("fail"));
            }
      }
}
```

Below, the TTCN-3 control actions for the MTC to interpret the simple test case of the MCUM are shown. After the module definition part that contains the definition of function `state1to2` abstracted by ”′ · · · ”′ the first control action starts the PTC of User1. If the expected result is received from the PTC the MTC logs this event and the verdict `pass` is given. Otherwise the test results in the verdict `fail` and the `errorState` is reached.

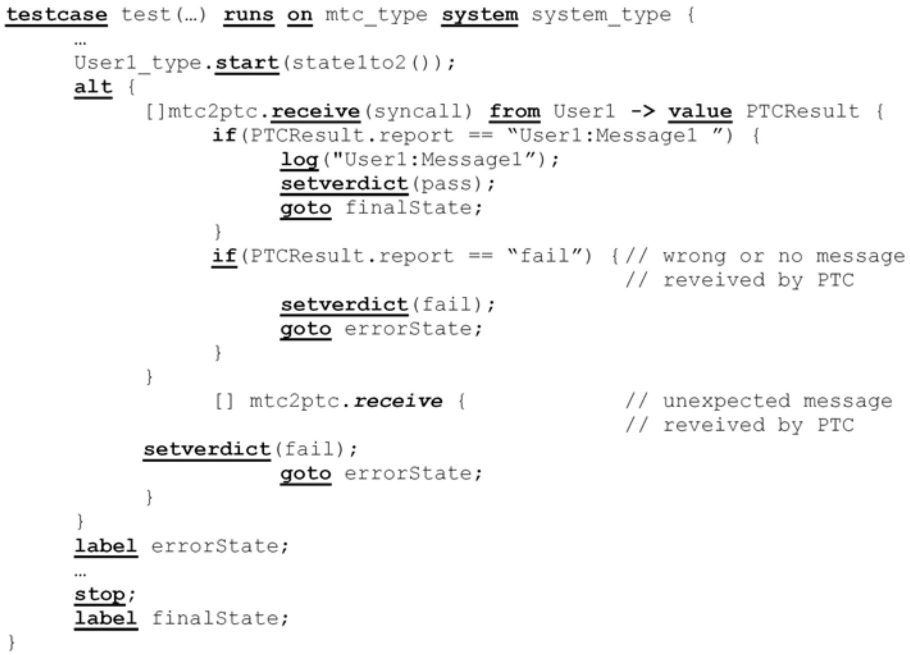
If there exists more than one possibility to leave a given state of the MCUM the MTC has to choose randomly the next transition based on the probability information of the leaving transitions that has to sum up to 1 for each state.

After logging the test verdict the MTC will select the next test case on-the-fly by continuing with the start state of the MCUM. At the end of the test typical statistics are calculated and presented to the test user, e.g. number of test cases, number of visited states and transitions, mean length of a test case and the reliability of the SUT.

5 A DECT Case Study

To validate the TestUS approach we have chosen the case study from Biegel ([3]) in order to compare the results. In [3], the main test goal was to demonstrate the correct intercommunication behavior of *DECT* protocol modules via the *DHCI* (DECT Host Controller Interface).

The configuration of the SUT is shown in Fig. 14. The DECT system consists of two base stations (FP: fixed part) and four portable parts (PP: portable part) that may be subscribed either to the first or the second FP. During the test the PPs are allowed



```
testcase test(…) runs on mtc_type system system_type {
      …
      User1_type.start(state1to2());
      alt {
            []mtc2ptc.receive(syncall) from User1 -> value PTCResult {
                  if(PTCResult.report == "User1:Message1 ") {
                        log("User1:Message1");
                        setverdict(pass);
                        goto finalState;
                  }
                  if(PTCResult.report == "fail") {// wrong or no message
                                                  // reveived by PTC
                        setverdict(fail);
                        goto errorState;
                  }
            }
                  [] mtc2ptc.receive {            // unexpected message
                                                  // reveived by PTC
            setverdict(fail);
                        goto errorState;
            }
      }
      label errorState;
      …
      stop;
      label finalState;
}
```

to change the FP in order to emulate roaming mobile users while talking in a voice conference.

Use case and interaction diagrams that express the requirement definitions of the DECT system contain about 230 sequence diagrams. XSLT style sheets are used to transform these diagrams to a UML protocol state machine consisting of around 900 states and over 3400 transitions that represents the MCUM as an executable test model.

The usage behavior was specified by TTCN-3 test cases and functions which are executed on the components. In addition templates for sending and receiving messages had to be defined. By matching template names to the signatures of the DECT messages that are used in the scenarios this mapping was done automatically during the test.

In our previous approach ([2]), the duration for generating and transforming a test case from the generated MCUM as show in Fig. 12 is in the order of six minutes related to the DECT case study. The TTCN-3 test suite consisted of over 200 test cases, which means that about 20 hours are needed to derive the TTCN-3 source code. After additional 24 hours to compile the executable test suite by means of Telelogic Tau G2 the actual test could be started and revealed around ten failures of different types, e.g. the reception of a wrong message type, wrong parameter values and even a non-functional violation of a given time constraint.

In the TestUS approach no overhead for generating, transforming and translating test cases in order to produce the TTCN-3 test suite is necessary. Instead, the transformation of the MCUM to the TTCN-3 source code for the DECT case study can be done within 5 seconds using a Java tool that was developed to do this task and which can be selected via an Eclipse plug-in. The compilation of the executable test suite is done by Telelogic Tau G2 within additional 15 minutes. Now, as long as one likes test cases can

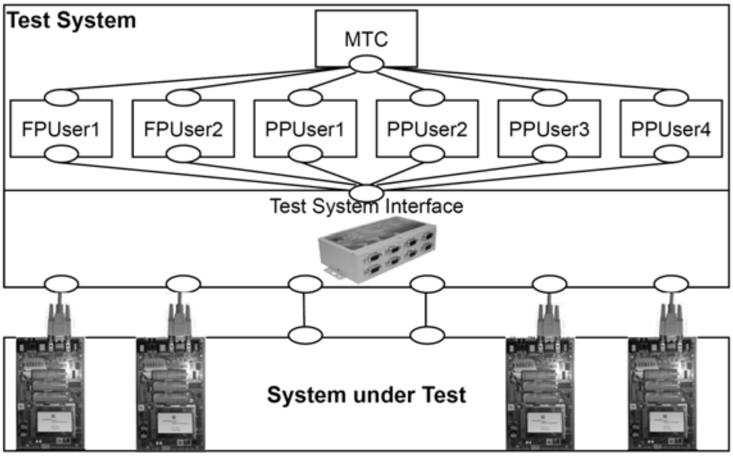


Fig. 14. TTCN-3 test suite for testing the DECT system

be performed and interpreted on-the-fly in real-time without any further modifications of the TTCN-3 test suite.

6 Conclusions and Ongoing Work

The advantage of the new approach implemented in the TestUS framework is obvious:

* The main effort at the beginning of the test process is to construct a MCUM in order to reflect the correct usage behavior between the SUT and all possible actors.
* Based on a UML 2.0 software engineering process, which starts from use case diagrams that contain interaction diagrams to refine the user interactions an automatic derivation of the MCUM protocol state machine representation is achieved by a proper tool chain.
* There is no need to calculate TTCN-3 test cases in advance. Therefore, it is possible to avoid the unfolding of finite loop fragments with upper and/or lower boundaries and the serialization of interleaved events that are responsible for a factorial growth of the length of the test cases.
* Once the MCUM is transformed to a TTCN-3 test suite, test cases and the evaluation of test verdicts are interpreted on-the-fly in the executable test suite.

We proved the new concept by means of a realistic case study for testing a DECT communication system. The previous generation and compilation time for the dedicated DECT test suite summing up in the order of 44 hours was reduced to only 15 minutes and we got a TTCN-3 test suite at the end that interprets as many test cases as one likes for the DECT system on-the-fly and in real-time.

Recently we have also shown that statistical testing based on MCUMs is a promising extension to existing deterministic testing approaches in the medical and automotive domain. In particular, a polyhedron method for calculating probability distributions from given constraints has been developed and compared to the maximum entropy approach [8].

From the usage models, a multitude of metrics can be obtained analytically to improve test planning or to support management decisions. A number of possible scenarios for using these metrics in order to compare customer classes by means of different MCUM usage profiles have also been identified and discussed ([8]).

All in all, model-based test case generation using MCUMs and dedicated profile classes enable many new and promising techniques for testing complex systems.

References

1. Beyer, M., Dulz, W.: Scenario-Based Statistical Testing of Quality of Service Requirements. In: Leue, S., Systä, T.J. (eds.) Scenarios: Models, Transformations and Tools. LNCS, vol. 3466, pp. 152–173. Springer, Heidelberg (2005)
2. Beyer, M., Dulz, W., Hielscher, K.-S.J.: Performance Issues in Statistical Testing. In: Proceedings MMB 2006, Nuremberg, Germany (2006)
3. Biegel, M.: StatisticalTesting of DECT Modules. In: Proceedings ITG Workshop on Model-Based Testing, Nuremberg, Germany (2006)
4. Broy, M., Jonsson, B., Katoen, J.-P. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 607–609. Springer, Heidelberg (2005)
5. ETSI: TTCN-3 Core Language. ES 201 873-1 V3.1.1 (2005)
6. Gutjahr, W.J.: Importance Sampling of Test Cases in Markovian Software Usage Models, Department of Statistics, Operations Research and Computer Science, University of Vienna (1997)
7. Haberditzl, T.: On-the-fly Interpretation von UML 2.0 Anwendungsszenarien in einer automatisch generierten TTCN-3 Testsuite. Masterthesis, Department of Computer Science, University of Erlangen-Nuremberg, Germany (2007)
8. Holpp, S.: Proving the Practicality of the Scenario-based Statistical Testing Approach within the Scope of the System Validation of a RIS/PACS System. Masterthesis, Department of Computer Science, University of Erlangen-Nuremberg, Germany (2008)
9. Musa, J.D.: Operational Profiles in Software-Reliability Engineering. IEEE Software (1993)
10. OMG: Unified Modeling Language - Superstructure. Version 2.1.1 (2007)
11. Rosaria, S., Robinson, H.: Applying models in your testing process. Information and Software Technology 42, 815–824 (2000)
12. Sayre, K.: Improved Techniques for Software Testing Based on Markov Chain Usage Models. PhD thesis, University of Tennessee, Knoxville (1999)
13. Tretmans, J., Brinksma, E.: Automated Model Based Testing. University of Twente (2002)
14. Walton, G.H., Poore, J.H., Trammell, C.J.: Statistical Ttesting of Software Based on a Usage Model. Software - Practice and Experience 25(1), 97–108 (1995)
15. Walton, G.H., Poore, J.H.: Generating transition probabilities to support model-based software testing. Software - Practice and Experience 30, 1095–1106 (2000)
16. Whittaker, J.A., Thomason, M.G.: A Markov Chain Model for Statistical Software Testing. IEEE Transactions on Software Engineering 20(10), 812–824 (1994)

Language-Critical Development of Process-Centric Application Systems

Tayyeb Amin, Tobias Grollius, and Erich Ortner

Development of Application Systems, Technische Universität Darmstadt
Hochschulstr.1, 65239 Darmstadt, Germany
{amin,grollius,ortner}@winf.tu-darmstadt.de

Abstract. The shortage of skilled IT-staff as well as the technological possibilities offered by Service-oriented Architectures (SOA) and Web 2.0 applications, leads us to the following consequences: working processes, job engineering and labor organization are going to be modeled and therefore made digital in the sense of IT-support. This goes along with modeling working processes being independent from the individual employee in areas to be rationalized resp. not to be staffed by qualified specialists. Hence, there will be a worldwide net based selection of those who are able and skilled to fulfill modeled work like e.g. "handling a damage event" or "creating an optimized data structure for master data" by means of the Unified Modeling Language (UML) in the most effective and efficient way. An enterprise will therefore neutrally manage its modeled work processes (HB-services) and IT-services (application programs) taking place as computer supported work equipment in any working process being located anywhere in the world without assigning it first to a specific performer (neutral or artificial actors). By doing so it is possible to control and dynamically execute working processes globally based on the division of labor, and on a database supported administration of "bills of activities" (work plans) by means of the World Wide Web. All that requires new and dynamic – in the sense of component based – job descriptions and other work equipment exceeding today's established skill and task management by far.

1 Introduction

According to Aristotle (384-322 BC), *Architectonics* refers to the art and science of building, while the term *architecture* denotes the structure. In civil engineering, the proportion of "art" within the structuring activity is usually drawn upon to distinguish an architect (emphasis on art) from an engineer (emphasis on methodology).

For vendors and users of information technology, the term *service-oriented architecture* (SOA) has now been an issue for about many years. Organization-centric Requirements Engineering (on a constructivistic basis) with reference to Applied Computer Science was proposed for the first time in 1980 [1]. The following observations are important to this concept today:

- *Organizational Processes*, which, to some extent, offer a free choice (e.g. specific knowledge gained from experience) to the acting persons in particular steps of an occurrence, are principally to be distinguished from the inherently different algorithmic

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 31–46, 2010.

computer processes (software and hardware). Therefore, organizational processes ought to be specified further by *language-critical organization theory* [2].
- Ubiquitous (= found everywhere) computer technology leads to the fact that presently almost any object (thing or occurrence) can be a medium in this technology.
- *Common languages*, (e.g. "ontologies", conceptual schema, ortho-languages) whether spoken by people or used in technology, always serve as a means for integration. This includes, to some extent, the integration of heterogeneous elements in a system or architecture.
- Today, the *object-language/meta-language* difference in application systems – the showpiece of system informatics – is firmly established within the overall architecture by means of repositories. For example, repositories are used to enable and facilitate the management of component-based solutions.
- In the field of constructive languages and consequently in research methods of various application fields, there has been considerable progress in the past few years ascribed to the *Unified Modeling Language* (UML), whose development is still ongoing.
- There are predominantly three factors, which have initiated the necessity to start *organizing work globally*. These are: The fact that Enterprise Organization Theory is a part of Applied Computer Science, the comprehensive concept of application systems as *service-oriented architectures*, and the development of new technologies on the internet (Web 2.0), for example interactive applications [3]. It is essential to approach this organizational task from the position of dynamically organized enterprise networks that interact globally.

In the following, we will describe how the ProCEM® method (Process-Centric Enterprise Modeling & Management) meets the above-mentioned requirements, taking into consideration the human needs. The basis for our description is the experience gained from accompanying the project "Best Process Architecture", a contribution to the BITKOM college competition 2007, the seminar "SOA in Accountancy" held in the summer term of the same year, and the lecture organization-centric "Development of Application Systems", which is an ongoing lecture at TU Darmstadt as of 1996.

The progression of software engineering to enterprise engineering in the last thirty years as well as the reversionary methodological development of a complete application system can be conceived as a basis for ProCEM®.

2 Application Systems Architecture in the 21st Century

Modern application system architectures have been evolved from the conventional layer architecture of an IT system (IT infrastructure, data, application software) to a comprehensive architecture of a whole organization – the application system. Now it has its span over the organizational structure e.g. human resources, the business processes that are partly automated as workflows, the application services (IT services) and the data inventory and IT infrastructure as a basis. The business processes resp. the workflows constitute the core of the process centric application system, executed and supported by employees as "human services" or IT services.

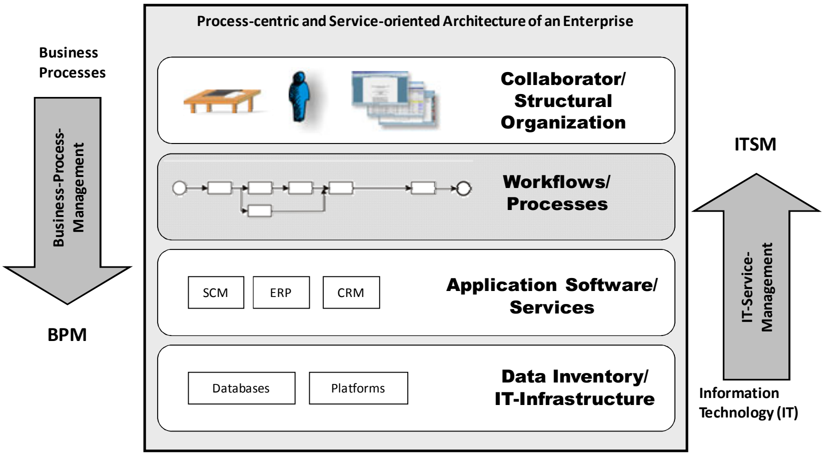

Process-centric and Service-oriented Architecture of an Enterprise
Business Processes
Business-Process-Management
BPM
Collaborator/ Structural Organization
Workflows/ Processes
Application Software/ Services
SCM
ERP
CRM
Data Inventory/ IT-Infrastructure
Databases
Platforms
ITSM
IT-Service-Management
Information Technology (IT)

Fig. 1. 4-tier architecture of an enterprise (application system)

From the information technology point of view on an application system, the field of IT service management is concerned with business process orientation, whereas business process management deals with the business process integration with IT applications from the business perspective.

3 Process-Centric Development of Application Systems

While process-centric application systems development can be equated with a shift from the “world of being” (entity schema) into the “world of doing” (occurrence schema), it can also be described as the change from static organization to dynamic organization of an application system.

In the center of the development are the (organizational, e.g. business) processes, which are reconstructed and composed of process schema parts. A process is defined as a directed sequence of occurrences. If a process has been schematized, then this process is available like any other appliance.

Following the process-centric application systems development style we involve and integrate different services concerned with organization structure (human resources) and IT applications to support, execute, or even automate the processes. As shown in figure 2, the development process is organized in three iterative sub processes:

- *Process Centric Cycle*: the main development cycle is responsible for the optimized workflow management. As the starting point the (work-) processes of the organization are reconstructed using different types of process modeling techniques e.g. BPMN. Simultaneously these work processes can be improved by optimizing and simulating based on the process models. Having optimized processes, internal or even external services are determined that execute the processes. At this point the two more cycles of the development process are involved: either work plans to be fulfilled by human resources (employees) or IT services can be selected as executing units. Finally, the elected but possibly heterogeneous services must be orchestrated to the application system.

- *Human Centric Cycle*: which is responsible of the dynamic nature of the organization structure with respect to human resources and offers the best use of these resources with minimum cost expenditures. In this cycle we select detailed work plans, which are schematized and therefore disposable like a product, for the single process activities or sub processes and determine the appropriate employee, who is able to execute the work plan.
- *IT Centric Cycle*: this is responsible of providing dynamically IT services that support or execute activitities or sub processes of the reconstructed process. IT services, or more precisely IT service schemas, are application software that implements work procedures [4]. They are commonly developed on the basis of software components and specified as algorithms.

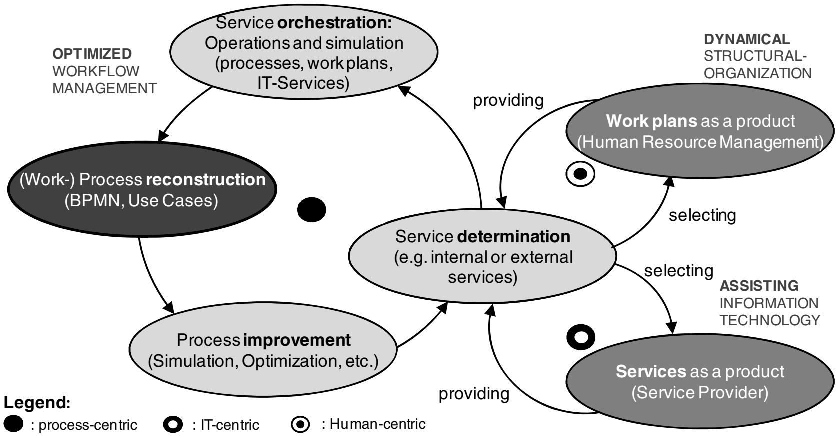


Fig. 2. Iteration paths of orchestration: processes – (information) technology – man

Rather than having oboes, violins or triangles at ones disposal, the orchestration of application systems make use of human beings, organizational structures and technology. Whereby anyone who specifies e.g. organizational processes in the same way as computer processes and does not distinguish between human-related symbol processing and computer-based symbol processing [5], is not suitable for the development of organization-centric application systems. Such a person lacks the interdisciplinary knowledge taught by some of the forward-looking chairs of Applied Computing and Application Systems at universities worldwide today.

4 Interdisciplinary Language-Critical Specification of IT-Use

Which skills and what kind of knowledge do developers, i.e. “business architects,” “application developers,” “solution architects,” and so on, need for organization-centric application development in order to successfully participate in projects of this kind or even execute such a project on their own? In his book “Der Flug der Eule” (The flight of the owl), Mittelstraß gives us an answer that is as clearly defined as it is simple [6]:

> "Anyone […] who has not studied interdisciplinary
> cannot perform interdisciplinary research."

The acquisition of interdisciplinary schemas and the understanding of them is a prerequisite for interdisciplinarity. Anyone who has only studied how to apply something will not be able to develop organization-centric application systems. The following is a simple example that illustrates the profound understanding of the development process. The example reconstructs a data schema in Applied Computer Science.

Schematize the following sentence in object-language,

a) "Smith is a customer who is willing to pay."

by means of computer sciences, specifically the meta-language "relational model":

b) "Relation Name (key attribute(s); non-key attributes)"

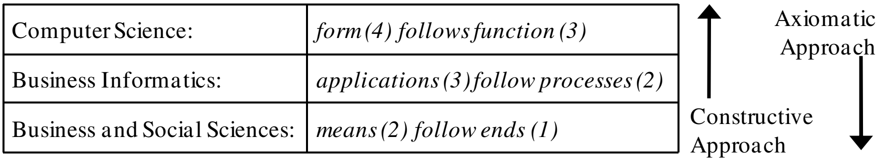
in an interdisciplinary way, i.e. by different disciplines simultaneously. In order to accomplish this, a developer must not solely understand the sentence in object-language a) with respect to its business-driven generalization (schematization, norm), but additionally, the user must have agreed on the norm derived from it. In our example, this would mean that a society (language community) tolerates the following object-language norm:

a') "If we identify a person as a customer, we are allowed to characterize the customer more closely by the attribute 'payment behavior'."

Furthermore, it is necessary to realize or understand that the "relational model" is merely a different grammar (meta-schema) for representing the standardized (schematized) object-language "content". It is our goal to maintain customer data efficiently on the computer. Through modeling, we achieve the significant result of our interdisciplinary schematization:

b') "Customer (name; payment behavior)".

Interdisciplinary schematization (modeling) is one of the core tasks in Applied Computer Science such as Business Informatics. For the acquisition of interdisciplinary knowledge, e.g. in university courses of study, there is even a so-called "methodical order" that is to start from the organization to come down to the layer of information technology. (see figure 1). We can formulate it as follows, whereby the figures in brackets indicate the "sequence", i.e., the methodical order. Today specified processes are means for ends.



Computer Science:	*form (4) follows function (3)*
Business Informatics:	*applications (3) follow processes (2)*
Business and Social Sciences:	*means (2) follow ends (1)*

Theoretically, the methodical order, or course can be avoided. However, in practice, it is recommended to adhere to it. It is most advisable to "put on the socks before putting on the shoes", although, at least in theory, it may be possible to consider the reversed order. The problem in some of the programs of study in Computing Sciences is that interdisciplinary knowledge is not taught – even at the recently appointed German superior universities, an apparent lack in IT-architects has lead to the fact that students in bachelor programs of study merely concern themselves with pure computer sciences, i.e. (3) and (4). For those students who have not entered a practical profession by then, the Masters program of study will "ensure that they are acquainted with matters of Applied Computer Science" [5]. Well, it is conceivable that disciplines become extinct.

4.1 Organization Modeling

For successful organization modeling (Enterprise Engineering) – especially with respect to optimization – differentiation is vitally important. Figure 3 illustrates the possibilities regarding work processes and structures.

The capability to differentiate clearly is critical to the ability to optimize. This is important for the object-language level, the application field, as well as for the meta-language level, the diagram language field. On both levels, the point is the reconstruction of connector words (e.g. to do) and topic words (e.g. to work). On the meta-language level, the developer gets to know the modeling method in greater detail.

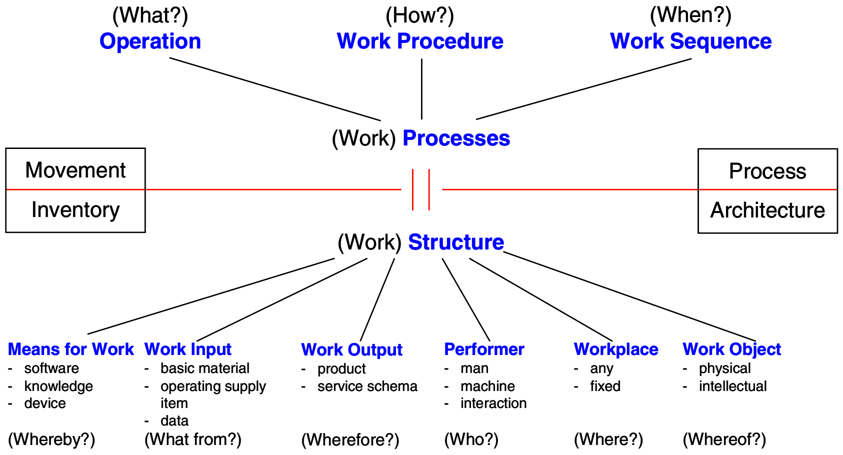


Fig. 3. Differentiated work organization

On the object-language level, the grammar of the modeling language plays a vital role. In the latter case, the organizational expert knowledge of the relevant application domain must be represented in a structured way.

Modeling (topic words of the application field) and structuring (connector words of the modeling language) are different but they supplement each other as complementary parts.

Even before the object-oriented system design, diagram languages have proved to be suitable modeling languages for the organization of a company's structures and processes. Use cases for example are especially useful for the structural aspect (see figure 4), while the Business Process Modeling Notation (BPMN) is suited ideally for the procedural aspect (see figure 5).

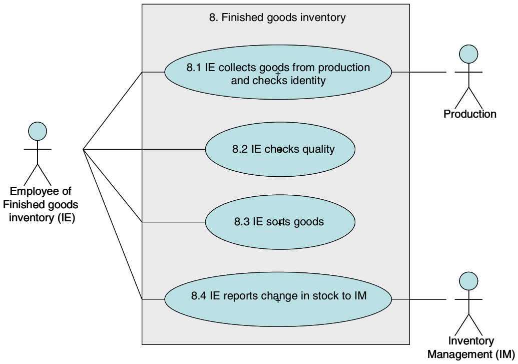

8. Finished goods inventory
8.1 IE collects goods from production and checks identity
8.2 IE checks quality
8.3 IE sorts goods
8.4 IE reports change in stock to IM
Employee of Finished goods inventory (IE)
Production
Inventory Management (IM)

Fig. 4. Use case diagram (example: finished goods inventory)

Detailed descriptions of modeling languages can be found in various case collections [7] or OMG manuals. However, anyone who later, in the system design, intends to specify the flow of work processes in greater detail is well advised to distinguish the aspects like "operations", "work procedures" and "sequence of work" or "workflows" orthogonally. The same applies to the organization structure and aspects such as "workplace", "vacancy", "employee" or "work material" (see figure 3). The optimization can now be considered sensibly and from different angles (aspects).

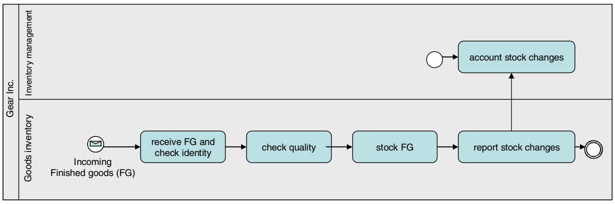


Fig. 5. BPMN diagram (example: incoming finished goods)

4.2 Method-Neutral Knowledge Reconstruction

The method-neutral knowledge reconstruction [8] is primarily communicative and hardly any diagram representations are used.

In order to get a first picture of the important tasks, which are performed in close cooperation with the users, we classify them roughly in the following three parts:

- Collection of propositions those are relevant for development by talking to the users.
- Clarification and reconstruction of the expert terminology that has been used.
- Establishment of a common enterprise expert language.

The collection of propositions relevant to the development can be done by using a model, as shown in figure 6.

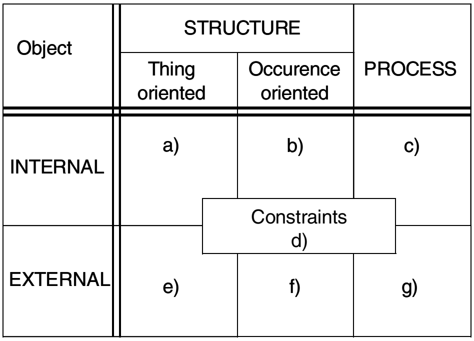


Fig. 6. Classification schema for propositions

The model is intended to aid the collection and to ensure that all potential types of results for the system design have been scrutinized in consideration of their underlying expert knowledge. The following list contains several propositions that can be assigned to the above fields (see figure 6):

a) An account has an account number.
b) An account is opened.
c) Opening an account results in an opening balance.
d) The total of all debit line items must be the same as the total of all credit line items in double entry accounting.
e) A (personal) account is assigned to a business partner.
f) Shipment of goods is related to posting business transactions.
g) At the end of an accounting period, all of the accounts are closed; their values are in a profit and loss account and ultimately gathered in the balance.

In order to clarify and reconstruct the identified expert terminology the following "defects" are discussed and examined thoroughly with the future users or the company's experts.

Checking ***synonyms***
Check for words with the same meaning (extension and intension) that can be interchanged.

e.g.: MEMBER and ASSOCIATE have the same meaning for DATEV[1].

Eliminating ***homonyms***
Check for words that are written or pronounced in the same way but have a different meaning.

e.g.: STALK, which can mean either part of a plant or to follow someone around

Identifying ***equipollences***
Different names are used for the same objects (extension) from different perspectives (intension).

[1] DATEV is a computer center and software house for the German-speaking tax profession where the author worked as executive manager in software development for seven years.

e.g.: Goods or merchandise of a company is referred to as STOCK from a quantitative perspective and INVENTORY ACCOUNT from a value perspective.

Clarifying ***vagueness***
As there is no clear delimitation (definition) of the terms in regard to their content (intension), it may not be clear which objects belong to each term (scope, extension)

e.g.: Does RESIDENCE, the place where a CONSULTANT works, belong to the term CHAMBERS for DATEV or not?

Replacing ***wrong designators***
Discrepancies between the actual meaning of a word and the meaning assumed at first (intension and extension)

e.g.: For DATEV, the CONSULTANT NUMBER does not define the function of a tax CONSULTANT, but it defines the USER RIGHTS a tax CONSULTANT has within DATEV.

This clarification results in further propositions relevant for development. Their relevance for the result types (system design) can be examined with the help of a classification schema (see figure 6). Work on building a common expert language for a company, which is aimed at integrating all of a company's knowledge resources, can be organized in different ways.

1. With the help of a repository, a kind of glossary will be created and administered. This glossary will contain all the terms that are important for an organization (language community), and should be designed for internal and external use.
2. A much more complex way, in comparison to (1.), of representing a company's knowledge is with an encyclopedia. The encyclopedia amounts to a conceptual schema for data but will go substantially further in respect to terminological coherences. This approach will distinguish inward and outward knowledge, which will be administered in a repository as an enterprise knowledge base.
3. The enterprise expert language is a rational interim language that is implemented on a meta-meta language level in the repository [9]. It is used for integrating and translating other languages used in a company. For users, it is not necessary to know the interim language itself.

Currently, the three variants discussed above can be found in industry worldwide. Vendor-independent research is done in the field of SOA under the catchword *Enterprise Application Integration* (EAI). Furthermore, companies like Oracle look into *Application Integration Architecture* (AIA) and offer products such as Fusion. Other vendors offer products like WebSphere (IBM) or NetWeaver (SAP) for the integration task.

4.3 Generally Object-Oriented System Design

After the development-relevant knowledge (see figure 7) has been reconstructed neutral to specific methods and technology (e.g. according to Ortner [8]), and integrated into the overall knowledge base of an enterprise using common language, then, in the system design, this knowledge is transformed into the result types of an object-oriented solution to the task. Figure 7 shows an object-oriented system design according to Schienmann [10] that has been extended for the design of service-oriented architectures of an enterprise.

When we speak of entirely object-oriented development of application systems, the enlightening step is the introduction of objects from computing sciences as grammatical objects. Grammatical objects are target points of language actions (e.g. writing, speaking, thinking) in a sentence, whereby they can also be replaced by pronouns in the sentence (e.g. one, he, him, this one). At school we have learned to speak of direct and indirect objects, genitive objects and various prepositional objects.

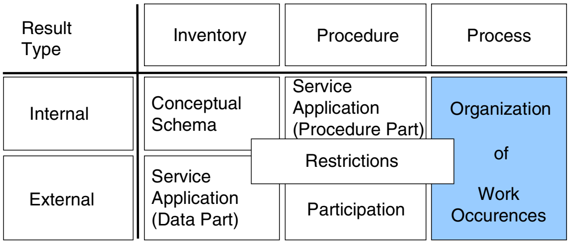

Result Type
Inventory
Procedure
Process
Internal
Conceptual Schema
Service Application (Procedure Part)
Organization of Work Occurences
Restrictions
External
Service Application (Data Part)
Participation

Fig. 7. Extended object-oriented enterprise design

In contradiction to what many computer scientists still believe, when modeling and programming in computer science we do not concern ourselves with concrete or "ontological objects" such as this chair, that apple or my laptop. When speaking of objects "informatically" (i.e. modeling and programming), it is of particular importance that abstract types such as "class" in a repository or the term "invoice" in an application, are to be thought of as target points. The concrete, "ontological objects" are usually found in the application fields.

Computer science is the science where students learn how to talk constructively about language (grammatical) objects, or more precisely, about abstract objects. Needless to say, we can still start from the concrete objects of the application fields in "Requirements Engineering" and when introducing the implemented solution, we can refer back to the users' concrete (ontological) objects.

The object-oriented approach in the development of application systems goes back to Platon. Platon classifies objects from the perspective of human beings and their languages into things (nouns, proper names) and actions, which can also be considered occurrences (verbs). If we transfer this classification to operating with data on a computer, the object-orientation (resp. its object) will be classified into the fields of data orientation (things) and procedure orientation (occurrences). This classification shows why object-orientation is universal. It encompasses data orientation (data classes) as well as procedure orientation (procedural classes).

Based on the results of organization modeling (enterprise engineering) and (embedded) system design (see figure 1), the results from figure 7 are modeled (software engineering) in the following methodical order:

1. Process modeling:
   - BPMN diagrams (from organization modeling)
   - State machine diagrams
   - Activity diagrams
   - ...
   - Constraints (e.g. in Object Constraint Language (OCL))

2. Participation modeling:
    - Use cases (from organization modeling)
    - Sequence diagrams
    - Job descriptions (in the sense of structural organization)
    - ...
    - Constraints (e.g. organizational standards such as signature regulations)
3. Procedure modeling:
    - Class diagrams (data classes and procedural classes)
    - State machine diagrams
    - Activity diagrams
    - ...
    - Constraints (e.g. plausibility checks at data entry in service-oriented applications)
4. Inventory modeling:
    - Object type diagrams (for the conceptual schema)
    - Dataflow diagrams (for specification of data that are exchanged)
    - External schemas (extended as data classes)
    - ...
    - Constraints (e.g. semantic integrity rules for DBMS-enforced integrity)
5. Business goal modeling:
    - SBVR
    - ...

Enterprise Engineering and the reconstruction of development-relevant expert knowledge from the application fields are highly communicative processes. Here, users and developers communicate very “intensively” (in great detail and clearly) with each other. Diagram languages play a minor role in this context. In contrast, the (entirely) object-oriented design of a SOA with diagram languages must be performed in an already highly "significative" way. This means that the terms of the object language and the meta-language (e.g. “invoice” as an object-language terminus and “procedural class” as a meta-language terminus) should be displayed as independent as possible from their use in the judgment-context. The focus is on disclosing the types (software, concepts) that shall be implemented later. Diagrams are ideally suited for this purpose.

The diagram languages for procedure modeling are of course very similar to the diagram languages for process modeling. Procedure modeling comprises of the process parts (algorithms), which run as service-oriented applications while a process is being executed, as well as of those process parts, which can be specified in less detail since they involve human work (e.g. following work plans).

The order (1.-4.) chosen here serves merely as a recommendation. The modeling process is an iterative process, as every well-educated developer will know from practical projects (see figure 2).

5 Concretion in the Large

Organization-centric development of application systems has been derived from data-centric [1] development. The development paradigm “applications follow processes”, which is valid in today’s service-oriented architectures, complements, but does not replace the data-centric approach. Therefore, SOA stands for a new paradigm, not a shift in paradigm. The data-centric approach remains as important as ever, but due to the triumphant progress of object-orientation and component-based development, it is integrated in the overall architecture and work processes in a more “intelligent” way (Platon was right!). In addition to data processing, work organization (enterprise engineering) has become a subject in applied computer science (software engineering).

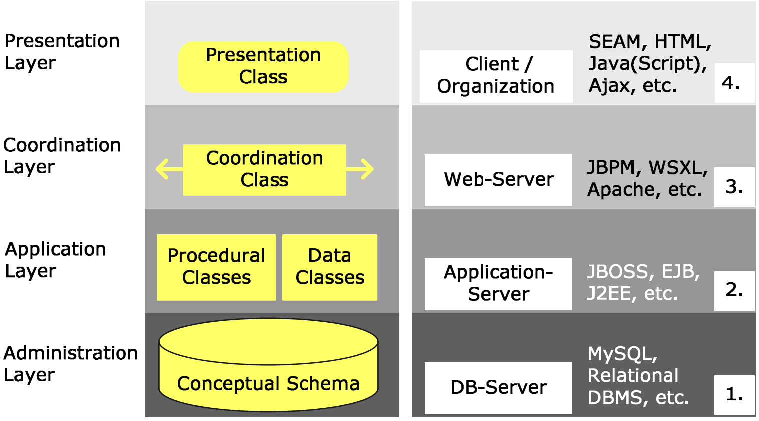


Fig. 8. Entirely object-oriented concretion of a service-oriented architecture

Figure 8 shows an enterprise represented in an entirely object-oriented way. On the right, implementation aspects can be found; the right side lists the specific details of the information technology available for implementation today. We are therefore only talking about a “concretion in the large”. “IT and solution architects”, “integration developers” and “deployment managers” must be able to deliver this concretion for the enterprises (domains) that use information technology [11].

“Code development”, which is of course also important, in particular from the point of view of the “service and solution testers” on site, is currently done in so-called “low-wage countries” by well-trained people (near and offshoring). In complementation to a “concretion in the large”, we are now talking about a “concretion in the small.”

6 Dynamic Support and Optimization of Work Processes

For the dynamic management of application systems (see figure 2), it is necessary to create and use a meta-information system whose most important part is the repository system [9] as for example described by Berbner et al. [12]. In accordance to the much-noted work “The Quest for Resilience” by Hamel and Välikangas [13], future enterprise networks will be implemented as elastic ecosystems [14] built from components of different categories. These systems must be assembled in the best possible way, thereby facilitating the systems to respond, possibly even self-actingly, to changing situations.

The ever changing job design (e.g. due to product changes) and work organization is crucial to this approach. This is done considering

1. the aspects: optimized processes, best possible employee assignment and dynamic IT-support (e.g. IT-services), as well as
2. the fact that some of the jobs that are part of these processes, are performed by employees who come from everywhere, or respectively, the jobs are done where personnel is available at low cost.

In this regard, organizing potential assignments for employees in work plans and establishing a global work base (see figure 9) is exceedingly relevant. Such a database allows neutral (assignment-free) storage and maintenance. It could contain the IT-services (data and program schemas) that are used anywhere in the world as program-technological means (to work plans) for work in those processes. This way, an enterprise’s IT-department organizes and controls the company’s work processes worldwide in division of labor and dynamically using the Internet.

The protruding innovation of SOA is the extension of the concept of application systems by work and organizational processes of enterprises. This makes organization theory as it is found in Business and Social Sciences, the Engineering disciplines or in Enterprise Engineering an “integral”, that is an interdisciplinary part, of Applied Computer Science. And this in a way that has not been seen in previous years.

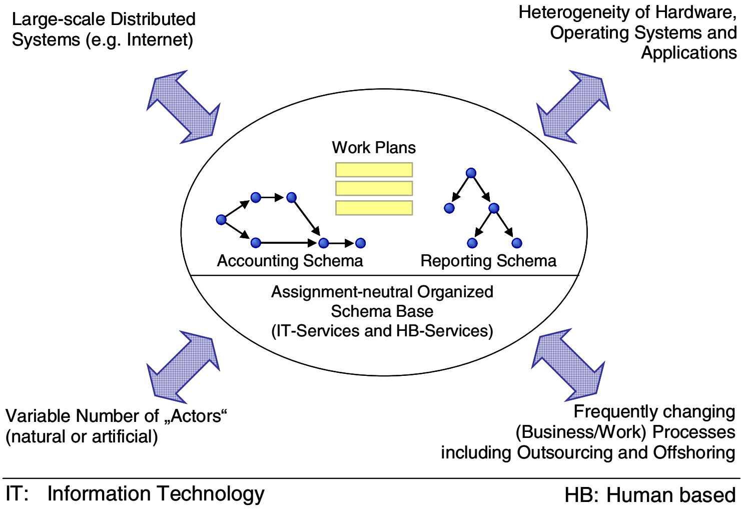

Large-scale Distributed Systems (e.g. Internet)
Heterogeneity of Hardware, Operating Systems and Applications
Work Plans
Accounting Schema
Reporting Schema
Assignment-neutral Organized Schema Base (IT-Services and HB-Services)
Variable Number of „Actors“ (natural or artificial)
Frequently changing (Business/Work) Processes including Outsourcing and Offshoring
IT: Information Technology
HB: Human based

Fig. 9. Labor as a product

Concepts and institutions like the German REFA-Association for Work Design or Methods-Time-Measurement (MTM) founded in 1924 (These are systems for time allotment that have been used in Sweden as of 1950, in Switzerland since 1957 and in Germany since 1960) suddenly constitute a field of activity and provide IT-businesses and enterprises worldwide with the knowledge that information and computing scientists possess. Due to Ubiquitous Computing, however, this also affects the courses of study of Enterprise Engineering, Business and Social Sciences, Mechanical Engineering, Electrical Engineering or Civil Engineering, as all of them are concerned with work science and process organization.

The following is a list of typical issues in optimization as they were recently elaborated by a student team of the TU Darmstadt at the Bitkom competition “Best Process Architecture” [15].

- *Parallelization*: Operations that are independent of each other must run in parallel and thereby shorten the overall process duration.
- *Optimization of single processes*: In addition, we must analyze each process individually to make improvements.
- *Integration*: Existing systems are integrated seamlessly in the new architecture so that the available resources can be used efficiently and effectively.
- *Elimination of information deficits*: The interfaces between operations are analyzed thoroughly so that the expected input or output will be found at the right time in the right place.
- *Reorganization or sequence optimization*: Process analysis takes into account an increase in efficiency due to reorganization of the order of single operations.
- *Outsourcing*: It is considered an alternative to outsource single processes, especially maintenance activities at the customer’s site, to external service providers.
- *Elimination*: During process analysis those operations must be eliminated that are useless or do not contribute beneficially to the process result.
- *Acceleration*: Specific measures for shortening the overall process duration are vital, but not at the expense of quality and cost.
- *Introduction of additional test steps*: To ensure higher quality, it is useful to integrate additional steps for checking the process.

From the perspective of an employee, there are three possibilities to be considered when setting out to optimize work processes using IT:

- to reduce people’s workload through *automation* (resource: “software”)
- to *support* human work as for example using interactive applications (resources: “software” and “knowledge”), or
- to improve people’s work *qualifications* (resource: “knowledge”)

Industrialization and automation were so successful in the previous decades that it is very advisable to revert our efforts with respect to the listing above. With globalization in mind as well as taking into consideration our worldwide division of labor, we should “invest much more in education and as little as possible in further automation efforts.” Technological progress cannot be stopped, but a world which is becoming increasingly compact, can only cope with progress, if it is flanked by human education.

7 Outlook

In Applied Computer Science, from a global standpoint, service-oriented architectures constitute a new paradigm, but do not result in a paradigm shift. Managing data and managing processes are complementary and lead to entirely new job descriptions. In the expert languages of globallyinteracting IT-enterprises these new professions are called:

- IT-Architect
- Business Analyst
- Application Developer
- Service and Solution Tester
- Software Developer
- Deployment Manager
- Integration Developer
- Solution Architect
- Code Developer
- etc.

Nevertheless, people, who perform these jobs throughout the world, have the least say in *who performs which kind of work when and where*.

There is nothing more important for our survival than that the humanities take up the challenge to newly enter in a process of enlightenment. *Logic*, *Mathematics*, *Linguistics* and *Computer Science*, for example, are studies of the humanities. “Normative Logic and Ethics” [16] as well as their advancement to an “Encyclopedia Philosophy and Philosophy of Science” [17] provide us with the necessary *fundamental education and terminology*, in the sense of a *Universal Literacy*, to fulfill this task.

Therefore, we appeal for constructive computer sciences [17] to become basic education for all citizens. As a matter of course, this basic education should be graded and differentiated into interdisciplinary (rather universities) and infradisciplinary (rather schools) knowledge.

The root of the matter is teaching a disciplined use of language.

Anyone who is a democrat and who is interested in participating in remodeling our pluralistic democracies into republics with a “plurality-tolerating form of life” [18] all over the world is well-advised to try this in a language-critical way. This form of life is characterized by the fact that it teaches people how they can think correctly instead of teaching them what they should think. – Parlemus!

References

1. Wedekind, H., Ortner, E.: Systematisches Konstruieren von Datenbankanwendungen – Zur Methodologie der Angewandten Informatik. Carl Hanser Verlag, Munich (1980)
2. Lehmann, F.R.: Fachlicher Entwurf von Workflow-Management-Anwendungen. B.G. Teubner Verlagsgesellschaft, Stuttgart, Leipzig (1999)

3. Nussbaum, D., Ortner, E., Scheele, S., Sternhuber, J.: Discussion of the Interaction Concept focusing on Application Systems. In: Proc. of the IEEE Intl. Conf. on Web Intelligence 2007, pp. 199–203. IEEE Press, Los Alamitos (2007)
4. Grollius, T., Lonthoff, J., Ortner, E.: Software industrialisierung durch Komponentenorientierung und Arbeitsteilung. HMD-Praxis der Wirtschaftsinformatik 256, 37–45 (2007)
5. Oberweis, A., Broy, M.: Informatiker disputieren über Anwendungsnähe der Disziplinen. Computer Zeitung 29 (2007)
6. Mittelstraß, J.: Der Flug der Eule – Von der Vernunft der Wissenschaft und der Aufgabe der Philosophie. Suhrkamp Verlag, Frankfurt (1989)
7. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley, Boston (2001)
8. Ortner, E.: Methodenneutraler Fachentwurf – Zu den Grundlagen einer anwendungsorientierten Informatik. B.G. Teubner Verlagsgesellschaft, Stuttgart, Leipzig (1997)
9. Ortner, E.: Repository Systeme, Teil 1: Mehrstufigkeit und Entwicklungsumgebung, Repository Systeme, Teil 2: Aufbau und Betrieb eines Entwicklungsrepositoriums. Informatik-Spektrum 22(9), 235–251 resp. 22(9), 351-363 (1999)
10. Schienmann, B.: Objektorientierter Fachentwurf – Ein terminologiebasierter Ansatz für die Konstruktion von Anwendungssystemen. B.G. Teubner Verlagsgesellschaft, Stuttgart, Leipzig (1997)
11. Jablonski, S., Petrov, I., Meiler, C., Mayer, U.: Guide to Web Application and Platform Architectures. Springer, Berlin (2004)
12. Berbner, R., Grollius, T., Repp, N., Heckmann, O., Ortner, E., Steinmetz, R.: Management of Service-oriented Architecture (SoA)-based Application Systems. Enterprise Modelling and Information Systems Architectures 2(1), 14–25 (2007)
13. Hamel, G., Välikangas, L.: The Quest for Resilience. Harvard Business Review (September 2003)
14. Corallo, A., Passiante, G., Prencipe, A.: Digital Business Ecosystems. Edward Elgar Publishing, Cheltenham (2007)
15. Ghani, H., Koll, C., Kunz, C., Sahin, T., Yalcin, A.: Concept for the BITKOM University Challenge 2007. Best Process Architecture (2007), http://www.bitkom.org
16. Lorenzen, P.: Normative Logic and Ethics, 2nd edn. B.I.-Wissenschaftsverlag, Zurich (1984)
17. Mittelstraß, J.: Enzyklopädie Philosophie und Wissenschaftstheorie, vol. 1 (1980), vol. 2 (1984), vol. 3 (1995), vol. 4 (1996) . J.B. Metzler Verlag, Stuttgart
18. Lorenzen, P.: Constructivism. Journal for General Philosophy of Science 25(1), 125–133 (1994)

Balancing Business Perspectives in Requirements Analysis

Alberto Siena[1], Alessio Bonetti[2], and Paolo Giorgini[2]

[1] FBK - Irst, via Sommarive, 18, 38050 - Povo, Trento, Italy
siena@fbk.eu
[2] University of Trento, via Sommarive, 14, 38050 - Povo, Trento, Italy
{bonetti,giorgini}@fbk.eu

Abstract. In modern organisations, the development of an Information System answers to strategic needs of the business. As such, it must be aligned at the same time with software and corporate goals. Capturing its requirements is a complex task as it have to deal with a variety of different and potentially conflicting needs. In this paper, we propose a novel conceptual framework for requirements modelling and validation, based on economic and business strategy theory.

Keywords: Software Engineering, Requirements Engineering, Business Strategy, Goal Analysis.

1 Introduction

Understanding and analysing the purpose of a software system before defining its desired functionalities, results crucial and more and more mandatory in the development of modern information systems [7]. Early requirements analysis [1] is currently gaining popularity in the requirements engineering community as an effective way to understand the reasons of a new system with respect to the organisational setting in which it will operate. In this context, goal-oriented techniques have been proposed in the last decade [17,9,5] to answer *why* questions, besides *what* and *how*, regarding system functionality. Answers to why questions ultimately link system functionality to stakeholder needs, preferences and objectives.

Goal analysis techniques [4] are useful to understand the structure and the correlations among goals, their decomposition into more fine-grained sub-goals, and their relation with operational plans. Moreover, reasoning techniques applied to goal models [8] can be very useful to verify properties of the model and possibly to support the analyst in the conflict resolution process. Although these techniques result very useful to reason about single goal models, they are inadequate to support strategic decisions of an organisation. This is mainly due to the fact that they assume the perspective of the designer of the system and do not consider other dimensions like the business or financial needs that are crucial in decision process of an organisation.

In this paper, we revise the Tropos methodology [3] extending its goal reasoning framework [9] with a more enterprise and business oriented approach, the *balanced scorecards*. The adoption of balanced scorecards allows the analyst to have a more comprehensive vision of the enterprise and consequently to adopt solutions that can

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 47–59, 2010.

be related to its strategic decisional process. The purpose of this paper is twofold. On the one hand, it *introduces* the conceptual framework and the methodology for guiding the analyst in the requirements analysis process, by assuming and combing multiple perspectives of analysis (i.e., financial perspective, internal processes perspective, customer perspective, learning and growth perspective). On the other hand, it *evaluates* such a framework with respect to its actual contribution to the software engineering process, by means of a real world example.

The paper is structured as follows. Section 2 presents our experience with a requirements analysis case and the problems it rises. Section 3 introduces the underlying concepts of goal analysis techniques and the Balanced Scorecards business modelling approach. Section 4 presents the new Balanced Goalcards framework. Section 5 discusses the results that emerge from the application of the framework to our case. Section 6 concludes the paper.

2 An IS for Logistics

Frioli S.n.C. is a small Italian transport company located in the Trentino region, interested in developing a new information system to support its business. The company was founded in the sixties as a family company, and today it has five associates and four full-time employees. It owns five trucks and four trailers, and it transports both construction material (mainly cement) and undifferentiated goods. The transport activity is 80% in northern Italy, 10% in the rest of Italy, and 10% among Austria, Germany and France. Its costs are mainly related to fuel, insurance and road tolls. First need of the company is to optimise travels, and a new information system seems to be a necessary step in this direction. Our purpose is to gather the requirements for such an information system, ensuring them to be aligned with the actual business culture of the company.

Requirements have been initially collected after a number of meetings with people in the company. The main objective of these meetings was to understand the company's strategy and related activities. Unfortunately, after that it was not possible to have further meetings with the company (mainly for reasonable business constraints). This methodological constraint is the major **motivation** for our work. In a perfect world, with full information and unlimited resources, we could elicitate the requirements perfectly with effectiveness and efficiency. But in the real world, the challenge of software development is to overcome any kind of *limitations* - time constraints, budget constraints, communication obstacles, domain-specific skills, implicit information, and so on - to deliver the right solution at the right time and for the right market [6].

Our experience with the company has confirmed such a problem. A modelling session followed the interviews, with the purpose to model and formally analyse the models. Already during the modelling phase, we have encountered suspicious inconsistencies. For instance, managers declared interest in reducing costs but they were not interested to reduce phone customer support and off-line marketing techniques in favour of on-line services. The reason of this can be shortly explained as follows: the company has currently a positioning in the market and in the local community and doesn't want to lose it. Its positioning is built on top of the company's *philosophy* that is implicit and depends on many factors, such as quality of directors and employees, local market characteristics, history and structure of the company, results obtained, etc. In this scenario, on-line marketing can produce a very negative impact and change heavily the customers' perception of the company. So, even if the company wants to reduce costs, choosing the lower-cost solution is not the right solution.

These and other considerations showed clearly the need to:

i) support the analyst in capturing during the interviews both technical requirements and business strategy needs;
ii) provide a formalism to represent business entities in the software development process.

3 Background

3.1 Tropos and Goal Analysis

Tropos [3] is an agent-oriented software development methodology, tailored to describe the system-to be in terms of its requirements. It is intended to capture both functional and non-functional requirements, and, as such, a special role is given to the capability to model and understand the stakeholders' goals. The methodology analyses the requirements of the system-to-be in terms of goal models. Goals basically represent the functional requirements, while the softgoals represent the non-functional requirements of the system.

Goal models are represented by means of **goal graphs** composed of goal nodes and goal relations. Goal relations can be AND and OR decomposition relations; or, they can be contribution relations, *partial* - the "$+$" and "$-$" relations - and *full* - "$++$" and "$--$" relations. In practice, a goal graph can be seen as a set of and/or trees whose nodes are connected by contribution relations arcs. Root goals are roots of and/or trees, whilst leaf goals are either leaves or nodes which are not part of the trees.

For each goal of a goal graph, we consider three values representing the current evidence of **satisfiability** and **deniability** of goal: F (full), P (partial), N (none). We admit also conflicting situations in which we have both evidence for satisfaction and denial of a goal. So for instance, we may have that for goal G we have fully (F) evidence for the satisfaction and at the same time partial (P) evidence for denial. Such an evidence is either known *a priori* or is the desired one. In both cases, the conflicts arise by reasoning on the graphs with the techniques explained below.

On goal graphs, it is possible to analyse it with both *forward reasoning* and *backward reasoning*.

Forward Reasoning. Given an initial values assignment to some goals, *input goals* from now on (typically leaf goals), forward reasoning focuses on the forward propagation of these initial values to all other goals of the graph according to the rules described in [9]. Initial values represent the evidence available about the satisfaction and the denial of a specific goal, namely evidence about the state of the goal. After the forward propagation of the initial values, the user can look the final values of the goals of interest, *target goals* from now on (typically root goals), and reveal possible conflicts. In other words, the user observes the effects of the initial values over the goals of interests.

Backward Reasoning. Backward reasoning focuses on the *backward search* of the possible input values leading to some desired final value, under desired constraints. We set the desired final values of the target goals, and we want to find possible initial assignments to the input goals which would cause the desired final values of the target goals by forward propagation. We may also add some desired constraints, and decide to avoid strong/medium/weak conflicts.

The present work is based on the consideration that this kind of systematic analysis of the stakeholders' goals is necessarily general-purpose and performed from the perspective of the requirements engineer. As such, it provides little help in understanding the business specific requirements of the organisation. To overcome these difficulties, we refer to the *"Balanced Scorecard"* approach.

3.2 Balanced Scorecards

The Balanced Scorecards (BSC) framework was introduced by Kaplan and Norton in the early nineties [15] as a new strategy management approach, able to overcome the limitations they found in the then existing management methodologies. Essentially, pre-existing financial analysis techniques used to focus on monetary indicators, without taking into account non-measurable capitals of a company, such as knowledge or customers loyalty. As opposite, the BSC approach relies on three basic ideas [13]: i) both the *material* and *immaterial* objectives are important for the company; ii) the objectives, material and immaterial, can be numerically measured via properly chosen *metrics*; iii) the strategy is the resultant of the *balancing* of different kinds of metrics. In a broad sense, the ***strategy*** consists in the set goals, which will determine the success of the organisation in the market [16]. The strategy is the actual realisation of the mission, values and vision: the ***Mission*** statement describes the reason of being of the organisation and its overall purpose; the ***Business Values*** represent the organisation's culture, and turn out in the priorities that drive the organisation's choices; the ***Vision*** describes the goals to be pursued in the medium and long term, and how the organisation wants to be perceived from the market.

In the BSC approach, the strategy is bounded with a conceptual tool, the ***Strategic Map*** [12]. It is commonly represented as a diagram, containing goals and their interconnections. The connections are cause-effect relationships, meaning that the achievement of a goal brings to the achievement of its consequent. The strategic map is comprised by four different perspectives, i.e., the the financial perspective, the internal processes perspective, the customer perspective and the growth perspective.

The Economic-Financial Perspective. This perspective looks at the company from the point of view of the profitability, as well as solvency, liquidity, stability and so on. It expresses how well are the companys finances managed to achieve the mission. The metrics associated with this perspective are monetary and economic such as ROI, ROE, and more low-level values.

The Customer Perspective. Customers are the key stakeholders for the success of a company, so it is important to identify the target customer and define the properties that meet his needs. The ultimate purpose is to make products attractive for the customers.

The Internal Processes Perspective. The goals associated with this perspective are those that can have an impact on the internal effectiveness and efficiency of the company. Also, the goals that attain human and organisational resources are relevant, and can be measured by metrics such as are time, turnaround time, internal communications rating and so on. For instance, the choice to use e-mails instead of the telephone could improve the processes.

The Learning and Growth Perspective. The fourth perspective describes the long-term success factors. It is important to understand the characteristics that should equip the human, informative and organisational resources. So for instance, the choice to train the employees to use IT resources could allow the company to stay on the market also in the evolving global economy.

Differently from the Tropos goal analysis framework, the BSC is not intended to support formal analysis. However, its conceptual model can successfully capture the business aspects of an organisation. Thus, we aim to extend the representation capabilities of Tropos goal diagrams with the conceptual expressiveness of the BSC.

4 Balanced Goalcards

With the term "Balanced Goalcards" we refer to a novel approach for conceptual modelling that aims at aligning the business-centred needs of an organisation with IT-oriented requirements. The approach extends the *i**/Tropos methodology in both its modelling and analysis capabilities, and turns out in the ability to capture strategic requirements that are consistent with economic principles. The methodological framework of the approach is shown in Fig. 1. Shortly, the first step is the **domain modelling** and consists in the definition of the basic organisational settings. The actors are identified and their strategic dependencies are modelled. This phase follows exactly the Tropos methodology guidelines (see [3] for more details). Then, in the **strategy modelling**, the business context is modelled. Firstly, the mission of the organisation should be made clear, and then business values and vision. Using as leading parameters these root entities, the four different perspectives are modelled separately. The perspectives are then joint in a goal diagram that represents the Strategic Map. Finally, the map is **validated** along three dimensions: conflicts detection and resolution, minimisation of costs, and evaluation of unpredictable events. The result of the validation is the actual requirements system.

4.1 Strategy Modelling

The idea is that the Tropos early requirements phase is led by the emergence of the strategic map. The strategic map is represented as a Tropos goal diagram, so that BSC's cause-effect relations are replaced by Tropos relations. AND-decompositions are used, when multiple cause-effect relations exist, and there is the evidence that the decomposed goal can't be reached if at least one of the sub-goals is not achieved. OR-decompositions are used, when multiple cause-effect relations exist, but achieving one of the sub-goal is enough to consider achieved the parent goal. If there is no clear evidence of

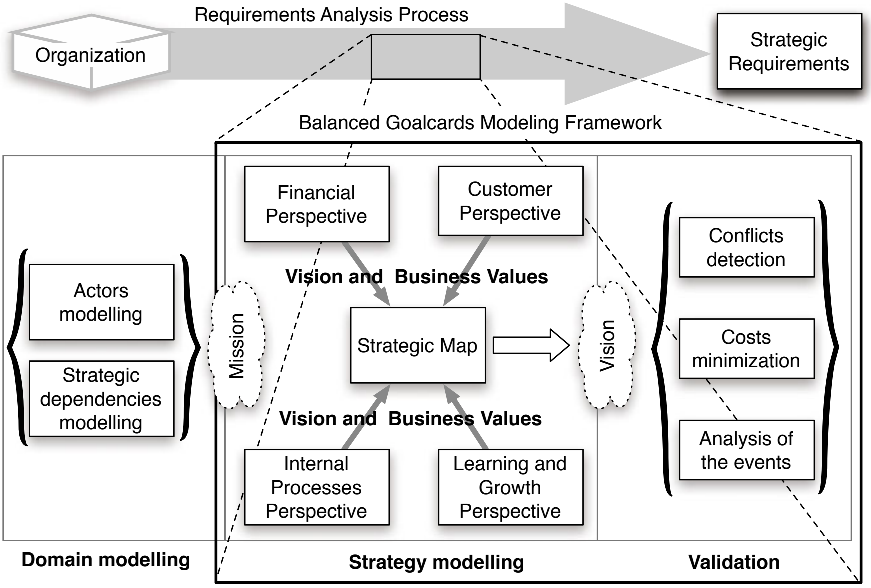


Fig. 1. The Balanced Goalcards methodological framework

a decomposition, a contribution link is used. Carefully evaluating the contribution metrics ("+", "−", "++" and "−−", see section 3 for the guidelines) allows the designer to describe in a more precise and realistic way the mutual influence among goals.

Mission. The concept of mission is mapped as the root goal. Frioli is a transportation company, so we have a root in the goal "Delivery service be fulfilled" as in Fig. 2(a). The root goal is further analysed by a decomposition into more operational goals ("Orders be received", "Goods be delivered" and so on). There could also be more that one root, if the company's business is wider.

Business Values. The business values are represented as softgoals. They emerge from both, an explicit indication of the organisation to be modelled, and the perception of the analyst. For instance, "Customer loyalty", "Timeliness", "Care corporate image", and so on (Fig. 2(b)) are business values. They are linked in weak cause-effect relations (represented as contributions in the picture). The "Long-term value for the associates" general-purpose goal is possibly reached directly, via a "Growth" of the company; or indirectly, via the "Customers loyalty" given by the "Quality of service". "Effectiveness" is related in particular with the inner processes of the company, whereas "Care corporate image" refers to how to company is perceived by the customer.

Vision. The vision represents the link between what the organisation is and what it wants to be. For the Frioli company, no *vision* has been clearly identified. This mainly because it is implicit in the managers' rationales, and the challenge is to capture it and make it explicit, so that we can elicit consistent requirements.

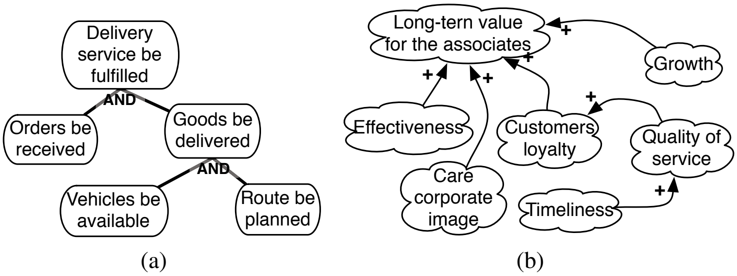


Fig. 2. (a) The mission of the company. (b) The business values. Ovals represent goals, and clouds represent softgoals.

Strategy. We want to make the strategy to emerge during the goal modelling, through the building and evaluation of the strategic map. We want to capture the business needs, and this means we need also to model the business profile of the company. In order to do this, we adopt here the classical BSC-based modelling approach. In detail, we build our goal model by taking into account the four perspective mentioned above: the economic-financial perspective, the customer perspective, the internal processes perspective and the learning and growth perspective.

Economic-financial Perspective. From the economic perspective, we observe an important effort of the Frioli company to contain costs (see Fig. 3). There are two kinds of costs: management costs and supply costs. Supply costs are intrinsic to the transport activity, such as fuel and tolls. On the other hand, management costs are related to the administration of the business; TLC are an important part, but also the extra payments the arise from unforeseens.

Customer Perspective. From this perspective, it is important to understand in which way a company can be attractive for the customer (Fig. 4). The overall image of the company ("Care corporate image") is important, as well as the details, such as the look of the personnel and of the documents. The communications with the customer ("Care the communications with customers") are important for the customers to be loyal to the company. Also, an important goal is to be able to offer an international transportation service. Even if not frequent, the lack of this service could affect the perception of the customer in the company's capabilities.

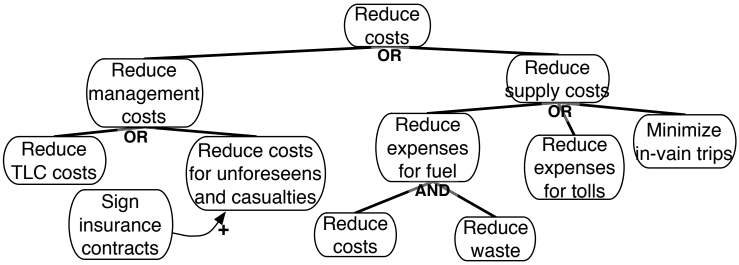


Fig. 3. The Economic-financial Perspective

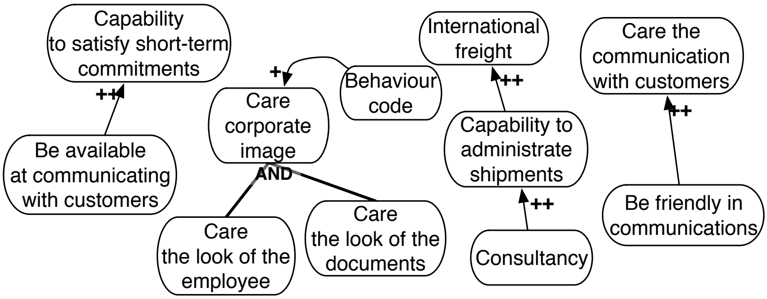

Capability to satisfy short-term commitments
++
Be available at communicating with customers
+
Care corporate image
Behaviour code
AND
Care the look of the employee
Care the look of the documents
International freight
++
Capability to administrate shipments
++
Consultancy
Care the communication with customers
++
Be friendly in communications

Fig. 4. The Customer Perspective

Internal Processes Perspective. Economic-financial goals and customer's strategy have to be translated into internal processes. Notice that we don't want to actually model the sequences of activities that form the company's business processes. What we want to capture here is the *why* of the processes. The processes shall allow the company to achieve its goals, so we model only the low-level goals that the internal activities are expected to fulfil (Fig. 5).

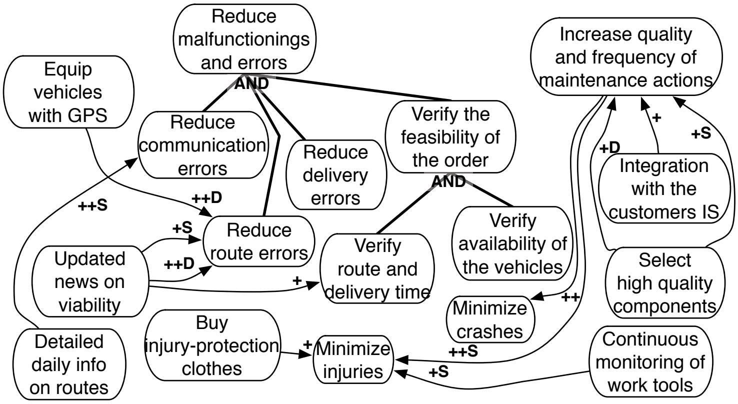

Reduce malfunctionings and errors
AND
Equip vehicles with GPS
Reduce communication errors
Reduce delivery errors
Verify the feasibility of the order
AND
Increase quality and frequency of maintenance actions
+D
+
+S
Integration with the customers IS
++S
++D
+S
Reduce route errors
Updated news on viability
++D
+
Verify route and delivery time
Verify availability of the vehicles
Select high quality components
++
Minimize crashes
Detailed daily info on routes
Buy injury-protection clothes
+
Minimize injuries
++S
+S
Continuous monitoring of work tools

Fig. 5. The Internal Processes Perspective

Learning and Growth Perspective. From this perspective, the company has a little margin of technical improvement. For instance, it could acquire new tools (for self-made reparations) or train the personnel (Fig. 6). Some other goals are related to the acquisition of new customers. More ambitious growth plans, such as investments in new market segments, are not present in the company.

4.2 Validation of the Strategic Map

The last step consists in putting together the perspectives and *balance* them into a consistent strategy. One of the strength points of the original BSC framework is its

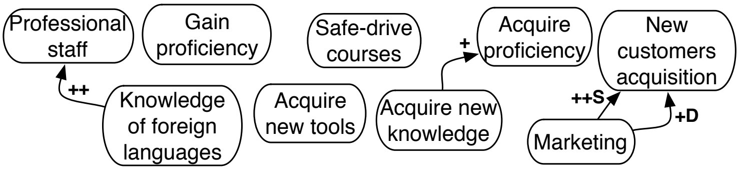


Fig. 6. The Learning and Growth Perspective

simplicity; using a goal diagram based approach causes the level of complexity to grow up (see Fig. 7), and this raises the need for formal analysis able verify and validate the models. The first problem (verification) can be solved with goal analysis techniques. The second problem (validation) is more complex, since it requires to align the models with economic principles. For this purpose, we have defined an analysis methodology comprised by three steps: conflicts resolution, costs minimisation and risk prevention.

Conflicts Resolution. A conflict is a situation where the satisfaction of some subgoals prevents (fully or partially) the top goals from being fulfilled. Such kind of situation is extremely dangerous because it can undermine the enforcement of the strategy. But it results difficult to be detected, specially when the strategic map becomes complex. The procedure that we use for conflicts detection and resolution is the following: *i*) we execute *backward reasoning* in order to find a possible values assignment for leaf nodes that satisfy all the top goals; *ii*) if the solution introduces a conflict for some of the top goal we start again backward reasoning for each of the conflicting goal; *iii*) the whole goal model is then modified accordingly to the partial results obtained in the previous step.

So for example, in Fig. 7, the goal "Verify the feasibility of the order", introduced in order to "Reduce malfunctioning and errors", caused a conflict with the goal "Reduce management costs". Since the company privileges the financial perspective over the internal processes one, the first goal has been discarded.

Costs Minimisation. Due to OR-decompositions and multiple contribution relations, different strategies can be adopted. To select the best one we recall the BSC fundamentals, arguing that each alternative has a different cost for the organisation. So we assign to each goal a numerical value, which represent its cost. If possible, we evaluate its monetary cost; so for instance the actual cost of the "Consultancy" can be accurately estimated. Otherwise, we search for a suitable metrics; for instance, for transport companies "timeliness" can be evaluated and translated into a numerical value. Costs are then associated to goals as meta tags, so that each possible strategy will have by this way a corresponding weigh in term of its resulting cost. The selection of the best strategy will be based on the comparison of that costs.

For instance, in Fig. 7, the goal "Customers loyalty" can be achieved either adopting a marketing strategy (e.g., promotional campaigns) or introducing a dedicated software able to reduce delivery errors. However, a new software can be very expensive and adopting marketing-based strategy could be more convenient.

Risk Prevention. Risk is something that heavily affects a company's life. Risky events are outside the control of an organisation and can prevent its strategy to be accomplished. We take into account this problem by introducing in the diagrams a new entity

- the “Event”. Events are linked via $--_S$ contribution relations to one or more goals. So, if an event occurs (i.e., its SAT value equals to P or F), then the affected goal is inhibited. We have no control over the occurrence of the event; however, by assigning the SAT value to the event, we can perform bottom-up analysis and see what is the potential effect of the event.

For example, in Fig. 7 the event “Crashes” can potentially compromise the whole long-term strategy having a negative impact on the reduction of costs.

5 Evaluation

In order to verify our approach, we have implemented a CASE tool, the B-GRTool (Balanced-GoalCards Tool). The tool has been implemented as an extension of the GR-Tool [9] and maintains all its functionalities, including reasoning techniques like forward and backward reasoning. Besides the standard GR-Tool scenarios, the B-GRTool supports views on single scenarios that are used to build the balanced perspectives. We used the tool to model the strategy for Frioli S.n.C. according to the approach shown in previous section. An almost complete goal model is illustrated in Fig. 7. It contains the mission and values of the company, together with the four perspectives; notice that during the validation phase, we have completed and refined the model by establishing

Table 1. Perspective-based comparison of four possible scenarios. The “S” columns contain Satisfiability values, whereas the “D” columns contain Deniability values.

	Scenario:	1		2		3		4	
		S	D	S	D	S	D	S	D
Values	Long-term value for the associates	P		P		P		P	
	Customer loyalty	P		P		P		P	
Economic perspective	Sign insurance contracts	P				T			
	Reduce management costs	P			P	T		P	
	Reduce supply costs	P		P		T		P	
	Minimize in-vain trips					T			
	Reduce costs for unforeseens and casualties	P			T	T			T
Customer's perspective	Capability to satisfy short-term commitments	P							
	Care the communication with customers	T					P		P
	Care corporate image	T		P	P		P		
	International freight	P		P			T		P
	Behavior code	T						T	
Internal processes perspective	Equip vehicles with GPS							T	
	Reduce malfunctioning and errors	P					T	P	
	New communication media	P		T		P		T	
	Improve customer-carrier-receiver comm.	T		P		P		P	
	Increase frequency of maintenance actions	P		P		P		P	
Learning and growth perspective	New customers acquisition	T		T		P		P	
	Acquire new tools	P		P		P		T	
	Marketing			T		P		P	
	Acquire new knowledge	P		T				P	

(P = “Partial”, T = “Total”)

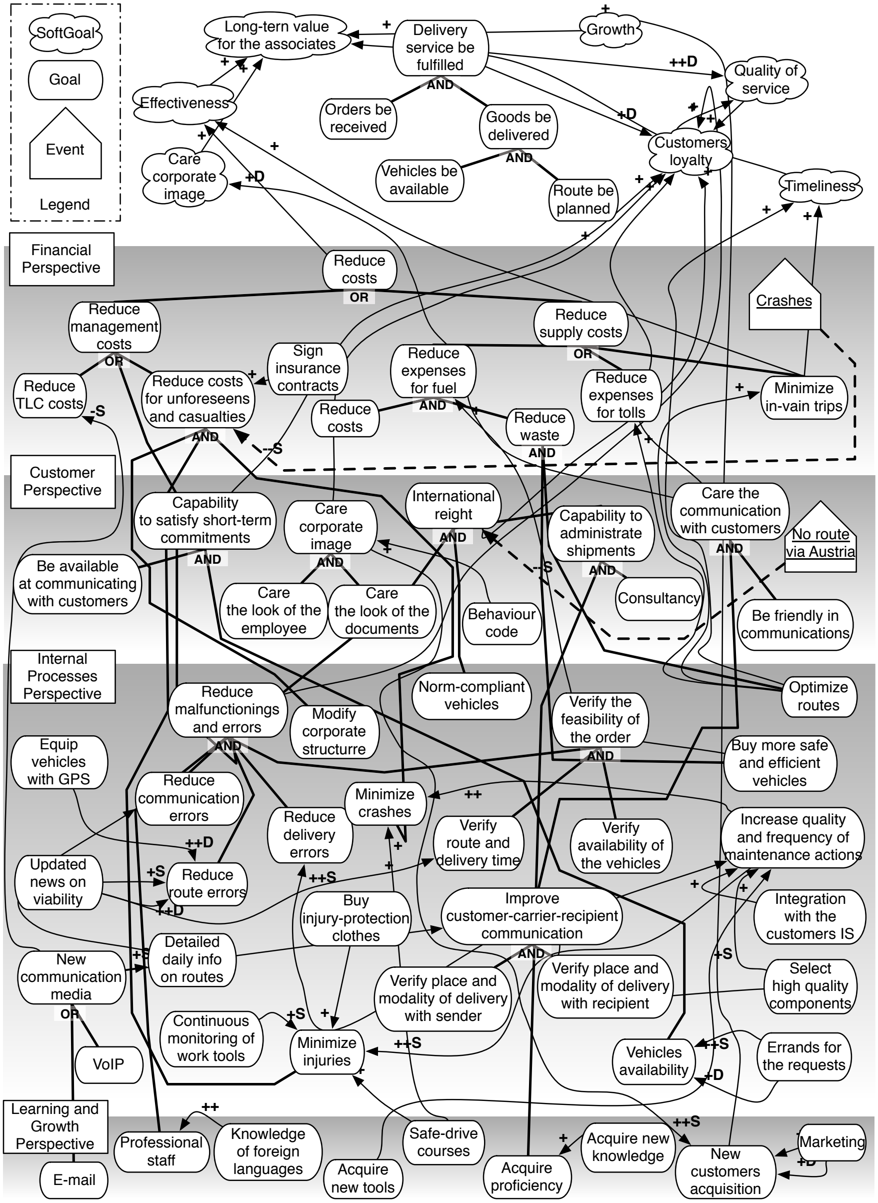

SoftGoal
Goal
Event
Legend
Long-tern value for the associates
Delivery service be fulfilled
Growth
Quality of service
Effectiveness
Orders be received
Goods be delivered
Customers loyalty
Care corporate image
Vehicles be available
Route be planned
Timeliness
Financial Perspective
Reduce costs
Crashes
Reduce management costs
Reduce supply costs
Sign insurance contracts
Reduce expenses for fuel
Reduce expenses for tolls
Minimize in-vain trips
Reduce TLC costs
Reduce costs for unforeseens and casualties
Reduce costs
Reduce waste
Customer Perspective
Capability to satisfy short-term commitments
Care corporate image
International reight
Capability to administrate shipments
Care the communication with customers
No route via Austria
Be available at communicating with customers
Care the look of the employee
Care the look of the documents
Behaviour code
Consultancy
Be friendly in communications
Internal Processes Perspective
Reduce malfunctionings and errors
Modify corporate structurre
Norm-compliant vehicles
Verify the feasibility of the order
Optimize routes
Equip vehicles with GPS
Buy more safe and efficient vehicles
Reduce communication errors
Minimize crashes
Reduce delivery errors
Verify route and delivery time
Verify availability of the vehicles
Increase quality and frequency of maintenance actions
Updated news on viability
Reduce route errors
Buy injury-protection clothes
Improve customer-carrier-recipient communication
Integration with the customers IS
Detailed daily info on routes
New communication media
Verify place and modality of delivery with sender
Verify place and modality of delivery with recipient
Select high quality components
Continuous monitoring of work tools
Minimize injuries
Vehicles availability
Errands for the requests
VoIP
Learning and Growth Perspective
Professional staff
Knowledge of foreign languages
Safe-drive courses
Acquire new knowledge
Marketing
New customers acquisition
E-mail
Acquire new tools
Acquire proficiency
AND
OR
+
++
+D
++D
+S
++S
-S
--S

Fig. 7. An (almost) complete goal model for the Frioli S.n.C. case study

further relations (contributions and decompositions). Table 1 shows the metrics that result from the goalcards; due to lack of space, only a subset of the goals can be shown. The four scenarios correspond to alternative strategies, each of which gives different priorities to different goals.

Scenario 1: The Current Strategy and Business Values of the Company. In this case, some goals are arbitrarily considered more important than others. For instance, it is extremely important the "Customer loyalty" obtained offering a reliable service. This requires a particular attention to "Care corporate image" and to "Care the communication with customers".

Scenario 2: A Growth-oriented Strategy. The focus of the analysis is on goals such as "Acquire new tools" or "New customers acquisition". What we obtain is a strategy that privileges the growth, but goals such as "Reduce management costs" and "Reduce costs for unforeseens and casualties" are denied. If a company wants to grow, it is extremely difficult to reduce at the same time the costs.

Scenario 3: A Costs-reduction Strategy. In this case, the focus is on the economic and the internal processes perspective, and particularly on goals such as "Reduce management costs" or "Reduce expenses for fuel". The resulting strategy allows the company to satisfy all goals, but suggests to abandon the international freight, and some non-essential goals are denied.

Scenario 4: An Innovation-oriented Strategy. In this case the learning and growth perspective has again a relevant influence on the strategy, but caring at the same time the internal processes one. The resulting strategy is similar to the one of scenario 2, but it is now possible to contain costs.

Some interesting results come from the case study. Despite their claim of the "Growth" as a business value, the actual strategy does not reflect such a will. It is possible to see in Fig. 7 that the learning and growth perspective has a few number of goals. The scenarios above also confirm this perception. So we observe that, despite the fact that the company wants to grow, its implicit values do actually privilege stability. This observation is reinforced by a look at the customer perspective: it has an important impact on the realisation of the business values. In particular, we see that many goals exist in order to satisfy the "Customer loyalty" soft-goal. Through the customer loyalty, the general-purpose "Long-term value for the associates" is intended to be reached. Thus, the company seems to have a well-established relation with customers, and wants to keep it, without going further into market risks. So the resulting requirements system should privilege this *status quo* arrangement.

6 Conclusions

The importance of business criteria is explicitly recognised in particular in the e-Commerce engineering, where value exchanges play a role in understanding and generating requirements for the system [10]. Also, in e-Business, some approaches exist, which focus on the need for the alignment of IT and business strategy [2,11]. In this paper we have presented a new methodological framework for modelling requirements and validating them against a business strategy: goal graphs are used to represent the strategic map, and the economic metrics are associated to goals' satisfiability and deniability; this allows to reason on the metrics and analyse the diagrams, building balanced requirements systems. We have reported the use of the framework in our experience with a transport company, trying to evaluate the results by comparing different scenarios and estimating the effectiveness gained in gathering requirements.

References

1. Alencar, F., Castro, J., Cysneiros, L., Mylopoulos, J.: From early requirements modeled by the i* technique to later requirements modeled in precise UML. In: Anais do III Workshop em Engenharia de Requisitos, Rio de Janeiro, Brazil, pp. 92–109 (2000)
2. Bleistein, S.J., Aurum, A., Cox, K., Ray, P.K.: Strategy-oriented alignment in requirements engineering: Linking business strategy to requirements of e-business systems using the soare approach, vol. 36 (2004)
3. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J., Perini, A.: Tropos: An agent-oriented software development methodology. Journal of Autonomous Agents and Multi-Agent Systems 8(8), 203–236 (2004)
4. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming 20(1-2), 3–50 (1993)
5. Delor, E., Darimont, R., Rifaut, A.: Software quality starts with the modelling of goal-oriented requirements. In: 16th International Conference Software & Systems Engineering and their Applications (2003)
6. Ebert, C.: Requirements before the requirements: Understanding the upstream impact. In: RE 2005: Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE 2005), Washington, DC, USA, pp. 117–124. IEEE Computer Society, Los Alamitos (2005)
7. Fuxman, A., Giorgini, P., Kolp, M., Mylopoulos, J.: Information systems as social structures. In: Second International Conference on Formal Ontologies for Information Systems (FOIS-2001), Ogunquit, USA (2001)
8. Giorgini, P., Mylopoulos, J., Nicchiarelli, E., Sebastiani, R.: Reasoning with goal models. In: Spaccapietra, S., March, S.T., Kambayashi, Y. (eds.) ER 2002. LNCS, vol. 2503, p. 167. Springer, Heidelberg (2002)
9. Giorgini, P., Mylopoulos, J., Sebastiani, R.: Goal-oriented requirements analysis and reasoning in the tropos methodology. Engineering Applications of Artifcial Intelligence 18/2 (2005)
10. Gordijn, J., Akkermans, H.: Value based requirements engineering: Exploring innovative e-commerce idea. Requirements Engineering Journal 8(2), 114–134 (2003)
11. Grembergen, W., Saull, R.: Aligning business and information technology through the balanced scorecard at a major canadian financial group: It's status measured with an it bsc maturity model. In: HICSS 2001: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34), Washington, DC, USA, vol. 8, p. 8061. IEEE Computer Society, Los Alamitos (2001)
12. Kaplan, R., Norton, D.P.: The Balanced Scorecard: Translating Strategy into Action. Harvard Business School Press, Massachusetts (1996)
13. Kaplan, R.S., Norton, D.P.: The strategy-focused organization. how balanced scorecard companies thrive in the new business environment. Harvard Business School Press, Boston (2001)
14. Niehaves, B., Stirna, J.: Participative enterprise modelling for balanced scorecard implementation. In: Ljunberg, J.A.M., et al. (eds.) The Fourteenth European Conference on Information Systems, Goteborg, pp. 286–298 (2006)
15. Norton, D., Kaplan, R.: The balanced scorecard: measures that drive performance. Harvard Business Review 70(1) (1992)
16. Porter, M.E.: What is strategy? Harvard Business Review 74(6), 61–78 (1996)
17. Rolland, C.: Reasoning with goals to engineer requirements. In: 5th International Conference on Enterprise Information Systems, Angers, France, April 22-26 (2003)

Using Fault Screeners for Software Error Detection*

Rui Abreu, Alberto González, Peter Zoeteweij, and Arjan J.C. van Gemund

Software Engineering Research Group, Delft University of Technology
P.O. Box 5031, NL-2600 GA Delft, The Netherlands
{r.f.abreu,a.gonzalezsanchez,p.zoeteweij}@tudelft.nl,
a.j.c.vangemund@tudelft.nl

Abstract. Fault screeners are simple software (or hardware) constructs that detect variable value errors based on unary invariant checking. In this paper we evaluate and compare the performance of three low-cost screeners (Bloom filter, bitmask, and range screener) that can be automatically integrated within a program, and trained during the testing phase. While the Bloom filter has the capacity of retaining virtually all variable values associated with proper program execution, this property comes with a much higher false positive rate per unit of training effort, compared to the more simple range and bitmask screeners, that compresses all value information in terms of a single lower and upper bound or a bitmask, respectively. We present a novel analytic model that predicts the false positive and false negative rate for ideal (i.e., screeners that store each individual value during training) and simple (e.g., range and bitmask) screeners. We show that the model agrees with our empirical findings. Furthermore, we describe an application of all screeners, where the screener's error detection output is used as input to a fault localization process that provides automatic feedback on the location of residual program defects during deployment in the field.

Keywords: Error detection, Program invariants, Analytic model, Fault localization, Program spectra.

1 Introduction

In many domains such as consumer products the residual defect rate of software is considerable, due to the trade-off between reliability on the one hand and development cost and time-to-market on the other. Proper *error detection* is a critical factor in successfully recognizing, and coping with (recovering from) failures during the *deployment phase* [25,19]. Even more than during testing at the *development phase*, errors may otherwise go unnoticed, possibly resulting in catastrophic later on.

Error detection is typically implemented through tests (invariants) that usually trigger some exception handling process. The invariants range from *application-specific* (e.g., a user-programmed test to assert that two state variables in two different components are in sync) to *generic* (e.g., a compiler-generated value range check). While application-specific invariants cover many failures anticipated by the programmer and have a low

* This work has been carried out as part of the TRADER project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the BSIK03021 program.

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 60–74, 2010.

false positive and false negative rate[1], their (manual) integration within the code is typically a costly, and error-prone process. Despite the simplicity of generic invariants, and their higher false positive and false negative rates, they can be *automatically* generated within the code, while their application-specific *training* can also be *automatically* performed as integral part of the testing process during the development phase. Furthermore, generic invariants correlate to some extent with application-specific invariants. Consequently, violation of the latter is typically preluded by violation of the former type [8].

In view of the above, attractive properties, generic invariants, often dubbed fault screeners, have long been subject of study in both the software and the hardware domain (see Section 6). Examples include value screeners such as simple bitmask [14,27] and range screeners [14,27], and more sophisticated screeners such as Bloom filters [14,27]. In most work the screeners are used for automatic fault detection [1] and fault localization [14,26].

In all the above work, the performance of screeners is evaluated empirically. While empirical information is invaluable, no analytical performance models are available that explain why certain screeners outperform other screeners. For example, it is known that, compared to a (sophisticated) Bloom filter, a simple range screener takes less effort to train, but has worse detection performance (higher false negative rate). Up to now there has been no modeling effort that supports these empirical findings.

In this paper we analytically and empirically investigate the performance of screeners. In particular, we make the following contributions:

- We develop a simple, approximate, analytical performance model that predicts the false positive and false negative rates in terms of the variable domain size and training effort. We derive a model for (ideal) screeners that store each individual value during training, one for bitmask screeners that express all observed values in terms of a bit array, and another model for range screeners that compress all training information in terms of a single range interval.
- We evaluate the performance of Bloom filters, bitmask, and range screeners based on instrumenting them within the Siemens benchmark suite, which comprises a large set of program versions, of which a subset is seeded with faults. We show that our empirical findings are in agreement with our model.
- As a typical application of screeners, we show how the Bloom filter, bitmask, and range screeners are applied as input for automatic fault localization, namely spectrum-based fault localization (SFL). It is shown that the resulting fault localization accuracy is comparable to one that is traditionally achieved at the design (testing) phase, namely for either Bloom filter or range screeners.

The significance of the latter result is that no costly, application-specific modeling is required for diagnostic purposes, paving the way for truly automatic program debugging.

The paper is organized as follows. In the next section we introduce the Bloom filter, bitmask and range screeners. In Section 3 the experimental setup is described and the empirical results are discussed. Section 4 presents our analytical performance model

[1] An error flagged when there is none is called false positive, while missing an error is called false negative.

to explain the experimental results. The application of screeners as input for SFL is discussed in Section 5. A comparison to related work appears in Section 6. Section 7 concludes the paper.

2 Fault Screeners

Program invariants, first introduced by Ernst *et al.* [8] with the purpose of supporting program evolution, are conditions that have to be met by the state of the program for it to be correct. Many kinds of program invariants have been proposed in the past [8,9,27]. In this paper, we focus on dynamic range invariants [27], bitmask invariants [14,27], and Bloom filter invariants [27]. Besides being generic, they require minimal overhead (lending themselves well for application within resource-constrained environments, such as embedded systems).

A *bitmask invariant* is composed of two fields: the first observed value (*fst*) and a bitmask (msk) representing the activated bits (initially all bits are set to 1). Every time a new value v is observed, it is checked against the currently valid msk according to:

$$violation = (v \oplus \mathit{fst}) \wedge msk \tag{1}$$

where $\oplus$ and $\wedge$ are the bitwise *xor* and *and* operators respectively. If the $violation$ is non-zero, an invariant violation is reported. In error detection mode (operational phase) an error is flagged. During training mode (development phase) the invariant is updated according to:

$$msk := \neg(v \oplus \mathit{fst}) \wedge msk \tag{2}$$

Although bitmask invariants were used with success by Hangal and Lam [14], they have limitations. First of all, their support for representing negative and floating point numbers is limited. Finally, the upper bound representation of an observed number is far from tight. To overcome these problems, we also consider *range invariants*, e.g., used by Racunas *et al.* in their hardware perturbation screener [27].

Range invariants are used to represent the (integer or real) bounds of a program variable. Every time a new value v is observed, it is checked against the currently valid lower bound l and upper bound u according to

$$violation = \neg(l < v < u) \tag{3}$$

If v is outside the bounds, an error is flagged in error detection mode (deployment phase), whereas in training mode (development phase) the range is extended according to the assignment

$$l := \min(l, v) \tag{4}$$

$$u := \max(l, v) \tag{5}$$

Bloom filters [4] are a space-efficient probabilistic data structure used to check if an element is a member of a set. This screener is stricter than the range screeners, as it is basically a compact representation of a variable's entire history.

All variables share the same Bloom filter, which is essentially a bit array (64*KB*, the size of the filter could be decreased by using a backup filter to prevent saturation [27]). Each 32-bit value `v` and instruction address `ia` are merged into a single 32-bit number `g`:

$$g = (v * 2^{16}) \vee (0xFFFF \wedge ia) \tag{6}$$

where $\vee$ and $\wedge$ are bitwise operators, respectively. This number `g` is used as input to two hash functions (h_1 and h_2), which index into the Bloom filter `b`. In detection mode an error is flagged according to

$$violation = \neg(b[h_1(g)] \wedge b[h_2(g)]) \tag{7}$$

During training mode, the outputs of the hash functions are used to update the Bloom filter according to the assignment

$$b[h_1(g)] := 1 \tag{8}$$

$$b[h_2(g)] := 1 \tag{9}$$

3 Experiments

In this section the experimental setup is presented, namely the benchmark set of programs, the workflow of the experiments, and the evaluation metrics. Finally, the experimental results are discussed.

3.1 Experimental Setup

Benchmark Set. In our study, we use a set of test programs known as the *Siemens set* [16]. The Siemens set is composed of seven programs. Every single program has a correct version and a set of faulty versions of the same program. The correct version can be used as reference version. Each faulty version contains exactly one fault. Each program also has a set of inputs that ensures full code coverage. Table 1 provides more information about the programs in the package (for more information see [16]). Although the Siemens set was not assembled with the purpose of testing fault diagnosis and/or error detection techniques, it is typically used by the research community as the set of programs to test their techniques.

Table 1. Set of programs used in the experiments

Program	Faulty Versions	LOC	Test Cases	Description
`print_tokens`	7	539	4130	Lexical Analyzer
`print_tokens2`	10	489	4115	Lexical Analyzer
`replace`	32	507	5542	Pattern Recognition
`schedule`	9	397	2650	Priority Scheduler
`schedule2`	10	299	2710	Priority Scheduler
`tcas`	41	174	1608	Altitude Separation
`tot_info`	23	398	1052	Information Measure

In total the Siemens set provides 132 programs. However, as no failures are observed in two of these programs, namely version 9 of `schedule2` and version 32 of `replace`, they are discarded. Besides, we also discard versions 4 and 6 of `print_tokens` because the faults in this versions are in global variables and the profiling tool used in our experiments does not log the execution of these statements. In summary, we discarded 4 versions out of 132 provided by the suite, using 128 versions in our experiments.

Workflow of Experiments. Our approach to study the performance of fault screeners as error detectors in the deployment phase comprises three stages. First, the target program is instrumented to generate program spectra (used by the fault localization technique, see Section 5) and to execute the invariants (see Figure 1). To prevent faulty programs to corrupt the logged information, the program invariants and spectra themselves are located in an external component ("Screener"). The instrumentation process is implemented as an optimization pass for the LLVM tool [20] in C++ (for details on the instrumentation process see [11]). The program points screened are all memory loads/stores, and function argument and return values.

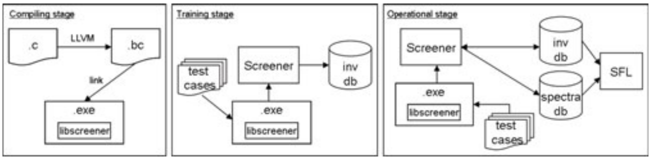


Fig. 1. Workflow of experiments

Second, the program is run for those test cases for which the program passes (its output equals that of the reference version), in which the screeners are operated in training mode. The number of (correct) test cases used to train the screeners is of great importance to the performance of the error detectors at the deployment (detection) phase. In the experiments this number is varied between 5% and 100% of all correct cases (134 and 2666 cases on average, respectively) in order to evaluate the effect of training.

Finally, we execute the program over all test cases (excluding the previous training set), in which the screeners are executed in detection mode.

Error Detection Evaluation Metrics. We evaluate the error detection performance of the fault screeners by comparing their output to the pass/fail outcome per program over the entire benchmark set. The ("correct") pass/fail information is obtained by comparing the output of the faulty program with the reference program.

Let N_P and N_F be the size of the set of passed and failed runs, respectively, and let F_p and F_n be the number of false positives and negatives, respectively. We measure the false positive rate f_p and the false negative rate f_p according to

$$f_p = \frac{F_p}{N_P} \tag{10}$$

$$f_n = \frac{F_n}{N_F} \tag{11}$$

3.2 Results

Figure 2 plots f_p and f_n in percents for bitmask (*msk*), range (*rng*), and Bloom filter (*bloom*) screeners for different percentages of (correct) test cases used to train the screeners, when instrumenting all program points in the program under analysis. The plots represent the average over all programs, which has negligible variance (between $0 - 0.2\%$ and $3 - 5\%$, for f_p and f_n, respectively). From the figure, the following conclusions can be drawn for f_p: the more test cases used to train the screeners, the lower f_p (as screeners evolve with the learning process). In addition, it can be seen that Bloom filter screeners learn slower than the range screener, which in turn learn slower than bitmask screeners. Furthermore, for all screeners f_n rapidly increases, meaning that even after minimal training many errors are already tolerated. This is due to:

- limited detection capabilities: only either single upper/lower bounds or a compact representation of the observed values are stored are screened, i.e., simple and invariants, in contrast to the host of invariants conceivable, based on complex relationships between multiple variables (typically found in application-specific invariants)
- program-specific properties: certain variables exhibit the same values for passed and failed runs, see Section 4. Those cases lead to false negatives.
- limited training accuracy: although the plots indicate that the *quantity* of pass/fail training input is sufficient, the *quality* of the input is inherently limited. In a number of cases a (faulty) variable error does not result in a failure (i.e., a different output than the correct reference program). Consequently, the screener is trained to accept the error, thus limiting its detection sensitivity.

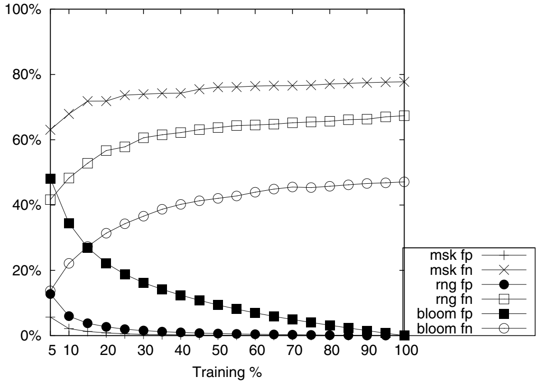


Fig. 2. Fsalse positives and negatives on average

Due to its strictness, Bloom filter screeners have on the one hand lower f_n than range screeners. On the other, this strictness increases f_p. In the next section we provide a more theoretic explanation for the observed phenomena.

Because of their simplicity, the evaluated screeners entail minimal computational overhead. On average, the 494 (0.40 cov[1]) program points screened introduced an

[1] Coefficient of variance (standard deviation divided by mean).

overhead of 14.2% (0.33% cov) for the range screener, and 46.2% (0.15% cov) was measured for the Bloom filter screener (when all program variable loads/stores and function argument/returns are screened).

4 Analytic Model

In this section we present our analytic screening performance model. First, we derive some main properties that apply without considering the particular properties that (simple) screeners exhibit. Next we present a performance model for the bitmask screening. Finally, we focus on the range screener, which is a typical example of a simple, yet powerful screener, and which is amongst the screeners evaluated.

4.1 Concepts and Definitions

Consider a particular program variable x. Let P denote the set of values x takes in all N_P passing runs, and let F denotes the set of values x takes in all N_F failing runs. Let T denote the set of values recorded during training. Let $|P|, |F|, |T|$ denote the set sizes, respectively. Screener performance can generally be analyzed by considering the relationship between the three sets P, F, and T as depicted in Fig. 3. In the figure we distinguish between five regions, numbered 1 through 5, all of which associate with false positives (fp), false negatives (fn), true positives (tp), and true negatives (tn). For example, values of x which are within P (i.e., OK values) but which are (still) outside of the training set T, will trigger a false positive (region 1). Region 3 represents the fact that certain values of x may occur in both passing runs, as well as failing runs, leading to potential false negatives. Region 4 relates to the fact that for many simple screeners the update due to training with a certain OK value (e.g., in region 2) may also lead to acceptance of values that are exclusively associated with failed runs, leading to false negatives (e.g., an upper bound 10, widened to 15 due to $x = 15$, while $x = 13$ is associated with a failed run).

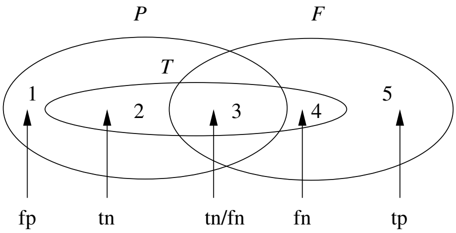


Fig. 3. Distribution of variable x

4.2 Ideal Screening

In the following we derive general properties of the evolution of the false positive rate f_p and the false negative rate f_n as training progresses. For each new value of x in a passing run the probability p that x represents a value that is not already trained equals

$$p = \frac{|P| - |T|}{|P|} = 1 - \frac{|T|}{|P|} \tag{12}$$

Note that for ideal screeners region 4 does not exist. Hence T grows entirely within P. Consequently, the expected growth of the training set is given by

$$t_k - t_{k-1} = p_{k-1} \tag{13}$$

where t_k denotes the expected value of $|T|$, $\mathsf{E}[|T|]$, at training step k, and p_k denotes the probability p at step k. It follows that t_k is given by the recurrence relation

$$t_k = \alpha \cdot t_{k-1} + 1 \tag{14}$$

where $\alpha = 1 - 1/|P|$. The solution to this recurrence relation is given by

$$t_k = \frac{\alpha^k - 1}{\alpha - 1} \tag{15}$$

Consequently

$$\mathsf{E}[|T|] = |P| \cdot \left(1 - (1 - \frac{1}{|P|})^k\right) \tag{16}$$

Thus the fraction of T within P initially increases linearly with k, approaching P in the limit for $k \to \infty$.

Since in detection mode the false positive rate f_p equals p, from (12) it follows

$$f_p = (1 - \frac{1}{|P|})^k \tag{17}$$

Thus the false positive rate decreases with k, approaching a particular threshold after a training effort k that is (approximately) *proportional* to $|P|$. As the false negative rate is proportional to the part of T that intersects with F (region 3) it follows that f_n is proportional to the growth of T according to

$$f_n = f \cdot \left(1 - (1 - \frac{1}{|P|})^k\right) \tag{18}$$

where f denotes the fraction of P that intersects with F. Thus the false negative rate increases with k, approaching f in the limit when T equals P. From the above it follows

$$f_n = f \cdot (1 - f_p) \tag{19}$$

4.3 Bitmask Screening

In the following we introduce the constraint that the entire value domain of a variable is compressed in terms of a bitmask. Let msk be a bit array with a indices. Without loss of generality, let $p_i = p$ be the probability that the bit in index i equals 0 after a value is observed. The expected number H of indices set to 1 (aka Hamming weight) after that observation follows a binomial distribution, and amounts to $\mathsf{E}[H] = (1-p) \cdot a$. Thus, msk has the following expected number of 1's after k observations

$$\mathsf{E}[H]_k = (1-p^k) \cdot a \tag{20}$$

Consequently,

$$\mathsf{E}[|T|]_k = 2^{(1-p^k)\cdot a} \tag{21}$$

Note that every time a bit is flipped in msk, the number of accepted values doubles. From (12) it follows that

$$f_p = 1 - \frac{2^{(1-p^k)\cdot a}}{|P|} \tag{22}$$

Thus the false positive rate decreases exponentially with k, approaching a particular threshold after a training effort k that is (approximately) *proportional* to $|P|$. The analysis of f_n is similar to the previous section with the modification that for simple screeners such as the bitmask screener the fraction f' of T that intersects with F is generally greater than the fraction f for ideal screeners (regions 3 and 4, as explained earlier). Thus,

$$f_n = f' \cdot (1 - f_p) = f' \cdot \frac{2^{(1-p^k)\cdot a}}{|P|} > f \cdot \frac{2^{(1-p^k)\cdot a}}{|P|} \tag{23}$$

4.4 Range Screening

In the following we introduce the constraint that the entire value domain of variable x available for storage is compressed in terms of only one range, coded in terms of two values l (lower bound) and u (upper bound). Despite the potential negative impact on f_p and f_n we show that the training effort required for a particular performance is *independent* of the entire value domain, *unlike* the two previous screeners.

After training with k values, the range screener bounds have evolved to

$$l_k = \min_{i=1,\ldots,k} x_i \tag{24}$$

$$u_k = \max_{i=1,\ldots,k} x_i \tag{25}$$

Since x_i are samples of x, it follows that l_k and u_k are essentially the lowest and highest *order statistic* [7], respectively, of the sequence of k variates taken from the (pseudo) random variable x with a particular probability density function (pdf). The order statistics interpretation allows a straightforward performance analysis when the pdf of x is known. In the following we treat two cases.

Uniform Distribution. Without loss of generality, let x be distributed according to a uniform pdf between 0 and r (e.g., a uniformly distributed index variable with some upper bound r). From, e.g., [7] it follows that the expected values of l_k and u_k are given by

$$\mathsf{E}[l_k] = \frac{1}{k+1} \cdot r \tag{26}$$

$$\mathsf{E}[u_k] = \frac{k}{k+1} \cdot r \tag{27}$$

Consequently,

$$\mathsf{E}[|T|] = \mathsf{E}[u_k] - \mathsf{E}[l_k] = \frac{k-1}{k+1} \cdot r \tag{28}$$

Since $|P| = r$, from (12) it follows ($f_p = p$) that

$$f_p = 1 - \frac{k-1}{k+1} = \frac{2}{k+1} \tag{29}$$

The analysis of f_n is similar to the previous section, thus

$$f_n = f' \cdot (1 - f_p) = f' \cdot \frac{k-1}{k+1} > f \cdot \frac{k-1}{k+1} \tag{30}$$

Normal Distribution. Let x be distributed according to a normal pdf with zero mean and variance σ (many variables such as loop bounds are measured to have a near-normal distribution over a series of runs with different input sets [10]). From, e.g., [12] it follows that the expected values of l_k and u_k are given by the approximation (asymptotically correct for large k)

$$\mathsf{E}[l_k] = \sigma \cdot \sqrt{2 \cdot \log(0.4 \cdot k)} \tag{31}$$

$$\mathsf{E}[u_k] = -\sigma \cdot \sqrt{2 \cdot \log(0.4 \cdot k)} \tag{32}$$

Consequently,

$$\mathsf{E}[|T|] = \mathsf{E}[u_k] - \mathsf{E}[l_k] = 2 \cdot \sigma \cdot \sqrt{2 \cdot \log(0.4 \cdot k)} \tag{33}$$

The false positive rate equals the fraction of the normal distribution (P) not covered by T. In terms of the normal distribution's cumulative density function (cdf) it follows

$$f_p = 1 - \text{erf}\, \frac{\sigma \cdot \sqrt{2 \cdot \log(0.4 \cdot k)}}{\sigma \cdot \sqrt{2}} \tag{34}$$

which reduces to

$$f_p = 1 - \text{erf}\, \sqrt{\log(0.4 \cdot k)} \tag{35}$$

Note that, again, f_p is independent of the variance of the distribution of x. For the false negative rate it follows

$$f_n = f' \cdot (1 - f_p) = f' \cdot \text{erf}\, \sqrt{\log(0.4 \cdot k)} \tag{36}$$

4.5 Discussion

Both the result for uniform and normal distributions show that the use of range screeners implies that the false positive rate (and, similarly, the false negative rate) can be optimized *independent* of the size of the value domain. Since the value domain of x can be very large this means that range screeners require much less training than "ideal" screeners to attain bounds that are close to the bounds of P. Rather than increasing one value at a time by "ideal" screeners, range screeners can "jump" to a much greater range at a single training instance. The associated order statistics show that $|T|$ approaches $|P|$ regardless their absolute size. For limited domains such as in the case of the uniform pdf the bounds grow very quickly. In the case of the normal pdf the bounds grow less quickly. Nevertheless, according to the model a 1 percent false positive rate can be attained for only a few thousand training runs (few hundred in the uniform case). Although bitmask screeners are dependent on the size of variable x, they learn much faster than range and "ideal" screeners. This is due to the fact that every time a bit is flipped in the bitmask, the number of accepted values doubles.

The model is in good agreement with our empirical findings (see Figure 2). While exhibiting better f_n performance, the Bloom filter suffers from a less steep learning curve (f_p) compared to the range screener, which has a higher f_p rate if compared to the bitmask screener. Although it might seem that even the Bloom filter has acceptable performance near the 100 percent mark, this is due to an artifact of the measurement setup. For 100 percent training there are no passing runs available for the evaluation (detection) phase, meaning that there will never be a (correct) value presented to the screener that it has not already been seen during training. Consequently, for the 100 percent mark f_p is zero by definition, which implies that in reality the Bloom filter is g expected to exhibit still a non-zero false positive rate after 2666 test cases (in agreement with the model). In contrast, for the range/bitmask screener it is clearly seen that even for 1066 tests f_p is already virtually zero (again, in agreement with the model).

5 Fault Screening and SFL

In this section we evaluate the performance of the studied fault screeners as error detector input for automatic fault localization tools. Although many fault localization tools exist [5,6,18,21,28,30], in this paper we use spectrum-based fault localization (SFL) because it is known to be among the best techniques [18,21].

In SFL, program runs are captured in terms of a spectrum. A program spectrum [15] can be seen as a projection of the execution trace that shows which parts (e.g., blocks, statements, or even paths) of the program were active during its execution (a so-called "hit spectrum"). In this paper we consider a program part to be a statement. Diagnosis consists in identifying the part whose activation pattern resembles the occurrences of errors in different executions. This degree of similarity is calculated using *similarity coefficients* taken from data clustering techniques [17]. Amongst the best similarity coefficients for SFL is the Ochiai coefficient [3,1,2]. The output of SFL is a ranked list of parts (statements) in order of likelihood to be at fault.

Given that the output of SFL is a ranked list of statements in order of likelihood to be at fault, we define quality of the diagnosis q_d as $1 - (p/(N - 1))$, where p is the position of the faulty statement in the ranking, and N the total number of statements, i.e., the number of statements that need not be inspected when following the ranking in searching for the fault. If there are more statements with the same coefficient, p is then the average ranking position for all of them (see [1] for a more elaborate definition).

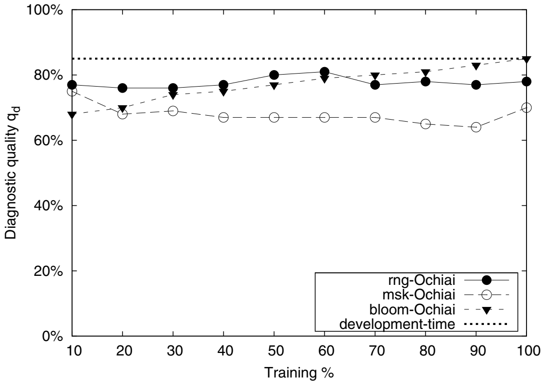


Fig. 4. Diagnostic quality q_d on average

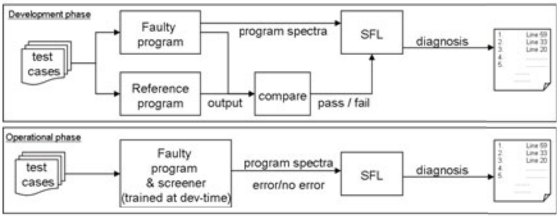


Fig. 5. Screener-SFL vs. reference-based SFL

Figure 4 plots q_d for SFL using the three screeners versus the training percentage as used in Figure 2. From the figure, we conclude that the bitmask screener is the worst performing one. In general, the performance of bloom filter and range screeners is similar. The higher f_n of the range screener is compensated by its lower f_p, compared to the Bloom filter screener. The best q_d, 81% for the range screener is obtained for 50% training, whereas the Bloom filter screener has its best 85% performance for 100% (although this is due to an artifact of the measurement setup as explained in the previous section). From this, we can conclude that, despite its slower learning curve, the Bloom filter screener can outperform the range screener if massive amounts of data are available for training (f_p becomes acceptable). On the other hand, for those situations where only a few test cases are available, it is better to use the range screener. Comparing the screener-SFL performance with SFL at development-time (85% on average [2], see Figure 5), we conclude that the use of screeners in an operational (deployment) context yields comparable diagnostic accuracy to using pass/fail information available in the testing phase. As shown in [1] this is due to the fact that the quantity of error information compensates the limited quality.

Due to their small overhead ([1,3], see also Section 3.2), fault screeners and SFL are attractive for being used as error detectors and fault localization, respectively.

6 Related Work

Dynamic program invariants have been subject of study by many researchers for different purposes, such as program evolution [8,9,29], fault detection [27], and fault localization [14,26]. More recently, they have been used as error detection input for fault localization techniques, namely SFL [1].

Daikon [9] is a dynamic and automatic invariant detector tool for several programming languages, and built with the intention of supporting program evolution, by helping programmers to understand the code. It stores program invariants for several program points, such as call parameters, return values, and for relationships between variables. Examples of stored invariants are constant, non-zero, range, relationships, containment, and ordering. Besides, it can be extended with user-specified invariants. Carrot [26] is a lightweight version of Daikon, that uses a smaller set of invariants (equality, sum, and order). Carrot tries to use program invariants to pinpoint the faulty locations directly. Similarly to our experiments, the Siemens set is also used to test Carrot. Due to the negative results reported, it has been hypothesized that program invariants alone may not be suitable for debugging. DIDUCE [14] uses dynamic bitmask invariants for pinpointing software bugs in Java programs. Essentially, it stores program invariants for the same program points as in this paper. It was tested on four real world applications yielding "useful" results. However, the error detected in the experiments was caused by a variable whose value was constant throughout the training mode and that changed in the deployment phase (hence, easy to detect using the bitmask screener). In [27] several screeners are evaluated to detect hardware faults. Evaluated screeners include dynamic ranges, bitmasks, TLB misses, and Bloom filters. The authors concluded that bitmask screeners perform slightly better than range and Bloom filter screeners. However, the (hardware) errors used to test the screeners constitute random bit errors which, although ideal for bitmask screeners, hardly occur in program variables. IODINE [13] is a framework for extracting dynamic invariants for hardware designs. In has been shown that dynamic invariant detection can infer relevant and accurate properties, such as request-acknowledge pairs and mutual exclusion between signals.

To the best of our knowledge, none of the previous work has analytically modeled the performance of the screeners, nor evaluated their use in an automatic debugging context.

7 Conclusions and Future Work

In this paper we have analytically and empirically investigated the performance of low-cost, generic program invariants (also known as "screeners"), namely range and Bloom-filter invariants, in their capacity of error detectors. Empirical results show that near-"ideal" screeners, of which the Bloom filter screener is an example, are slower learners than range invariants, but have less false negatives. As major contribution, we present a novel, approximate, analytical model to explain the fault screener performance. The model shows that the training effort required by near-"ideal" screeners, such as Bloom filters, increases with the variable domain size, whereas simple screeners, such as range screeners, only require constant training effort. Despite its simplicity, the model is in total agreement with the empirical findings. Finally, we evaluated the impact of using such error detectors within a fault localization approach aimed at the deployment (operational) phase, rather than just the development phase. We verified that, despite the simplicity of the screeners (and therefore considerable rates of false positives and/or negatives), the diagnostic performance of SFL is similar to the development-time situation. This implies that fault diagnosis with an accuracy comparable to that in the development phase can be attained at the deployment phase with no additional programming effort or human intervention.

Future work includes the following. Although other screeners are more time-consuming and program-specific, such as relationships between variables or components' state machine-based program invariants [22], they may lead to better diagnostic performance, and are therefore worth investigating. Finally, inspired by the fact that only a limited number of (so-called collar) variables are primarily responsible for program behavior [23,24], we also plan to study the impact of (judiciously) reducing the amount of screened program points (overhead).

References

1. Abreu, R., González, A., Zoeteweij, P., van Gemund, A.J.C.: Automatic software fault localization using generic program invariants. In: Proc. SAC 2008. ACM Press, New York (2008)
2. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: An evaluation of similarity coefficients for software fault localization. In: Proc. PRDC 2006. IEEE CS, Los Alamitos (2006)
3. Abreu, R., Zoeteweij, P., van Gemund, A.J.C.: On the accuracy of spectrum-based fault localization. In: Proc. TAIC PART 2007. IEEE CS, Los Alamitos (2007)
4. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13(7), 422–426 (1970)
5. Cleve, H., Zeller, A.: Locating causes of program failures. In: Proc. ICSE 2005. IEEE CS, Los Alamitos (2005)
6. Dallmeier, V., Lindig, C., Zeller, A.: Lightweight defect localization for Java. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 528–550. Springer, Heidelberg (2005)
7. David, H.A.: Order Statistics. John Wiley & Sons, Chichester (1970)
8. Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. In: Proc. ICSE 1999. IEEE CS, Los Alamitos (1999)
9. Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The Daikon system for dynamic detection of likely invariants. In: Science of Computer Programming (2007)
10. Gautama, H., van Gemund, A.: Low-cost static performance prediction of parallel stochastic task compositions. IEEE Trans. Parallel Distrib. Syst. 17(1), 78–91 (2006)
11. González, A.: Automatic error detection techniques based on dynamic invariants. Master's thesis, Delft University of Technology and Universidad de Valladolid, Delft (2007)
12. Gumbel, E.: Statistical theory of extreme values (main results). In: Sarhan, A., Greenberg, B. (eds.) Contributions to Order Statistics. John Wiley & Sons, Chichester (1962)
13. Hangal, S., Chandra, N., Narayanan, S., Chakravorty, S.: IODINE: A tool to automatically infer dynamic invariants for hardware designs. In: Proc. DAC 2005. ACM Press, New York (2005)

14. Hangal, S., Lam, M.: Tracking down software bugs using automatic anomaly detection. In: Proc. ICSE 2002. IEEE CS, Los Alamitos (2002)
15. Harrold, M., Rothermel, G., Wu, R., Yi, L.: An empirical investigation of program spectra. ACM SIGPLAN Notices 33(7) (1998)
16. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proc. ICSE 1994. IEEE CS, Los Alamitos (1994)
17. Jain, A., Dubes, R.: Algorithms for clustering data. Prentice-Hall, Inc., Englewood Cliffs (1988)
18. Jones, J., Harrold, M.: Empirical evaluation of the tarantula automatic fault-localization technique. In: Proc. ASE 2005. ACM Press, New York (2005)
19. Kephart, J., Chess, D.: The vision of autonomic computing. Computer 36 (2003)
20. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: Proc. CGO 2004. ACM Press, New York (2004)
21. Liu, C., Fei, L., Yan, X., Han, J., Midkiff, S.: Statistical debugging: A hypothesis testing-based approach. IEEE TSE 32(10), 831–848 (2006)
22. Lorenzoli, D., Mariani, L., Pezzè, M.: Automatic generation of software behavioral models. In: ICSE 2008. ACM Press, New York (2008)
23. Menzies, T., Owen, D., Richardson, J.: The strangest thing about software. Computer 40(1), 54–60 (2007)
24. Pattabiraman, K., Kalbarczyk, Z., Iyer, R.K.: Application-based metrics for strategic placement of detectors. In: Proc. PRDC 2005. IEEE CS, Los Alamitos (2005)
25. Patterson, D., Brown, A., Broadwell, P., Candea, G., Chen, M., Cutler, J., Enriquez, P., Fox, A., Kiciman, E., Merzbacher, M., Oppenheimer, D., Sastry, N., Tetzlaff, W., Traupman, J., Treuhaft, N.: Recovery Oriented Computing (ROC): Motivation, definition, techniques, and case studies. Technical Report UCB/CSD-02-1175, U.C. Berkeley (2002)
26. Pytlik, B., Renieris, M., Krishnamurthi, S., Reiss, S.: Automated fault localization using potential invariants. In: Proc. AADEBUG 2003. ACM Press, New York (2003)
27. Racunas, P., Constantinides, K., Manne, S., Mukherjee, S.: Perturbation-based fault screening. In: Proc. HPCA 2007. IEEE CS, Los Alamitos (2007)
28. Renieris, M., Reiss, S.: Fault localization with nearest neighbor queries. In: Proc. ASE 2003, Montreal, Canada. IEEE CS, Los Alamitos (2003)
29. Yang, J., Evans, D.: Automatically inferring temporal properties for program evolution. In: Proc. ISSRE 2004. IEEE CS, Los Alamitos (2004)
30. Zhang, X., He, H., Gupta, N., Gupta, R.: Experimental evaluation of using dynamic slices for fault location. In: Proc. AADEBUG 2005. ACM Press, New York (2005)

Language Support for Service Interactions in Service-Oriented Architecture

Sven De Labey[1,⋆], Jeroen Boydens[2], and Eric Steegmans[1]

[1] K.U. Leuven, Department of Computer Science
Celestijnenlaan 200A, B3000 Leuven, Belgium
[2] KHBO Department of Industrial Engineering Science & Technology
Zeedijk 101, B8400 Oostende, Belgium
{svendl,eric}@cs.kuleuven.be, jeroen.boydens@khbo.be

Abstract. The Open Services Gateway initiative (OSGi) is a platform for running service-oriented Java applications. OSGi provides a central service registry to allow application components (so-called *bundles*) to share functionality. From the viewpoint of programming language development, OSGi leaves a lot of room for improvement. Its service query language, for instance, bypasses important compile-time guarantees and it works only for service metadata that never changes during the lifetime of a service. A second problem is that the event notification system requires programmers to write a considerable amount of boilerplate logic for reacting to service events. This obfuscates the business logic, which in turn decreases code comprehension and increases the odds for introducings bugs when implementing client-service interactions.

This paper evaluates OSGi as a platform for programming client-service interactions in Java. After focusing on problems that relate to OSGi's integrated service query language and its event notification system, we propose a solution based on a programming language extension. We also show how this extension is transformed to regular Java code so as to maintain interoperability with the OSGi specification.

Keywords: Service-Oriented Architecture, Language Concepts.

1 Introduction

Object-Oriented programming languages such as Java are increasingly adopting the paradigm of Service-Oriented Computing [1]. One of the most popular SOA adopters is the Open Services Gateway initiative [2]. OSGi technology provides a service-oriented, component-based environment and offers standardized ways to manage the software lifecycle [3]. It subscribes to the *publish-find-bind* model by providing a *central service registry* which is used by application components (so-called *bundles*) to publish their services along with relevant metadata. These services can then be retrieved by other bundles by means of an LDAP-based search mechanism. OSGi also provides a notification system to signal lifecycle changes of services. This support for *implicit invocation*,

[⋆] Research funded by the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 75–90, 2010.

where bundles are given a chance to react to events that are relevant for them is a key feature of OSGi because service architectures are inherently dynamic and volatile.

But OSGI also imposes a lot of responsibilities on the programmer [4]. Its integrated service query language, for instance, is very weak and bypasses compile-time guarantees on the syntactic correctness of a query. Also, the benefits of the notification system are overshadowed by the requirement to write a considerable amount of boilerplate code, which bypasses compile-time guarantees in a similar way and which lacks support for advanced event notification methodologies, such as Event-Condition-Action rules.

In this paper, we evaluate OSGi as a means for implementing client-service interactions in Java-based service-oriented architectures. We identify a number of problems and we show how these can be solved by introducing a language extension that (1) increases the level of abstraction and (2) provides compile-time guarantees on the correctness of service queries and event definitions.

This paper is structured as follows. Section 2 provides a comprehensive review of OSGi in the context of client-service interactions. Based on this evaluation, Sections 3 and 4 propose a language extension that aims at solving the problems of OSGi. The implementation of this extension is described in Section 5. Related work is presented in Section 6 and we conclude in Section 7.

2 Evaluation of Client-Service Interactions in OSGi

Section 2.1 first evaluates *explicit* client-service interactions, which occur when a client directly invokes a method that is published by the public interface of the target service. Section 2.2 then evaluates *implicit* interactions, which occur when a service reacts to the notification of an event.

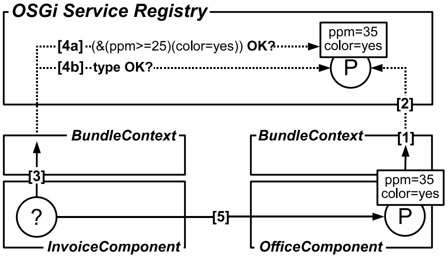


Fig. 1. Service registration and retrieval via the `BundleContext`

2.1 Explicit Client-Service Interactions

Explicitly invoking an operation from the public interface of a target service requires a client to find the service first. OSGi follows the *publish-find-bind* methodology of Service-Oriented Computing to enable service retrieval via the OSGi Service Registry. This section evaluates (1) service registration and (2) service retrieval.

Service Registration. A bundle uses the central service registry to register the services it offers to other bundles. Registration is done through the bundle's `BundleContext` reference, which is injected into the *bundle activator class* by the OSGi runtime system. The *service object* can be added to the registry along with service-specific metadata represented as a *dictionary* containing key-value pairs. This is depicted conceptually in steps 1–2 of Figure 1, where an `OfficeComponent` registers a printer of type `P` with metadata pairs describing (1) the printer throughput (`ppm`) and whether it supports color printing (`color`). The code for realizing this registration is shown in Listing 1. Lines 7-10 show how the printer and its metadata are added to the service registry.

Evaluation. The major problem is that OSGi's metadata system bypasses compile-time guarantees on the soundness of key-value pairs. Erroneous pairs such as `(ppm,true)` are accepted by the Java compiler because keys and values are statically typed as `String`, as shown on lines 3–4 in Listing 1. Moreover, clients must *know* the names of the properties that were added during registration (such as "`ppm`") as well as the domain of possible values for each property, but there is no standardized way to retrieve this information.

```
1  //--1-- Specify metadata
2    Properties metadata = new Properties();
3    metadata.put("ppm","35");
4    metadata.put("color","true");
5   //--2-- Register Service and Metadata
6    ServiceRegistration registration =
7       context.registerService(          //use of the BundleContext reference
8         PrinterServiceImpl.class.getName(),
9         printer,
10        metadata);|
```

Listing 1. Registering services and metadata in OSGi

Service Retrieval. Bundles use *service queries* to find services that were registered by other bundles. A service query is written as an *LDAP expression* containing (1) the *service type* in order to indicate what kind of service the bundle needs (e.g. `PrinterService`) and (2) a boolean expression constraining the values of the service's metadata. Steps 3–4 in Figure 1 depict the retrieval of a color printer, `P`, with the constraint that it must print at least 25 pages per minute. The variables used in the LDAP query (`ppm` and `color`) refer to metadata entered by the service provider. Listing 2 shows how such a retrieval is realized by means of an LDAP expression containing the service type (line

```
1  public PrinterService searchPrinterService(){
2     ServiceReference[] printerReferences;
3     try {
4       String servType="(objectClass="+PrinterService.class.getName()+")";
5       String serviceFilter="(&"+servType+"(&(ppm>=25)(color=true)))";
6       references[] = context.getServiceReferences(null,serviceFilter);
7       return (PrinterService)context.getService(references[0]);
8     }
9     catch(InvalidSyntaxException ise){ return null; }
10 }|
```

Listing 2. Retrieving services from the OSGi Service Registry

4) and a boolean expression (line 5). Again, the `BundleContext` is used to shield the requesting bundle from complex interactions with OSGi framework classes.

Evaluation. Since we are mainly interested in reviewing the expressive power of the LDAP query mechanism, we consider three kinds of *service properties*: (1) *static*, (2) *dynamic* and (3) *derived* properties:

- *Static Service Properties.* OSGi's LDAP-based query mechanism is ideally suited for static properties, as these properties never change during the lifetime of the service. Metadata such as *pages per minute* can be registered along with a `PrinterService` instance because this information is assumed not to change during the lifetime of the printer.
- *Dynamic Service Properties.* Properties that *do* change when a service is operational, introduce major problems. The queue of a `PrinterService`, for instance, grows and shrinks as jobs arrive and get processed. Obviously, these properties cannot be added during registration, so OSGi's metadata system does not support them and neither does its LDAP query language.
- *Derived Service Properties.* A third class of characteristics comprises information that depends on input provided by the client bundle. The cost for printing a file, for instance, may be calculated by combining the file's page count with the `PrinterService`'s cost for printing one page. But the OSGi metadata system does not support this. Thus, clients cannot specify constraints on information that is *derived* from the metadata of a service.

OSGi also lacks a *statically typed*, *comprehensible* query language (as shown on lines 4–5 in Listing 2). Similar to the service registration process where syntactically incorrect metadata could be added, it is possible to write inconsistent or syntactically incorrect queries. These errors will only be detected at runtime when the query is parsed. Programmers using LDAP-based queries must therefore expect to catch an `InvalidSyntaxException` *every time* they want to retrieve services (cfr line 9 in Listing 2). This is in sharp contrast with regular method invocations, about which the Java compiler provides strong guarantees about their syntactical correctness and the absence of typing errors.

In summary, the OSGi query mechanism is a *dynamically* typed query language that only deals with *static* service metadata. What we need, however, is a *statically* typed query language that is able to deal with *dynamic, volatile* service properties.

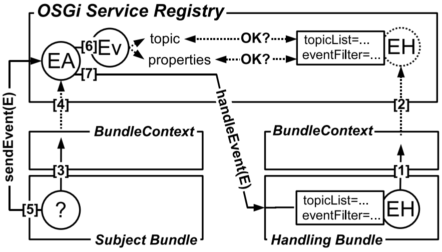


Fig. 2. Event Notification in OSGi

2.2 Implicit Client-Service Interactions

Next to explicit client-service interactions, OSGi also supports *implicit* interactions, which occur when a service reacts to the notification of an event. This section focuses on how bundles can register event handlers that allow them to react to events sent by other services, bundles, or by the OSGi framework itself. Two roles are important: the *event handler* and the *subject*. Interactions between both are mediated by the OSGi `EventAdmin` service:

- **Event Handler.** An `EventHandler` is a class that reacts to events. Its `handleEvent(Event ev)` method is called by the `EventAdmin` when events occur. Handlers are *registered as services* following the approach explained in Section 2.1 (see steps 1–2 in Fig. 2). Registration can be done along with metadata so as to filter the set of events that will be signalled to the handler. Listing 3 shows how a *topic* (line 3) and a *filter* (line 4) are added to a metadata dictionary (lines 5–7) which is added to the `ItemTracker` event handler that is registered as a service in the OSGi registry (line 8). A topic can be seen as a *channel* on which related events are published; the filter expression constrains the values of *event-specific metadata*, such as `type` and `price`.

```
1 public void start(BundleContext bundleContext){
2   ItemTracker tracker = new ItemTracker(); |/*implements|EventHandler|*/|
3   String topic = "org/osgi/service/Item";
4   String filter = "(&(type=sold)(price>1000))";
5   Dictionary dict = new Hashtable();
6   dict.put(EventConstants.EVENT_TOPICS,topic);
7   dict.put(EventConstants.EVENT_FILTER,filter);
8   bundleContext.registerService("EventHandler.class",tracker,dict);
9 }|
```

Listing 3. Registration of a constrained `EventHandler`

- **Subject.** A bundle that publishes events is called an *event source* or a *subject*. Events typically contain metadata to finetune the notification phase. The `Item` event, for instance, contains information about the `type` of the event, and the `price` of the `Item`. Listing 4 shows how this metadata is added to the event (lines 2–5) and Listing 3 already indicated how event handlers may constrain these metadata properties. To publish events, subjects use their `BundleContext` reference to search for the OSGi `EventAdmin` service (shown in steps 3–5 in Figure 2 and on lines 7–10 in Listing 4). The `EventAdmin` matches the metadata of the event with the constraints (topic and filter) of all the registered event handlers and notifies those handlers for which the matching process succeeds (steps 6–7 in Figure 2). This decoupled way of notifying handlers, in which a central `EventAdmin` is responsible for the technical wiring between subjects and their handlers, is called the *Whiteboard pattern* [5].

```
 1 public void start(BundleContext bundleContext){
 2    Dictionary metadata = new Hashtable();              /*| Create Event |*/
 3    metadata.add("price","1500");
 4    metadata.add("type","sold");
 5    Event event = new Event("org/osgi/service/Item",metadata);
 6
 7    String adminName=EventAdmin.class.getName();        /*| Send Event |*/
 8    ServiceReference sr = bundleContext.getServiceReference(adminName);
 9    EventAdmin admin = (EventAdmin)bundleContext.getService(sr);
10    admin.sendEvent(event);
11 }|
```

Listing 4. Creating and sending events using the `EventAdmin`

Evaluation. We focus on three problems concerning implicit service interactions: (1) the lack of compile-time guarantees, (2) the limited expressiveness of event definitions and (3) the lack of support for complex event processing.

- **Limited Compile-time Guarantees.** Similar to the lack of compile-time guarantees for explicit service interactions, OSGi fails to provide important guarantees on the soundness of (1) topics and filters of event handlers, as shown on lines 3–4 in Listing 3 and (2) on the metadata of the events and their topic, as shown on lines 3–5 in Listing 4.
- **Limited Expressiveness.** Again, LDAP constraints can only refer to service properties that never change during the lifetime of the service, i.e. *static* service properties. This is not a problem for the metadata of the `Event` instance, as events are typically shortlived. But it is a major problem for the handlers: they cannot introduce *dynamic* or *derived* propertes in their topics and filters. This inability to expressively filter events increases the complexity of event handlers and it leads to a large amount of unsolicited notifications.
- **Lack of Support for Complex Event Processing.** Bundles may be interested in *patterns* among events, rather than events themselves, thus requiring a mechanism for correlating event occurrences. The problem with OSGi's event notification mechanism is that the `EventAdmin` automatically forwards event occurrences from the subject to the handlers, without providing support for complex event detection. This puts the burden of creating data structures and algorithms for detecting composite events on the programmer, thus decreasing the level of abstraction and significantly reducing the readability of the code due to the interweaving of technical detection code with the business logic.

The evaluation presented here drives our research for language extensions, which we introduce in the next two sections. We first introduce constructs for explicit service interactions [6] (Section 3) and then discuss concepts for implicit service interactions (Section 4) before explaining how this extension is transformed to Java so as to remain compatible with the OSGi specification (Section 5).

3 Abstractions for Explicit Service Interactions

ServiceJ introduces (1) *type qualifiers* to identify variables depending on services, as discussed in Section 3.1 and (2) *declarative operations* for fine-tuning the set of assignable services to such a service variable, as discussed in Section 3.2.

3.1 Support for Basic Service Interactions

Type qualifiers are used to distinguish variables holding service references from variables pointing to local objects. This differentiation allows the ServiceJ-to-Java transformer to inject additional operations for transparently dealing with the typical challenges introduced by service architectures (this is explained in detail in Section 5). Currently, two type qualifiers are defined in ServiceJ:

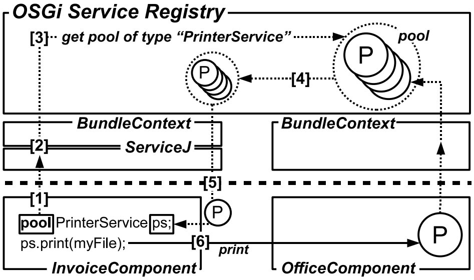


Fig. 3. The pool qualifier triggers transparent service lookup and service injection

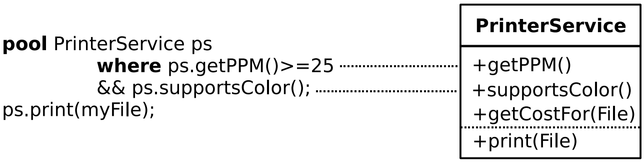
1. *The* `pool` *qualifier*. This type qualifier is used to indicate that a variable depends on a service published by another bundle. In Figure 3, for instance, the variable `ps` of type `PrinterService` is decorated with the `pool` qualifier to indicate that the variable depends on a service that must be provided by another bundle. The advantage of explicitly stating this dependency is that our compiler can *inject* code for service retrieval, thus increasing the level of abstraction. The `pool` qualifier causes the transformer (see Section 5 for details) to transparently inject operations for (1) service retrieval, (2) non-deterministic service binding, and (3) service failover.
2. *The* `sequence` *qualifier*. The `pool` qualifier is the most basic qualifier for service interactions in that it selects *any* service from the registry that has a compatible type. Sometimes, however, bundles may prefer some service implementations above others, based on service-specific metadata. In that case, a *deterministic* service selection procedure is required. The `sequence` qualifier, which is a *subqualifier* of the `pool` qualifier, is used to decorate these variables. The `sequence`qualifier is used exclusively in conjunction with the `orderby` operation, which is explained in Section 3.2.

Example. Figure 3 depicts how `ps`, a variable of type `PrinterService`, is decorated with the `pool` qualifier. It shows how the programmer is exonerated from implementing interactions with the OSGi middleware in order to obtain service references. Programmers only need to declare their service variables and invoke operations on them. Initialization is now the responsibility of the ServiceJ middleware (steps 1–2 in Figure 3), which interacts via the `BundleContext` with the OSGi service registry (steps 3–4) before non-deterministically injecting a service reference in `ps` (step 5) and invoking the `print` operation (step 6). Should the injected service fail during this interaction, then ServiceJ automatically injects another service into `ps` before transparently reinvoking the operation.

3.2 Support for Constrained Service Interactions

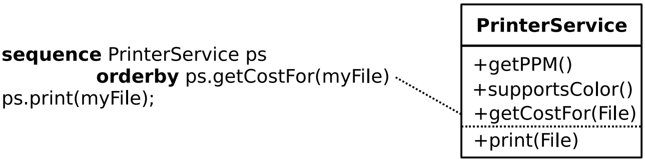
Similar to the LDAP-based query language provided by OSGi, ServiceJ incorporates specialized support for fine-tuning service selection. In contrast with OSGi, however, these operations are now fully integrated within the programming languages in the form of *declarative operations*. In ServiceJ, queries no longer refer to untyped metadata, but instead, they directly relate to the operations that are exported by the service's interface. In stead of using an untyped property such as “`ppm`”, for instance, queries in ServiceJ refer to a *public inspector method* such as `getPPM()`, which is exported by the `PrinterService` interface. This provides better compile-time guarantees on the syntactical and conceptual correctness of queries. Currently, two declarative operations are defined for fine-tuning service selection:

- *The* `where` *operation*. This operation is used to *constrain* a set of candidate services by means of a boolean expression and thus replaces OSGi's untyped query language. The service query from Listing 2, for instance, can be translated using the `where` operation as follows:



```
pool PrinterService ps
          where ps.getPPM()>=25
          && ps.supportsColor();
ps.print(myFile);
```

  This system not only fosters compile-time guarantees, but it is also more expressive than LDAP expressions. Our query language can take into account the most up-to-date information of a service since it refers directly to public methods of the service's API, thus allowing programmers to impose constraints on *dynamic* and *derived* service properties. An additional benefit is that the `where` operation is combined with the `pool` qualifier, implying that these constrained sets of candidate services still provide transparent service selection, injection and fail-over.



```
sequence PrinterService ps
          orderby ps.getCostFor(myFile)
ps.print(myFile);
```

- *The* `orderby` *operation*. This operation is used to sort a set of candidate services according to the preferences of a user. In the `PrinterService` example, a programmer can use the `orderby` operation to select the printer that minimizes the cost for printing a given file, again referring to public API methods, as shown in the sample code above.

  Note that this query necessitates the use of the `sequence` qualifier because a *deterministic* service selection policy is requested. Detailed information about the use of type qualifiers and their associated service selection strategies can be found in our previous paper [7].

4 Abstractions for Implicit Service Interactions

This section introduces concepts that solve the problems identified in Section 2.2 by (1) providing compile-time guarantees for both subjects and their event handlers, (2) improving the expressiveness of event constraints and (3) increasing the level of abstraction so as to exonerate programmers from implementing boilerplate code. We look at *atomic events* (Section 4.1) and show how type-safe support for *composite event notification* is integrated in ServiceJ (Section 4.2).

```
public interface Item{
  |/* events that can be published by all Item services */|
  public event Sold{
     private final double price;
     public Sold(double price){ this.price=price; }
     public double getPrice(){ return this.price; }
  }
  public event Bought{ |/* event body omitted */| }

  |/* regular Java interfaces methods */|
  void sell();
}|
```

Listing 5. Event definitions are directly integrated into the service interface

4.1 Language-Integrated Atomic Event Notification

OSGi offers no standardized way for event handlers to find out what events a subject signals. This drawback is overcome by integrating events as inner types in the public service interface. Listing 5 shows how a `Sold` event is integrated into the `Item` interface. Subjects implementing this `Item` interface can publish `Item.Sold` events. Each event class inherits from a common base class, `Event`, that offers general-purpose methods such as `getSource()` to retrieve the subject and `getOccurrenceTime()` to return the event's publication time.

```
1  public class Flight implements Item{
2    public void sell(){
3       //sell item at <price>
4       new Item.Sold(price).publish();
5    }
6  }|
```

Listing 6. The `publish()` operation makes event publication transparent

Event Publication. Subjects create new instances of the `Item.Sold` event and then call the `publish()` operation of `Event` to start the notification process. This approach completely hides interactions with the `BundleContext` and the `EventAdmin`, as shown on line 4 of Listing 6. All technical details are left to our preprocessing tool and our underlying notification middleware (see Section 5).

Reacting to Events. Rather than filtering events using LDAP constraints, our language allows event handlers to use event filters based on (1) the *subject* that sends them and (2) the *event* type and its characteristics. The language constructs are shown in Listing

7 on lines 1–7, but due to space constraints, we explain them by means of the example on lines 9–16:

- **Subject.** The event handler uses the `observe()` clause to indicate what type of events it needs to track. Line 10 in Listing 7 indicates that the handler is willing to listen to events sent by subjects of type `Item`. These subjects are further constrained by means of a boolean `where` expression. In our example, the item must be contained in a list, as required by the boolean expression, `this.getStoredItems().contains(item)`, on line 10.
- **Event.** The observed subject may notify various types of events, so the `on()` clause is used to specify what types of events are of interest to the handler. A second `where` clause may be attached to the `on` clause to further constrain that type of events. Line 11 in Listing 7 shows an event handler soliciting `Item.Sold` events with a selling price of more than 3000. When a matching event is found, the `sale` variable is automatically bound to that event instance, allowing the handler to retrieve event-specific data using traditional inspector methods, such as `sale.getPrice()` on line 12.

```
 1  class Monitor{   // -- Language Concepts, Definition
 2     observe(|subject type|) where(|subject constraint|){
 3        on(|event expression|) where(|event constraint|){
 4           |reaction|
 5        }
 6     }
 7  }
 8
 9  class Monitor{   // -- Language Concepts, Example
10     observe(Item item) where(this.getStoredItems().contains(item)){
11        on(Item.Sold sale) where(sale.getPrice()>3000){
12           Logger.archive("We sold an item for " + sale.getPrice());
13        }
14        on(...) //other events can be handled here.
15     }
16  }|
```

Listing 7. Observing subjects and conditionally reacting to events

4.2 Language-Integrated Composite Event Notification

Composite events are structural combinations of atomic events and/or other composite events. OSGi does not support composition because it lacks a language for defining composite events and because the `EventAdmin` currently does not have a detection algorithm. This section provides an overview of event composition operators and our strategy for composite event handling in ServiceJ.

We define a *sublanguage*, embedded in our DSL, that allows to define event compositions in OSGi. This is a sublanguage because its expressions may only appear inside the `on` clause, which was introduced in Section 4.1. Keeping the event composition sublanguage completely independent of Java avoids pollution of the Java grammar definition and allows both languages to evolve independently of each other. We focus on the following composition operators:

Operator	Example	Occurs when...
&&	A && B	when both A and B have occurred
||	A || E	when A or E has been notified
->	A -> C	when the notification of A is followed by a notification of C
[*i*]	C[*3*]	when an event of type C has occurred 3 times
!	!E(A->D)	when an event E is not detected during the detection of A->D

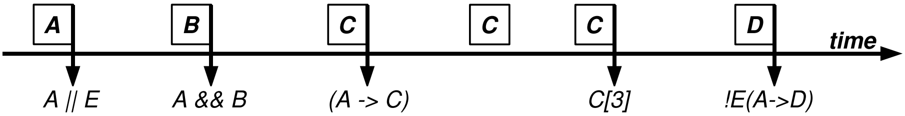


Fig. 4. Composite events are defined as structural combinations of other events

Listing 8 shows how a composite event Profit is fired when an Item.Bought event is followed by an Item.Sold event. This composition is specified in the on clause on line 10. Listing 8 contains two new language constructs:

1. **Constrained Unification Variables.** These are variables that are matched with events that were signalled by observed subjects. Line 7 in Listing 8, for instance, declares a unification variable buy of type Item.Bought. This CUV is *matched* with Item.Bought instances signalled by the subject itself (as required by observe(this) on line 6). Note also that this variable is *constrained*: it contains a where clause that imposes additional constraints on events. Unification only succeeds if three rules are satisfied: (1) the subject matches the observe clause, (2) the event type matches the CUV type and (3) the composite event satisfies the where clause on line 10.

```
1  public event Profit{
2     public double amount;
3     public Profit(double amt){ this.amount=amt; }
4  }
5
6  observe(this){
7     Item.Bought buy where buy.getPrice()>3000;              |/* CUV */|
8     Item.Sold sale;
9
10    on(buy->sale) where(buy.getPrice() < sale.getPrice()){    |/* definition */|
11       new Profit(sale.getPrice()-buy.getPrice()).publish();
12    }
13 }|
```

Listing 8. Composite events are defined in the on clause on line 10

2. **Composite Event Specification.** The specification of the composite event is isolated in the on clause, which is the only place where our event composition sublanguage is supported. Such a specification relates to the CUVs, buy and sale, declared on lines 7–8 in Listing 8. If these variables are matched by event occurrences according to the specification buy->sale, and if the accompanying

where clause on line 10 is satisfied at the moment of the detection, then the reactive part (line 11) is executed. This triggers the creation and publication of a new Profit event. Thus, the reactive part of the on clause can be used for two purposes: (1) for reacting to a composite event, but also (2) for signalling new events.

5 Implementation

Figure 5 shows how ServiceJ code is read by a lexer and a parser ① so as to create a *ServiceJ metamodel instance* ②. The ServiceJ metamodel is an extension of Jnome [8], our Java metamodel, to which we added constructs that represent type qualifiers (pool and sequence), declarative operations (where and orderby), and constrainable observation structures (observe and on). The metamodel representation of the source code is fed to the ServiceJ-to-Java transformer ③, which is responsible for building a Java metamodel instance that replaces the ServiceJ constructs of the ServiceJ metamodel instance with equivalent Java constructs. Service interactions are transformed as follows:

- **Explicit Interactions.** When a method invocation on a pool variable is detected in the metamodel instance representing the source code, the transformer injects code for service retrieval. First, it inserts interactions with the OSGi service registry via a BundleContext reference so as to select a proper candidate set ④. Then, it selects the service that satisfies the where clause, if any, and optimizes this selection based on the orderby clause. Finally, it makes the service interaction more robust by injecting *service fail-over* code.
- **Implicit Interactions.** Composite event patterns are transformed back to atomic event subscriptions so as to remain compatible with the existing OSGi EventAdmin. The constraints found in the where expressions accompanying the observe and on clauses are transformed to classes following the Command pattern [9]. These commands are then organized as *filters* following the Pipes and Filters design pattern [10]. The result of this process is that each atomic subscription is decorated with a chain of filters, represented as commands. The transformer then builds a directed *event graph* based on the composite event definition found in the on clause, using the atomic subscriptions as *nodes* of that graph.

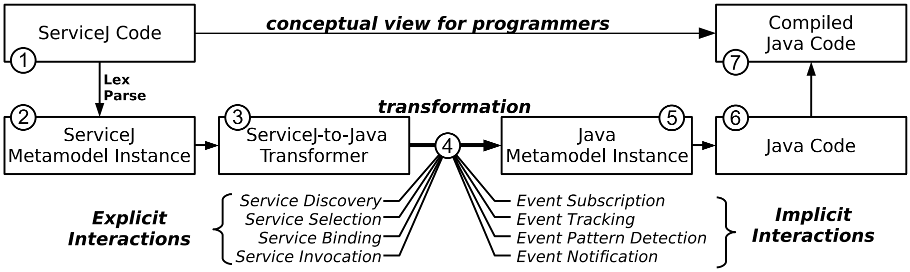
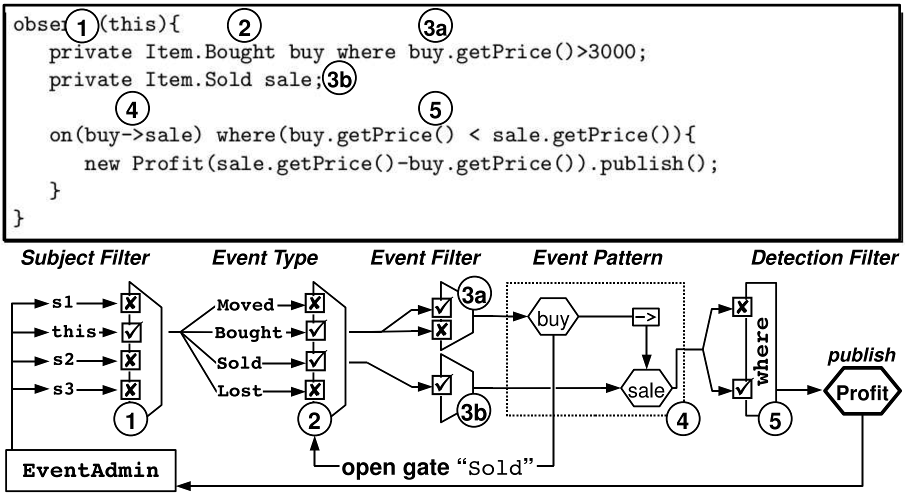


Fig. 5. Transformation of ServiceJ code to compiled Java code



```
observe(this){
   private Item.Bought buy where buy.getPrice()>3000;
   private Item.Sold sale;

   on(buy->sale) where(buy.getPrice() < sale.getPrice()){
      new Profit(sale.getPrice()-buy.getPrice()).publish();
   }
}
```

Fig. 6. Transformation of an event definition according to the Pipes and Filters design pattern

After the equivalent Java metamodel has been built by the ServiceJ-to-Java transformer, a simple code writer ⑤ transforms the Java metamodel instance to *.java* files ⑥, which can be compiled by the standard Java compiler, thus finishing the compilation process ⑦. All in all, this is an extensive process, but it is important to note that it acts as a black box consuming ServiceJ source files and producing compiled Java classes. Intervention of developers or deployers is never required during the compilation process.

Transforming Implicit Interactions, Example. Figure 6 shows an event handler that signals a composite event `Profit` when an `Item.Bought` event is followed by an `Item.Sold` event. The OSGi `EventAdmin` distributes event notifications, that are sent to a series of connected filters. The first filter is a *subject filter* ①. As required by `observe(this)`, this subject filter only allows events sent by the subject itself to pass through. Assuming that four types of events can be signalled (`Moved`, `Bought`, `Sold`, and `Lost`), the *event filter* ② then blocks `Moved` and `Lost` events since these are not solicited in order to detect the composite event. Events of type `Sold` are also blocked at this point because the `buy->sale` specification is not interested in `Item.Sold`: Figure 6 shows that the `Sold` gate is opened only when a `Bought` event is signalled. Events that satisfy the type constraint are passed to the third filter ③, which checks the `where` clause of the constrained unification variables `buy` and `sale`. Events that pass this third filter are injected in the *event graph* ④ that represents the composite event definition `buy->sale`. This graph is traversed as events are signalled, and when it reaches a *sink*, the next filter is activated. This filter represents the `where` expression ⑤ that further constrains the `on` clause. If that condition is satisfied, then the `EventAdmin` is called to signal the `Profit` event.

6 Related Work

Jini [11] is a close competitor to OSGi, since it also attempts to bring Service-Oriented Computing to the world of object-oriented programming [12]. One drawback of Jini with respect to OSGi, is that Jini's query mechanism is much weaker than OSGi's LDAP-based queries. Jini relies on *entry objects* representing the *exact* value that a property must have, whereas OSGi allows programmers to use operators such as "<" and ">=" to specify *ranges* of values, rather than a single value. Moreover, Jini does not install an event notification architecture, but relies on a *leasing* system instead. Clients retrieving a Jini service are given a lease representing the time they are allowed to use the service. Leases must be renewed temporarily, which creates additional programming overhead.

Cervantes and Hall observed that OSGi does not provide any support for managing service dependencies apart from the basic event notification system in [13] and [14]. They propose to improve dependency management based on *instance descriptors*. Such instance descriptors are XML files describing how a bundle depends on external services. A `<requires>` tag is introduced to specify the *service type*, the *cardinality* of the dependency, and the *filter condition*. Programmers may also define the *bind* and *unbind* methods that should be called when a service is to be bound or unbound. One problem with this approach is that the information to be specified bypasses compile-time guarantees. Also, the filter condition is now isolated in an XML file, but it is still an LDAP-based query string, which lacks support for dynamic and derived service properties. Moreover, instance descriptors create a strong dependency between a Java file and an XML file, and they divide important decisions concerning the business logic between these two files, which reduces the comprehensibility of the code.

The Observer pattern [9] is the pattern on which the Whiteboard pattern [5] is based. Bosch [15] proposes *LayOM*, a *layered object model* that integrates events directly into a programming language. Layers intercept method calls so as to enable pre- and postprocessing, such as event notification. The main goal of *LayOM* is to increase the visibility of events. Other problems, such as the monolithic `handleEvent` method are not solved. Moreover, events are modelled as strings, similar to OSGi event metadata, thus reducing compile-time guarantees. Programmers are also responsible for wiring handlers to subjects, thus reducing the level of abstraction. The `VetoableChangeListen- er` and the `PropertyChangeListener` interfaces from the *JavaBeans* package [16] have similar problems.

Hedin introduces language support for events based on attribute extension [17]. Programmers can make their subjects and events more visible and the compiler can verify whether event notification is implemented correctly, thus increasing both traceability and verifiability. This approach resembles ours although its expressiveness is limited: event occurrences cannot be constrained or combined, whereas our approach supports (1) subject constraints, (2) constrained unification variables and (3) constrained event composition.

Riehle introduces a programming model where state changes and dependencies are modelled as first class objects in the programming language [18]. The resulting system is highy flexible and can be used for dependency management between objects in an object-oriented programming language. But no compiler can guarantee whether these dependencies are wired correctly. Programmers are also required to implement a large amount of classes. The Observer-Conditioned-Observable pattern [19] has similar problems.

Our approach of introducing type qualifiers and qualifier inference is based on the approach followed in Javari [20]. We have proven the type soundness of this language extension in a way similar to the Java extension presented in [21]. For more information about the formal development of ServiceJ, we refer to [22] and [7].

7 Conclusions

The Open Services Gateway Initiative is a successful attempt to bridge the gap between object-oriented programming and service-oriented computing, but a number of challenges remain unsolved. In this paper, we have focused on problems stemming from explicit and implicit client-service interactions, where the lack of compile-time guarantees and the limited expressiveness of the service query and event constraint language remain the most important drawbacks.

To solve these problems, we propose an integration of ServiceJ language concepts into the OSGi programming model. Type qualifiers and declarative operations enable programmers to invoke services that satisfy their requirements (using the `where` clause) and that most closely approximate their expectations (using the `orderby` clause). Moreover, specialized constructs such as the `observe` and `on` clause allow them to define conditional reactions to complex event patterns. In summary, our concepts increase the level of abstraction, at the same time fostering the compile-time guarantees of a statically typed programming lannguage that remains fully compatible with the existing OSGi specification.

References

1. Papazoglou, M.: Service Oriented Computing: Concepts, Characteristics and Directions. In: Proceedings of the 4th International Conference on Web Information Systems Engineering (2003)
2. OSGi: Open Services Gateway Initiative Specification v4.0.1 (2006), http://www.osgi.org
3. Marples, D., Kriens, P.: The open service gateway initiative: An introductory overview. IEEE Communications Magazine 39 (2001)
4. Hall, R., Cervantes, H.: Challenges in building service-oriented applications for OSGi. IEEE Communications Magazine 42, 144–149 (2004)
5. OSGi: Listeners considered harmful: The whiteboard pattern (2004), www.osgi.org/documents/osgi_technology/
6. De Labey, S., Steegmans, E.: Typed Abstractions for Client-Service Interactions in OSGi. In: Proceedings of the Third International Conference on the Evaluation of New Approaches to Software Engineering (2008)
7. De Labey, S., Steegmans, E.: ServiceJ. A Type System Extension for Programming Web Service Interactions. In: Proceedings of the Fifth International Conference on Web Services, ICWS 2007 (2007)
8. van Dooren, M., Vanderkimpen, K., De Labey, S.: The Jnome and Chameleon Metamodels for OOP (2007)

9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Professional, Reading (1995)
10. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions (2003)
11. Sun: The Jini Architecture Specification and API Archive – (2005), http://www.jini.org
12. Huang, Y., Walker, D.: Extensions to Web Service Techniques for Integrating Jini into a Service-Oriented Architecture for the Grid. In: Proceedings of the International Conference on Computational Science (2003)
13. Hall, R., Cervantes, H.: Gravity: supporting dynamically available services in client-side applications. SIGSOFT Software Engineering Notes 28, 379–382 (2003)
14. Cervantes, H., Hall, R.: Automating Service Dependency Management in a Service-Oriented Component Model. In: Proceedings of the 6th Workshop on Foundations of Software Engineering and Component Based Software Engineering, pp. 379–382 (2003)
15. Bosch, J.: Design patterns as language constructs. Journal of Object-Oriented Programming 11, 18–32 (1998)
16. Java SE 6.0: VetoableChangeListener API (1998), http://java.sun.com/javase/6/
17. Hedin, G.: Language Support for Design Patterns Using Attribute Extension. In: Bosch, J., Mitchell, S. (eds.) ECOOP 1997 Workshops. LNCS, vol. 1357, pp. 137–140. Springer, Heidelberg (1998)
18. Riehle, D.: The Event Notification Pattern–Integrating Implicit Invocation with Object-Orientation. Theor. Pract. Object Syst. 2 (1996)
19. Lyon, D.A., Weiman, C.F.R.: Observer-conditioned-observable design pattern. Journal of Object Technology 6 (2007)
20. Tschantz, M.S., Ernst, M.D.: Javari: Adding reference immutability to Java. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005), San Diego, CA, USA, pp. 211–230 (2005)
21. Pratikakis, P., Spacco, J., Hicks, M.: Transparent Proxies for Java Futures. In: OOPSLA 2004: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 206–223. ACM, New York (2004)
22. De Labey, S., van Dooren, M., Steegmans, E.: ServiceJ: Service-Oriented Programming in Java. Technical Report KULeuven, CW451 (June 2006)

Part II
Evaluation of Novel Approaches to Software Engineering 2009

Automating Component Selection and Building Flexible Composites for Service-Based Applications

Jacky Estublier, Idrissa A. Dieng, and Eric Simon

Grenoble University - LIG, 220 rue de la Chimie, 38041 Grenoble, BP53 Cedex 9, France
{Jacky.Estublier,Idrissa.Dieng,Eric.Simon}@imag.fr

Abstract. Service Oriented Computing allows defining applications in which components (services) can be available and selected very late during the development process or even "discovered" at execution time. In this context, it is no longer possible to describe an application as a composite entity containing all its components; we need to perform component selection all along the application life-cycle, including execution. It requires describing an application at least partially by its requirements and goals, leaving room for delaying selection; the development system, and the run-time must ensure that the current component selection satisfies, at all time, the application description.

In this paper, we propose a concept of composite addressing the needs of advanced and flexible service-based applications, automating component selection and building composites satisfying the application description and enforcing minimality, completeness and consistency properties. We also propose tools and environment supporting these concepts and mechanisms in the different phases of the application life-cycle.

Keywords: Service oriented computing, Service selection, Service composition, Composite services, Software engineering environments.

1 Introduction

Service Oriented Computing (SOC) [13] like its predecessor, Component Based Software engineering (CBSE), allows defining an application by composing (assembling) a set of software elements named services or components and it relies on a clear separation between interface (service) and implementation (component) that increases their decoupling. But SOC emphasizes the fact that services (implementations or running instances):

- may be already available and provided by third parties,
- can appear or disappear at any time,
- may be available locally, on the net or elsewhere,
- their selection can be performed at any time including at execution,
- different implementations or instances of the "same" service may be simultaneously used in the same application,
- the same service can be used simultaneously by different applications.

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 93–106, 2010.

SOC increases thus the flexibility in the selection of required services which constitute the software application. These characteristics enable SOC to be well suitable to new kind of software applications like those managing captors, sensors and actuators.

This context does not fit the usual component based technology, which implicitly, hypothesizes a static structure of the application, with a single component implementation and instance per service, known before hand, not shared and so on. It does not mean that it is not possible to develop, with traditional technology, software applications with more relaxed hypothesis, like the one mentioned above, but in this case the designers and developers are left alone with complex and low-level technology, without any tools and methods to help them; and development turns out to be more a hacking nightmare than software engineering. To a lesser degree this also applies to service-based applications due to the current lack of support tools.

In traditional Software engineering approaches, an application is often described as a composite entity; but depending on the life-cycle phases, the composite elements and their relationships are of different nature. For example, at design time, the application can be described in terms of coarse grain functional elements with constraints and characteristics; while at deployment time, it can be a set of packaged binaries with dependencies. The "usual" composite concept fits mostly the development phase with elements being components and relationships between these components. This concept of composite is unsatisfactory for at least two reasons:

- It is too rigid to accommodate for the flexibility required by advanced applications.
- It is a low-level implementation view of the application.

Different kinds of composites have been proposed so far, adapted to different needs, different technologies and/or different contexts. Orchestration and choreography [14], [15], as well as ADL (Architecture Description Language) and configurations are different kinds of composite. For example, orchestration has been proposed to solve some issues found in service-based applications, with the hypothesis that services (most often web services) can be discovered at execution, that services do not have dependencies, and that the structure of the application is statically defined (the workflow model).

Service-based technology is becoming widespread, and an increasing number of applications applying this technology are under development. Despite this success, developing service-based applications, today, is a challenging task. Designers and developers require high level concepts, mechanisms, tools and Software engineering environments which natively support the characteristics required by advanced applications. Our main objective is to facilitate the realization of such advanced service-based software applications. In this work, we propose a concept of composite that addresses the needs of the advanced service-based applications, proposing technical concepts and mechanisms, tools and environment allowing designing, developing and executing applications which require high levels of flexibility and dynamism.

This paper is structured as follows: In Section 2, we briefly introduce our SAM / CADSE approach. Then in Section 3, we propose a way to define and manage the concept of composite addressing several needs of service-based applications. In Section 4, we present our composition environment and its associated runtime for service composites execution. Section 5 highlights related work. Finally, we present our conclusion and future work in Section 6.

2 The SAM / CADSE Approach

2.1 The Approach

In “pure” Service Oriented Architecture (SOA) like web services [1], there are not explicit dependencies and the orchestration model is a static architectural description. In dynamic service frameworks like OSGi [11], there is no explicit architectural description; it implies a dynamic behavior in which the framework is in charge to resolve dynamically the service dependencies; but conversely the application is not explicitly defined, it has no architecture; no explicit structure. There is a conflict between making explicit the application structure and content; and providing the application a large degree of dynamism.

It is interesting to mention that this conflict also exists between the early design phase when only the purpose, constraints and gross structure are defined, and the implementation phase that (usually) requires knowing the exact structure and content of the application.

To solve this conflict, we propose to see a software project as a succession of phases which purpose is to gradually select, adapt and develop components until the structure and content of an application is fully defined and complete. These phases are either performed by humans, before execution, or by machines at execution. Of course, the machine can only perform selections (as opposed to develop code). But the components automatically selected must satisfy the application needs which requires making explicit the application goal, purpose and constraints and the availability of one or more component repository.

Our solution relies on two systems: SAM (Service Abstract Machine) [5] a service framework for executing the service-based application, and a set of CADSEs (Computer Aided Domain Specific Engineering Environments) [4] in which are performed the Software engineering activities. The approach makes the hypothesis that each software engineering activity receives a composite as input, and produces a composite (the same or another one) as output. The output composite represents the same application as provided as input, but more precisely. In the same way, SAM receives a composite in input; it executes those parts that are defined, and complete those that are not fully defined, dynamically selecting the missing services.

Therefore our approach relies on a composite concept which can describe the application in abstract terms, through the properties and constraints it must satisfy, and which can describe that same application in terms of services and connections, as understood by the underlying service platform(s) e.g. OSGi or Web services. We believe that there is a continuum between these two extremes; each point being represented by a composite. All these composites and environments share the same basic SAM core metamodel, presented bellow in Section 2.2.

2.2 SAM Core

The goal of Service Abstract Machine (SAM) is to dynamically delegate the execution performed in SAM toward real service platforms, like OSGi, J2EE or Web services. Its basic metamodel, called SAM core therefore subsumes the metamodels supported by the current service platforms.

A composite, during execution, is expressed in terms of the concepts exposed by SAM core; but composites also represent the application during the early phases of the life-cycle; SAM core is the metamodel shared by all composites, both in the Software engineering activities and at execution; for that reason it must be abstract enough and independent from specific platforms and technologies.

The central concept is *Service*. But *service*, in SAM core is an abstraction containing a Java interface along with its properties (a set of attribute/values pairs) and constraints (a set of predicates). Its real subclasses are *Specification*, *Implementation* and *Instance*, seen as different materializations of the concept of *service*. *Specifications* are services indicating, through relationships *requires*, their dependencies toward other specifications. Despite being a rather abstract concept (it does not include any implementation, platform or technical concern), it is possible to define a structural composite only in terms of service specifications, as well as semantic composites in term of constraints expressing the characteristics (functional or non functional) required from the services that will be used. Still, the system is capable to check completeness (no specification is missing), and consistency (all constraints are valid) on abstract composites, making it relevant for the early phases.

An *implementation* represents a code snippet which is said to *provide* one or more *specifications*. Conversely, a specification may be provided by a number of implementations. The *provides* relationship has a strong semantics: the implementation object inherits all the properties values, relationships and interfaces of its specifications and it must implement (in the Java sense) the interfaces associated with its specifications. In particular, if specification *A requires* specification *B,* all *A*'s implementations will *require B*. An implementation can add dependencies, through relationship *requires*, toward other specifications. It is important to mention that, in contrast to most systems, an implementation cannot express dependencies toward other implementations.

Instances are run-time entities corresponding to the execution of an implementation. An instance inherits all the properties and relationships of its associated implementation. Fig. 1 illustrates the concepts of our SOA model:

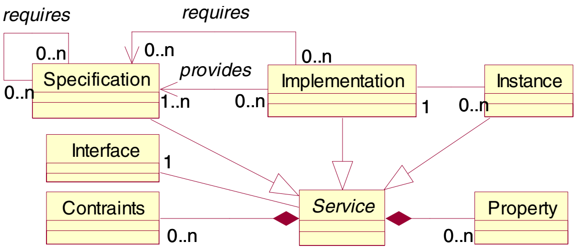


Fig. 1. SAM Core Metamodel

2.3 An Example

Developers use our CADSE environment to develop services that are conforming to the previous SAM Core metamodel. These services are available in the SAM repository. Suppose the SAM repository has the content shown in Fig. 2. In this repository, *MediaPlayerImpl* is an implementation which integrates the functionalities of both a media renderer and a controller; therefore it *provides MediaPlayer* specification. The *Log* specification has two implementations namely *LogImpl_1* and *LogImpl_2*. *LogImpl_1 requires* a *DB* (a database) and *Security* (for secure logging). *DivX* is an implementation that provides *Codec* specification, and that has a property “*Quality* = *loss*”. *Codec* has property *Unique=false* which means that an application may use more than one *Codec* implementation simultaneously. *Unique* is a predefined attribute whose semantic is known by the system, “*Unique=true”* is the default value. *Shared=true|false* (e.g. property of *LogImpl_2*) is another predefined attribute expressing the fact that an implementation or an instance can be shared by different applications (*false* is the default value).

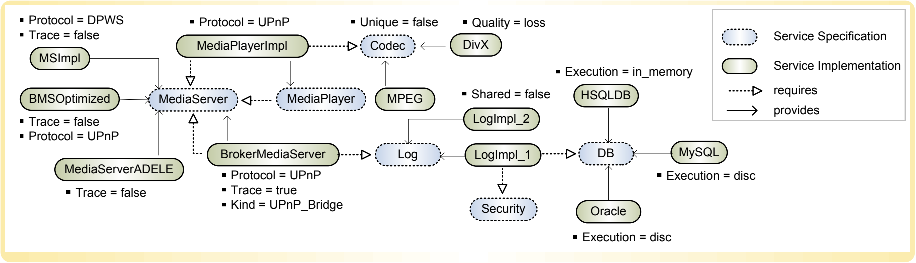


Fig. 2. A SAM Repository

The *MediaPlayer* specification *requires MediaServer* specification which means that all its implementations (e.g. *MediaPlayerImpl*) also require *MediaServer* specification. From a service (e.g. *BrokerMediaServer*) we can navigate its relationships to obtain for example the service(s) it requires (*Log*) or its available implementations (*LogImpl_1* and *LogImpl_2*) following the *provides* relationship.

Services may have constraints which, like in OCL, are predicates. In contrast with OCL, these constraints can be associated both on types and on service objects (i.e. Specifications, Implementations and Instances). The language allows both navigating over relationships and defining LDAP search filters as defined in [6]. For example, *LogImpl_1* implementation may declare that it *requires* an *in_memory DB*. This constraint can be expressed as follows:

```
Self.requires(name=DB)..provides (execution = in_memory);
```

Self denotes the entity context on which the constraint is associated (*LogImpl_1). Self.requires* denotes the set of entities required by Self (*{DB, Security}*); *(name = DB)* select the elements of the set satisfying the expression (*DB*), *..provides* is a reverse navigation which returns all elements that provide *DB* (*{Oracle, MySQL, HSQLDB}*); finally the expression returns the set {HSQLDB} since it is the only *DB* implementation with property (*execution= in_memory*). An expression returning at least an element is considered true. The constraint means that from the point of view of the object origin of the constraint (*LogImpl_1*), *DB* has a single valid implementation: *HSQLDB*. The *BrokerMediaServer* implementation may declare that it requires *MediaServer* implementations that are *UPnP*, but not a bridge:

```
Self.requires(name=MediaServer)..provides
              (&(kind!=UPnP_Bridge)(protocol=UPnP));
```

These previous constraints must be respected by all applications that use services defining them. If the constraint follows a single relationship type, it expresses which relationships are valid. When the relationship is created, the constraint is evaluated for that relationship, if false the relationship cannot be created. For example, if we want that *Codec* implementations cannot have more dependencies than *Codec* itself, we can set the constraint:

```
equals(Self.requires, Self..provides.requires);
```

In this example, *Self* denotes the *Codec* specification that defines the constraint. *Self..provides* denotes the set of implementations which provide it (*MPEG* and *DivX*). This constraint is relevant for **all** implementations providing *Codec*. Therefore, such constraints enforce some repository integrity.

Let us introduce our Media Player Application (MPA) example that we use thereafter in this paper. In our system, each MPA to be built is a composite which consists of a media renderer (which consumes a flux, for example a video stream), a media server (which provides flux found in a storage), and a controller that interacts with the customer and connects servers and renderers. Each MPA must fulfill a set of characteristics (properties and constraints) to be consistent. Its components (services) should be chosen among those available in the SAM repository if available, if not they have to be developed, but in any case these services must be compatible and deliver the desired characteristics of the composite.

3 Composite

Suppose that we want to build an UPnP-based home appliance MPA such that, when running in a particular house, it is capable of discovering the media servers available in that house and to provide *MediaPlayer* functionalities. Defining that MPA in the traditional way, as a composite which gives the full list of components, is inconvenient, or even impossible for a number of reasons:

- Some components may not exist or not be known (yet) and
- there is no guaranty of composite's completeness, consistency and optimality properties.

Creating a complex composite on real cases, is not only a time consuming and error prone task; but it may be simply impossible when components are missing (they must be developed like the *Security* service in Fig.2) or when components are selected in a later phase, or even discovered during execution (like *MediaServers* and *Codecs*). Our goal is to propose a way to build and manage composites that avoid the above pitfalls. Such composites are rather demanding; indeed they require the following properties:

- ***Completeness control.*** A composite must be capable of being explicitly incomplete; this is the case when components will be developed, when design choices have not made or when the components cannot be known before execution time. The composite must tell what is missing and why.

- ***Consistency control.*** The composite must be able to detect and report inconsistencies and constraints violation.
- ***Automation and optimality.*** The system should be capable to compute an optimal and consistent list of components, satisfying the composite requirements and constraints, but also the degree of completeness required.
- ***Evolution.*** Incomplete composite must be such that they can be incrementally completed.

The following sections present the concepts and mechanisms for composite management.

3.1 Static Composite Definition

In our system, we define a composite as an extension of the SAM core presented above. The extension presented in Fig. 3 is only one of the different SAM Core extensions; indeed other simpler composite concepts have also been defined and implemented.

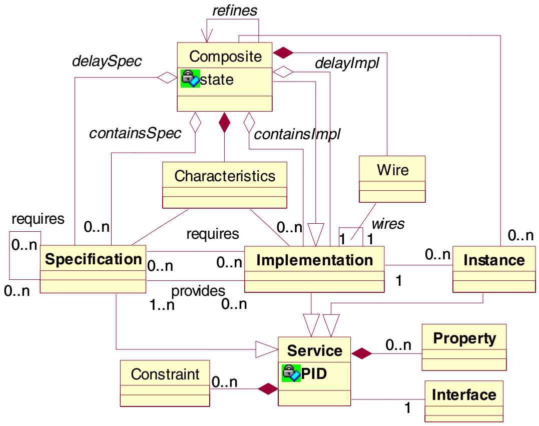


Fig. 3. Composite Metamodel

As shown in Fig. 1 a SAM composite is a service implementation that can contains specifications (*containsSpec* relationship), implementations (atomics or composites; *containsImpl* relationship) and instances. Classically, a SAM composite can be defined by the list of its service components (specifications and/or implementations) setting explicitly the *containsSpec* and *containsImpl* relationships.

Being an implementation, a composite is not necessarily self-contained; it may have *requires* relationships toward other services. Similarly, it is not necessarily complete. As explained above, a composite must be capable of being incomplete by giving explicitly the delayed choices of components. Thus, a *delaySpec* or *delayImpl* relationships toward an entity *E* express the fact that the selection process should not follow the *E* dependencies. Delayed service selections can be carried out at any later time for example during development, at deployment or at execution; the strategy is up to the user.

3.2 Automatic Composite Building

Selecting manually the components of a composite, and creating explicitly the associated relationships is tedious and error prone since this manual process does not guaranty *minimality* (all components are useful), *completeness* (all required components are present) nor *consistency* (constraints are all valid). To simplify the process of defining a composite and to enforce the properties of *completeness*, *consistency* and *minimality*, we need an automatic composite construction mechanism. Thus, we need a language in which it is possible to specify the required characteristics of the composite to build, and an interpreter, which analyses the composite description and selects, in the repository, the components that together constitute a composite satisfying the description, complete and consistent.

Therefore, a SAM composite can also be defined by its goal i.e. by its characteristics, properties and constraints. We use a language to describe the intended properties and constraints of composites. To create a composite, the designer first defines the Specification(s) it provides, and then, optionally, imposes some choices explicitly creating relationships indicating the Specifications it *requires*, the Specifications or Implementations it *contains* and those it *delays*. Then the designer expresses the expected composite properties; our system performs the rest of the job. For example, to build our UPnP-based home appliance MPA, one could first create a *provides* relationship to *MediaPlayer*, a *delaySpec* relationship to *Codec*, and a *containsImpl* relationship to *MPEG* as in Fig. 4:

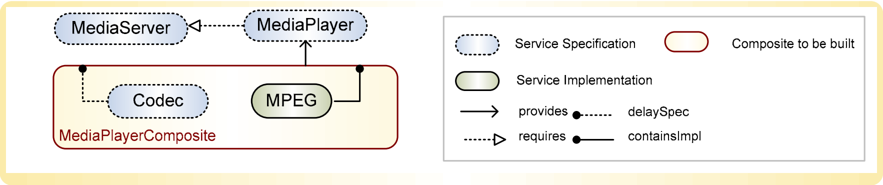


Fig. 4. Composite initial MPA definition

Then we can declare the intended characteristics of the MPA as follows:

- We want to create a MPA that uses UPnP as protocol:
  `Select Implementation (Protocol=UPnP);`
- The MPA to build must provides a trace of the executed actions:
  `Optional Implementation (Trace=true);`
- The MPA should foster service sharing whenever possible.
  `Select Implementation (Shared=true);`

This language is an extension of the constraint language, in which *Self* can be replaced by any set, including a complete type extension, like *Implementation* meaning all actual implementations found during the selection process. In our example, traces are preferred but not required. Since the system is weakly typed, an expression is ignored if no element (in the selection set) defines the attribute; if only one element defines the attribute with the good value, it is selected. The first sentence means that we must select an implementation with *protocol=UPnP* for those Specifications for which at least one Implementation defines the *protocol* attribute; in our example, this selection applies only to *MediaServer*.

These expressions are interpreted when computing which are the required services and selecting the implementations and instances that fulfil (1) the intended composite constraints, and (2) all the constraints expressed by the already selected components. For instance, if we select the *LogImpl_1* implementation then we will necessarily select *HSQLDB* since it is the only DB that satisfies the *LogImpl_1* constraints.

Based on the description of a composite i.e. its initial relationships and its constraints, an interpreter computes and selects the required services that satisfy the composite characteristics. The interpreter "simply" starts from the specification provided by the composite (e.g. *MediaPlayer* in our MPA), and follows the *requires* relationships to obtain all required specifications. It also follows the *requires* relationships of its contained services (*containsSpec* and *containsImpl* relationships). For each *specification* found (except those delayed and those explicitly required by the composite itself), it tries to select one or more implementations satisfying all the constraints associated with the services already selected and the selection expressions defined by the composite itself. For each selected implementation the interpreter iterates the above steps to found other required services. For instance, from the repository in Fig. 2 and the selection expressions declared by the MPA, the interpreter builds the following composite Fig. 5:

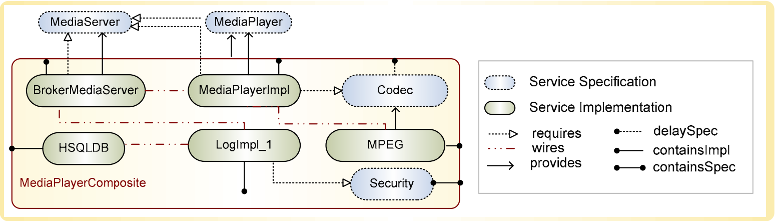


Fig. 5. A configuration of the MPA composite

For the *MediaPlayer* specification, the interpreter selected its unique implementation *MediaPlayerImpl* but *MediaPlayerImpl* requires a *MediaServer,* therefore *BrokerMediaServer* is selected since it is the only *MediaServer* implementation satisfying the composite selection *(Protocol=UPnP)*. In turn *BrokerMediaServer* being selected, a *Log* service is required, and *LogImpl_1* is selected, because *LogImpl_2* does not satisfy the composite constraint *(Shared=true)*; and consequently *HSQLDB* (because of the *LogImpl_1* constraint) and *security* because of the *requires* relationship. Unfortunately S*ecurity* has no available (or no convenient) implementation; the state of the composite is "incomplete" and this specification is added in the MPA composite through the *containsSpec* relationship. *Codec* being delayed, the system does not try to select any of its implementations; at run-time, depending on the discovered *MediaServers*, the required *Codecs* will be installed. Since the *Codec* implementations cannot have other dependencies (because of the *Codec* constraint), there is no risk, at execution, to depend on an unexpected service. Composite dynamic execution behaviour will not be discussed in this paper.

Our system guarantees minimality (nothing is useless), completeness (except for explicit delays) and consistency since all constraints and selection expressions are satisfied. But we no not guaranty optimality because the system may fail to find the "best" solution or even a solution when one does exist. Indeed, during the selection process, if a specification with *(unique= true)* has more than one satisfactory implementations, the system selects one of them arbitrarily. It may turn out to be a bad choice if the selected one sets a constraint that will later conflict with another component constraint. The solution consists in backtracking and trying all the possibilities, which turns out to be too expensive in real cases.

3.3 Composite Contextual Characteristics

SAM composite extends SAM Core defining new concepts (e.g. composite), new relationships (e.g. contains, delay); but any individual composite can also extend the existing services with properties that are only relevant for the composite at hand. These relationships and properties are called contextual characteristics (see Fig. 3).

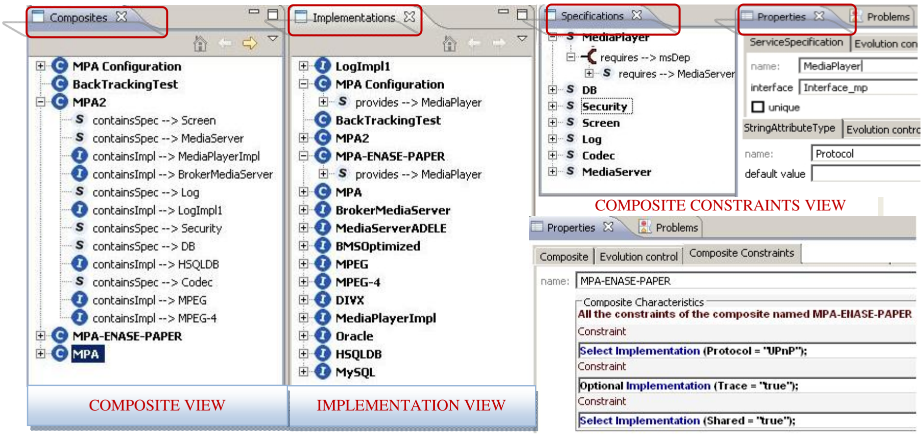
For example, *wire* is a contextual relationship; it means that two implementations are directly linked but for a given composite point of view only; this is not true in general. We may wish to add property to some implementations such that constant values, parameters or configuration information which are those required in a given composite only, like bufferSize, localPath and so on. Implementations, along with their contextual properties are called components in SCA [12]. Contextual characteristics may apply to implementations, in order to create instances with the right initial values, as in SCA, but also to specifications, when specific implementation can be generated out of some parameters. More generally, contextual characteristics, including constraints, apply to any delayed service, when the selection must be performed in the scope of the current composite, and with the properties only relevant in that scope. This is fundamental when selections are delayed until execution.

4 SAM Composite System: Tools, Environment and Runtime

The process of undertaking a software application has a rather complex life-cycle consisting of several phases like software specification and design, development of the software elements (components), developing and performing a series of unit tests, deploying these elements, composing the application, software execution and so on. In each phase, several stakeholders handle a set of concepts for achieving the related software engineering activities. These concepts may be very different depending on the abstraction level and the task at hand. Stakeholders need therefore assistance and software engineering support to facilitate and automate theirs activities. For "each" phase, we propose to use a CADSE (Computer Aided Domain Specific Engineering Environment) dedicated to that phase. We see a software application project as a succession of phases which purpose is to gradually develop new components, select or adapt existing ones that constitute the full and consistent software. We try to identify the link between these phases in order to automate the phase's transitions, using the concept of composite. In this paper, we present only our environment and tools for service composition and its associated runtime for executing and managing service composites. The following sections describe the main properties of our tools and environment, and the composite runtime.

4.1 Composite Designing Environment

The environment we propose for designing and computing a composite service is a set of Eclipse features and plug-ins generated by our CADSE technology [4]. CADSE is a model-driven approach in which designers and developers interact only with models views, editors and wizards, and not directly with the Eclipse IDE artifacts. The Composition CADSE generates the associated IDE artifacts and source code from these models. Thus, our environment allows specifying the composition at a high level of abstraction. It is also based on the separation of concerns idea: the concepts and aspects relevant for a class of stakeholders are gathered through a set of specific views, for instance those allowing defining service specifications, implementations or composites. Screenshot (Fig. 6) illustrates these views and describes the definition, properties and constraints of a service (specification, implementation or composite).



name: MediaPlayer
interface: Interface_mp
□ unique

name: Protocol
default value:

name: MPA-ENASE-PAPER

Composite Characteristics
All the constraints of the composite named MPA-ENASE-PAPER
Constraint
Select Implementation (Protocol = "UPnP");
Constraint
Optional Implementation (Trace = "true");
Constraint
Select Implementation (Shared = "true");

Fig. 6. A screenshot of our CADSE composition

The Environment integrates several tools and editors allowing for example:

- defining the intended application through the constraints and properties required from the services that will be part of that application;
- validating service constraints and showing errors and/or warnings messages. The system verify the syntax during constraint edition, if the name or attributes (properties) of services used in an expression exist and/or are correct or not;
- performing automatic component (service) selection satisfying the composite characteristics (requirements and constraints) in order to compute and build automatically a consistent configuration (list of components) of the composite;
- delaying (or not) some component selection and evaluating the validity of the new composite (its selected components);
- ….

Our CADSE Composition environment is extensible to support the addition of new concepts, features or functionalities. Easy extensibility, maintenance and evolution are one of its essential properties. It allows enforcing completeness and consistency properties of computed composites.

4.2 Composite Runtime

The SAM system provides a Composite runtime support for executing and managing composite services. This runtime platform receives as input a composite description as produced by the composite CADSE and performs the needed actions in order to execute that composite enforcing, at each time, the satisfaction of its description (its constraints and requirements). Our composite runtime executes the parts (services) of a composite that are defined and completes those that are not fully defined by dynamically selecting the missing services, if they are available and if they fulfill the composite characteristics.

The composite runtime gradually and incrementally transforms the composite description until it contains only service instances connected by wires satisfying the constraints. To do so, depending on

- the current execution context (the running services), and
- the implementations available in the currently reachable repositories, and
- the currently defined implementations (in the composite description), and
- the composite constraints and requirements, and
- the already selected services constraints and requirements.

The composite runtime creates *wires* relationships linking two service implementations, establishes *containsSpec* or *containsImpl* relationships specifying the selected services, sets attributes like *complete* or *delayed* to make explicit the composite state (it is complete or not, delayed or not and so on). We use the same constraints language, selection algorithm and tools in the CADSE Composition environment (design phase) and in the Composite runtime (at execution time). The following figure illustrates our system from design to execution of a composite:

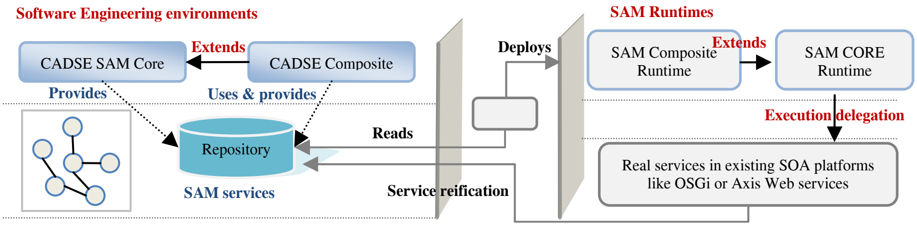


Fig. 7. Our system architecture

5 Related Work

We can classify the approaches and languages for service composition as orchestration, structural and semantic [8], [3], [10].

Orchestration [14] is a recent trend fuelled by web services for which de facto standard is Business Process Execution Language for Web Services (WS-BPEL) [9]. Structural composition defines the application in terms of service component linked by dependency relationships. Service Component Architecture (SCA) specification [12] is a structural SOA composition model [14]. Most efforts to automate service composition are performed in the web semantic community. The hypothesis here is that services do not have dependencies, and that specifications include a semantic description using ontology languages such as OWL (Ontology Web Language) or WSML (Web Service Modeling Language). The goal is to find an orchestration that satisfies the composite semantic description (OWL-S [16] or WSMO [17]).

Automaton of service selection has been addressed in many research works focusing on quality-of-service (QoS) criteria like reliability or response time. [7] propose a QoS based service selection and discuss about the optimisation of this selection using heuristic approaches. [2] propose an approach for dynamic service composite and reduce the dynamic composition to a constraints (ontology based) satisfaction problem. In most of these systems, QoS requirements are specified at the overall application level. Therefore, it becomes unclear how to derive the QoS goals from participating services [18]. In our approach, service properties and requirements can be specified both on the individual services that will participate in a given composition and on the composites themselves.

6 Conclusions and Future Work

SOC represents the logical evolution we are witnessing in software engineering: increasing the decoupling between specifications (interfaces) and implementations, increasing the flexibility in the selection of implementations fitting specifications, delaying as much as possible the selection including the execution time, allowing multiples implementations and instances of the same service to pertain to the same application, and finally allowing some services to be shared between different applications during execution. Building a complex (service-based) application in this context is very challenging.

In the traditional way, an application is defined in the early phases by a number of documents and models in which the purpose, constraint and architecture of the application are defined; but in the later phases of the application life-cycle an application is specified as a full list of its components (being services or not), often called a composite (or configuration). Building the composite manually is tedious and error prone when components are developed independently and have conflicting requirements. It becomes virtually impossible when some selections are to be done very late in the life-cycle.

In this paper, we propose to extend the concept of composite in order to represent faithfully the application along the different life-cycle phases, from design to execution. To that end, the composite must contain a high level description of the application, in terms of properties and constraints it must satisfy, and on the other hand in terms of components, bundles and run-time properties. We also propose a full-fledged set of tools, environments and runtime for designing, computing from a goal (the application requirements and constraints) and executing complex service-based applications using a flexible and extensible concept of composite. In this work, we show how it is possible to go seemingly from the high level description to the execution, and how the system, all along this long process, is able to compute and enforce the conformity and compatibility of the different composite descriptions, while enforcing minimality, completeness and consistency properties. Our work is a step toward the above goal, but even in its current form, it provides a large fraction of the properties discussed above and show the feasibility of the approach. We expect future work to introduce how the dynamism of service-based applications is managed in our system and to present more precisely the service composite execution.

SAM is available at http://sam.ligforge.imag.fr and CADSE at http://cadse.imag.fr.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, H.: Web Services – Concepts, Architectures and Applications. Springer, Heidelberg (2003)
2. Channa, N., Li, S., Shaikh, A.W., Fu, X.: Constraint Satisfaction in Dynamic Web Service Composition. In: 6th International Workshop on Database and Expert Systems Applications, pp. 658–664 (2005)
3. Dustdar, S., Schreiner, W.: A survey on web services composition. International Journal of Web and Grid Services (IJWGS) 1, 1–30 (2005)
4. Estublier, J., Vega, G., Lalanda, P., Leveque, T.: Domain Specific Engineering Environments. In: APSEC 2008 Asian Pacific Software engineering Conference (2008)
5. Estublier, J., Simon, E.: Universal and Extensible Service-Oriented platform. Feasibility and Experience: The Service Abstract Machine. In: The 2nd International Workshop on Real-Time Service-Oriented Architecture and Applications (RTSOAA). IEEE, Los Alamitos (2008)
6. Howes, T.: RFC 1960: a String Representation of LDAP Search Filters (1996), http://www.ietf.org/rfc/rfc1960.txt
7. Jaeger, M.C., Mühl, G.: QoS-based Selection of Services: The implementation of a Genetic Algorithm. In: KiVS Workshop: Service-Oriented Architectures and Service Oriented Computing, pp. 359–370 (2007)
8. Milanovic, N., Malek, M.: Current solutions for web service composition. IEEE Internet Computing 8, 51–59 (2004)
9. OASIS, Web Service Business Process Execution Language Version 2.0. (2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
10. Orriens, B., Yang, J., Papazoglou, M.P.: Model Driven Service Composition. In: Orlowska, M.E., Weerawarana, S., Papazoglou, M.P., Yang, J. (eds.) ICSOC 2003. LNCS, vol. 2910, pp. 75–90. Springer, Heidelberg (2003)
11. OSGi Release 4, http://www.osgi.org/Specifications/HomePage
12. OSOA, Service Component Architecture: Assembly Model Specification Version 1.0. (2007), http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications
13. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. IEEE 40, 38–45 (2007)
14. Papazoglou, M.P., Van den Heuvel, W.J.: Service oriented architectures: approaches, technologies and research issues. VLDB Journal 16, 389–415 (2007)
15. Pedraza, G., Estublier, J.: An extensible Services Orchestration Framework through Concern Composition. In: Proceeding in International Workshop on Non-functional System Properties in Domain Specific Modeling Languages, NFPDSML (2008)
16. W3C, Semantic Markup for Web Services (2004), http://www.w3.org/Submission/OWL-S/
17. WSML: Web Service Modeling Language, http://www.wsmo.org/wsml/
18. Yen, I.-L., Ma, H., Bastani, F.B., Mei, H.: QoS-Reconfigurable Web Services and Composition for High-Assurance Systems, vol. 41, pp. 48–55. IEEE Computer Society Press, Los Alamitos (2008)

An Aspect-Oriented Framework for Event Capture and Usability Evaluation

Slava Shekh and Sue Tyerman

School of Computer and Information Science
University of South Australia, Australia
shesy006@students.unisa.edu.au, Sue.Tyerman@unisa.edu.au

Abstract. Recent work in usability evaluation has focused on automatically capturing and analysing user interface events. However, automated techniques typically require modification of the underlying software, preventing non-programmers from using these techniques. In addition, capturing events requires each event source to be modified and since these sources may be spread throughout the system, maintaining the event capture functionality can become a very arduous task. Aspect-oriented programming (AOP) is a programming paradigm that separates the concerns or behaviours of a system into discrete aspects, allowing all event capture to be contained within a single aspect. Consequently, the use of AOP for usability evaluation is currently an area of research interest, but there is a lack of a general framework. This paper describes the development of an AOP-based usability evaluation framework that can be dynamically configured to capture specific events in an application.

Keywords: Aspect-oriented programming, Usability evaluation, Human-computer interaction.

1 Introduction

Usability evaluation is a technique used in the area of human-computer interaction (HCI) to assess how easily a user can interact with an interface. Software applications generate user interface events as part of their operation, and these events are used to assist in usability evaluation. Recent work in this area has examined techniques for automating usability evaluation by automatically capturing and analysing user interface events [1]. Automation allows the events to be captured with minimal human intervention and the event capture is unobtrusive from the perspective of the user, so the act of observation does not affect the activities being observed.

However, in order to automatically capture these events, the underlying software needs to be modified, requiring further preparation time for the usability evaluation and the involvement of a programmer to make the modifications. In comparison, a non-automated method, such as heuristic evaluation, can be performed by a non-programmer [2].

Additionally, user interface events occur in different components of the application, making it necessary to modify the system in each of these places to facilitate event capture. These modifications become increasingly difficult to manage in larger systems and extending the software becomes a very arduous task, because the software developer

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 107–119, 2010.

needs to find and change each individual part. For instance, consider a situation where the current system captures events by displaying them on the screen. If the need arises for these events to also be stored in a database, then each component of the system needs to be individually modified in order to implement the change. If the system contains hundreds or thousands of components, then modification can become infeasible.

Nonetheless, the limitations described can be overcome using aspect-oriented programming (AOP). AOP is a programming paradigm that focuses on separating the cross-cutting concerns of a system. A *concern* is a function or behaviour of the application, and can be categorised as either a *core concern* or a *cross-cutting concern*. A core concern is a primary behaviour, while a cross-cutting concern is a behaviour that spreads across multiple parts of the system [3].

Consider a simple banking application, which supports three functions: *withdraw money*, *view balance* and *log event*. An event is logged when either *withdraw money* or *view balance* is executed. *Withdraw money* and *view balance* are core concerns, because they provide the primary behaviour of the system, while *log event* is a cross-cutting concern, as its functionality needs to be integrated into both *withdraw money* and *view balance* in order for it to be able to log events generated by those components.

Traditional paradigms, such as object-oriented programming (OOP) provide limited support for separation of concerns. For example, developing a software system with OOP is based on the idea of *objects*. A single object can be used to handle a single core concern, such a system function, but is not able to encapsulate a cross-cutting concern, such as event logging. Conversely, AOP addresses separation of concerns using *aspects,* which are able to effectively encapsulate both core and cross-cutting concerns. Each aspect is designed separately from the rest of the system and all aspects are then integrated into the system using an *aspect weaver*. A weaver adds the aspect code into the base code of the application at specific points that are defined by the programmer, called *join points* (see Figure 1).

2 Related Work

AOP offers a number of opportunities that could benefit human-computer interaction.

2.1 Event Tracing

The use of AOP for event tracing was demonstrated by Low [4] in the development of Toolkit for Aspect-oriented Tracing (TAST), which handles all trace-related activities in Java applications. TAST is a support tool for the Test and Monitoring Tool, which had previously been developed at Siemens.

In addition to Low's work on event tracing, AOP has been used in the development of a trace monitor [5]; a tool that observes a system to detect sequences of events and takes action when events occur in a particular pattern. The tool was developed using a combination of AspectJ (a popular Java-based implementation of AOP) [6], and another language called Datalog. However, AspectJ can only trigger code at the occurrence of a single event, so the authors introduced a new language feature to AspectJ called "tracematches", which allows code to be triggered when an event sequence matches a pattern.

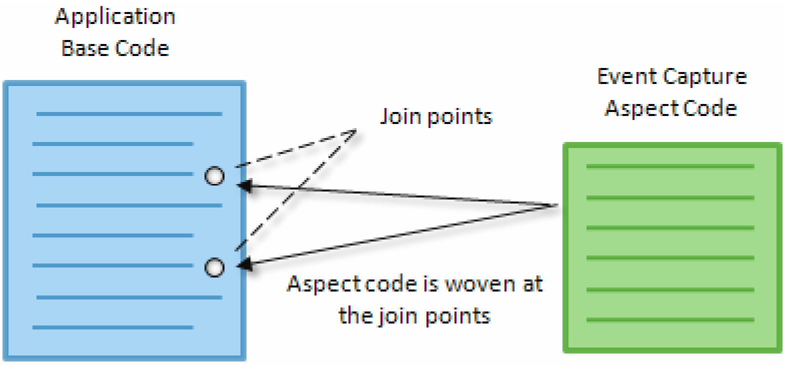


Fig. 1. Basic operation of an aspect weaver

2.2 Event Capture

Hartman and Bass [7] described and implemented an event logging system for graphical applications to capture additional event logging information at the architectural boundaries of the application. The system was implemented using Java and Swing, and cross-cutting concerns, such as logging, were handled by AspectJ. The system was then applied to two applications and tested with a small user group. The results showed that the new system addressed many of the log-related problems that were described by earlier authors, including Hilbert and Redmiles [1], in their survey of usability evaluation techniques.

Tao further explored AOP-based event capture by using AOP as a means of capturing user interface events that occurred in different parts of the application [3, 8]. Aspects were added to the application, with the responsibility of capturing different events, along with contextual information. The use of an aspect-oriented approach allowed system-wide events to be captured in a single location.

2.3 Usability Evaluation

Extending on aspect-based event capture, Tarta and Moldovan [9] used AOP to automate the process of usability evaluation by developing a usability evaluation module, which captured events using an aspect-oriented approach. This module was integrated as part of a working application and then tested with a small user group. The testing showed promising results when compared to traditional usability evaluation techniques, such as event log analysis. Their research also provides ample opportunities for future work, such as the need for a usability evaluation framework based on AOP. The implementation of such a framework is the primary focus of this paper.

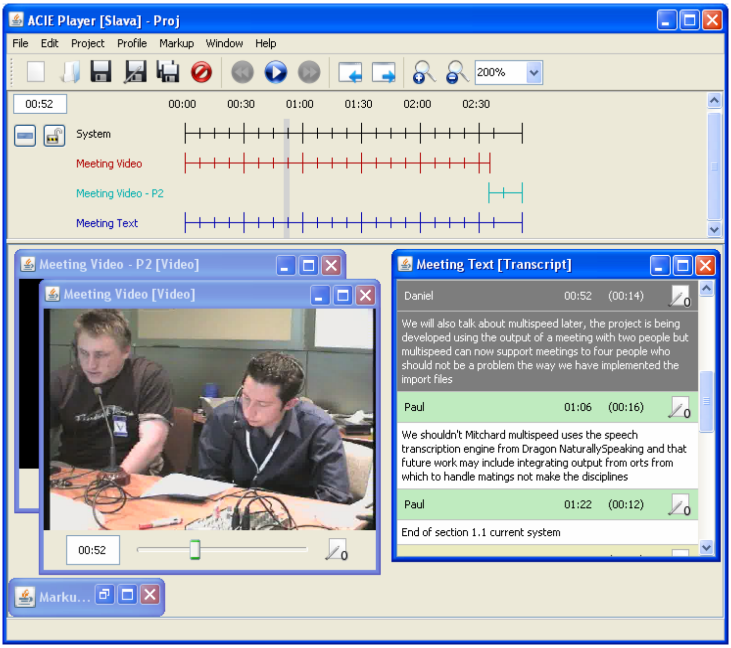
Although AOP is a suitable approach for conducting usability evaluation, there are other techniques that may also be useful. One group of researchers compared two approaches for conducting usability evaluation: aspect-oriented programming and agent-based architecture [10]. The comparison described both techniques as being effective for usability evaluation, and showed that AOP could support usability testing without the need to modify the original application code. The comparison provides valuable insights into different techniques for conducting usability evaluation and highlights the validity of an aspect-oriented approach.

3 Framework

The primary aim of the research presented in this paper was to develop a usability evaluation framework using AOP, with no impact on the source code. If AOP was not used, in the worst case scenario, twenty duplicate sets of logging code would be needed to record mouse events alone, leading to potential mistakes and difficulties with maintenance. With over 100 different events being observed, AOP has a clear advantage over traditional approaches.

The main purpose of the framework is to capture user interface events and assist in usability evaluation. At the same time, the framework aims to be as re-usable as possible, so that it can be used in different applications, following the recommendations of [9].

In order to capture events, the framework needs a software application to act as a source of events; an event being any change of state in the system. In this implementation, a Java-based application called the ACIE Player has been used as the event source. The Player provides an integrated environment for viewing synchronised data streams, including audio, video and transcribed text (see Figure 2). In addition, the Player also allows the user to modify the transcribed text and annotate the different data streams.



Daniel 00:52 (00:14) 0

We will also talk about multispeed later, the project is being developed using the output of a meeting with two people but multispeed can now support meetings to four people who should not be a problem the way we have implemented the import files

Paul 01:06 (00:16) 0

We shouldn't Mitchard multispeed uses the speech transcription engine from Dragon NaturallySpeaking and that future work may include integrating output from orts from which to handle matings not make the disciplines

Paul 01:22 (00:12) 0

End of section 1.1 current system

Fig. 2. Screenshot of the ACIE Player

The ACIE Player has been chosen as the event source for the framework, because it is a GUI-oriented application, which generates a large number of user interface events. This provides the framework with a lot of event data to capture and analyse. Since the ACIE Player is a Java-based application, a popular Java AOP implementation called AspectJ has been used for developing the aspect-oriented parts of the framework.

The framework consists of four main components: *aspect interface*, *event capture module*, *framework frontend* and *output module*. These components and their interactions are shown in Figure 3, and described in more detail below.

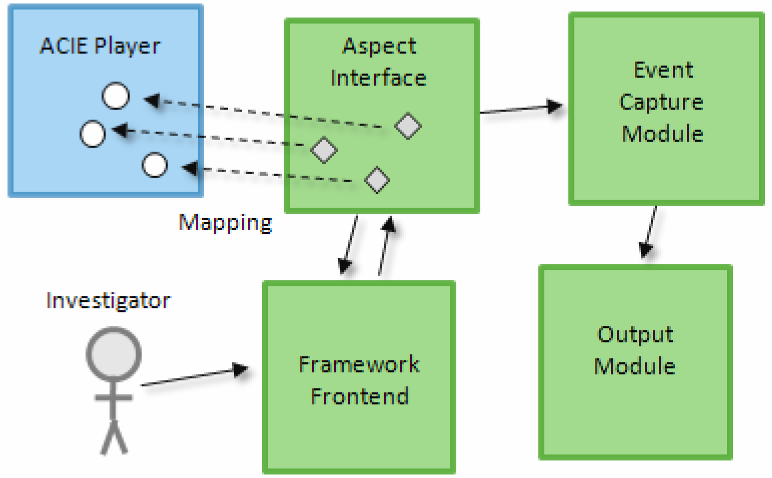


Fig. 3. Overview of the framework architecture

3.1 Aspect Interface

The aspect interface is specific to the ACIE Player, while the other components can be re-used by other applications that adopt the framework. Initially, a programmer needs to create the aspect interface for the target application, so that the framework has knowledge of the events that can be generated by the application. The aspect interface is populated with a series of join points that map to the GUI components which can generate events.

For instance, the work area is a GUI component in the ACIE Player and contains a method for cascading windows, which generates the event of windows being cascaded. The following join point in the aspect interface maps to this method and thus, captures the event:

```
after() : target(WorkArea) && execution(* cascadeWindows()) {
  String action = "Windows Cascaded";
  Event event = new Event(action);
  CaptureModule.logEvent(event);
}
```

Although the joint point mapping needs to be done by a programmer, each unique application only needs to be mapped once.

3.2 Framework Frontend

Once the programmer has created the aspect interface, an investigator can configure the interface and perform a usability evaluation without any more input from the programmer. In order to configure the interface, the investigator uses the framework frontend. This component provides a GUI with a series of checkboxes (see Figure 4), which are automatically generated based on the join points present in the aspect interface. The investigator then selects checkboxes based on the GUI components and specific events that they are interested in logging.

In addition, the final tab of the framework frontend allows the investigator to select their preferred output formats. The investigator's overall selection in the frontend is saved to a configuration file.

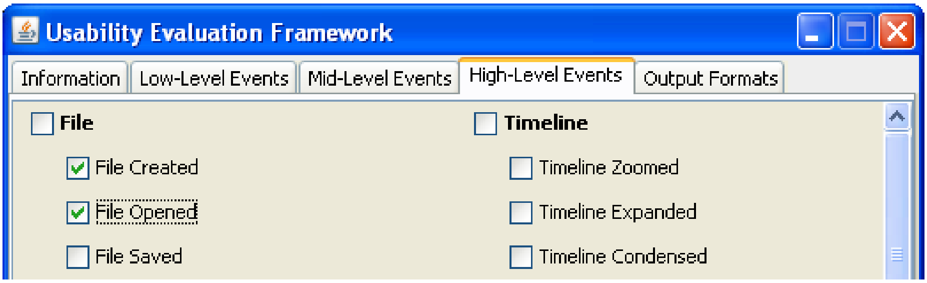
3.3 Event Capture Module

Once the configuration file has been set up, the framework can begin logging events. While a user is interacting with the ACIE Player, the framework runs in the background and records their actions. However, the framework only logs the events that are specified in the configuration file, which is determined by the investigator's selection in the framework frontend. Each time an event occurs and has a corresponding entry in the configuration file, the aspect interface requests the event capture module to log that event. The event capture module stores the event internally, and then passes it on to the output module.

3.4 Output Module

The output module is responsible for processing events into different output formats. Currently, the module is capable of producing the following formats:

1. Displaying events in the system console
2. Writing events to a plain text file
3. Writing events to a comma-separated values (CSV) file
4. Writing events to an XML file
5. Generating a linear graph visualisation of the events
6. Generating a complex graph visualisation of the events



□ File

☑ File Created

☑ File Opened

□ File Saved

□ Timeline

□ Timeline Zoomed

□ Timeline Expanded

□ Timeline Condensed

Fig. 4. Screenshot of a section of the framework frontend

The first four output formats are all textual representations of the event data. The console output is a temporary format, existing only during the execution of the software, while the other formats provide permanent storage of data. Each output format stores the event data in a different structure, which offers alternative data representations for event analysis. In addition to textual formats, the framework is also able to generate graph visualisations of the event data. Each event is modeled as a node and relationships between events are shown as directed edges in the graph.

The output module offers two types of graph visualisations (referred to as linear and complex), which differ in their placement of nodes, but present the same content. A linear graph visualisation is shown in Figure 5. The rectangles represent GUI components in the ACIE Player, while the other shapes correspond to different events. The colour shading of the shapes indicates the time of the event's occurrence, where darker shapes represent earlier events.

The six output formats currently supported by the output module show the flexibility of the framework in being able to generate different types of output. Additional formats could be added quite easily, meaning that the framework could potentially generate output for a wide range of purposes.

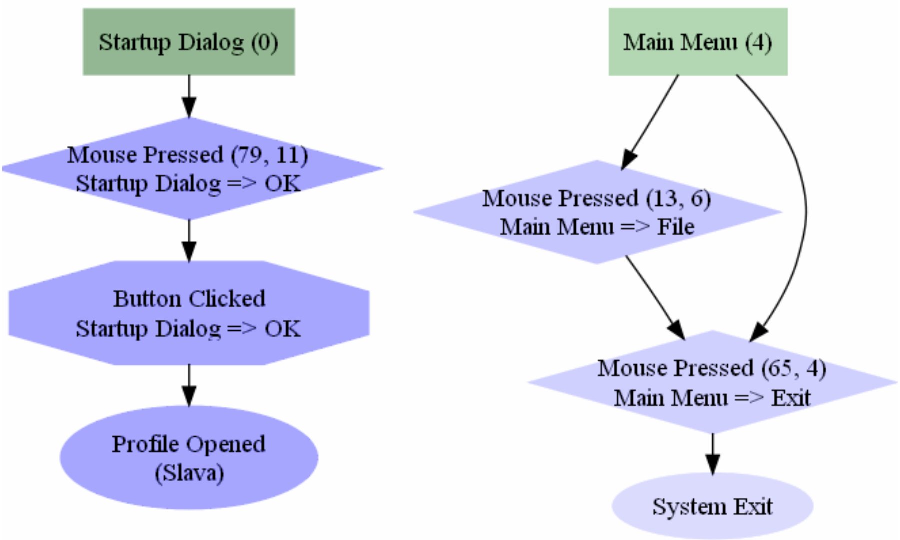


Fig. 5. Linear graph visualization

4 Case Study

The framework was tested and validated using a case study. The study was targeted at the Information Systems Laboratory (InSyL) of the University of South Australia, which is a specific user group of the ACIE Player [11]. The case study consisted of an investigation experiment and a usability experiment. The investigation experiment evaluated the use of the framework itself, while the usability experiment tested the effect of AOP on system performance to identify whether AOP was a suitable driving technology for the framework.

The participants selected for both experiments were a combination of programmers and non-programmers, which helped evaluate the general effectiveness of the framework for all user types. All participants completed the usability experiment first, and thus, this was their initial point of contact with the ACIE Player and framework.

4.1 The Case

Consider the following scenario, which illustrates how the ACIE Player is used by members of InSyL and is the target case for the study.

An analyst is interested in observing a recorded meeting and analysing the behaviour and interaction of the meeting participants. The analyst has access to the video and audio streams of the recorded meeting, and a transcript that is automatically generated from the audio. The analyst observes the video and transcript in small sections (i.e. 10 seconds at a time), and makes textual annotations based on the behaviour of participants. The analyst records the annotations using a standard syntax to make the information easier to categorise. For instance, if a participant stands up and says something, the analyst might record “P1: stand, speak”.

Performing this task without an integrated tool is problematic, because the analyst needs to run multiple applications concurrently, and manually synchronise them. For example, they might run a video player to watch the meeting, a transcription program to read the meeting transcript and a text editor to record their annotations. Since all of these are separate applications, the analyst will have to constantly check that the data from each application is synchronised.

The ACIE Player is preferable for performing the task, because it is able to provide all of the required capabilities in an integrated and synchronised environment. Using the ACIE Player minimises the cognitive overhead for the analyst in performing their task, since the work is managed within a single application. This also reduces the likelihood of errors and inconsistencies in the analysis, because the analyst no longer needs to spend time managing and synchronising different media and software applications, and is able to focus more on their actual task.

4.2 Usability Experiment

In the usability experiment, all participants took on the role of a user. The main purpose of this experiment was to evaluate whether or not users could detect any degradation in system performance due to the presence of the framework. This helped to identify whether AOP was a suitable technology for capturing events and running the framework.

Each user carried out two simple tasks using the ACIE Player, based on a set of instructions. In both tasks, the overall flow of activities was as follows:

1. Launch the ACIE Player
2. Create a project (a project is simply a collection of related files)
3. Add some files to the project
4. Customise the work environment
5. Play the video

6. Edit the transcript
7. Create, edit and remove an annotation
8. Close the application

During one of the tasks, the participant used the ACIE Player by itself, while during the other task they used the Player with the framework enabled. The presence of the framework was randomised during each task to minimise the existence of bias, and to prevent learning effects from affecting survey results. Participants were not told whether the framework was enabled during a particular task, and care was taken to ensure that there were no clues to reveal this information.

When enabled, the framework was configured to capture all available events in three different output formats: plain text, CSV and XML. This produced a large amount of event data and was therefore an accurate way of testing the user's ability to detect the framework.

Each user completed a short survey upon completing the first task of the experiment and another survey at the end of the experiment. Each survey presented the user with two questions:

> *Question 1* – were you aware of the presence of the framework?
> *Question 2* – did the framework interfere with your task?

It was assumed that the participants had basic computer skills and experience with Window-based applications, but they were not required to have any prior experience in using the ACIE Player or the framework. Each user was simply required to follow a set of instructions as best as they could, without any practice or training.

4.3 Investigation Experiment

In the investigation experiment, participants were randomly divided into three groups, where each group consisted of a user and an investigator. All participants had already completed two tasks with the ACIE Player in the usability experiment, so each person had the same amount of experience in using the Player and the framework.

The user of each group carried out the task from the first experiment, while the investigator observed the user and conducted a simple usability evaluation. The investigator performed this evaluation by following a set of instructions and completing an objective sheet – counting the occurrence of specific events and comparing this to other events. For example, one of the objectives asked the investigator to identify whether *tile window* events or *cascade window* events occurred more often.

The overall flow of activities during the investigation experiment was as follows:

1. Investigator receives an instruction sheet and an objective sheet
2. Investigator launches the usability evaluation framework
3. Investigator configures the framework to capture the events described on the objective sheet
4. User launches the ACIE Player and performs the activities from the usability experiment
5. User closes the ACIE Player
6. Investigator examines the event logs that were generated by the framework
7. Investigator completes the objective sheet

Upon completion of the experiment, the investigator completed a survey that was answered by circling a number on a 7-point Likert scale. The number 1 represented a very negative answer, while 7 represented a very positive answer. For instance, the first question was “how would you rate your computer skill level?” The number 1 matched to the verbal answer of “beginner”, while 7 matched to “expert”.

Once again, participants were not required to have any prior experience in using the ACIE Player or framework.

5 Experimental Results

The same group of six participants was involved in both experiments of the study. All participants were members of the InSyL group.

5.1 Usability Experiment

All six participants took on the role of a user in the usability experiment. The primary source of data in this experiment was the two surveys that users were required to complete, in which they provided responses to the questions described in Section 4.2. Almost all users indicated that they did not notice the presence of the framework, and that it did not interfere with their task.

In one case, a user thought that they noticed the framework running during both of their tasks, but it was only running during one task. This suggests that what the user noticed was not actually the framework. Thus, the results indicate that AOP does not affect software usability and is a suitable technology for driving the framework.

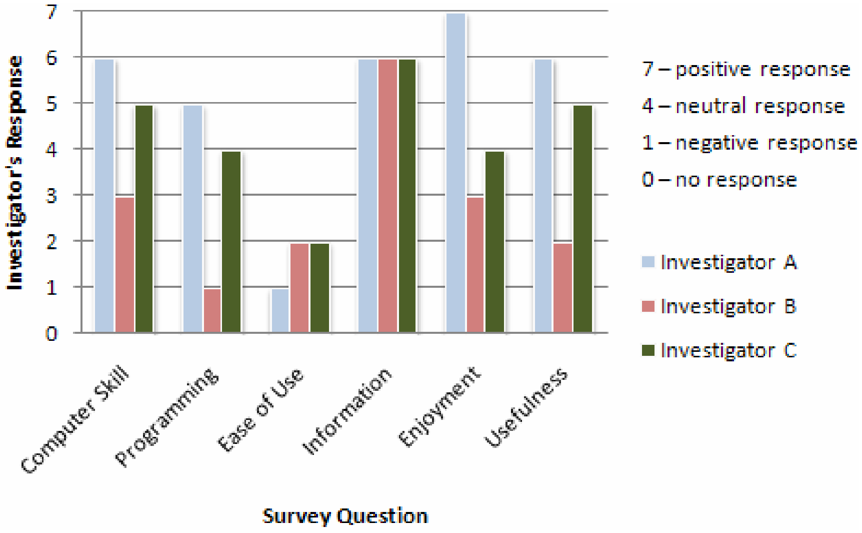


Fig. 6. Investigator survey results

5.2 Investigation Experiment

In the investigation experiment, the six participants were divided into three groups. In each group, one participant took on the role of a user and the other participant became the investigator. At the end of the experiment, the investigator filled in a survey by answering questions on a 7-point Likert scale.

The graph in Figure 6 shows the results obtained from the investigator survey. The survey questions have been abbreviated to improve the readability of the graph. The full questions are as follows:

- *Computer Skill* – how would you rate your computer skill level?
- *Programming* – how much experience have you had with computer programming?
- *Ease of Use* – how easy was the framework to use?
- *Information* – did the framework provide you with the information you needed to complete your objectives?
- *Enjoyment* – overall, did you enjoy using the framework?
- *Usefulness* – do you see the framework as something that you could use in your own work?

The graph shows that the investigators varied greatly in computer skill and programming ability, based on their self-assessment. However, all of the investigators found the framework difficult to use, even those with prior computer and programming experience.

Investigators described that the main difficulty was encountered while configuring the framework. The instruction sheet and the framework provided a large amount of information, which was very overwhelming for the investigators. One user explained that the framework GUI was "cramped" with information and that they were confused about what they needed to do to meet the requirements of their task.

Nonetheless, after thoroughly explaining the task to the investigator and describing how the framework should be configured, all investigators were able to complete the task successfully, even those with limited computing / programming experience.

Although configuring the framework generally proved to be a difficult task, the final task of interpreting the event data and completing the objective sheet was considered much easier. All investigators were able to fill in the objective sheet without encountering any major problems, and all of their answers proved to be 100% correct, when compared to the generated event data.

6 Future Work

The framework does not log every single event generated by the ACIE Player. Currently, only the user-initiated events are captured, but internal application events are ignored. The framework could be extended to capture all events generated by an application, which may prove to be useful. For example, the logging of internal events could provide the programmer with a trace of program execution to assist in debugging.

Some members of InSyL are not only interested in observing the participants of recorded meetings, but also in observing those who are observing the meeting. Since AOP has proven to be particularly useful in capturing a user's actions as they interact with an application, it may also be possible to capture and analyse the actions of an observer.

One of the current limitations of the framework is that a programmer needs to create an aspect interface for each unique application. This involves examining the application source code, discovering the join points and adding these join points to the aspect interface. The framework could be extended to automatically discover the join points and generate the aspect interface, potentially saving time for the programmer.

The case study showed that investigators found the framework difficult to use. One reason for the difficulties was the large amount of information presented in the framework frontend. Therefore, further analysis needs to be done in improving the GUI or creating a completely new one. For instance, the framework could run the ACIE Player in a *framework configuration mode*, where the investigator clicks on various GUI components in the Player and the framework then enables event logging of those particular components. This approach may be more intuitive, because the investigator would be clicking on the actual components and performing the actual operations that they want to log, as opposed to selecting checkboxes with the names of those operations.

Although the framework has been evaluated using a case study, the study only examined a single scenario of usage and the framework has only been tested with a single application (the ACIE Player). Since the framework is designed to be reusable, it needs to be tested with other applications to evaluate its effectiveness in different environments.

7 Conclusions

Aspect-oriented programming allows usability evaluation concerns, such as event logging, to be separated from the rest of the system. Researchers have already begun to explore this area, and in particular, Tarta and Moldovan [9] suggested the development of an AOP-based usability evaluation framework. Implementing this framework has been the primary accomplishment of this research project.

The framework was developed as an extension of a software application called the ACIE Player. The use of AOP enables the framework to be dynamically configured to capture a specific subset of all of the mapped events within the Player. The configuration is provided through a frontend, making the framework accessible to both programmers and non-programmers.

A case study, consisting of a usability and an investigation experiment, was used to evaluate the implementation. The results showed that the framework could assist in performing usability evaluation. Furthermore, the data from the usability experiment revealed that AOP did not create any performance constraints on the working environment, suggesting that AOP is a suitable technology for driving the framework.

Acknowledgements. This research project was in part funded through the UniSA/DSTO Research Agreement "Human Interaction Studies on the use of Speech Technology and Speaker Localisation in support of LiveSpaces: 2006/1170285: ACIE Player v2.0 Development Project – Honours Scholarship 2008". We also thank Ahmad Hashemi-Sakhtsari and Michael Coleman for their assistance.

References

1. Hilbert, D.M., Redmiles, D.F.: Extracting Usability Information from User Interface Events. ACM Computing Surveys 32(4), 384–421 (2000)
2. Ivory, M.Y., Hearst, M.A.: The State of the Art in Automating Usability Evaluation of User Interfaces. ACM Computing Surveys 33(4), 470–516 (2001)
3. Tao, Y.: Capturing User Interface Events with Aspects. In: Jacko, J.A. (ed.) HCI 2007. LNCS, vol. 4553, pp. 1170–1179. Springer, Heidelberg (2007)
4. Low, T.: Designing, Modelling and Implementing a Toolkit for Aspect-oriented Tracing (TAST). In: AOSD 2002 Workshop on Aspect-Oriented Modeling with UML (2002)
5. Avgustinov, P., Bodden, E., Hajiyev, E., Hendren, L., Lhotak, O., Moor, O., Ongkingco, N., Sereni, D., Sittampalam, G., Tibble, J., Verbaere, M.: Aspects for Trace Monitoring. In: Formal Approaches to Testing and Runtime Verification, pp. 20–39. Springer, Heidelberg (2006)
6. The AspectJ Project, http://www.eclipse.org/aspectj/
7. Hartman, G.S., Bass, L.: Logging Events Crossing Architectural Boundaries. In: Costabile, M.F., Paternó, F. (eds.) INTERACT 2005. LNCS, vol. 3585, pp. 823–834. Springer, Heidelberg (2005)
8. Tao, Y.: Toward Computer-Aided Usability Evaluation for Evolving Interactive Software. In: ECOOP 2007 Workshop on Reflection, AOP and Meta-Data for Software Evolution. University of Magdeburg (2007)
9. Tarta, A.M., Moldovan, G.S.: Automatic Usability Evaluation Using AOP. In: 2006 IEEE International Conference on Automation, Quality and Testing, Robotics, pp. 84–89. IEEE Computer Society, Los Alamitos (2006)
10. Tarby, J., Ezzedine, H., Rouillard, J., Tran, C.D., Laporte, P., Kolski, C.: Traces Using Aspect Oriented Programming and Interactive Agent-Based Architecture for Early Usability Evaluation: Basic Principles and Comparison. In: Jacko, J.A. (ed.) HCI 2007. LNCS, vol. 4550, pp. 632–641. Springer, Heidelberg (2007)
11. Information Systems Laboratory, http://www.insyl.unisa.edu.au/

Implementing Domain Specific Process Modelling

Bernhard Volz and Sebastian Dornstauder

Chair for Applied Computer Science IV, University of Bayreuth, Bayreuth, Germany
{bernhard.volz,sebastian.dornstauder}@uni-bayreuth.de

Abstract. Business process modelling becomes more productive when modellers can use process modelling languages which optimally fit to the application domain. This requires the proliferation and management of domain specific modelling languages and modelling tools. In this paper we address the issue of providing domain specific languages in a systematic and structural way without having to implement modelling tools for each domain specific language separately. Our approach is based on a two dimensional meta modelling stack.

1 Introduction

"The only constant is change" is a quotation that is often used to characterize process management. And indeed, changes can occur on all levels of process management from running process instances over process models to modelling languages. Starting at the "lowest" level, running process instances might have to be changed to react to a sudden shift in the application. Among others, [28], [4] and [26] are investigating this issue and suggest adequate solutions. Stepping one level up, the process model (definition) might have to be changed since it has become obvious that from now on a certain application will be performed in a different way [28] [10]. Nevertheless, it is even possible to step up another level in the hierarchy. Change on this level means to alter the modelling language, which is the focus of our paper. For process aware information systems this kind of change means an evolution of the whole system over time.

Why is the change of a process modelling language an issue that is worth investigating? One can argue that a process modelling language should always remain untouched. However, we fully comply with the interpretation of change as being related to diversity [5]. Although that book discusses change in the context of programming languages, we can transfer the results to process management. The authors of [5] notice that different domains will be characterized by different customer requirements. This observation can seamlessly be adopted in the business process management domain. Here, the programming language is the modelling language and the deployment platform corresponds to the process execution infrastructure.

We fully subscribe to the argument of [5] that the right languages enable developers to be significantly more productive. Besides we agree with the requirement that "we need the ability to rapidly design and integrate semantically rich languages in a unified way". This means on the one hand that each domain may and finally has to create its individual, specific language (domain specific language, DSL). On the other hand it means that a common basis for these languages facilitates their developments. It is important to sustain – despite the diversity of DSLs – a kind of comparability and

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 120–132, 2010.

compatibility between them. We finally agree that meta modelling provides capabilities to achieve this.

Changes of a modelling language need not to be huge. For example, in [20] process steps are tagged to indicate whether they are prohibitive or mandatory. Although being unspectacular, this tagging is very valuable for the execution and evaluation of a process model. Standard process modelling languages like BPMN [22] do not offer this special kind of tagging a priori.

At this point we also have to discuss whether changing a process modelling language is counterproductive since it diminishes the possibility to exchange process models with partners. Here, we assume that each development of a DSL takes a standard language (e.g. BPMN) as a starting point. The following two arguments support the idea of domain specific process modelling and – therefore – the adaptation of a standard modelling language:

First, domain specific adaptations are decisively enhancing the applicability of a process model within that domain. Adaptations are almost exclusively of interest within a domain. Thus, it is favourable to support adaptations.

Second, the use of a standard modelling language especially pays off when process models have to be exchanged with partners. Using a meta modelling approach, it is easy to distinguish between modelling elements of the standard language and those of a domain specific adaptation. Thus domain specific adaptations can be filtered out before a process model is exchanged. Although filtered process models lose information they are relevant and readable for receiving partners since the latter merely contains standard modelling elements. Assuming that domain specific extensions are primarily of interest for the domain developing them, this loss of information is tolerable.

Building up on these assumptions we present a meta modelling approach which supports the definition of domain specific process modelling languages. The special feature of our approach is that DSLs are derived from a common basic language; this language will be most probable a sort of standard language. All language definitions will be based on a meta model. This strategy bears major advantages.

- All derived DSL share a common set of modelling constructs. Thus, they remain compatible and comparable to a certain extend.
- The definition of a DSL is performed in a systematic way by extending the meta model of such a language.
- Extensions made for one DSL could be inherited by other domains, i.e. DSLs, if it is considered to be valuable for the new domain as well. This feature supports reuse of modelling constructs greatly.
- Tools can be built that support different DSLs at the same time. It is not necessary to build a special tool for each DSL.

So, we deliberate on the benefit of a standard notation and of a customized notation. We definitely favour the latter one – as argued in [5] – since productivity is supported decisively better. Nevertheless, data exchange is still feasible.

The focus of this paper lies on tool support for domain specific modelling languages. The foundations of a domain specific processes modelling tool are discussed in Section 2. Section 3 illustrates its basic part, a meta model stack. Several use cases of change are analyzed in Section 4; Section 5 presents the foundation of a repository which was implemented in order to get a working system and Section 6 finally discusses related work.

2 Foundations

The foundation of Perspective Oriented Process Modelling (POPM) were already presented about 15 years ago in [13] and [14]; runtime and visualization aspects of POPM are discussed in [15] and [16], respectively. Since POPM combines a couple of matured modelling concepts in a synergetic manner, these modelling concepts will be introduced in the following.

2.1 Layered Meta Modelling

In literature, the term “Meta Model” is often defined as a model of models – e.g. [27]. Thus a meta model defines the structure of models and can be seen as language for defining models. We also use a model to define the structure (syntax) of our process modelling languages. According to the Meta Object Facility (MOF) [23] this model then becomes part of a meta model stack which consists of linearly ordered layers. Since MOF restricts modellers (e.g. it allows only instanceOf relationships between layers), our solution is only inspired by them.

In Fig. 1, the actual process models are defined on modelling layer M1 (right boxes). A process model uses process (and data, organization etc.) definitions which are collected in the “Type library” on M1 (left box). All process types are defined first and then "used" in process models (e.g. as sub-processes) to define the latter. M0 contains running instances of processes defined on M1 (right boxes).

All process definitions on M1 are defined in a DSL previously specified at M2. M2 further contains the definition of an abstract process meta model (APMM) defining a set of general language features such as Processes, Data Flow or Control Flow. Each DSL is a specialization of elements contained in the APMM. M2 is therefore the layer where a modelling language like BPMN (left boxes) and its derivations (cf. Section 1, right boxes) are defined. It is noteworthy to mention that the elements of M2 reference those on M3 (MOF only allows “instanceOf” relationships).

An abstract process meta meta model (APM^{2}M) at M3 defines basic modelling principles; for instance, it defines that processes are modelled as directed graphs that also support nesting of nodes; this defines the fundamental structure for process modelling languages. Following the architecture of Fig. 1 (logical stack) allows for exchanging the modelling paradigm (graph based process models) at M3, defining DSLs at M2 as specializations of a general modelling language (APMM) and establishing type libraries at M1 which allow the re-use of existing process types in different contexts. One of the most powerful features of our approach is that most of the above mentioned kinds of changes can be applied by users and do not require a re-implementation of the modelling tool.

Without going into details we want to introduce one more feature which is most relevant for multi layer modelling. We borrow the Deep Instantiation pattern from [2] that allows defining on which level of a modelling hierarchy a type or an attribute of a type must be instantiated. Thus it is possible to introduce runtime instance identification on M3 that enforces that all derived types must carry this identification. However, this identification is not instantiated before M0.

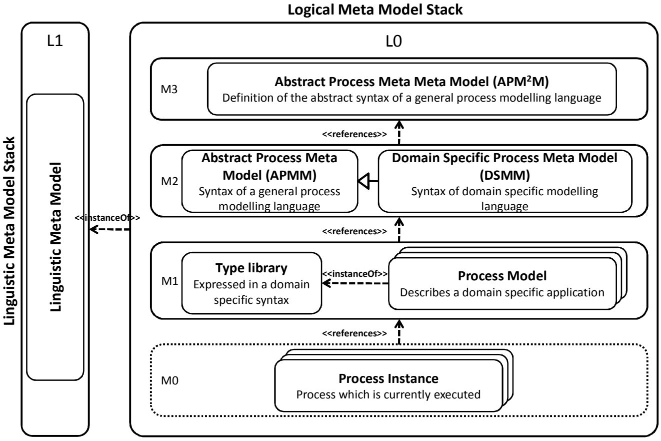

Logical Meta Model Stack
L1
L0
Linguistic Meta Model Stack
Linguistic Meta Model
M3
Abstract Process Meta Meta Model (APM²M)
Definition of the abstract syntax of a general process modelling language
<<references>>
M2
Abstract Process Meta Model (APMM)
Syntax of a general process modelling language
Domain Specific Process Meta Model (DSMM)
Syntax of domain specific modelling language
<<instanceOf>>
<<references>>
M1
Type library
Expressed in a domain specific syntax
<<instanceOf>>
Process Model
Describes a domain specific application
<<references>>
M0
Process Instance
Process which is currently executed

Fig. 1. Meta layer stack of POPM

2.2 Extended Powertypes

As mentioned, the APM^{2}M on M3 defines process models to be interpreted as graphs; for a tool it is then often necessary to interpret the capabilities (features) of each element. For example, both "Process" and "Start-Interface" are nodes of a process model graph. As in a graph usually each node can be connected with others, also the Start-Interface could have incoming arcs; a fact that needs to be prohibited. Therefore already at modelling layer M3 a capability *canHaveIncomingControlFlows* can be defined that describes whether a node accepts incoming flows or not.

Traditional approaches for implementing these capabilities are class hierarchies or constraint languages such as the Object Constraint Language (OCL) [24]. Both approaches are not very useful since either the complexity of the required type hierarchy explodes with an increasing number of capabilities or the user, who should be the one to extend the language, must be familiar with an additional language. Therefore we have chosen to extend the Powertype modelling pattern [25]. In our extension the capabilities (e.g. to have incoming flows) are defined as attributes of the powertype. These values then specify which capabilities of the partitioned type should be activated. Furthermore, only those attributes of a partitioned type are inherited by new constructs whose capability attribute has been set to “true”. Thus our extension removes features physically from a new construct.

2.3 Logical and Linguistic Modelling

In [1] an orthogonal classification approach is introduced. It contains two stacks that are orthogonal to each other (cf. Fig. 1, Linguistic Meta Model Stack). The Linguistic Meta Model Stack contains a meta model describing how models (including meta models) of the application domain are stored. An orthogonal Logical Meta Model Stack hosts one or more models which are purely content related.

It is crucial for this architecture that each layer of the logical stack can be expressed in the same linguistic model. As a result a modelling tool can be built that allows users to modify all layers of the logical stack in the same way. Conventional modelling tools do not support an explicit linguistic model and thus can usually modify only one layer of a logical model hierarchy [1]. Therefore a profound linguistic meta model is a good basis for creating a modelling tool that allows users to modify arbitrary layers and models.

Due to our extension of the powertype construct, the problem oriented conception of the meta model stack of the logical model, and the application of the orthogonal classification approach, a powerful foundation for an infrastructure for domain specific modelling tools is created. The following sections detail this infrastructure with respect to the most important part – the logical meta model stack.

3 Content of the Logical Meta Model Stack

Our goal is to implement a tool for the POPM framework that is capable of handling changes on the various levels of our meta modelling hierarchy. In this section we will introduce the logical models our actual implementation is based on.

3.1 Abstract Process Meta Meta Model (APM^{2}M)

As explained, the APM^{2}M located at M3 provides basic structures for process modelling languages defined on layer M2, i.e. it prescribes the structure of the modelling elements a process modelling language can offer. The most common graphical notation for process models in POPM is based on directed graphs whose meta meta model is depicted in Fig. 2 (standard UML notation). It is important to differentiate between modelling and visualization in this context. In Fig. 2 only the (content related) structure of a process modelling language – and respectively the process models derived from it – is defined. How these models are visualized is not part of this model; visualization is defined in an independent – but certainly related and integrated – model that is published in parts in [16].

Nodes of a process graph are represented by Node in the APM^{2}M (Fig. 2). NodeKind then describes the characteristics (features) of nodes in the graph where each feature corresponds to an attribute of NodeKind. The Powertype pattern between Node and NodeKind is established through the “partitions” relationship; Node represents the partitioned type and NodeKind is the powertype of the Powertype pattern.

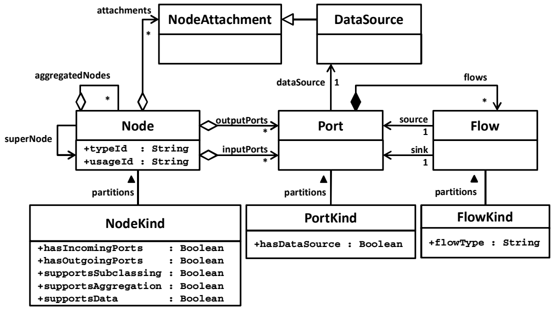


Fig. 2. APM^{2}M of POPM

Processes are just one type of nodes in such a graph; another type of nodes is e.g. Start-Interface. The different behaviours and capabilities of these two types are determined by the attributes within NodeKind. Features defined and implemented by the partitioned type Node are:

- HasIncomingPorts determines whether a modelling construct can be a destination of incoming flows. It is deactivated for constructs defining the start of a process (Start-Interface).
- HasOutgoingPorts defines if a modelling construct can be the origin of flows. For example a “Stop” interface cannot have outgoing connections.
- SupportsData specifies whether a construct accepts inbound and outbound data flows. If this feature is set to “false” but any of the has…Ports feature attributes has been set to “true”, this defines connectivity through control flow(s) only.
- SupportsSubclassing determines if a construct can have another construct as “super type”. The child construct will then inherit all attributes from the parent.
- SupportsAggregation defines whether a construct can contain usages of other elements. Typically this feature is activated for process steps but not for interfaces. Thus if activated, hierarchies of modelling elements can be built.

In summary, the features presented above determine whether elements of Node can establish relationships of a certain kind (e.g. superNode, aggregatedNodes, inputPorts) to other types of the APM^{2}M. The extended Powertype concept is also used for the type PortKind – here it determines whether a port can be bound to data sources; FlowKind is using the normal Powertype semantics.

3.2 Abstract Process Meta Model (APMM)

Fig. 3 shows the APMM of POPM, which defines the fundamental components of a POPM-related process model: process, connector, data container, control and data flow, organization, etc.

In the APMM a process is an element in a graph that can be interconnected with other nodes (hasIncoming/OutgoingPorts = true), can receive and produce data (supportsData = true), can be defined in terms of an already existing process (supportsSubclassing = true) and can be used as a container for other elements (supportsAggregation = true). A process – and in general every element on layer M2 – is an instance of a corresponding type (sometimes a powertype) on M3. For instance, Process is an instance of

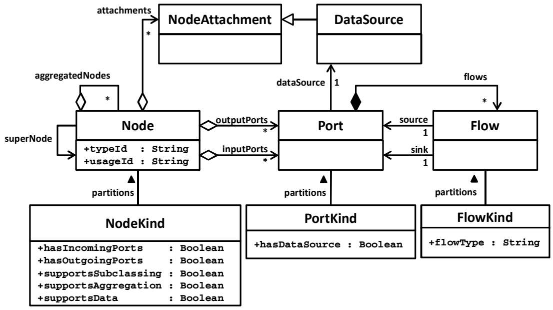


Fig. 3. Abstract Process Meta Model of POPM

the powertype NodeKind and inherits all activated features from the partitioned type Node. The type StartInterface is also an instance of the powertype NodeKind but does neither support the creation of hierarchies (supportsAggregation = false) nor incoming connections (hasIncomingPorts = false).

3.3 Domain Specific Meta Models (DSMMs)

According to Fig. 1, DSMMs are specializations of the APMM. As with object oriented programming languages, abstract types cannot be instantiated. Thus, a DSMM must first provide specializations for each element of the APMM (abstract model) which can be instantiated. Then it can be enriched by additional modelling constructs which determine its specific characteristics. We will show a simple example DSMM from the medical domain in the following.

We decided to provide for each modelling element of the APMM at least one modelling element in the DSMM for the medical domain. These domain specific modelling elements can furthermore be modified in order to capture specific characteristics of the medical realm. For instance the attribute stepType for the modelling element Medical Process (specialization of the APMM element Process) is introduced to determine whether a given step is an administrational or a medical task. Also tags as requested by [20] can be implemented in this way. Completely new modelling constructs can be introduced as well, like the so-called MedicalDecisionElement. In Section 4 we detail this feature.

At level M1 the "normal" modelling of processes takes place. Real (medical) processes use the types defined in the DSMM on M2; for example each process uses MedicalProcess as basis. Accordingly, input and output data for each process can be defined; the same applies to organizations and operations. In Fig. 4c an example is shown. Note that all modelling elements must be defined before being used. For instance, the process Anamnesis must be modelled (and put into the type library) before it can be used as sub-processes within HipTEP.

3.4 Modelling Processes on Level M1

In Fig. 4c, a part of a real-world process HipTEP [8] which describes a hip surgery is depicted. It consists of a start interface and two process steps namely Anamnesis and Surgery. The start interface is connected with the Anamnesis step via a control flow whereas Anamnesis and Surgery are also connected with data flows indicating the transport of data items between them. The symbols (document, red cross) inside the two steps are tags that indicate whether a step is more of medical or administrational interest (this is valuable information when the process model has to be analyzed). The tags correspond to the attribute stepType defined in the Medical DSMM for MedicalProcess.

3.5 Stepwise Design of a Process Model

In Fig. 4 the three decisive layers of a flexible modelling tool are clearly arranged. The figure illustrates how concepts evolve from very abstract (APM2M), to more concrete (APMM), to domain specific (DSMM). Some of the metamorphoses of modelling elements are explained in detail.

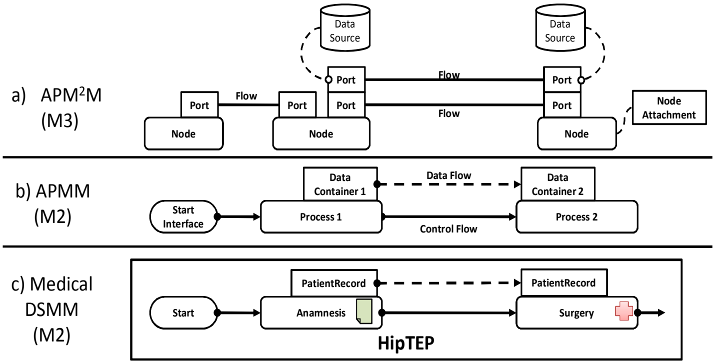


Fig. 4. Stepwise design of a process models on M1

M3 defines that nodes exist which carry ports (Fig. 4a). Ports are sometimes connected with data sources and can be interconnected by Flows. In the derived APMM (Fig. 4b) this definition is refined. Nodes are divided into two kinds: StartInterfaces and Processes. Ports which are not connected to data containers have evolved into gluing points for control flows between nodes (StartInterface and Process). Ports connected to data sources demarcate output from input data container for processes which are connected by data flows. Fig. 4c then depicts a concrete example written in the language predetermined by the APMM of Fig. 4b. A part of a medical process (HipTEP) is shown which consists of the processes Anamnesis and Surgery. One data item is passed between these processes, namely PatientRecord.

Fig. 4 demonstrates the power of this approach since each artefact of a process model is explicitly defined on clearly separated meta levels.

4 Dealing with Change

We will now explain concrete use cases of changes. These scenarios are ordered according to their relevance in practice based on our experience. We also depict how users can use them in a safe and structured way.

4.1 Change I: New Feature for an Existing Construct (Tagging)

Often it is necessary to distinguish processes from each other. Frequently, special tags are attached to processes and visualized in a suitable form [8] [20]. Speaking in terms of our logical meta model stack this means that an attribute is added to the corresponding modelling element in the DSMM that holds the tag. In Section 3 we have already shown this extension by adding the stepType attribute to the MedicalProcess type. Depending on the actual value of this attribute a visualization algorithm can then e.g. display icons appropriately.

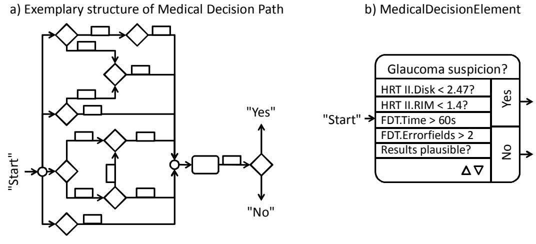
4.2 Change II: Introducing New Constructs

One reason for adapting modelling constructs is the evolution of the application domain. For example, due to more insight into the domain more powerful and semantically richer modelling constructs have to be created.

A new construct can either be defined “from scratch” or by redefining an already existing constructs of the DSMM or APMM. Fig. 5 gives an example for this kind of change in the medical domain. Fig. 5a outlines the complex structure of a medical decision path whereas Fig. 5b depicts a newly created modelling construct MedicalDecisionElement which is a macro comprising the functionality of the complex process structure of Fig. 5a. The problem with the process in Fig. 5a is that it is not comprehensible easily (only the complex structure of the decision path is of interest; therefore we did not show any details in Fig. 5a). Thus we decided to introduce a new compact modelling construct MedicalDecisionElement (Fig. 5b). This construct comprises the same functionality but is much easier to interpret. First, the construct has a title clearly showing its purpose. Then the most interesting decisions are shown in the list below the title and the two possible outcomes – yes or no – are depicted on the right side. The introduction of this compact construct – together with the consequent elimination of unreadable process models – was one of the major factors why process modelling was accepted as adequate means to illustrate the medical applications in the Ophthalmological Clinics of the University of Erlangen [17]. This project convincingly demonstrated that a domain specific modelling language is not just "nice-to-have" but is crucial for the acceptance of process management in general.

4.3 Change III: Enhancing / Changing the Modelling Method

So far all changes of process modelling languages were applied to DSLs individually. In our approach it is also possible to change the modelling method as such. This change happens on layer M3 and affects all process modelling languages defined below. For instance, from now on we will prohibit control flows between nodes. Referring to the APM2M in Fig. 4 this means to remove ports which are not connected with data sources. Consequently all flow derived from this constellation must be removed from all process modelling languages on M2 and also from all defined process models on M1.


a) Exemplary structure of Medical Decision Path

b) MedicalDecisionElement

Fig. 5. The MedicalDecisionElement (b) subsumes many single decisions (a)

5 Language for Linguistic Meta Modelling

As already mentioned earlier, linguistic meta modelling according to [1] is one of the key-points in our solution for domain specific modelling. Therefore, we shortly want to introduce our linguistic meta model and explain how models can be represented.

Fig. 6 shows the hierarchy of all elements of the linguistic meta model (LMM) which is inspired by commonly known models such as MOF [23] but contains elements which are beyond the scope of them such as the possibility to fully represent Powertypes and Deep Instantiation. All elements are derived from the basic type MElement that defines fundamental features such as the existence of a name and an ID for each element. Elements which are directly derived from MElement are MModel for representing models and MLevel for representing levels. Each model in the LMM consists of at least one meta level; several levels can be interconnected via relationships represented as instances of type MRelationship. Since arbitrary relationships can be defined between levels, the LMM does not enforce a certain structure in the sequence or hierarchy of levels. Each level then can hold concepts (MConcept) and relationships (MRelationship) which can be organized in packages (MPackage). Therefore the contents of a level are derived from the type MPackableElement.

A concept in the LMM implements a clabject [3] (which is especially useful when implementing powertypes, also cf. [18]) and thus consists of an instance and a type facet (MInstanceFacet, MTypeFacet). The type facet of a concept holds attribute definitions (MAttributeDefinition) whereas the instance facet of a concept stores values for previously defined attributes (MAttributeValue). Relationships are again used to establish some kind of link in between concepts.

Powertypes and Deep Instantiation are implemented as extensions (PTFeatureDefinition, DeepInstantiation) that can be assigned to each element within the LMM. Nevertheless, these two extensions can only be added to concepts and not to a model, a level or a package.

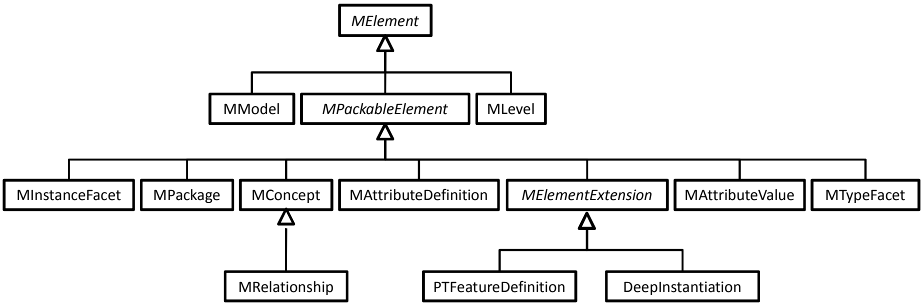


Fig. 6. Hierarchy of the elements of the linguistic meta model

The APM^{2}M of POPM (cf. Section 3.1) can be represented in terms of the LMM as follows (we are restricting ourselves to the concepts Node and NodeKind of the APM^{2}M and will also leave out packages):

```
model POPM {    // definition of a model
  level M3 { // definition of a level
    concept Node { // definition of the concept Node
      definitions: // attribute definitions
        outPorts  : Port[];
        superNode : Node;

    }
    // NodeKind is the powertype, Node is the partitioned type
    concept NodeKind partitions Node {
      definitions: // 'enables' is linked to the ext. powertypes
        hasOutgoingPorts   : Boolean(false) enables Node.outPorts;
        supportsSubclassing: Boolean(false) enables Node.superNode;
    }
}
```

Then, the concepts on M2 (APMM and Medical DSMM) can be represented as follows:

```
model POPM {
  level M3 { ... }
  level M2 references M3 {

    package APMM {

      concept Process instanceOf M3::NodeKind {
        values:
          hasOutgoingPorts    = true;
          supportsSubclassing = true;
      }
    }

    package MedicalDSMM {

      concept MedicalProcess extends APMM::Process { ...
      }
    }
  }
}
```

The LMM then provides to kinds of extensibility – an internal extensibility and an external extensibility. 'Internal' in the sense that the LMM can be extended itself thus allows for more expressive language. Examples are the introduction of new relationship types and the introduction of a representation for logic that are not yet part of the LMM. The external extensibility refers to the possibility of extending and combining models expressed in the LMM. In this scenario, the LMM can be seen as a common meta model for representing arbitrary models which supports relating models of different kinds.

6 Related Work

We now give an overview on existing technologies and systems (beside those already introduced in Section 2) that aim at increasing the sustainability of information systems. We will show that these are – per se – not appropriate for domain experts because they require extensive programming skills or are not flexible enough.

Generative Programming [6] and Software Factories [9] are techniques for the reuse of code. Generative Programming aims at the generation of code out of a set of templates. Requiring programming skills to produce valid and correct results, Generative Programming is unusable for end-users or domain experts. Software Factories in contrast aim at reducing the cost factors (time, resources etc.) during application development. This again is not suitable for end-users or domain experts. Even more harmful is that both approaches are meant to be applied during the development phase of an application but not during runtime.

Beside programming techniques, we also investigated complete meta modelling systems e.g. the Microsoft Domain Specific Language Tools for Visual Studio [21], the Eclipse Modeling Framework (EMF) [7] (along with related technologies that support the generation of graphical editors) or MetaEdit+ [19]. Most of them use only two levels in which the type level defines the storage format for the user models. Beside this the modelling freedom is restricted by a fixed underlying meta model. Also many solutions are not able to use a new modelling language without generating a new modelling environment.

Summarizing, there are solutions that provide some means for building modelling tools. But either they require too much programming skills or they are not flexible enough.

Even though we used the ISO/IEC 24744:2007 standard [12] and its preceding work (e.g. [11]) as an inspiration of how powertypes can be used to express complex concepts and relations, the ISO/IEC 24744:2007 standard and the solution presented here differ significantly such that they cannot be compared. For instance, our solution does not restrict extensions to additions – also elements of the standard POPM language can be removed via the extended powertype pattern. Furthermore, the inclusion of Deep Instantiation provides means for defining more freely when an attribute must be instantiated.

7 Conclusions

In this paper we introduced our approach for a more sustainable process modelling environment that can be easily adapted by domain experts to their realm without programming. We have based our system on many established technologies for the development of flexible and adaptive systems. But only their combination in our comprehensive approach allows them to unfold their real power. We have then shown how different adaptation scenarios can be performed with the help of these concepts. Here the important key-point is that all common change requests can be performed without writing code; instead only a new configuration for the system has to be provided. This is easy to set up even though the domain expert who is pursuing these changes has not much knowledge about the system internals. Thus domain experts are empowered to adapt the whole system perpetually to changing requirements which we believe is a fundamental step towards more sustainability.

References

1. Atkinson, C., Kühne, T.: Concepts for Comparing Modeling Tool Architectures. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 398–413. Springer, Heidelberg (2005)
2. Atkinson, C., Kühne, T.: The Essence of Multilevel Metamodeling. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. Springer, Toronto (2001)
3. Atkinson, C., Kühne, T.: Meta-level Independent Modelling. In: International Workshop Model Engineering 2000, Cannes, France (2000)
4. Clarence, E., Karim, K., Grzegorz, R.: Dynamic change within workflow systems. In: Conference on Organizational Computing Systems. ACM, Milpitas (1995)
5. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling - A Foundation For Language Driven Development. CETEVA (2008)
6. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley Professional, Reading (2000)
7. Eclipse Foundation. Eclipse Modeling Framework Project, EMF (2008)

8. Faerber, M., Jablonski, S., Schneider, T.: A Comprehensive Modeling Language for Clinical Processes. In: Hein, A., Thoben, W., Appelrath, H.-J., Jensch, P. (eds.) European Conference on eHealth 2007, GI, Oldenburg, Germany. Lecture Notes in Informatics (LNI), pp. 77–88 (2007)
9. Greenfield, J., Short, K., Cook, S., Kent, S.: Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, Chichester (2004)
10. Heinl, P., Horn, S., Jablonski, S., Neeb, J., Stein, K., Teschke, M.: A comprehensive approach to flexibility in workflow management systems. SIGSOFT Softw. Eng. Notes 24(2), 79–88
11. Henderson-Sellers, B., Gonzalez-Perez, C.: A powertype-based metamodelling framework. Software and Systems Modeling 5(1), 72–90
12. ISO/IEC Software Engineering - Metamodel for Development Methodologies. International Organization for Standardization / International Electrotechnical Commission, ISO/IEC 24744 (2007)
13. Jablonski, S.: MOBILE: A Modular Workflow Model and Architecture. In: 4th International Working Conference on Dynamic Modelling and Information Systems, Noordwijkerhout, NL (1994)
14. Jablonski, S., Bussler, C.: Workflow Management: Modeling Concepts, Architecture and Implementation. International Thomson Computer Press (1996)
15. Jablonski, S., Faerber, M., Götz, M., Volz, B., Dornstauder, S., Müller, S.: Integrated Process Execution: A Generic Execution Infrastructure for Process Models. In: 4th International Conference on Business Process Management (BPM), Vienna, Austria (2006)
16. Jablonski, S., Götz, M.: Perspective Oriented Business Process Visualization. In: ter Hofstede, A.H.M., Benatallah, B., Paik, H.-Y. (eds.) BPM Workshops 2007. LNCS, vol. 4928, pp. 144–155. Springer, Heidelberg (2008)
17. Jablonski, S., Lay, R., Meiler, C., Müller, S., Hümmer, W.: Data logistics as a means of integration in healthcare applications. In: 2005 ACM symposium on Applied computing. ACM, Santa Fe (2005)
18. Jablonski, S., Volz, B., Dornstauder, S.: A Meta Modeling Framework for Domain Specific Process Management. In: 1st International Workshop on Semantics of Business Process Management (SemBPM) in conjunction with the 32nd Annual IEEE Int'l Computer Software and Applications Conference (COMPSAC), Turku, Finland (2008)
19. Kelly, S., Lyytinen, K., Rossi, M.: MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: 8th International Conference on Advanced Information System Engineering, pp. 1–21. Springer, Heraklion (1996)
20. Lu, R., Sadiq, S.: On the discovery of preferred work practice through business process variants. In: Parent, C., Schewe, K.D., Storey, V.C., Thalheim, B. (eds.) 26th International Conference on Conceptual Modeling, pp. 165–180. Springer, Auckland (2007)
21. Microsoft. Domain-Specific Language Tools (2008)
22. Object Management Group. BPMN 1.1 Specification (2008)
23. Object Management Group. MOF 2.0 Specification (2008)
24. Object Management Group. OCL 2.0 Specification (2008)
25. Odell, J.: Advanced Object-Oriented Analysis and Design Using UML. Cambridge University Press, New York (1998)
26. Rinderle, S., Reichert, M., Dadam, P.: Correctness criteria for dynamic changes in workflow systems–a survey. Data & Knowledge Engineering 50(1), 9–34
27. Seidewitz, E.: What Models Mean. IEEE Software 20, 26–32
28. van der Aalst, W.M.P., Jablonski, S.: Dealing with workflow change: identification of issues and solutions. International Journal of Computer Systems Science and Engineering 15(5), 267–276

Bin-Packing-Based Planning of Agile Releases

Ákos Szőke

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
http://home.mit.bme.hu/∼aszoke/

Abstract. Agile software development represents a major approach to software engineering. Agile processes offer numerous benefits to organizations including quicker return on investment, higher product quality, and better customer satisfaction. However, there is no sound methodological support of agile release planning – contrary to the traditional, plan-based approaches. To address this situation, we present i) a conceptual model of agile release planning, ii) a bin-packing-based optimization model and iii) a heuristic optimization algorithm as a solution. Four real life data sets of its application and evaluation are drawn from the lending sector. The experiment, which was supported by prototypes, demonstrates that this approach can provide more informed and established decisions and support easy optimized release plan productions. Finally, the paper analyzes benefits and issues from the use of this approach in system development projects.

Keywords: Agile conceptual model, Release planning, Bin-packing.

1 Introduction

Development governance covers the steering of software development projects. Traditional governance usually applies command-and-control approaches which explicitly direct development teams. Experiences with these approaches – such as Control Objectives for Information-Related Technology (CobiT) [1], and the Organizational Project Management Maturity Model (OPM) [2] – show that they are too heavy in practice for many organizations, although they provide a wealth of advice [3]. As a reaction to so-called *heavyweight* methodologies [4], many practitioners have adopted the ideas of agility [5]. Agile approaches are quickly becoming the norm, probably because recent surveys showed agile teams are more successful than traditional ones [6,7]. Several studies pointed out $\approx$ 60% increase in productivity, quality and improved stakeholder satisfaction [7,8], and 60% and 40% reduction in pre-, and post-release defect rates [10].

In recent years, several agile methods have emerged. The most popular methods are Extreme Programming (XP) [9](58%), Scrum [11](23%), and Feature Driven Development (FDD) [12](5%) [13]. Despite variety of methods all of them share the common principles and core values specified in the *Agile Manifesto* [5].

Release planning is an activity concerned with the implementation of the selected requirements in the next version of the software. Agile release planning is usually based on a prioritized list of requirements (typically User stories) and is made up of the following major steps: the team i) performs estimation on every requirement, ii) determines

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 133–146, 2010.

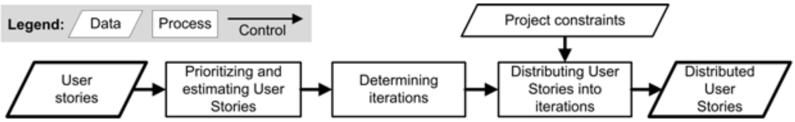


Fig. 1. Release planning in agile software development

the number and the length of iterations using historical iteration velocity (i.e. how much work can be done during each iteration), iii) distributes requirements into iterations considering constraints (Figure 1).

Problems. The essential aim of release planning is to determine an *optimal* execution plan of development respect to scarcity of available resources and dependencies between requirements. However, distributions of requirements are iteratively selected and assigned *manually* into iterations (see Figure 1). As a consequence, the following factors are managed implicitly: **P1**) *precedences* (temporal constraints between requirements), **P2**) *resource capacities* (resource demands during iterations), and **P3**) *priorities* (importance of each requirement delivery). Therefore, optimality of plans (i.e. maximal business value or minimal cost) is heavily based on the manager's right senses – nevertheless optimized project plans are crucial issues from the economic considerations of both customer and developer sides.

Objectives. Our proposed method intends to mitigate previous problems (**P1-3**) by i) constructing model of agile release planning which is intended to guide the design of release planning tools, ii) formulating release planning task as an *optimization model* that considers all the previous factors, and iii) providing a solution to this model by a *heuristic algorithm* to easily produce release plans.

Outline. The rest of the paper arranged as follows: Sec. 2 presents common notions of agile planning; Sec. 3 introduces our conceptual model, optimization model and algorithm; Sec. 4 describes experiences; Sec. 5 discusses our solution; Sec. 6 focuses on related work; Sec. 7 concludes the paper.

2 Agile Release Planning

In this section, we introduce agile release planning to provide the necessary background information for the proposed method.

2.1 Requirements Specification

Common to all software development processes in any projects is the need to capture and share knowledge about the requirements and design of the product, the development process, the business domain, and the project status. Contrary to the traditional methods (e.g. waterfall), agile methods advocate '*just enough*' documentations where the fundamental issue is communication, not documentation [4,11]. The agreed requirements not only drive the development, but direct planning of projects, provide basis for acceptance testing, risk management, trade-off analysis and change control [14]. In agile methods *User stories*, *Features*, and *Use cases* (see XP [9], FDD [12]) are the primary models of requirements and the source of effort estimation.

Estimating Effort. Agile teams focus on '*good enough*' estimates and try to optimize their estimating efforts. A good estimation approach takes short time, and can be used for planning and monitoring progress [15]. Effort estimation models usually based on size metrics: Story points [16], Feature points [12], and Use case points [17]. These metrics are abstract units, express the whole size of development and usually not directly convertible to person hours/days/months.

User Stories and Story Points. The most popular requirements modeling technique in agile methods is the User story technique [16]. User stories are usually materialized on electronic or paper Story cards with the description of i) the feature demanded by stakeholders, ii) the goal of the customers, and iii) the estimated size of the development work is expressed by a Story point and interpreted as person day effort (usually classified into Fibonacci-like effort sequence: 0.5, 1, 2, 3, 5, 8, 13) [15].

Dependencies. The complexity of scheduling arises from the interaction between requirements (User stories) by *implicit* and *explicit* dependencies. While the previous is given by the scarcity of resources, the latter one is emerged from different precedences between tasks' realizations [18,20]:

i) *functional implication* (j demands i to function),

ii) *cost-based dependency* (i influences the implementation cost of j, so useful to realize i earlier),

iii) *time-related dependency* (expresses technological and/or organizational demands).

3 Optimized Release Planning

In this section first, we formulate conceptual model of agile release planning, and we point out release planning can be characterized as a special bin-packing problem. Then we formulate a bin-packing-related optimization model for release planning, and present a solution to this model in the form of a heuristic algorithm.

3.1 Conceptual Model of Agile Release Planning

In order to formulate the release planning model, first, we have to identify the main concepts of agile release planning. These concepts are presented in the following list and visualized with UML notation in Fig. 2 [19,15,21].

- **Project:** is a planned endeavor, usually with specific *requirements* and rolled out in several deliverable stages i.e. *releases*.
- **Release:** produces (usually external) *selected* deliverable *requirements* for the customer, contains 1-4 iterations with start/end date and an iteration count.
- **Iteration:** is a development timebox that *delivers* intermediate deliverables. It is characterized by available resource capacity of the team – often expressed by iteration velocity.

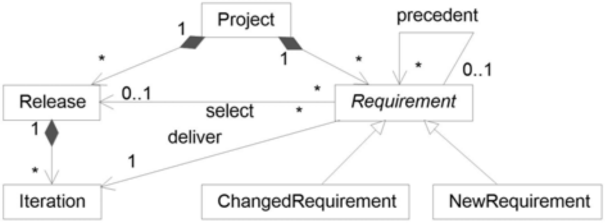


Fig. 2. Information Model of Agile Release Planning

- **Requirement:** deliverable that the customer values. They can be classified two kind of set of elements: new and changed *requirements* (functional and non-functional). In most cases requirements mandates several realization steps that requires cooperation of some developers. Requirement usually requires several working days realization effort that is estimated by developers or some method (see Sec. 2.1).
- **Precedence:** realization dependencies between *requirements*. Precedences emanate from the following sources (j', j denotes requirements) [18,20]:

  i) *functional implication* (j demands j' to function),
  ii) *cost-based dependency* (j' influences the implementation cost of j, so useful to realize j' earlier),
  iii) *time-related dependency* (expresses technological/organizational demands).

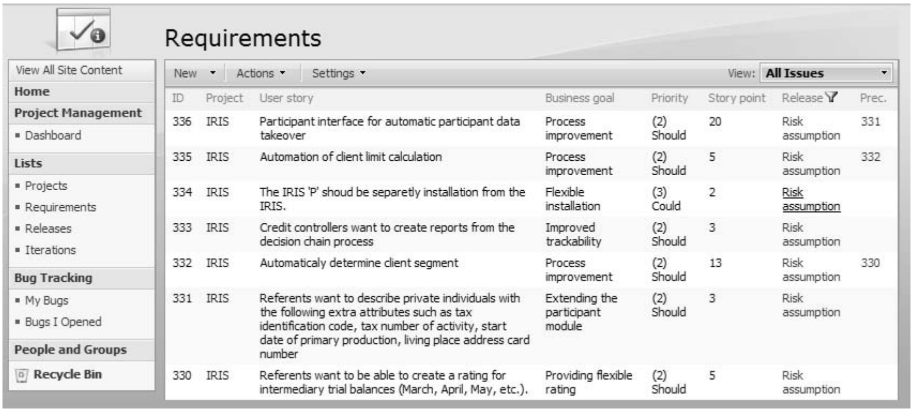
These concepts not only help to identify the objects and the subject of the optimization model but with the precise relationships it can also be used as database schema definition for agile release planning applications.

3.2 A Prototype for Collaborative Agile Release Planning Data Collection

Previously presented conceptual model is realized by our MS Sharepoint-based website (named *Serpa*) at Multilogic [22,23]. Sharepoint is browser-based collaboration and a document-management platform, and its capability includes creating different lists (as database tables). The previously constructed agile planning information model (see Fig. 2) were implemented as Sharepoint lists. Figure 3 shows the visual appearance of the prototype pointing out the Requirements list with User story, Priority, Story point and Precedence properties. Thus, the portal was targeted as a collaborative workspace for developers, and a tool for the management to collect all planning information. With this web-based tool, the team can list requirements, indicate precedences, set effort estimation and priorities, and they also can share these information to facilitate communication.

3.3 Mapping to Bin-Packing

Generally, a bin-packing problem instance is specified by a set of items and a standard bin size. The objective is to pack every item in a set of bins while minimizing the total number of bins used [24,25]. The analogy between release planning and bin-packing problem can be explained as follows.



ID	Project	User story	Business goal	Priority	Story point	Release	Prec.
336	IRIS	Participant interface for automatic participant data takeover	Process improvement	(2) Should	20	Risk assumption	331
335	IRIS	Automation of client limit calculation	Process improvement	(2) Should	5	Risk assumption	332
334	IRIS	The IRIS 'P' shoud be separetly installation from the IRIS.	Flexible installation	(3) Could	2	Risk assumption	
333	IRIS	Credit controllers want to create reports from the decision chain process	Improved trackability	(2) Should	3	Risk assumption	
332	IRIS	Automaticaly determine client segment	Process improvement	(2) Should	13	Risk assumption	330
331	IRIS	Referents want to describe private individuals with the following extra attributes such as tax identification code, tax number of activity, start date of primary production, living place address card number	Extending the participant module	(2) Should	3	Risk assumption	
330	IRIS	Referents want to be able to create a rating for intermediary trial balances (March, April, May, etc.).	Providing flexible rating	(2) Should	5	Risk assumption	

Fig. 3. Serpa: a prototype website for collaborating workspace of agile projects

The team's resource capacity in an iteration stands for a bin size, while a requirement's resource demand represents an item size. In the resource-constrained project scheduling problem (RCPSP) context, we can view each iteration within release as a bin into which we can pack different deliverable requirements. Without loss of generality, we can ensure that the resource demand of each requirement is less than team's resource capacity in an iteration. Then minimizing makespan (i.e. finding the minimum time to completion) of this RCPSP is equivalent to minimizing the number of bins used in the equivalent bin-packing problem [26].

We extend this ordinary bin-packing problem and interpretation with the following elements to provide further computational capabilities for wide-ranging release planning situations (c.f. **P1-3**): i) precedences between items (requirements) (c.f. Sec. 2.1), ii) varying capacities of bins (iterations), and iii) item priorities. From now on we call this extended problem as bin-packing-related RCPSP (BPR-RCPSP).

3.4 Formulating BPR-RCPSP Model

Henceforth, without loss of generality, we focus on the User story technique to be more concrete. Given a set of deliverable User stories j ($j \in A : |A| = n$) with *required efforts* w_j, and iterations n with different *capacities* c_i ($i \in \{1, 2, ..., n\}$) within a release. Let assign each User story into one iteration so that the total required effort in iteration i does not exceed c_i and the number of iteration used as a minimum while precedence relation (matrix) $P_{j,j'} \in \{0, 1\}$ (where $P_{j,j'} = 1$ if j precedes j', otherwise $P_{j,j'} = 0$ – c.f. Sec. 2.1) holds. A possible mathematical formulation is:

$$\text{Minimize } z = \sum_{i=1}^{n} y_i \tag{1a}$$

$$\text{subject to} \quad \sum_{j=1}^{n} w_j x_{i,j} \leq c_i y_i \tag{1b}$$

$$i' - i \geqslant P_{j,j'} \quad : x_{i',j'} = x_{i,j} = 1 \tag{1c}$$

$$\sum_{i=1}^{n} x_{i,j} = 1 \tag{1d}$$

where $y_i = 0$ or $1,$ and $x_{i,j} = 0$ or 1 $(i, j \in N)$, and

$$x_{i,j} = \begin{cases} 1 & \text{if } j \text{ is assigned to iteration } i \\ 0 & \text{otherwise} \end{cases} \tag{2a}$$

$$y_i = \begin{cases} 1 & \text{if iteration } i \text{ is used} \\ 0 & \text{otherwise} \end{cases} \tag{2b}$$

The equations denote minimization of iteration count of release (1a), resource constraints (1b), temporal constraints (1c), and an item j can be assigned to only one iteration (1d). We will suppose, as is usual, that the efforts w_j are positive integers. Without loss of generality, we will also assume that

$$c_i \text{ is a positive integer} \tag{3a}$$

$$w_j \leq c_i \text{ for } \forall i, j \in N \tag{3b}$$

If assumption (3a) is violated, c_i can be replaced by $\lfloor c_i \rfloor$. If an item violates assumption (3b), then the instance is treated as trivially infeasible. For the sake of simplicity we will also assume that, in any feasible solution, the lowest indexed iterations are used, i.e. $y_i \geq y_{i+1}$ for $i = 1, 2, ..., n-1$.

3.5 Solving the BPR-RCPS Problem

For the previously formulated optimization model we developed a *Binscheduling* algorithm (Algorithm 1). It is a constructive heuristic algorithm, which iteratively selects and schedules an item (*User story*) into an *iteration* – where it fits best. In the program listing lowercase and uppercase letters with indices denote vectors and matrices (e.g. c_i, $P_{j,j'}$). While bold-faced letters show concise (without indices) forms (e.g. **c**, **P**).

In the *require* section the preconditions are given. Each w_j is the weight (required effort) for User story j in Story point. Precedences between User stories can be represented by a precedence matrix where $P_{j,j'} = 1$ means that User story j precedes User story j', otherwise $P_{j,j'} = 0$. Both conditions $P_{j,j} = 0$ (no loop) and P is directed acyclic graph (DAG) ensures that temporal constraints are not trivially unsatisfiable. Priorities p_j express stakeholders' demands and are used by the scheduler algorithm as a rule when choosing between concurrent schedulable User stories. Capacities of iterations are calculated by taking the historical values of iteration velocities into consideration. The *ensure* section prescribes the postcondition on the return value (**X**): every User story j has to be assigned to exactly one iteration i.

During scheduling steps, first the initial values are set (line $1 - 5$). The iteration value (n) is equal to the number of User stories (line 1). The residual capacity denotes the remained capacity of an iteration after a User story is assigned – so it is initially set to capacity (line 3). The algorithm uses a *ready list* (**rlist**) and a *scheduled list* (**slist**) to keep track of schedulable and scheduled User stories. *Potentially* schedulable items (**pot**) are unscheduled items from which the algorithm can choose in the current control

Algorithm 1. *Binsched* algorithm with BF strategy

```
Require:
    $w_j \in N$                                   /* weights of each User story $j$ */
    $P_{j,j'} \in 0,1 \wedge P_{j,j} = 0 \wedge P$ is DAG              /* precedences */
    $p_j \in N, c_i \in N$      /* priority values and capacity of each iteration */
Ensure: $X_{i,j} \in 0,1 \wedge \forall j \exists! i \; X_{i,j} = 1$
 1: $n \Leftarrow length(\mathbf{w})$                    /* schedulable User stories */
 2: $\mathbf{X} \Leftarrow [0]_{n,n}$                /* assignment matrix initialization */
 3: $\mathbf{r} \Leftarrow \mathbf{c}$              /* residual capacities of each iteration */
 4: rlist $\Leftarrow \emptyset$                       /* 'ready list' initialization */
 5: slist $\Leftarrow \emptyset$                   /* 'scheduled list' initialization */
 6: for $fj = 0$ to $n$ do
 7:     pot $\Leftarrow findNotPrecedentedItems\,(\mathbf{P})$
 8:     rlist $\Leftarrow$ pot $\setminus$ slist                   /* construct ready list */
 9:     if rlist $==\emptyset$ then
10:         print 'Infeasible schedule!'
11:         return $\emptyset$
12:     end if
13:     $j \Leftarrow \min\{p_j\} : j \in$ rlist
14:     $i \Leftarrow selectBestFittingBin\,(w_j, \mathbf{r})$
15:     $X_{i,j} \Leftarrow 1$                  /* assign User story $j$ to iteration $i$ */
16:     $r_i \Leftarrow r_i - w_j$                       /* decrease residual capacity */
17:     slist $\Leftarrow$ slist $\cup \{j\}$
18:     $P_{\{1,\ldots,n\},j} = 0$                    /* delete scheduled User story */
19: end for
20: return $\mathbf{X}$
```

step without violating any precedence constraint (line 7). Previously assigned items are extracted from potentially schedulable items to form the ready list (line 8). As long as the ready list contains schedulable items, the algorithm chooses items from that list – otherwise the schedule is infeasible (line 9). The minimum priority item is selected from the ready-list to schedule (line 13). To find the proper iteration term for the selected item, the *best fit* (i.e. having the smallest residual capacity) strategy (line 14) is applied, and an item j is assigned to iteration i (i.e. $X_{i,j} = 1$). As a consequence residual capacity of iteration i is decreased by item weight w_j (line 16). Finally, scheduled list (**slist**), is updated with scheduled item (lines 17), and no longer valid precedence relations are also deleted from **P** after scheduling of the given item (lines 18). Iteration proceeds until all items are assigned to iterations (line 6-19).

After termination, $\mathbf{X}$ contains the User story assignments to iterations, where the number of nonzero columns denotes the packed iterations (i.e.

$z \Leftarrow length\left(nonZeros\left(\sum_{j=1}^{n} w_j x_{i,j}\right)\right)$ – c.f. (1a)).

There can be used several strategies (e.g. *FirstFit, BestFit*) to find the appropriate release plan, but we used only one (the *best fit*) for simple demonstration (line 14). This greedy strategy makes a series of local decisions, selecting at each point the best step without backtracking or lookahead. As a consequence, local decisions miss the global optimal solution, but produce quick (time complexity is clearly *O(nlogn)*) and usually sufficient results for practical applications [25].

Figure 4 illustrates the packing concept. This example shows the post-mortem release planning result of a real life development situation using the previous algorithm.

Figure 4 (left) demonstrates the histogram of schedulable User stories. The x-axis enumerates Story point categories (weights), while y-axis shows how many User stories fall into these categories. Figure 4 (right) depicts planning results produced by *Binsched* algorithm in stacked bar chart form: the schedulable User stories are packed into four iterations (x-axis) with capacities $30, 30, 29,$ and 28 (y-axis). Bar colors on the Figure 4 (left) point out how Story points are distributed on Figure 4 (right).

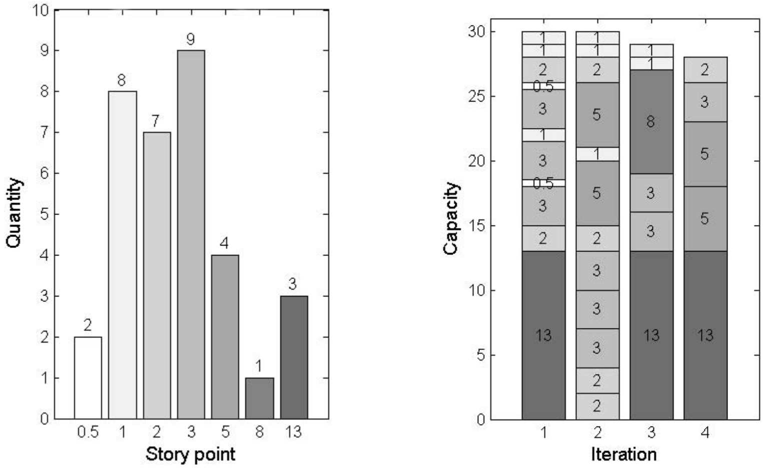


Fig. 4. Release plan applying the BPR-RCPSP approach: Schedulable User stories (i) (left) and User story assignments ($X_{i,j}$) (right)

4 Experimentation

To obtain a proof-of-concept we implemented the bin-packing-based algorithm in Matlab [27]. Four past release data sets – extracted from the backlog of IRIS application developed by Multilogic Ltd [23] – were compared against the results of simulations applying the same inputs [28].

4.1 Context and Methodology

IRIS is a client risk management system (approx. 2 million SLOC) for credit institutions for analyzing the non-payment risk of clients. It has been continual evolution since its first release in the middle of 90s. The system was written in Visual Basic and C# the applied methodology was a custom agile process. The release planning process were made up of the following steps. First, the project manager used intuitive rule for selecting User stories from the backlog into a release. Then the team estimated on every User story and determined the number and the length of iterations within the release

– based on iteration velocity. Finally, the team distributed User stories into iterations considering priorities and precedences.

4.2 Data Collection and Results

Four data sets (Collateral evaluation, Risk assumption, Ukrainian deal flow I/II – respectively $R_A - R_D$) were selected to make a comparison between the algorithmic method and the manual release planning carried out previously at Multilogic. The R_A data set is used to present the concept in the previous example (Figure 4). All the releases had same project members (6 developers), iteration length (2 weeks), iteration velocity (30 Story point), domain, customer, and development methodology, but they were characterized by different User story counts (USC), Iteration counts (IC), Buffer per releases (BpR) (for contingencies), and delivered Story point per iteration (SP_i). Table 1 summarizes the variables of $R_A - R_D$ collected with the previously introduced Serpa website – see Sec. 3.2.

To determine the usefulness of our proposed method, we used historical data as input of the Binsched algorithm (Algorithm 1). This method made it possible to compare performance of the algorithmic (optimized) approach against the manual one. Computed values ($R^*_A - R^*_D$) are shown in Table 2 (since USC, IC, BpR were the same as Table 1 they are not shown).

Table 1. Historical release plan values ($R_A - R_D$)

	USC	IC	BpR	SP_1	SP_2	SP_3	SP_4	SP_5	$\sum_1^5 SP_i$
R_A	33	4	3.0	28.0	35.0	24.0	30.0	0.0	117.0
R_B	25	3	4.5	33.0	34.5	18.0	0.0	0.0	85.5
R_C	27	5	12.5	31.5	33.0	23.0	26.0	24.0	137.5
R_D	27	4	3.5	29.5	33.0	27.0	27.0	0.0	116.5

4.3 Analysis

The analysis goal was to compare the manual and the optimized approaches using the same *input variables*. The following key questions were addressed: **Q1**: *What are the staffing requirements over time?*; **Q2**: *How many iterations do we need per release?*; and **Q3**: *How buffers for contingencies are allocated?*

To answer to these questions, 1) we carried out Exploratory Data Analysis (EDA) [29,30] to gaining insight into the data sets, then 2) we performed descriptive statistical analysis to compare the main properties of the two approaches.

Qualitative Analysis. The following EDA techniques (called 4P EDA) are simple, efficient, and powerful for the routine testing of underlying assumptions [29]:

1. *run sequence plot* (Y_i versus iteration i)
2. *lag plot* (Y_i versus $Y_i - 1$)
3. *histogram* (counts versus subgroups of Y)
4. *normal probability plot* (ordered Y versus theoretical ordered Y)

Table 2. Optimized release plan values ($R^*_A - R^*_D$)

	SP^*_1	SP^*_2	SP^*_3	SP^*_4	SP^*_5
R^*_A	30.0	30.0	29.0	28.0	0.0
R^*_B	30.0	28.5	27.0	0.0	0.0
R^*_C	29.5	30.0	30.0	29.0	19.0
R^*_D	29.5	30.0	30.0	27.0	0.0

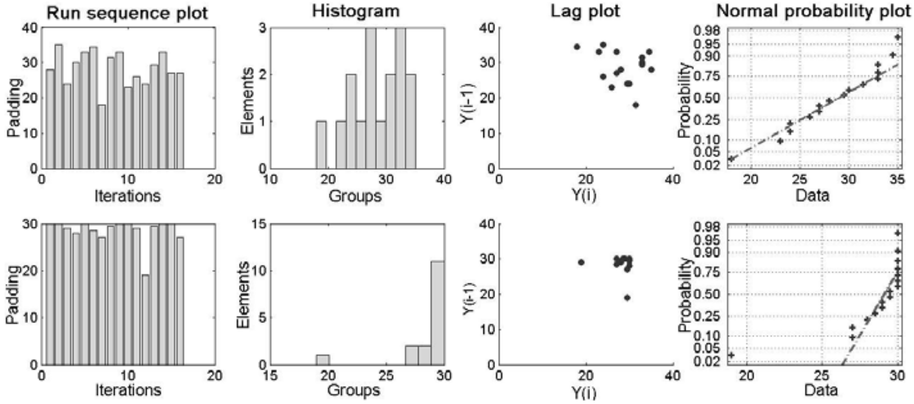


Fig. 5. 4P of historical (upper) and optimized (lower) plans

where $Y_i \triangleq \sum_{j=1}^{n} w_j x_{i,j}$ (i.e. sum of assigned Story point of each iteration (c.f. 1b) were identified as *result variables* to test or questions (**Q1-3**).

The four EDA plots are juxtaposed for a quick look at the characteristics of the data (Figure 5). The assumptions are addressed by the graphics:

A1: The run sequence plots indicate that the data do not have any significant shifts in location but have significant differences in variation over time.

A2: The upper histogram depicts that the data are skewed to the left, there do not appear to be significant outliers in the tails, and it is reasonable to assume the data are from approximately a normal distribution. Contrary, lower one shows asymmetricity (skewed to the left heavily), data are more peaked than the normal distribution. Additionally, there is a limit in the data (30) that can be explained by the subject of the optimization (c.f. 1b).

A3: The lag plots do not indicate any systematic behavior pattern in the data.

A4: The normal probability plot in upper approximately follows straight lines through the first and third quartiles of the samples, indicating normal distributions. On the contrary, normality assumption is in fact not reasonable on the right.

From the above plots, we conclude that there is no correlation among the data (**A3**), the historical data follow approximately a normal distribution (**A4**), and the optimized approach yields more smooth release padding and less variance (**A1**,**A2**).

Quantitative Analysis. Due to **A3** data sets could be analyzed with summary (descriptive) statistics (Table 3), and hypothesis test. Table 3 shows important differences between the historical and optimized data:

- **D1:** in the optimized case sample standard deviation is approximately halved, which supports **A1**,
- **D2:** despite of the fact that iteration velocity was 30 Story points the release plan prescribed 35 in the historical case which resulted 17% resource overload (c.f. **A2**),
- **D3:** relatively large skewness of the optimized case (histogram in Figure 5) can be interpreted by the capacity constraints of the optimization (see 1b),
- **D4:** relatively large kurtosis of the optimized case (histogram in Figure 5) can be explained by the subject of the optimization (see 1a).

After statistical analysis, Lilliefors test is carried out to quantify the test of normality (c.f. **A4**) at $\alpha = 95\%$ significance level: historical data comes from a normal distribution ($H_0 : F(Y_i) = \Theta(Y_i)$), against the alternative hypothesis ($H_1 : F(Y_i) \neq \Theta(Y_i)$). The result yielded *p-value* = 0.5 (observed significance level). As *p-value*$> (1 - \alpha)$ so historical data follow normal distribution (so H_0 was accepted at 95% significance level). Since the sample was relatively small, the Lilliefors test was adequate [29].

Finally, maximum likelihood estimation (MLE) procedure was accomplished to find the value of μ (expected value) and σ (standard deviation) parameters of the normal distribution. The estimation resulted $\mu = 28.53$ and $\sigma = 4.63$ values [30].

As a consequence, in the optimized case: staffing requirements (c.f. **Q1**) showed more smooth release padding, with less variance and an upper limit, therefore constituted less risk level regarding resource overload; iteration counts per releases (c.f. **Q2**) did not exhibit any differences contrary to the historical data; finally release buffers (c.f. **Q3**) were moved from the end of iterations to the end of releases which more advisable to mitigate risks [31].

5 Discussion and Related Work

Without loss of generality, we have selected User story as the most popular agile requirements modeling technique as a subject of release planning. User stories have many advantages including i) comprehensible to both customers and the developers, ii) emphasize verbal rather than written communication, iii) represent a discrete piece of functionality, iv) work for iterative development, and finally v) right sized for estimating (i.e. Story points [16]) and planning [9,15].

Table 3. Comparison with descriptive statistics

	Mean	Min	Max	Std.dev.	Skewness	Kurtosis
R_{A-B}	28.53	18.0	35.0	4.78	-0.48	2.50
R^*_{A-B}	28.53	19.0	30.0	2.75	-2.82	10.35

We applied the popular Story point method to estimate realization duration of each User story. Up to now, several case studies reported that the Story point is a reliable method to estimate the required effort at the release planning phase [6,7,16].

We constructed a proposed conceptual model of agile release planning (c.f. Sec. 3.1). This model not only helped to identify the objects and the subject of the optimization model but promoted to implement our prototype of collaborative data collection website (Sec. 3.2). Moreover it can help in the design of future release planning tools also.

Then we formulated release planning as BPR-RCPSP to provide algorithmic User story distribution considering i) team's resource capacity in an iteration and ii) minimizing the number of iteration used scheduling objective. Our proposed BP-RCPSP is an extension of bin-packing optimization model to cover wide-ranging release planning situations with the expression of: i) precedences between requirements (c.f. Sec. 2.1), ii) varying capacities of iterations, and iii) requirements priorities (c.f. **P1-3**). This interpretation makes it possible to adapt extremely successful heuristic algorithms applied to solving bin-packing situations. Generally, bin-packing problems are combinatorial NP-hard problems to which a variety approximation and only a few exact algorithms are proposed. The most popular heuristics in approximation algorithms are *First-Fit* (FF), Next-Fit Decreasing (NFD), First-Fit Decreasing (FFD), where the time complexity is $O(nlogn)$ – considering the worst-case performance ratio of the algorithm [25].

We developed a bin-packing algorithm (*Binsched*) for the BP-RCPSP model which illustrated the iteration capacities are filled more smoothly (c.f. **Q1**) and release buffers are adjusted to the end of the last iterations (c.f. **Q3**) to prevent slippage of schedule by the optimal usage of buffers [31]. Metrics indicated that the algorithmic approach balanced the workload by halved the dispersion (coefficient of variation ($c_v = \sigma/\mu$): $c_v^{hist} = 0.17 > c_v^{optm} = 0.09$) therefore provided less risky release plans besides satisfying the same constraints. Moreover, the easy and fast computation allows the user to generate alternative selections and what-if analysis to tailor the best plan for the specific project context and considering the stakeholders' feedbacks by altering constraints, capacities and priorities.

The growing pressure to reduce costs, time-to-market and to improve quality catalyzes transitions to more automated methods and tools in software engineering to support project planning, scheduling and decisions [14]. Scheduling requirements into the upcoming release version is complex and mainly manual process. In order to deal with this optimization problem some method have been proposed. Compared to the extensive research on requirements priorization [32,33], interdependencies [20,18], and estimation [20], only few researches performed requirements release planning. In [20] release planning was formulated as Integer Linear Programming (ILP) problem, where requirement dependencies were treated as precedence constraints. The ILP technique is extended with stakeholders' opinions in [34], and with some managerial steering mechanism that enabled what-if analysis [35]. In [36] a case study showed that integration of requirements and planning how significantly ($> 50\%$) can accelerate UML-based release planning. Although, [37] deals with the specialities of agile methods, it provides planning support at a lower level, at the level of iterations.

6 Conclusions

Up to our best knowledge, the proposed optimized model formulation of agile release planning is novel in the field. Although, there are some tenets to manual planning [6,15] algorithmic solution could not be found. To evaluate our model a simulation was carried out that demonstrated the method could easily cope with the previously manually managed planning factors i.e. precedences, resource constraints and priorities (c.f. **P1-3**) besides providing optimized plans. Additionally, this approach provides more informed and established decisions with application of what-if analysis, and mitigates risks with more smooth and limited requirements allocation and with moving buffers to the end of releases. We believe the results are even more impressive in more complex (more of constraints, user stories etc.) situations.

We think that our proposed method is a plain combination of the present theories and methods, that is demonstrated by the empirical investigation and the prototypes. It lead us to generalize our findings beyond the presented experiments.

References

1. Information Systems Audit and Control Association, http://www.isaca.org
2. Organizational PM maturity model, http://www.pmi.org
3. Ambler, S.W.: Best practices for lean development governance. The Rational Edge (2007)
4. Chau, T., Maurer, F., Melnik, G.: Knowledge Sharing: Agile Methods vs. Tayloristic Methods. In: Proceedings of the 12th IEEE International Workshops on Enabling Technologies, pp. 302–307. IEEE Press, Los Alamitos (2003)
5. Manifesto for agile software development, http://www.agilemanifesto.org
6. Dybå, T., Dingsøyr, T.: Empirical studies of agile software development: A systematic review. Information and Software Technology 50(9-10) (2008)
7. Ambler, S.W.: Survey says: Agile works in practice. Dr. Dobb's Journal (2006), http://www.ddj.com
8. Layman, L., Williams, L., Cunningham, L.: Motivations and measurements in an agile case study. Journal of Systems Architecture 52(11) (2006)
9. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd edn. Addison-Wesley Professional, Reading (2004)
10. Layman, L., Williams, L., Cunningham, L.: Exploring extreme programming in context: An industrial case study. In: ADC 2004: Proceedings of the Agile Development Conference, pp. 32–41 (2004)
11. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. Prentice Hall PTR, Upper Saddle River (2001)
12. Palmer, S.R., Felsing, M.: A Practical Guide to Feature-Driven Development. Pearson Education, London (2001)
13. Chow, T., Cao, D.B.: A survey study of critical success factors in agile software projects. Journal of System and Software 81(6) (2008)
14. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: ICSE 2000: Proceedings of the International Conference on Software Engineering, pp. 35–46. IEEE Press, Los Alamitos (2000)
15. Cohn, M.: Agile Estimating and Planning. Prentice Hall PTR, NJ (2005)
16. Cohn, M.: User Stories Applied For Agile Software Development. Addison-Wesley, Reading (2004)
17. Anda, B., Dreiem, H., Sjøberg, D.I.K., Jørgensen, M.: Estimating software development effort based on use cases - experiences from industry. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 487–502. Springer, Heidelberg (2001)
18. Li, C., van den Akker, J.M., Brinkkemper, S., Diepen, G.: Integrated requirement selection and scheduling for the release planning of a software product. In: Sawyer, P., Paech, B., Heymans, P. (eds.) REFSQ 2007. LNCS, vol. 4542, pp. 93–108. Springer, Heidelberg (2007)
19. Ambler, S.W., Jeffries, R.: Agile modeling: effective practices for extreme programming and the unified process. John Wiley & Sons, Inc., NY (2002)

20. Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., Dag, J.N.: An industrial survey of requirements interdependencies in software product release planning. In: RE 2001: Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, pp. 84–93. IEEE Press, Los Alamitos (2001)
21. Object Management Group, `http://www.uml.org`
22. Microsoft SharePoint (2007), `http://sharepoint.microsoft.com`
23. Multilogic Ltd. homepage, `http://www.multilogic.hu`
24. Hartmann, S.: Packing problems and project scheduling models: an integrating perspective. Journal of the Operational Research Society 51 (2000)
25. Martello, S., Toth, P.: Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc., New York (1990)
26. Schwindt, C.: Resource Allocation in Project Management. Springer, Heidelberg (2005)
27. Mathworks Inc. homepage, `http://www.mathworks.com`
28. Kellner, M., Madachy, R., Raffo, D.: Software process simulation modeling: Why? what? how? Journal of Systems and Software 46(2-3) (1999)
29. Martinez, W.L.: Exploratory Data Analysis with MATLAB. Chapman & Hall/CRC, Boca Raton (2004)
30. Shao, J.: Mathematical Statistics: Exercises and Solutions. Springer, Heidelberg (2005)
31. Tukel, O.I., Rom, W.O., Eksioglu, S.D.: An investigation of buffer sizing techniques in critical chain scheduling. European Journal of Operational Research 172(2) (2006)
32. Berander, P., Andrews, A.: Requirements Prioritization. In: Engineering and Managing Software Requirements, pp. 69–94. Springer, Heidelberg (2005)
33. Karlsson, L., Thelin, T., Regnell, B., Berander, P., Wohlin, C.: Pair-wise comparisons versus planning game partitioning–experiments on requirements prioritisation techniques. Empirical Software Engineering 12(1) (2007)
34. Ruhe, G., Saliu, M.: The art and science of software release planning. IEEE Software 22(6) (2005)
35. van den Akker, M., Brinkkemper, S., Diepen, G., Versendaal, J.: Software product release planning through optimization and what-if analysis. Information and Software Technology 50(1-2) (2008)
36. Szőke, A.: A proposed method for release planning from use case-based requirements. In: Proceedings of the 34th Euromicro Conference, pp. 449–456. IEEE Press, Los Alamitos (2008)
37. Szőke, A.: Decision support for iteration scheduling in agile environments. In: Bomarius, F., Oivo, M., Jaring, P., Abrahamsson, P. (eds.) PROFES 2009. LNBIP, vol. 32, pp. 156–170. Springer, Heidelberg (2009)

A Method to Measure Productivity Trends during Software Evolution

Hans Christian Benestad, Bente Anda, and Erik Arisholm

Simula Research Laboratory and University of Oslo
P.O. Box 134, 1325 Lysaker, Norway
{benestad,bentea,erika}@simula.no
http://www.simula.no

Abstract. Better measures of productivity are needed to support software process improvements. We propose and evaluate indicators of productivity trends that are based on the premise that productivity is closely related to the effort required to complete change tasks. Three indicators use change management data, while a fourth compares effort estimates of benchmarking tasks. We evaluated the indicators using data from 18 months of evolution in two commercial software projects. The productivity trend in the two projects had opposite directions according to the indicators. The evaluation showed that productivity trends can be quantified with little measurement overhead. We expect the methodology to be a step towards making quantitative self-assessment practices feasible even in low ceremony projects.

1 Introduction

The productivity of a software organization that maintains and evolves software can decrease over time due to factors like code decay [1] and difficulties in preserving and developing the required expertise [2]. Refactoring [3] and collaborative programming [4] are practices that can counteract negative trends. A development organization might have expectations and gut feelings about the total effect of such factors and accept a moderate decrease in productivity as the system grows bigger and more complex. However, with the ability to quantify changes in productivity with reasonable accuracy, organizations could make informed decisions about the need for improvement actions. The effects of new software practices are context dependent, and so it would be useful to subsequently evaluate whether the negative trend was broken.

The overall aim for the collaboration between our research group and two commercial software projects (henceforth referred to as MT and RCN) was to understand and manage evolution costs for object-oriented software. This paper was motivated by the need to answer the following practical question in a reliable way:

Did the productivity in the two projects change between the baseline period P0 (Jan-July 2007) and the subsequent period P1 (Jan-July 2008)?

The project RCN performed a major restructuring of their system during the fall of 2007. It was important to evaluate whether the project benefitted as expected from the restructuring effort. The project MT added a substantial set of new features since the

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 147–162, 2010.

start of *P0* and queried whether actions that could ease further development were needed. The methodology used to answer this question was designed to become part of the projects' periodic self-assessments, and aimed to be a practical methodology in other contexts as well.

Our contribution is i) to define the indicators within a framework that allows for a common and straightforward interpretation, and ii) to evaluate the validity of the indicators in the context of two commercial software projects. The evaluation procedures are important, because the validity of the indicators depends on the data at hand.

The remainder of this paper is structured as follows: Section 2 describes current approaches to measuring productivity, Section 3 describes the design of the study, Section 4 presents the results and the evaluation of the indicators and Section 5 discusses the potential for using the indicators. Section 6 concludes.

2 Current Approaches to Measuring Productivity

In a business or industrial context, productivity refers to the ratio of output production to input effort [5]. In software engineering processes, inputs and outputs are multidimensional and often difficult to measure. In most cases, development effort measured in man-hours is a reasonable measure of input effort. In their book on software measurement, Fenton and Pfleeger [6] discussed measures of productivity based on the following definition of software productivity:

$$\text{productivity} = \frac{\text{size}}{\text{effort}} \tag{1}$$

Measures of developed size include *lines of code*, *affected components* [7], *function points* [8-10] and *specification weight metrics* [11]. By plotting the productivity measure, say, every month, projects can examine trends in productivity. Ramil and Lehman used a statistical test (CUSUM) to detect statistically significant changes over time [12]. The same authors proposed to model development effort as a function of size:

$$\text{effort} = \beta_0 + \beta_1 \cdot \text{size} \tag{2}$$

They suggested collecting data on effort and size periodically, e.g., monthly, and to interpret changes in the regression coefficients as changes in *evolvability*. *Number of changed modules* was proposed as a measure of size. The main problem with these approaches is to define a size measure that is both meaningful and easy to collect. This is particularly difficult when software is changed rather than developed from scratch.

An alternative approach, corresponding to this paper's proposal, is to focus on the *completed change task* as the fundamental unit of output production. A change task is the development activity that transforms a change request into a set of modifications to the source components of the system. When software evolution is organized around a queue of change requests, the completed change task is a more intuitive measure of output production than traditional size measures, because it has more direct value to complete a change task than to produce another *n* lines of code. A corresponding input measure is the development effort required to complete the change task, referred to as *change effort*.

Several authors compared average change effort between time periods to assess trends in the maintenance process [13-15]. Variations of this indicator include average change effort per maintenance type (e.g., corrective, adaptive or enhancive maintenance). One of the proposed indicators uses direct analysis of change effort. However, characteristics of change tasks may change over time, so focusing solely on change effort might give an incomplete picture of productivity trends.

Arisholm and Sjøberg argued that *changeability* may be evaluated with respect to the same change task, and defined that changeability had *decayed* with respect to a given change task *c* if the effort to complete *c* (including the consequential change propagation) increased between two points in time [16]. We consider *productivity* to be closely related to *changeability*, and we will adapt their definition of *changeability decay* to *productivity change*.

In practice, comparing the same change tasks over time is not straightforward, because change tasks rarely re-occur. To overcome this practical difficulty, developers could perform a set of "representative" tasks in periodic *benchmarking* sessions. One of the proposed indicators is based on benchmarking identical change tasks. For practical reasons, the tasks are only estimated (in terms of expected change effort) but are not completed by the developers.

An alternative to benchmarking sessions is using naturally occurring data about change tasks and adjusting for differences between them when assessing trends in productivity. Graves and Mockus retrieved data on 2794 change tasks completed over 45 months from the version control system for a large telecommunication system [17]. A regression model with the following structure was fitted on this data:

$$\text{Change effort} = \text{f}\,(\text{developer}, \text{type}, \text{size}, \text{date}) \tag{3}$$

The resulting regression coefficient for *date* was used to assess whether there was a time trend in the effort required to complete change tasks, while controlling for variations in other variables. One of our proposed indicators is an adaption of this approach.

A conceptually appealing way to think about productivity change is to compare change effort for a set of completed change tasks to the hypothetical change effort *had the same changes been completed at an earlier point in time*. One indicator operationalizes this approach by comparing change effort for completed change tasks to the corresponding effort estimates from statistical models. This is inspired by Kitchenham and Mendes' approach to measuring the productivity of finalized projects by comparing actual project effort to model-based effort estimates [18].

3 Design of the Study

The projects, the collected data and the proposed productivity indicators are described in the following subsections.

3.1 Context for Data Collection

The overall goal of the research collaboration with the projects RCN and MT was to better understand lifecycle development costs for object-oriented software.

MT is mostly written in Java, but uses C++ for low-level control of hardware. RCN is based on Java-technology, and uses a workflow engine, a JEE application server, and a UML-based code generation tool. Both projects use management principles from Scrum [19]. Incoming change requests are scheduled for the monthly releases by the development group and the product owner. Typically, 10-20 percent of the development effort was expended on corrective change tasks. The projects worked under time-and-material contracts, although fixed-price contracts were used in some cases. The staffing in the projects was almost completely stable in the measurement period.

Project RCN had planned for a major restructuring in their system during the summer and early fall of 2007 (between *P0* and *P1*), and was interested in evaluating whether the system was easier to maintain after this effort. Project MT added a substantial set of new features over the two preceding years and needed to know if actions easing further development were now needed.

Data collection is described in more detail below and is summarized in Table 1.

Table 1. Summary of data collection

	RCN	MT
Period *P0*	Jan 01 2007 - Jun 30 2007	Aug 30 2006 - Jun 30 2007
Period *P1*	Jan 30 2008 - Jun 30 2008	Jan 30 2008 - Jun 30 2008
Change tasks in *P0/P1*	136/137	200/28
Total change effort in *P0/P1*	1425/1165 hours	1115/234 hours
Benchmarking sessions	Mar 12 2007, Apr 14 2008	Mar 12 2007, Apr 14 2008
Benchmark tasks	16	16
Developers	4 (3 in benchmark)	4

3.2 Data on Real Change Tasks

The first three proposed indicators use data about change tasks completed in the two periods under comparison. It was crucial for the planned analysis that data on change effort was recorded by the developers, and that source code changes could be traced back to the originating change request. Although procedures that would fulfill these requirements were already defined by the projects, we offered an economic compensation for extra effort required to follow the procedures consistently.

We retrieved data about the completed change tasks from the projects' change trackers and version control systems by the end of the baseline period (P0) and by the end of the second period (*P1*). From this data, we constructed measures of change tasks that covered requirements, developers' experience, size and complexity of the change task and affected components, and the type of task (corrective vs. non-corrective). The following measures are used in the definitions of the productivity indicators in this paper:

- *crTracks* and *crWords* are the number of updates and words for the change request in the change tracker. They attempt to capture the volatility of requirements for a change task.
- *components* is the number of source components modified as part of a change task. It attempts to capture the dispersion of the change task.

- i*sCorrective* is 1 if the developers had classified the change task as corrective, or if the description for the change task in the change tracker contained strings such as *bug*, *fail* and *crash*. In all other cases, the value of *isCorrective* is 0.
- *addCC* is the number of control flow statements added to the system as part of a change task. It attempts to capture the control-flow complexity of the change task.
- *systExp* is the number of earlier version control check-ins by the developer of a change task.
- *chLoc* is the number of code lines that are modified in the change task.

A complete description of measures that were hypothesized to affect or correlate with change effort is provided in [20].

3.3 Data on Benchmark Tasks

The fourth indicator compares developers' effort estimates for benchmark change tasks between two *benchmarking sessions*. The 16 benchmark tasks for each project were collaboratively designed by the first author of this paper and the project managers. The project manager's role was to ensure that the benchmark tasks were representative of real change tasks. This meant that the change tasks should not be perceived as artificial by the developers, and they should cross-cut the main architectural units and functional areas of the systems.

The sessions were organized approximately in the midst of *P0* and *P1*. All developers in the two projects participated, except for one who joined RCN during *P0*. We provided the developers with the same material and instructions in the two sessions. The developers worked independently, and had access to their normal development environment. They were instructed to identify and record affected methods and classes before they recorded the estimate of most likely effort for a benchmark task. They also recorded estimates of uncertainty, the time spent to estimate each task, and an assessment of their knowledge about the task. Because our interest was in the productivity of the *project*, the developers were instructed to assume a normal assignment of tasks to developers in the project, rather than estimating on one's own behalf.

3.4 Design of Productivity Indicators

We introduce the term *productivity ratio* (PR) to capture the change in productivity between period *P0* and a subsequent period *P1*.

The productivity ratio with respect to a single change task *c* is the ratio between the effort required to complete *c* in *P1* and the effort required to complete *c* in *P0*:

$$\mathrm{PR}(\mathrm{c}) = \frac{\mathrm{effort}(\mathrm{c}, \mathrm{P1})}{\mathrm{effort}(\mathrm{c}, \mathrm{P0})} \tag{4}$$

The productivity ratio with respect to a set of change tasks *C* is defined as the set of individual values for *PR(c)*:

$$\mathrm{PR}(\mathrm{C}) = \{\, \frac{\mathrm{effort}(\mathrm{c}, \mathrm{P1})}{\mathrm{effort}(\mathrm{c}, \mathrm{P0})} \mid \mathrm{c} \in \mathrm{C} \,\} \tag{5}$$

The central tendency of values in *PR(C)*, *CPR(C),* is a useful single-valued statistic to assess the typical productivity ratio for change tasks in *C*:

$$\mathrm{CPR}(C) = \mathrm{central}\{\frac{\mathrm{effort}(c,P1)}{\mathrm{effort}(c,P0)} \mid c \in C\} \tag{6}$$

The purpose of the above definition is to link practical indicators to a common theoretical definition of productivity change. This enables us to define scale-free, comparable indicators with a straightforward interpretation. For example, a value of 1.2 indicates a 20% increase in effort from *P0* to *P1* to complete the same change tasks. A value of 1 indicates no change in productivity, whereas a value of 0.75 indicates that only 75% of the effort in *P0* is required in *P1*. Formal definitions of the indicators are provided in Section 3.4.1 to 3.4.4.

3.4.1 Simple Comparison of Change Effort

The first indicator requires collecting only change effort data. A straightforward way to compare two series of unpaired effort data is to compare their arithmetic means:

$$\mathrm{ICPR}_1 = \frac{\mathrm{mean}(\mathrm{effort}(c1) \mid c1 \in P1)}{\mathrm{mean}(\mathrm{effort}(c0) \mid c0 \in P0)} \tag{7}$$

The Wilcoxon rank-sum test determines whether there is a statistically significant difference in change effort values between *P0* and *P1*. One interpretation of this test is that it assesses whether the median of all possible differences between change effort in *P0* and *P1* is different from 0:

$$\mathrm{HL} = \mathrm{median}(\mathrm{effort}(c1) - \mathrm{effort}(c0) \mid c1 \in P1, c0 \in P0) \tag{8}$$

This statistic, known as the Hodges-Lehmann estimate of the difference between values in two data sets, can be used to complement ICPR_1. The actual value for this statistic is provided with the evaluation of ICPR_1, in Section 4.1.

ICPR_1 assumes that the change tasks in *P0* and *P1* are comparable, i.e. that there are no systematic differences in the properties of the change tasks between the periods. We checked this assumption by using descriptive statistics and statistical tests to compare measures that we assumed (and verified) to be correlated with change effort in the projects (see Section 4.2). These measures were defined in Section 3.2.

3.4.2 Controlled Comparison of Change Effort

ICPR₂ also compares change effort between *P0* and *P1*, but uses a statistical model to control for differences in properties of the change tasks between the periods. Regression models with the following structure for respectively RCN and MT are used:

$$\log(\mathrm{effort}) = \beta_0 + \beta_1 \cdot \mathrm{crWords} + \beta_2 \cdot \mathrm{chLoc} + \beta_3 \cdot \mathrm{filetypes} + \beta_4 \cdot \mathrm{isCorr} + \beta_5 \cdot \mathrm{inP1}. \tag{9}$$

$$\log(\mathrm{effort}) = \beta_0 + \beta_1 \cdot \mathrm{crTracks} + \beta_2 \cdot \mathrm{addCC} + \beta_3 \cdot \mathrm{components} + \beta_4 \cdot \mathrm{systExp} + \beta_5 \cdot \mathrm{inP1}. \tag{10}$$

The models (9) and (10) are project specific models that we found best explained variability in change effort, c.f. [20]. The dependent variable *effort* is the reported change effort for a change task. The variable *inP1* is 1 if the change task *c* was completed in *P1* and is zero otherwise. The other variables were explained in Section 3.2. When all explanatory variables except *inP1* are held constant, which would be the case if one applies the model on the same change tasks but in the two, different time periods *P0* and *P1*, the ratio between change effort in *P1* and *P0* becomes

$$\begin{aligned}\mathrm{ICPR}_2 &= \frac{\mathrm{effort}(\mathrm{Var1..Var4,inP1=1})}{\mathrm{effort}(\mathrm{Var1..Var4,inP1=0})}\\ &= \frac{e^{\beta0+\beta1\cdot \mathrm{Var1}+\beta2\cdot \mathrm{Var2}+\beta3\cdot \mathrm{Var3}+\beta4\cdot \mathrm{Var4}+\beta5\cdot 1}}{e^{\beta0+\beta1\cdot \mathrm{Var1}+\beta2\cdot \mathrm{Var2}+\beta3\cdot \mathrm{Var3}+\beta4\cdot \mathrm{Var4}+\beta5\cdot 0}} = e^{\beta5}.\end{aligned} \tag{11}$$

Hence, the value of the indicator can be obtained by looking at the regression coefficient for *inP1*, β5. Furthermore, the p-value for β5 is used to assess whether β5 is significantly different from 0, i.e. that the indicator is different from 1 (e^0=1).

Corresponding project specific models must be constructed to apply the indicator in other contexts. The statistical framework used was Generalized Linear Models assuming Gamma-distributed responses (change effort) and a *log* link-function.

3.4.3 Comparison between Actual and Hypothetical Change Effort

ICPR$_3$ compares change effort for tasks in *P1* with the hypothetical change effort had the same tasks been performed in *P0*. These hypothetical change effort values are generated with a project-specific prediction model built on data from change tasks in *P0*. The model structure is identical to (9) and (10), but without the variable *inP1*.

Having generated this paired data on change effort, the definition (6) can be used directly to define ICPR$_3$. To avoid over-influence of outliers, the median is used as a measure of central tendency.

$$\mathrm{ICPR}_3 = \mathrm{median}\{\frac{\mathrm{effort(c)}}{\mathrm{predictedEffort(c)}} \mid c \in \mathrm{P1}\} \tag{12}$$

A two-sided sign test is used to assess whether actual change effort is higher (or lower) than the hypothetical change effort in more cases than expected from chance. This corresponds to testing whether the indicator is statistically different from 1.

3.4.4 Benchmarking

ICPR$_4$ compares developers' estimates for 16 benchmark change tasks between *P0* and *P1*. Assuming the developers' estimation accuracy does not change between the periods, a systematic change in the estimates for the same change tasks would mean that the productivity with respect to these change tasks had changed. Effort estimates made by developers *D* for benchmarking tasks **C$_b$** in periods *P1* and *P0* therefore give rise to the following indicator:

$$\mathrm{ICPR}_4 = \mathrm{median}\{\frac{\mathrm{estEffort(P1,d,c)}}{\mathrm{estEffort(P0,d,c)}} \mid c \in C_b, d \in D\} \tag{13}$$

A two-sided sign test determines whether estimates in *P0* were higher (or lower) than the estimates in *P1* in more cases than expected from chance. This corresponds to testing whether the indicator is statistically different from 1.

Controlled studies show that judgement-based estimates can be unreliable, i.e. that there can be large random variations in estimates by the same developer [21]. Collecting more estimates reduces the threat implied by random variation. The available time for the benchmarking session allowed us to collect 48 (RCN – three developers) and 64 (MT – four developers) pairs of estimates.

One source of change in estimation accuracy over time is that developers may become more experienced, and hence provide more realistic estimates. For project RCN, it was possible to evaluate this threat by comparing the estimation bias for *actual changes* between the periods. For project MT, we did not have enough data about estimated change effort for real change tasks, and we could not evaluate this threat.

Other sources of change in estimation accuracy between the sessions are the context for the estimation, the exact instructions and procedures, and the mental state of the developers. While impossible to control perfectly, we attempted to make the two benchmarking sessions as identical as possible, using the same, precise instructions and material. The developers were led to a consistent (bottom-up) approach by our instructions to identify and record affected parts of the system before they made each estimate.

Estimates made in P1 could be influenced by estimates in P0 if developers remembered their previous estimates. After the session in P1, the feedback from all developers was that they did not remember their estimates or any of the tasks.

An alternative benchmarking approach is comparing change effort for benchmark tasks that were actually completed by the developers. Although intuitively appealing, the analysis would still have to control for random variation in change effort, outcomes beyond change effort, representativeness of change tasks, and also possible learning effects between benchmarking sessions.

In certain situations, it would even be possible to compare change effort for change tasks that recur naturally during maintenance and evolution (e.g., adding a new data provider to a price aggregation service). Most of the threats mentioned above would have to be considered in this case, as well. We did not have the opportunities to use these indicators in our study.

3.5 Accounting for Changes in Quality

Productivity analysis could be misleading if it does not control for other outcomes of change tasks, such as the change task's effect on system qualities. For example, if more time pressure is put on developers, change effort could decrease at the expense of correctness. We limit this validation to a comparison of the amount of corrective and non-corrective work between the periods. The evaluation assumes that the change task that introduced a fault was completed within the same period as the task that corrected the fault. Due to the short release-cycle and half-year leap between the end of *P0* and the start of *P1*, we are confident that change tasks in *P0* did not trigger fault corrections in *P1*, a situation that would have precluded this evaluation.

4 Results and Validation

The indicator values with associated p-values are given in Table 2.

Table 2. Results for the indicators

	RCN		MT	
Indicator	Value	p-value	Value	p-value
$ICPR_1$	0.81	0.92	1.50	0.21
$ICPR_2$	0.90	0.44	1.50	0.054
$ICPR_3$	0.78	<0.0001	1.18	0.85
$ICPR_4$	1.00	0.52	1.33	0.0448

For project RCN, the analysis of real change tasks indicate that productivity increased, since between 10 and 22% less effort was required to complete change tasks in *P1*. *$ICPR_4$* indicates no change in productivity between the periods. The project had refactored the system throughout the fall of 2008 as planned. Overall, the indicators are consistent with the expectation that the refactoring initiative would be effective. Furthermore, the subjective judgment by the developers was that the goal of the refactoring was met, and that change tasks were indeed easier to perform in *P1*.

For project MT, the analysis of real change tasks (*$ICPR_1$*, *$ICPR_2$* and *$ICPR_3$*) indicate a drop in productivity, with somewhere between 18 and 50% more effort to complete changes in *P1* compared with *P0*. The indicator that uses benchmarking data ($ICPR_4$) supports this estimate, being almost exactly in the middle of this range. The project manager in MT proposed post-hoc explanations as to why productivity might have decreased. During *P0*, project MT performed most of the changes under fixed-price contracts. In *P1*, most of the changes were completed under time-and material contracts. The project manager indicated that the developers may have experienced more time pressure in *P0*.

As discussed in Section 3.5, the indicators only consider trends in change effort, and not trends in other important outcome variables that might confound the results, e.g., positive or negative trends in quality of the delivered changes. To assess the

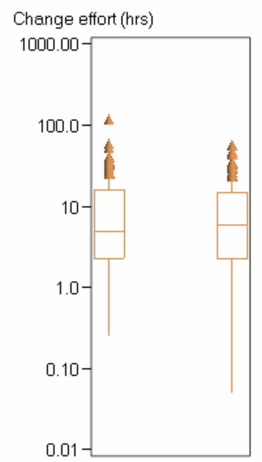


Fig. 1. Change effort in RCN, *P0* (left) vs. *P1*

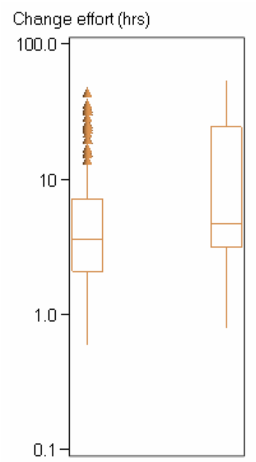


Fig. 2. Change effort in MT, *P0* (left) vs. *P1*

validity of our indicators with respect to such confounding effects, we compared the amount of corrective versus non-corrective work in the periods. For MT, the percentage of total effort spent on corrective work dropped from 35.6% to 17.1% between the periods. A plausible explanation is that the developers, due to less time pressure, expended more time in *P1* ensuring that the change tasks were correctly implemented. So even though the productivity indicators suggest a drop, the correctness of changes was also higher. For RCN, the percentage of the total effort spent on corrective work increased from 9.7% to 15%, suggesting that increased productivity was at the expense of slightly lesser quality.

4.1 Validation of ICPR$_1$

The distribution of change effort in the two periods is shown in Fig. 1 (RCN) and Fig. 2 (MT). The square boxes include the mid 50% of the data points. A log scale is used on the y-axis, with units in hours. Triangles show outliers in the data set.

For RCN, the plots for the two periods are very similar. The Hodges-Lehmann estimate of difference between two data sets (8) is 0, and the associated statistical test does not indicate a difference between the two periods. For MT, the plots show a trend towards higher change effort values in *P1*. The Hodges-Lehmann estimate is plus one hour in *P1*, and the statistical test showed that the probability is 0.21 that this result was obtained by pure chance.

If there were systematic differences in the properties of the change tasks between the periods, ICPR$_1$ can be misleading. This was assessed by comparing values for variables that capture certain important properties. The results are shown in Table 3 and Table 4. The Wilcoxon rank-sum test determined whether changes in these variables were statistically significant. In the case of *isCorrective*, the Fischer's exact test determined whether the proportion of corrective change tasks was significantly different in the two periods.

For RCN, *chLoc* significantly increased between the periods, while there were no statistically significant changes in the values of other variables. This indicates that

Table 3. Properties of change tasks in RCN

Variable	*P0*	*P1*	p-value
chLoc (mean)	26	104	0.0004
crWords (mean)	107	88	0.89
filetypes (mean)	2.7	2.9	0.50
isCorrective (%)	38	39	0.90

Table 4. Properties of change tasks in MT

Variable	*P0*	*P1*	p-value
addCC (mean)	8.7	44	0.06
components (mean)	3.6	7	0.09
crTracks (mean)	4.8	2.5	<0.0001
systExp (mean)	1870	2140	0.43

larger changes were completed in *P1*, and that the indicated gain in productivity is a conservative estimate

For MT, *crTracks* significantly decreased between *P0* and *P1*, while *addCC* and *components* increased in the same period. This indicates that more complex changes were completed in *P1*, but that there was less uncertainty about requirements. Because these effects counteract, it cannot be determined whether the value for ICPR_1 is conservative. This motivates the use of ICPR_2 and ICPR_3, which explicitly control for changes in the mentioned variables.

4.2 Validation of ICPR_2

ICPR_2 is obtained by fitting a model of change effort on change task data from *P0* and *P1*. The model includes a binary variable representing period of change (*inP1*) to allow for a constant proportional difference in change effort between the two periods. The statistical significance of the difference can be observed directly from the p-value of that variable. The fitted regression expressions for RCN and MT were:

$$\begin{gathered}\log(\text{effort}) = 9.5 + 0.0018 \cdot \text{crWords} + 0.2258 \cdot \text{filetypes} + 0.00073 \cdot \text{changed} - \\ 0.79 \cdot \text{isCorrective} - 0.10 \cdot \text{inP1}.\end{gathered} \tag{14}$$

$$\begin{gathered}\log(\text{effort}) = 9.1 + 0.088 \cdot \text{crTracks} + 0.0041 \cdot \text{addCC} + 0.098 \cdot \text{components} - \\ 0.00013 \cdot \text{systExp} + 0.40 \cdot \text{inP1}.\end{gathered} \tag{15}$$

The p-value for *inP1* is low (0.054) for MT and high (0.44) for RCN. All the other model variables have p-values lower than 0.05. For MT, the interpretation is that when these model variables are held constant, change effort increases by 50% ($e^{0.40}$=1.50). A plot of deviance residuals in Fig. 3 and Fig. 4 is used to assess whether the modelling framework (GLM with gamma distributed change effort and log link function) was appropriate. If the deviance residuals increase with higher outcomes (overdispersion) the computed p-values would be misleading. The plots show no sign of overdispersion. This validation increases the confidence in this indicator for project MT. For project RCN, the statistical significance is too weak to allow confidence in this indicator alone.

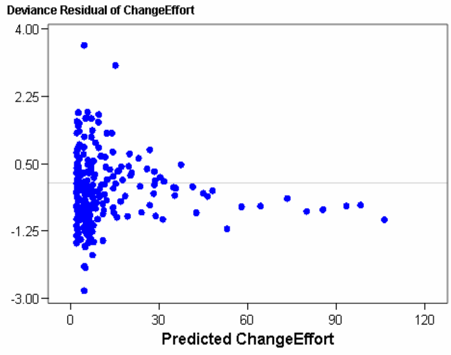


Fig. 3. Residual plot for RCN model (14)

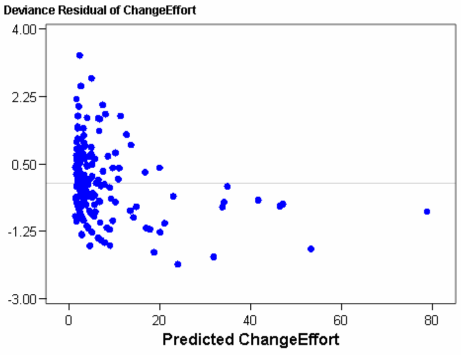


Fig. 4. Residual plot for MT model (15)

4.3 Validation of ICPR$_3$

ICPR$_3$ compares change effort in *P1* with the model-based estimates for the same change tasks had they been completed in *P0*. The model was fitted on data from *P0*. Fig. 5 shows that actual change effort tends to be higher than estimated effort for MT, while the tendency is opposite for RCN. For RCN, the low p-value shows that that actual change effort is systematically lower than the model-based estimates. For project MT, the high p-value means that actual effort was not systematically higher.

If the variable subset is overfitted to data from P0, the model-based estimates using data from P1 can be misleading. To evaluate the stability of the model structure, we compared the model residuals in the *P0* model with those in a new model fitted on data from *P1* (using the same variable subset). For MT, the model residuals were systematically larger (Wilcoxon rank-sum test, p=0.0048). There was no such trend for RCN (Wilcoxon rank-sum test, p=0.78), indicating a more stable model structure.

Another possible problem with ICPR$_3$ is that model estimates can degenerate for variable values poorly covered by the original data set. Inspection of the distributions for the independent variables showed that there was a potential problem with the variable *chLoc*, also indicated by the large difference in mean, shown in Table 3. We re-calculated ICPR$_3$ after removing the 10 data points that were poorly covered by the original model, but this did not affect the value of the indicator.

In summary, the validation for ICPR$_3$ gives us high confidence in the result for project RCN, due to high statistical significance, and evidence of a stable underlying model structure. For project MT, the opposite conclusion applies.

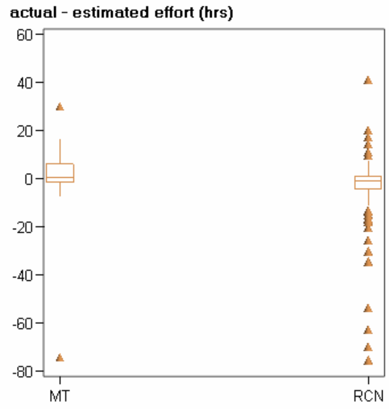


Fig. 5. Model estimates subtracted from actual effort

4.4 Validation of ICPR$_4$

ICPR$_4$ is obtained by comparing the estimates that were made in the benchmarking sessions in *P0* and *P1*. Fig. 6 shows that for project MT, the estimates tended to be higher in *P1* than in *P0*. For project RCN, there was no apparent difference.

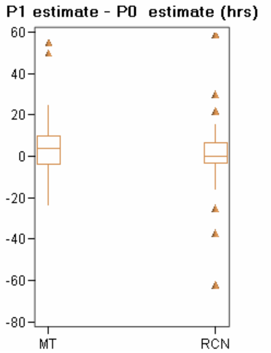


Fig. 6. Differences in estimates

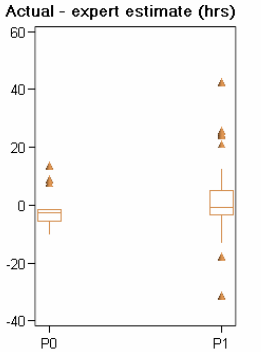


Fig. 7. RCN: Estimates subtracted from actual effort

A two-sided sign determines whether the differences are positive or negative in more cases than could be expected by pure chance. For project MT, the low p-value shows that estimates in *P1* are systematically higher than estimates in *P0*. For project RCN, the high p-value means that estimates in *P1* were not systematically different from in *P0*.

A change in estimation accuracy constitutes a threat to the validity of $ICPR_4$. For example, if developers tended to underestimate changes in *P0*, experience may have taught them to provide more relaxed estimates in *P1*. Because this would apply to real change tasks as well, we evaluated this threat by comparing estimation accuracy for real changes between the periods. The required data for this computation (developers' estimates and actual change effort) was only available for RCN. Fig. 7 shows a difference in estimation bias between the periods (Wilcoxon rank-sum test, p=0.086).

Changes tended to be overestimated in *P0* and underestimated in *P1*. Hence, the developers became more optimistic, indicating that $ICPR_4$ can be biased towards a higher value. This agrees with the results for the other indicators.

In summary, the benchmarking sessions supported the results from data on real change tasks. An additional result from the benchmarking session was that uncertainty estimates consistently increased between the periods in both projects. The developers explained this result by claiming they were more realistic in their assessments of uncertainty.

5 Discussion

The described approach to measuring productivity of software processes has some notable features compared with earlier work in this area. First, rather than searching for generally valid indicators of productivity, we believe it is more realistic to devise such indicators within more limited scopes. The targets for the proposed indicators are situations of software evolution where comparable change tasks were performed during two time intervals. Second, rather than claiming general validity, we believe it is more prudent to integrate validation procedures with the indicators. Third, our indicators are flexible within the defined scope, in that the structure of the underlying change effort models can vary in different contexts.

In a given project context, it may not be obvious which indicator will work best. Our experience is that additional insight was gained about the projects from using and assessing several indicators. The three first indicators require that data on change effort from individual change tasks is available. The advantage of $ICPR_1$ is that data on change effort is the *only* requirement for data collection. The caveat is that additional quantitative data is needed to assess the validity of the indicator. If this data is not available, a development organization may choose to be more pragmatic, and make qualitative judgments about potential differences in the properties of change tasks between the periods.

$ICPR_2$ and $ICPR_3$ require projects to collect data about factors that affect change effort, and that statistical models of change effort are established. To do this, it is essential to track relationships between change requests and code changes committed to the version control system. Furthermore, the models should be based on variables that it is meaningful to control for in the analysis. Although we controlled for differences in *developer experience* in one of our models, a given project could choose to not control for changes to such an attribute. Hence, the interpretation of the indicators is dependent on the actual variables used in the models. Whenever possible, variables that characterize the change request at the level of its requirements should be included, rather than considering code-oriented variables only.

One advantage of $ICPR_3$ is that any type of prediction framework can be used to establish the initial model. For example, data mining techniques such as decision trees or neural networks might be just as appropriate as multiple regression. Once the model is established, spreadsheets can be used to generate the estimates, construct the indicator and perform the associated statistical test.

$ICPR_2$ relies on a statistical regression model fitted on data from the periods under consideration. This approach better accounts for underlying changes in the cost drivers between the periods, than does $ICPR_3$. In organizations with a homogenous process and a large amount of change data, the methodology developed by Graves and Mockus could be used to construct the regression model [17]. With their approach, data on development effort need only be available on a more aggregated level (e.g., monthly), and relationships between change requests and code commits need not be explicitly tracked.

$ICPR_4$ most closely approximates the hypothetical measure of comparing change effort for identical change tasks. However, it can be difficult to design benchmarking tasks that resemble real change tasks, and to evaluate whether changes in estimation accuracy have affected the results. If the benchmarking sessions are organized frequently, developers' recollection of earlier estimates would constitute a validity threat.

As part of our analysis, we developed a collection of scripts to retrieve data, construct basic measures and indicators, and produce data and graphics for the evaluation. This means that it is straightforward and inexpensive to continue to use the indicators in the studied projects. It is conceptually straightforward to streamline the scripts so that they can be used with other data sources and statistical packages.

6 Conclusions

We conducted a field study in two software organizations to measure productivity changes between two time periods, using a new method that assumed that productivity during software evolution is closely related to the effort required to complete change tasks. Three of the indicators used the same data from real change tasks, but different methods to control for differences in the properties of the change tasks. The fourth indicator compared estimated change effort for a set of benchmarking tasks designed to be representative of real change tasks.

The indicators suggested that productivity trends had opposite directions in the two projects. It is interesting that these findings are consistent with major changes and events in the two projects. Instead of claiming general validity of the indicators, evaluation procedures for specific data sets are provided.

The paper makes a contribution towards the longer term goal of using methods and automated tools to assess trends in productivity during software evolution. We believe such methods and tools are important for software projects to assess and optimize development practices.

Acknowledgements. We thank Esito AS and KnowIT Objectnet for providing us with high quality empirical data, and the Simula School of Research and Innovation for funding the research.

References

1. Eick, S.G., Graves, T.L., Karr, A.F., Marron, J.S., Mockus, A.: Does Code Decay? Assessing the Evidence from Change Management Data. IEEE Transactions on Software Engineering 27(1), 1–12 (2001)
2. DeMarco, T., Lister, T.: Human Capital in Peopleware. Productive Projects and Teams, pp. 202–208. Dorset House Publishing (1999)
3. Mens, T., Tourwé, T.: A Survey of Software Refactoring. IEEE Transactions on Software Engineering 30(2), 126–139 (2004)
4. Dybå, T., Arisholm, E., Sjøberg, D.I.K., Hannay, J.E., Shull, F.: Are Two Heads Better Than One? On the Effectiveness of Pair Programming. IEEE Software 24(6), 12–15 (2007)
5. Tonkay, G.L.: Productivity in Encyclopedia of Science & Technology. McGraw-Hill, New York (2008)
6. Fenton, N.E., Pfleeger, S.L.: Measuring Productivity in Software Metrics, a Rigorous & Practical Approach, pp. 412–425 (1997)
7. Ramil, J.F., Lehman, M.M.: Cost Estimation and Evolvability Monitoring for Software Evolution Processes. In: Proceedings of the Workshop on Empirical Studies of Software Maintenance (2000)
8. Abran, A., Maya, M.: A Sizing Measure for Adaptive Maintenance Work Products. In: Proceedings of the International Conference on Software Maintenance, pp. 286–294 (1995)
9. Albrecht, A.J., Gaffney Jr., J.E.: Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. IEEE Transactions on Software Engineering 9(6), 639–648 (1983)
10. Maya, M., Abran, A., Bourque, P.: Measuring the Size of Small Functional Enhancements to Software. In: Proceedings of the 6th International Workshop on Software Metrics (1996)

11. DeMarco, T.: An Algorithm for Sizing Software Products. ACM SIGMETRICS Performance Evaluation Review 12(2), 13–22 (1984)
12. Ramil, J.F., Lehman, M.M.: Defining and Applying Metrics in the Context of Continuing Software Evolution. In: Proceedings of the Software Metrics Symposium, pp. 199–209 (2001)
13. Abran, A., Hguyenkim, H.: Measurement of the Maintenance Process from a Demand-Based Perspective. Journal of Software Maintenance: Research and Practice 5(2), 63–90 (1993)
14. Rombach, H.D., Ulery, B.T., Valett, J.D.: Toward Full Life Cycle Control: Adding Maintenance Measurement to the Sel. Journal of Systems and Software 18(2), 125–138 (1992)
15. Stark, G.E.: Measurements for Managing Software Maintenance. In: Proceedings of the 1996 International Conference on Software Maintenance, pp. 152–161 (1996)
16. Arisholm, E., Sjøberg, D.I.K.: Towards a Framework for Empirical Assessment of Changeability Decay. Journal of Systems and Software 53(1), 3–14 (2000)
17. Graves, T.L., Mockus, A.: Inferring Change Effort from Configuration Management Databases. In: Proceedings of the 5th International Symposium on Software Metrics, pp. 267–273 (1998)
18. Kitchenham, B., Mendes, E.: Software Productivity Measurement Using Multiple Size Measures. IEEE Transactions on Software Engineering 30(12), 1023–1035 (2004)
19. Schwaber, K.: Scrum Development Process. In: Proceedings of the 10th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications, pp. 117–134 (1995)
20. Benestad, H.C., Anda, B., Arisholm, E.: An Investigation of Change Effort in Two Evolving Software Systems. Technical report 01/2009, Simula Research Laboratory (2009)
21. Grimstad, S., Jørgensen, M.: Inconsistency of Expert Judgment-Based Estimates of Software Development Effort. Journal of Systems and Software 80(11), 1770–1777 (2007)

Design Pattern Detection in Java Systems: A Dynamic Analysis Based Approach

Francesca Arcelli[1], Fabrizio Perin[2], Claudia Raibulet[1], and Stefano Ravani[1]

[1] Università degli Studi di Milano-Bicocca, Dipartimento di Informatica Sistemistica e
Comunicazione, Viale Sarca 336, Edificio U14, 20126, Milan, Italy
{arcelli,raibulet}@disco.unimib.it,stefano.ravani@gmail.com
http://essere.disco.unimib.it

[2] University of Berne, Institute of Computer Science and Applied Mathematics
Software Composition Group, Neubrűckstrasse 10, Bern, Switzerland
perin@iam.unibe.ch
http://scg.unibe.ch

Abstract. In the context of reverse engineering, the recognition of design patterns provides additional information related to the rationale behind the design. This paper presents our approach to the recognition of the behavioral design patterns based on dynamic analysis of Java software systems. The idea behind our solution is to identify a set of rules capturing information necessary to identify a design pattern instance. Rules are characterized by weights indicating their importance in the detection of a specific design pattern. The core behavior of each design pattern may be described through a subset of these rules forming a macrorule. Macrorules define the main traits of a pattern. JADEPT (JAva DEsign Pattern deTector) is our software for design pattern identification based on this idea. It captures static and dynamic aspects through a dynamic analysis of the software by exploiting the JPDA (Java Platform Debugger Architecture). The extracted information is stored in a database. Queries to the database implement the rules defined to recognize the design patterns. The tool has been validated with positive results on different implementations of design patterns and on systems such as JADEPT itself.

Keywords: Reverse engineering, Design pattern detection, Rules, Dynamic analysis, Java.

1 Introduction

The usefulness of design patterns in forward engineering is well-known. They guarantee the creation of transparent structures which allow software to be easily understood, extended, and reused. The description of design patterns [11] provides information about the structure, the participant's roles, the interaction between participants and, above all, the intent for which they should be used.

In the context of reverse engineering, information related to the presence of a pattern is useful to understand not only the code, but to realize also the concepts behind its design. This has a significant implication for further improvement or adaptive changes on a software system. Implicitly, it leads to an enhancement of the life cycle

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 163–179, 2010.

with lower maintenance costs. For the design pattern detection, it is possible to use different approaches both for the identification logic (e.g., searching for subcomponents of design patterns, identifying the entire structure of a design pattern at once) and for the information extraction method (e.g., static, dynamic, or both). Static analysis [18, 26] consists in the analysis of data gathered directly from source code or, if possible, from compiled code [7]. Dynamic analysis [15, 30] deals with data obtained during the execution of a system, gathered by means of third party applications as debuggers or monitoring interfaces. Hybrid approaches identify statically the main candidates to be considered as implementations of design patterns and they exploit dynamic analysis to verify their actual existence [17]. Alternative solutions identify the structural characteristics of design patterns through the static analysis and the behavioral aspects through dynamic analysis [14, 29].

One of the advantages of using static analysis is the complete coverage of the software under examination. This is not always achieved through dynamic analysis. On the other hand, by exploiting static analysis, it is not possible to determine properly the behavior of a system. One of the advantages of using dynamic analysis is the capability to monitor each of the functionalities of a software independently of the others. In this way it is possible to consider a smaller part of code to increase precision and limit false positive or false negative results.

Problems raised by the identification of design patterns are related not only to the search aspect, but also to design and development choices. There are at least three important decisions that should be taken when developing a pattern detection tool. These decisions may influence significantly the final results. The first issue regards the evaluation of how to extract the interesting data from the examined software, including the type of the analysis to be performed. The second issue considers the data structure in which to store the gathered information: it may not model in a proper way the aspects of the software under investigation. The most important risk is related to the loss of knowledge at the data or the semantic level: this would generate inferences about something that is no more the analyzed software, but an incorrect abstraction of it. The third one highlights the importance to find a way to process the extracted data and to identify design pattern instances. Independently of the adopted data structure for the extracted information (e.g., a text file, XML, database), the following three issues should be considered: memory occupation, processing rate and, most important, the effective recognition process of design patterns with a minimum rate of false positives and false negatives. While the first two issues could be solved through an upgrade of the machine on which elaboration is performed, the last is strictly related to the efficiency of the recognition logic applied for design pattern detection due to the significant number of possible implementation variants.

In this paper, we present a new approach based on dynamic analysis to detect behavioral design patterns. We define a set of rules describing the properties of each design pattern. Properties may be either structural or behavioral and may define relationships between classes or families of classes. We define a family of classes as a group of classes implementing the same interface or extending a common class. Weights have been associated to rules indicating how much a rule is able to describe a specific property of a given design pattern. Rules have been written after we have deeply studied the books on design patterns of [11] and [8], evaluated different implementations of patterns, and implemented ourselves various variants of patterns.

JADEPT (JAva Design Pattern deTector) is the software prototype we have developed for design pattern detection based on these rules. An early version of JADEPT has been presented in [3, 4]. JADEPT collects structural and behavioral information through dynamic analysis of Java software by exploiting JPDA (Java Platform Debugger Architecture). Nevertheless part of the extracted information can be obtained by a static analysis of the software, JADEPT extracts all the information during the execution of the software adopting an approach based exclusively on dynamic analysis. The extracted information is stored in a database. The advantages of keeping information stored in database are: (1) the possibility to perform statistics and (2) the possibility to memorize information about various executions of the same software. A rule may be implemented by one or more queries. The presence of a design pattern is verified through the validation of its associated rules.

There are various approaches that aim to detect design patterns based on a static analysis of source code such as: FUJABA RE [18], [19], SPQR [25], [26], PINOT [24], PTIDEJ [13] or MAISA [28]. They adopt different detection techniques such as for example the search for information defining an entire pattern [13], the search for sub-elements of patterns which can be combined to build patterns [2] or the evaluation of the similarity of the code structure with higher-level models (e.g., UML diagrams [5] or graph representations [27]). Other approaches exploit both static and dynamic analysis as in [1], [6], [10], [14], [22] or only dynamic analysis as in [23], [29]. There are also several tools performing dynamic analysis of Java applications [9], [12], [31], but their main objective is not to identify design patterns. For example, Caffeine [12] is a dynamic analyzer for Java code which may be used also to support design patterns detection. The comparison of the different approaches exploiting dynamic analysis or both static and dynamic analysis is difficult to achieve since a standard benchmark for comparing design pattern detection tools is not yet available.

The rest of the paper is organized as follows. Section 2 describes our approach to design pattern detection based on rules and macrorules. Section 3 presents the rules and macrorules for three of the behavioral design patterns. The software implementing our approach is introduced in Section 4. Section 5 describes several aspects concerning the validation of JADEPT. Conclusions and further work are dealt with in Section 6.

2 A Rule Based Approach for Design Pattern Detection

We aim to develop a new approach for design pattern detection exploiting dynamic analysis. This kind of analysis allows the monitoring of the Java software at runtime, thus it is strictly related to the behavior of the system under analysis. A set of rules capturing the dynamic properties of design patterns and the interactions among classes and/or interfaces of design patterns are necessary for the detection of patterns through dynamic analysis. Our identification rules consider those static aspects which provide information further exploited in dynamic rules. For example, to check the existence of a particular behavior, it is necessary to verify in the software under analysis the presence of a method having a specific signature. We consider behavioral design patterns because they are particularly appropriate for dynamic analysis. In fact, their traces may be better revealed at runtime by analyzing all the dynamic aspects including: object instantiation, accessed/modified fields, method calls flows.

In the first step of our work, the identification rules have been written using the natural language. This approach avoids introducing constraints regarding the implementation of rules. In JADEPT, rules are translated into queries, but they can be used also outside the context of our tool and hence, represented through a different paradigm (e.g., graphs).

In the second step weights have been added to the rules. Weights denote the importance of a rule in the pattern detection process. Weights' range is 1 to 5. These values are used to compute the probability score indicating the probability of the presence of a pattern instance. A low weight value denotes a rule that describes a generic characteristic of a pattern as the existence of a reference or a method with a specific signature. A high weight value denotes a rule that describes a specific characteristic of a pattern as a particular method call chain linking two class families.

Even if each behavioral design pattern has its own particular properties, an absolute scale for the weights value has been defined. Rules whose weight value is equal to 1 or 2 describe structural and generic aspects of code (e.g., abstract class inheritance, interface implementation or the presence of particular class fields). Rules whose weight value is equal to 3 or higher, describe a specific static or dynamic property of a pattern. For example, the fifth rule of Chain of Responsibility in Table 1, specifics that each call to the *handle()* method has always the same caller-callee objects pair. This is the way objects are linked in the chain. A weight whose value is equal to 5 describes a native implementation of the design pattern we are considering (see Table 3). The weights of rules are used to determine the probability of the pattern presence in the examined code.

The next step regarded the definition of the relationship between rules [21]. There are two types of relationships. The first one is *logical*: if the check of a rule does not have a positive value, it does not make sense to proof the rules related to it. For example, the fifth rule of Chain of Responsibility in Table 1 cannot be proved if the fourth rule has not been proved first. The second one is *informative*: if a rule depends on another one, and the latter is verified by the software detector, its weight increases. The second type of relationship determines those rules which are stronger for the identification of design patterns.

Finally, we have introduced macrorules. A macrorule is a set of rules which describes a specific behavior of a pattern. If the rules that compose a macrorule are verified, the core behavior of a pattern has been detected so the final probability value increases. The value added to the probability is different for each pattern because the number of rules which belong to a macrorule varies from one macrorule to another.

3 Detection Rules for Design Patterns

We provide the detection rules and macrorules for three examples of behavioral design patterns: Chain of Responsibility, Observer, and Visitor. For further details related to the detection of the other behavioral design patterns refer to [20], [21].

3.1 Detection Rules for the Chain of Responsibility Design Pattern

The rules we have defined for the Chain of Responsibility are shown in Table 1. In the first column we assign a unique identifier to the rule in the context of a specific design pattern. The second column contains a textual description of each rule. In the third column are indicated the weights associated to each rule. A question mark after a weight value indicates a variable weight. For example, the rule 6 has a variable weight because of its relation with rule 4 and 5. If these two rules are verified then the weight of rule 6 is increased by one, hence associating a higher probability to the pattern instance recognition.

The fourth column indicates the type of information needed to verify a rule. If a rule describes a static property, which can be verified through an analysis of static information, then the value in this column is S (*static*). If a rule describes a dynamic property, which can be verified through an analysis of dynamic information, then the value in this column is D (*dynamic*). In the case we have to verify a property by performing analysis of static and dynamic information, then the value specified is S-D (*static* and *dynamic*). However, in JADEPT both static and dynamic information are extracted through a dynamic analysis of the software under inspection.

A relationship among two or more rules is indicated in the fifth column of Table 1.

For this pattern it is important to identify clues which capture its chain structure and behavior.

Rules 1 and 2 require that chain classes must implement a common interface or extend a common class. For the Chain of Responsibility pattern the common class/interface must declare a method for sending a request to its successor in the chain. In the following we call this method *handle()*.

Rule 3 claims for the presence in each chain class of a field whose type is the implemented interface or the extended class. At runtime, this reference indicates the successor of an instance in the chain, and it is used to call the *handle()* method. This reference is assigned to the successor during execution and it is used to call the *handle()* method.

Rule 4 is verified if the interface or considered class declares a method which can be a *handle()* method. Rules 3 and 4 are preconditions for the fifth rule. These rules define the Chain of Responsibility peculiar behavior.

Rule 5 specifies that each object in the chain must always be called by the same object, which is its predecessor if objects are unchanged during execution. In fact, each time a request management is needed, if the chain elements have not been modified, the chain is preserved

Eventually, rule 6 checks if the name of the common interface or common class contains the *chain* or *handle* string.

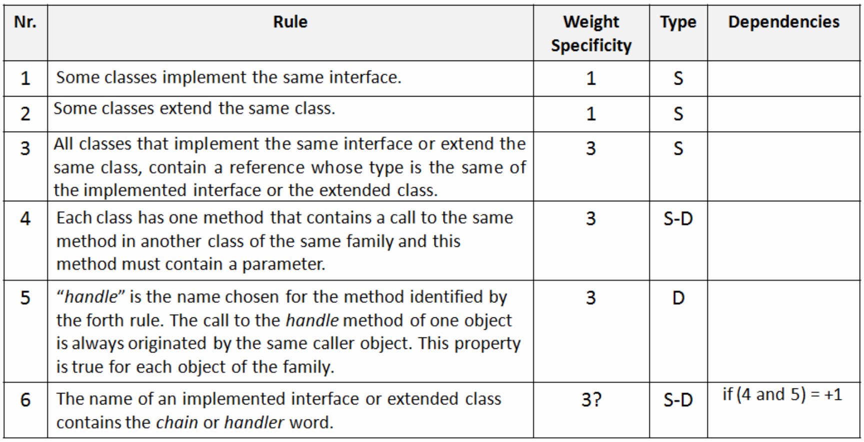
A logical dependency is between rule 4 and 5. Rule 5 cannot be proved if rule 4 is not previously verified.

The informative dependency we have defined for this pattern involves the 4th, 5th and 6th rules. Rule 6 can increment by one its weight if rules 4 and 5 are verified.

The macrorule for this pattern is called *sequential redirection* (see Table 2).

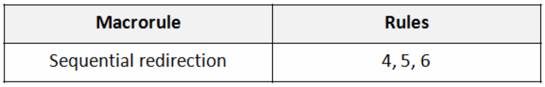
The macrorule includes the rules that describe the core of the Chain of Responsibly pattern. If rules 4 and 5 are checked the macrorule is verified. This means that code satisfies the basic criteria for the recognition of this pattern. If the macrorule is proved, the total probability score increases. This score indicates the confidence of the presence of the Chain of Responsibility in the examined software.

Table 1. Detection Rules for the Chain of Responsibility Design Pattern



Nr.	Rule	Weight Specificity	Type	Dependencies
1	Some classes implement the same interface.	1	S	
2	Some classes extend the same class.	1	S	
3	All classes that implement the same interface or extend the same class, contain a reference whose type is the same of the implemented interface or the extended class.	3	S	
4	Each class has one method that contains a call to the same method in another class of the same family and this method must contain a parameter.	3	S-D	
5	"*handle*" is the name chosen for the method identified by the forth rule. The call to the *handle* method of one object is always originated by the same caller object. This property is true for each object of the family.	3	D	
6	The name of an implemented interface or extended class contains the *chain* or *handler* word.	3?	S-D	if (4 and 5) = +1

Table 2. The Macrorule for the Chain of Responsibility Design Pattern



Macrorule	Rules
Sequential redirection	4, 5, 6

3.2 Detection Rules for the Observer Design Pattern

The rules we have defined for the Observer design pattern are shown in Table 3.

Rule 1 requires that both the Observer and Subject should implement an interface. In this pattern the presence of an interface is fundamental in order to allow the increment of the number of the Observers and Subjects.

Rule 2 specifies that a Concrete Subject should have a field to store all Observers interested in its changes.

Rule 3 is verified if an Observer has a reference to the Subject whose changes are interesting for it. This reference is used to get the new states of the Subject when modifications occur. In the native interface this field is not required because the Subject passes itself as a parameter to the update method of the Observer [16].

Rules 4 and 5 claim for the presence of two methods of the Subject that have a parameter of the same type of the Observers. These rules have the purpose to identify the Subject's attach and detach methods used by Observers to register or unregister their interest in a Subject's changes.

Rule 6 highlights the interaction between the Subject's notification method and the Observer's update method.

Rule 7 specifies that there should be an interaction between the update method of the Observer and a getter method of the Subject. If rules 4 and 6 are not verified, then it is impossible to identify the complete interaction between an Observer and a Subject. In Fig. 1 it is shown a sample of a complete interaction between a Subject and the Observers after a Subject's modifications (as defined by [11]).

Table 3. Detection Rules for the Observer Design Pattern

Nr.	Rule	Weight Specificity	Type	Dependencies
1	Some classes implement the same interface (two class families).	1	S	
2	The observed object (the Subject) should store a list of all the Observer objects interested in it.	2	S-D	
3	The Observer object should have a field whose type is the same of the Subject. This field is initialized by the constructor with a parameter whose type is the same of the Subject.	2	S-D	
4	The Subject object should have a method (attach) with a parameter whose type is the same of an Observer object. This method is invoked by an Observer or by another class.	2	S-D	
5	The Subject should have another method (detach) with a parameter whose type is the same of an Observer. This method is used from an Observer or from another class.	2?	S-D	If 2 =+1
6	There is a sequence of invocations between a method of the Subject (the candidate *notify()* method) and a method of the Observer objects which are stored in a field of the Subject objects. The invoked Observer method (the candidate *update()* method) is the same for all the Observer objects.	3?	S-D	if (2 and 4) = +1
7	Prerequisites: rules 4 and 5. Invocation in an Observer method of a Subject method (a getter) with the return value not *void*.	3?	D	if (2 and 4) = +1
8	A class implementing the native interface *Observable* is a Subject.	5	S	
9	A class implementing the native interface *Observer* is an Observer.	5	S	

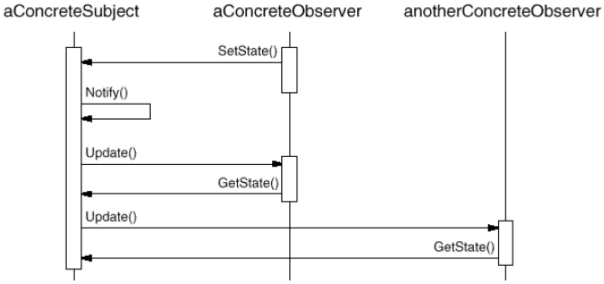


Fig. 1. Subject-Observer interaction

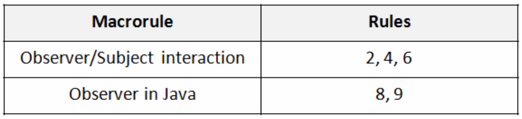
The last two rules identify an implementation of the Observer pattern using the native interfaces defined in Java. Rule 8 claims the presence of one or more classes that implement the native Observable interface. Those classes are considered as Subjects. Rule 9 requires that one or more classes implement the native interface Observer. These classes are considered as Observers.

The informative dependency we have defined for this pattern involves the 5th, 6th and 7th rules. Rule 5 gets an increment if rule 2 is verified. It means that a collection of references of objects of the same type has been identified. Rules 6 and 7 increment their weights if rules 2 and 4 are verified. It means that a method which populates the collection that verifies the rule 2 has been also exposed.

In Table 4 are summarized the two macrorules for the Observer design pattern.

The first macrorule is called *Observer/Subject interaction*. It describes the main characteristic of the Observer pattern that is the interaction between an Observer and a Subject when the latter modifies its state. The second macrorule is called *Observer in Java*. It is verified if an implementation of the Observer pattern using the Java native interfaces is detected [16].

Table 4. The Macrorule for the Observer Design Pattern



Macrorule	Rules
Observer/Subject interaction	2, 4, 6
Observer in Java	8, 9

3.3 Detection Rules for the Visitor Design Pattern

The rules we have defined for the Visitor design pattern are shown in Table 5.

Rule 1 requires two families of classes, each implementing an interface. Using an interface instead of an abstract class makes the structure easier to modify by adding new elements. However, an implementation of the pattern using abstract classes is not restrictive to validate the other rules.

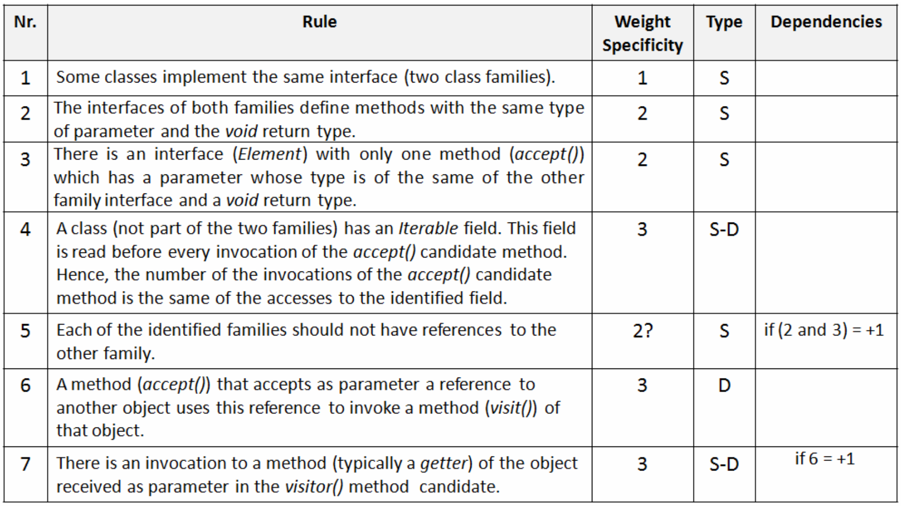
Rule 2 and 3 are verified if it is possible to identify a *visit()* candidate method and an *accept()* candidate method. These methods are searched in the interfaces (or abstract classes) specified in the first rule.

Rule 4 has the purpose to identify the Client, which is the owner of the objects that have to be investigated by the Visitor. For the other patterns the Client has not been taken into account (playing a secondary role), but in this case it has been necessary to consider it. During the analysis of different pattern implementations it became obvious that using the native interface *Iterable* or implementing the pattern manually, a class out of both families described in rule 1 was the owner of the structure that has to be visited. The second part of the rule aims to identify a relation between accesses to the structure that is visited and the method that implements the iteration.

Rule 5 tries to distinguish the Visitor from the Observer pattern. The problem is that the interaction between the *accept*() and *visit*() in the Visitor and the interaction between *notify*() and *update*() in the Observer could be misunderstood. The difference is that in the Observer, the Subject's and the Observer's instances are usually assigned in the constructor. In the Visitor implementation instead the relation between the structure Visitor and Element is build on the fly through methods' parameters.

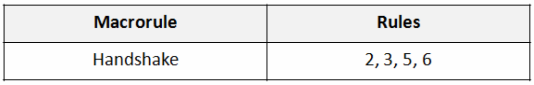
Rule 6 highlights the core behavior of the Visitor by defining which sequence of invocations should be verified at run-time. The rule claims that the *accept()* candidate method should be called after an invocation of the *visitor()* candidate method and that the *visit()* method has to receive as parameter the reference to the object containing Rule 7 describes the interaction which is verified after the typical scenario described the element which should be analyzed. In Fig. 2 it is shown an example of the interaction among the Visitor's participants (as defined by [11]). by rule 6. This rule is verified if after the interaction in rule 6 there is an invocation to a method (typically a getter) defined in the *Element* participant.

Table 5. Detection Rules for the Visitor Design Pattern



Nr.	Rule	Weight Specificity	Type	Dependencies
1	Some classes implement the same interface (two class families).	1	S	
2	The interfaces of both families define methods with the same type of parameter and the *void* return type.	2	S	
3	There is an interface (*Element*) with only one method (*accept()*) which has a parameter whose type is of the same of the other family interface and a *void* return type.	2	S	
4	A class (not part of the two families) has an *Iterable* field. This field is read before every invocation of the *accept()* candidate method. Hence, the number of the invocations of the *accept()* candidate method is the same of the accesses to the identified field.	3	S-D	
5	Each of the identified families should not have references to the other family.	2?	S	if (2 and 3) = +1
6	A method (*accept()*) that accepts as parameter a reference to another object uses this reference to invoke a method (*visit()*) of that object.	3	D	
7	There is an invocation to a method (typically a *getter*) of the object received as parameter in the *visitor()* method candidate.	3	S-D	if 6 = +1

Table 6. The Macrorule for the Observer Design Pattern



Macrorule	Rules
Handshake	2, 3, 5, 6

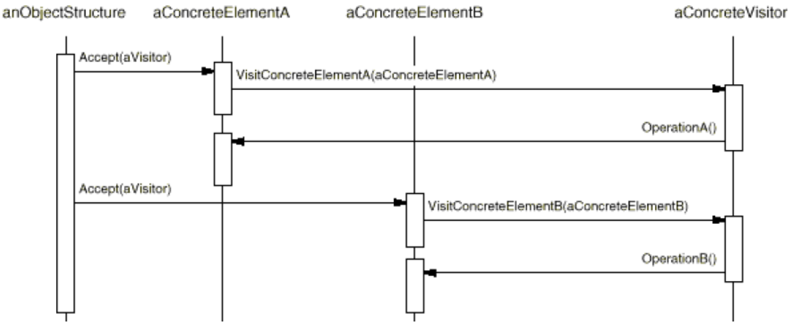


Fig. 2. The Visitor's Participants Interaction

In Table 6 it is described the macrorule for the Visitor design pattern.

The macrorule is called *handshake* and it highlights which rules are involved in the description of the behavior of this design pattern. In particular rules 2 and 3 are validated if it is possible to identify an *accept()* and a *visitor()* method. Rules 5 and 6 describe the interaction between those methods.

4 A Java Design Pattern Detector

JAva DEsign Pattern deTector is a Java application composed of four main modules: Graphic User Interface (GUI), Launcher and Capture Module (LCM), Design Pattern Detector Module (DPDM) and JDEC Interface Module (see Fig. 3).

JADEPT's GUI allows users: (1) to set up a JADEPT XML configuration file, (2) to launch the software to be monitored, (3) to start the analysis on the stored information and (4) to create the JDEC database.

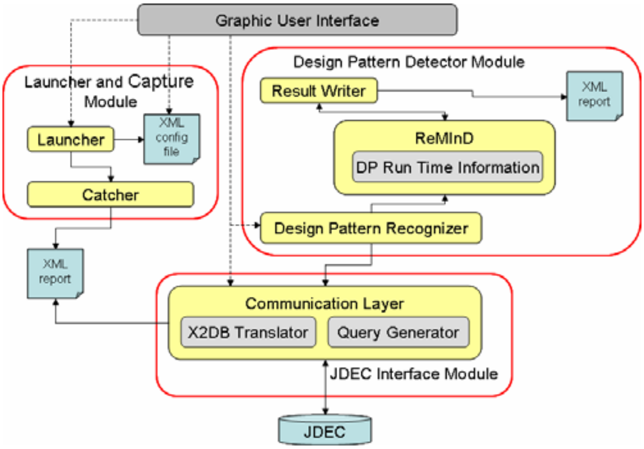


Fig. 3. JADEPT Architecture

The Capture and Launcher Module is composed of (1) the Launcher, which starts the execution of the software under analysis and the execution of the Catcher Module, and (2) the Catcher, which captures events occurred in the JVM created by the Launcher for the analyzed software. Events regard classes and interfaces loading, method calls, field accesses and modifications. Through these events JADEPT extracts various types of information exploited in the detection process.

During the analyzed software execution the Catcher Module writes the XML Report File containing all the collected information and, at the end of analyzed software execution it invokes the Communication Layer insertion method. Thus, the XML Report File is inserted in the JDEC database. In this way, information is available to the Design Pattern Detector Module (DPDM).

The Communication Layer represents the link between JDEC and the other software modules. Its functions are provided by XML2DBTranslator and Query Generator. XML2DBTranslator interprets the XML Report File created by the Catcher Module and creates inserting queries to fill JDEC. The Query Generator is used during the analysis phase to create the appropriate queries.

The Design Pattern Detector Module is composed of: the Design Pattern Recognizer, the Result and Metric Information Dispenser (ReMInD) and the Result Writer. The Design Pattern Recognizer contains the classes used in the detection process. Each class defines a thread representing a specific pattern role and each thread performs the analysis on a class family. A thread checks the rule on the family assigned to it and writes the results on an object metaphorically called *whiteboard* [20]. The ReMInD module provides objects which support the analysis threads. Each object is a whiteboard used by a thread to store information about temporary results obtained during the analysis. The Result Writer receives results coming from the Design Pattern Recognizer and it disposes them into an output file, called Result File. The last is divided into various sections each of them referring to a different pattern. For more details on JADEPT see [3, 4, 20].

5 Experimental Results with JADEPT

JADEPT has been validated using different implementation samples of design patterns more or less closer to their GoF's definitions [11]. The results of the analysis on different implementations of design patterns are shown in Tables 7, 8 and 9 and are related to the detection of three of the behavioral design patterns: Chain of Responsibility, Observer and Visitor. Table 10 shows the results of JADEPT that analyzes itself.

The first column of each table contains the identification name for the implementations considered. The remaining columns show the results provided by the Chain of Responsibility, Observer and Visitor detectors. The `-' symbol means that JADEPT has not detected any instance for a given design pattern. The `X' symbol indicates that the considered sample does not provide any implementation of a specific pattern.

Table 7 illustrates the results obtained during the detection of Chain of Responsibility. The second column indicates the results obtained through the Chain of Responsibility detector, while column three and four the results obtained through the Observer and Visitor detectors. The last two detectors have been used to verify if they provide false positives. The same approach has been applied in Table 8 and Table 9.

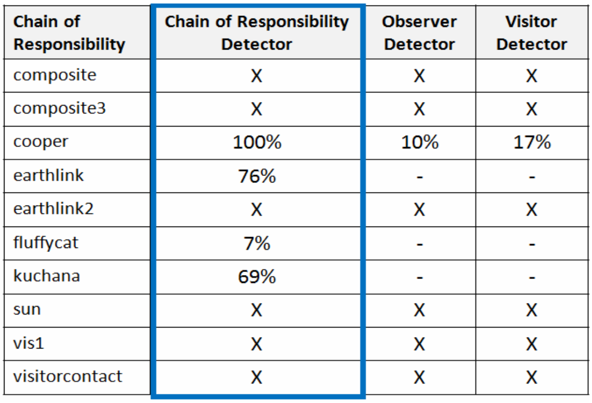
JADEPT recognizes the Chain of Responsibility pattern in three implementations with reliable values. The Chain implementation in fluffycat is detected as a false negative because JADEPT is not able to find a good *handle()* candidate in this pattern instance. This argument indicates the request that should be managed by one of the classes which implements the interface. Moreover, each class implementing the interface declares a field whose type is the type of the common interface. The successor element in the chain is assigned to this field during execution.

Fig. 4 shows the class diagram related to the implementation of the Chain of Responsibility in the *fluffycat* example. According to the GoF's definition, this pattern should define a common interface (e.g., called *Chain*) which is implemented further by two or more classes. The interface defines a method (e.g., called *sendToChain(String)*) which accepts only one argument.

The fluffycat implementation is not closed to the GOF's definition: it defines a common interface called *TopTitle*, but this interface declares methods which accept no arguments. One of the purposes of the Chain of Responsibility pattern is to build a structure which is able to handle requests generated by a sender. Hence, it is not possible that the *handle()* method accepts no parameters. Moreover, the three classes do not declare any field for a successor whose type is the interface type. *DvdCategoryClass* does not declare any field which indicates a reference to its successor. The *DvdSubCategory* class declares a field of *DvdSubSubCategory* type. The instances of these classes can be chained only in one way: the knowledge indicating which object must be used as a successor of another is built-in the classes and not in an external class which should define how the chain must be created. Hence, such an implementation is unacceptable and does not comply with the GoF's definition. This is reflected in the low values associated to the fluffycat implementation in Table 7.

The Observer and Visitor detectors obtain satisfying results. There are cases (e.g., earthlink, fluffycat and kuchana) in which detectors have not even start their analysis due to the absence of a common class or interface in these implementation instances. Moreover, the Chain structure is deeply different from the Observer and Visitor structures. It requires only one role, while the Observer and Visitor require two. This is the main reason why the two detectors cannot perform the analysis. In the Observer and Visitor implementations of the cooper sample it is revealed a Chain instance with a very low probability score, hence it cannot be even considered significant.

Table 7. Chain of Responsibility implementation analyzed by three design pattern detectors



Chain of Responsibility	Chain of Responsibility Detector	Observer Detector	Visitor Detector
composite	X	X	X
composite3	X	X	X
cooper	100%	10%	17%
earthlink	76%	-	-
earthlink2	X	X	X
fluffycat	7%	-	-
kuchana	69%	-	-
sun	X	X	X
vis1	X	X	X
visitorcontact	X	X	X

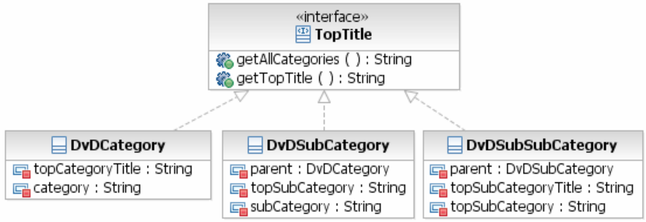
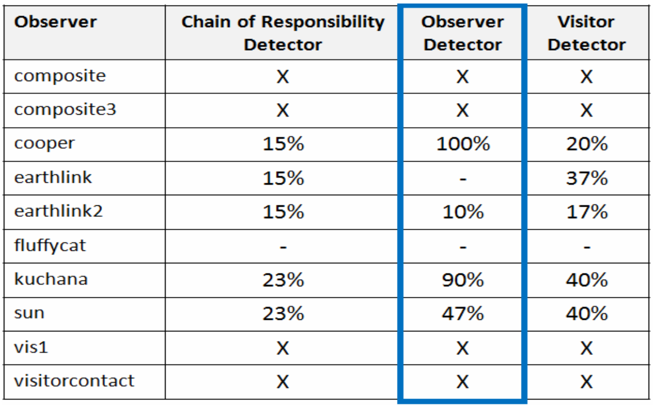


Fig. 4. Class Diagram for the Chain of Responsibility Pattern in fluffycat

The results of the detection of the Observer pattern are shown in Table 8. The Observer detector provides significant results for kuchana and cooper implementations. It obtains false negative values for earthlink2 and sun implementations and it does not provide any result for earthlink and fluffycat. The reason why the Observer detector cannot perform the analysis is related to the correctness of the implementations and to how JADEPT partitions the software system under examination to perform analysis. Even if Observer and Visitor are similar, the Visitor detector provides low score false positive results. The highest values were obtained for kuchana and sun and cooper implementations. It may be possible that the *notify()* and *update()* methods are considered by the static rules as *accept()* and *visit()* candidates. The fluffycat implementation cannot be analyzed due to the lack of the common classes/interfaces. The Chain of Responsibility detector provides very low probability scores, and also it cannot analyze the fluffycat implementation.

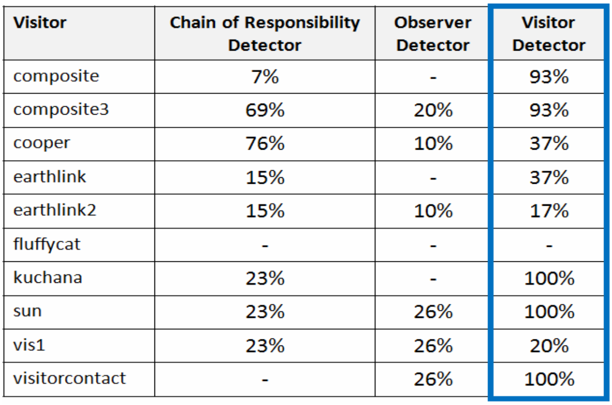
Table 9 shows the results obtained during the detection of the Visitor pattern. The Visitor detector provides satisfying results for five implementations. The Observer detector provides low false positive results, and it cannot analyze kuchana and composite implementations. The Chain of Responsibility detector obtains low false positive results, except for the composite3 and cooper implementations. In these cases, one method is wrongly considered as a *handle()* candidate. The Chain of Responsibility detector cannot perform the analysis on the visitorContact implementation. This result depends on the differences between the two pattern structures.

Table 8. Observer implementation analyzed by three design pattern detectors



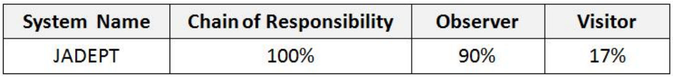
Observer	Chain of Responsibility Detector	Observer Detector	Visitor Detector
composite	X	X	X
composite3	X	X	X
cooper	15%	100%	20%
earthlink	15%	-	37%
earthlink2	15%	10%	17%
fluffycat	-	-	-
kuchana	23%	90%	40%
sun	23%	47%	40%
vis1	X	X	X
visitorcontact	X	X	X

Table 9. Visitor implementation analyzed by three design pattern detectors



Visitor	Chain of Responsibility Detector	Observer Detector	Visitor Detector
composite	7%	-	93%
composite3	69%	20%	93%
cooper	76%	10%	37%
earthlink	15%	-	37%
earthlink2	15%	10%	17%
fluffycat	-	-	-
kuchana	23%	-	100%
sun	23%	26%	100%
vis1	23%	26%	20%
visitorcontact	-	26%	100%

Table 10. JADEPT analyzed by JADEPT



System Name	Chain of Responsibility	Observer	Visitor
JADEPT	100%	90%	17%

Table 10 shows the results of JADEPT that analyzes itself. JADEPT is composed of 151 classes. The analysis reveals the presence of Chain of Responsibility and Observer, which are implemented in the code. There are no Visitor instances in JADEPT. This analysis was performed only to test if any false positives are revealed.

To summarize, there are two main reasons why JADEPT cannot perform analysis on some implementations. The first is related to the quality of implementations themselves because they are very different from the UML structure of patterns defined by GoF. For example, classes do not implement the same interface or extend the same class. We mean that such implementations cannot be retained as valid ones. Common interfaces and classes are used to easily extend software and their use is a principle of good programming as much as other design pattern features.

The second problem concerns the information partitioning technique of JADEPT. Our tool can work on families retrieved from the information collected in JDEC. Before starting the analysis, JADEPT identifies all the possible families and assigns to each family a specific role, according to the design pattern it is looking for. If the analyzed system is unstructured, meaning that common interfaces or classes are absent, JADEPT cannot build correctly the families and perform further analysis.

6 Concluding Remarks

As mentioned in the introduction, there are several main issues which should be addressed during the development of a design pattern detection tool. Considering these issues, our contribution includes:

- the definition of recognition rules able to capture static and dynamic properties;
- the use of dynamic analysis to extract all the information needed in the detection process;
- the specification of an entity-relationship schema to store and organize the extracted information from Java applications to be easily used for the recognition of design pattern instances.

The recognition rules regard in particular the dynamic nature of patterns. Rules focus on the behavior of the design patterns and not on their static aspects. Rules capturing static properties have been introduced because they express pre-conditions for the dynamic ones. Furthermore we have defined logical and informative dependencies among rules, established the importance of rules in the detection process through scores, and identified a group of rules characterizing the specific behavior of each pattern through macrorules.

Dynamic analysis may obviously provide significant information for design pattern recognition. Through dynamic analysis it is possible to observe objects, their creation and execution during their entire life-cycle and overcome part of the limitations of the static analysis (i.e., polymorfism) which may be determinant in pattern recognition. There are also two disadvantages of the dynamic approach. The first is related to the reduced performance of the analyzed application. To improve its performance we have used a filtering system to trace only the meaningful events. Nevertheless the execution time of the monitored applications is still longer than the ordinary execution time, especially for software having a Graphic User Interface. If the software under examination does not require user interaction, execution time should not be a critical factor. The second, concerns the code coverage problem. If the analyzed software needs a user interaction, it could be necessary a human-driven selection of code functions to reveal all possible behaviors. Thus, it is necessary to cover the entire code and test all code functions one by one.

We have validated our idea through the implementation of the JADEPT prototype. Modularity is one of the main characteristic of the JADEPT architectural model. Furthermore, the tool can be easily extended to other programming languages. It may use alternative ways to extract information or to perform analysis. It is possible also to exclude the database and to use another approach to detect design patterns due to the existence of the XML Report file. Or, the database model can be used in another design pattern detector or a software architecture reconstruction tool.

The decision to use a database to store the extracted information is due to two main reasons. The first is related to the large amount of information which should be extracted during software execution and which should be further considered to identify design patterns. The second is related to the time persistence of the extracted information, the comparison among two or more executions of the software code or among executions of different applications, and the statistics which may be done. The issues related to this second aspect are not implemented in the current version of our prototype but will be addressed in the future developments of JADEPT.

Further work will regard also the extension of JADEPT to the creational and structural design patterns, as well as to its validation on systems of larger dimensions.

References

1. Abd-El-Hafiz, S.k., Shawky, D.M., El-Sedeek, A.-L.: Recovery of Object-Oriented Design Patterns Using Static and Dynamic Analyses. International Journal of Computers and Applications (2008)
2. Arcelli, F., Masiero, S., Raibulet, C., Tisato, F.: A Comparison of Reverse Engineering Tools based on Design Pattern Decomposition. In: 2005 IEEE Australian Software Engineering Conference, pp. 262–269. IEEE Press, Los Alamitos (2005)
3. Arcelli, F., Perin, F., Raibulet, C., Ravani, S.: JADEPT: Behavioural Design Pattern Detection through Dynamic Analysis. In: 4th International Conference on Evaluation of Novel Approaches to Software Engineering, pp. 95–106. INSTICC Press (2009)
4. Arcelli, F., Perin, F., Raibulet, C., Ravani, S.: Behavioural Design Pattern Detection through Dynamic Analysis. 4th International Workshop on Program Comprehension through Dynamic Analysis, Technical report TUD-SERG-2008-036, 11–16 (2008)
5. Bergenti, F., Poggi, A.: Improving UML Designs Using Automatic Design Pattern Detection. In: 12th International Conference on Software Engineering and Knowledge Engineering, pp. 336–343 (2000)

6. Birkner, M.: Object-Oriented Design Pattern Detection Using Static and Dynamic Analysis of Java Software. Master Thesis, University of Applied Sciences Bonn-Rhein-Sieg, Sankt Augustin, Germany (2007)
7. Byte-Code Engineering Library (BCEL), `http://jakarta.apache.org/bcel`
8. Cooper, J.W.: The Design Pattern Java Companion. Addison-Wesley, Reading (1998)
9. Demeyer, S., Mens, K., Wuyts, R., Guéhéneuc, Y.-G., Zaidman, A., Walkinshaw, N., Aguiar, A., Ducasse, S.: Workshop on Object-Oriented Reengineering (2005)
10. Fan, L., Qing-shan, L., Yang, S., Ping, C.: Detection of Design Patterns by Combining Static and Dynamic Analyses. Journal of Shanghai University (English Edition) 11(2), 156–162 (2007)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: elements of reusable object-oriented software. Addison Wesley, Reading (1994)
12. Guéhéneuc, Y.-G., Douence, R., Jussien, N.: No Java without Caffeine. A Tool for Dynamic Analysis of Java Programs. In: 17th IEEE International Conference on Automated Software Engineering, pp. 117–126. IEEE Press, Los Alamitos (2002)
13. Guéhéneuc, Y.G.: PTIDEJ: Promoting Patterns with Patterns. In: 1st ECOOP Workshop on Building Systems using Patterns, pp. 1–9. Springer, Heidelberg (2005)
14. Heuzeroth, D., Holl, T., Löwe, W.: Combining Static and Dynamic Analysis to Detect Interaction Patterns. In: 6th World Conference on Integrated Design and Process Technology (2002)
15. Hu, L., Sartipi, K.: Dynamic Analysis and Design Pattern Detection in Java Programs. In: 20th International Conference on Software Engineering & Knowledge Engineering, pp. 842–846 (2008)
16. Java documentation,
`http://java.sun.com/j2se/1.4.2/docs/api/java/util/Observer.html`
17. Lee, H., Youn, H., Lee, E.: Automatic Detection of Design Pattern for Reverse Engineering. In: 5th ACIS International Conference on Software Engineering Research, Management & Applications, pp. 577–583 (2007)
18. Nickel, U., Niere, J., Zündorf, A.: The FUJABA Environment. In: 22nd International Conference on Software Engineering, pp. 742–745 (2000)
19. Niere, J., Schäfer, W., Wadsack, J.P., Wendehals, L., Welsh, J.: Towards Pattern-Based Design Recovery. In: 24th International Conference on Software Engineering, pp. 338–348 (2002)
20. Perin, F.: Dynamic analysis to detect the design patterns in Java: gathering information with JPDA. MSc Thesis, University of Milano-Bicocca, Milan (2007)
21. Ravani, S.: Dynamic analysis for Design Pattern detecting on Java code: information relationship modelling, MSc Thesis, University of Milano-Bicocca, Milan (2007)
22. Pettersson, N.: Measuring Precision for Static and Dynamic Design Pattern Recognition as a Function of Coverage. In: Workshop on Dynamic Analysis, ACM SIGSOFT Software Engineering Notes, vol. 30(4), pp. 1–7 (2005)
23. Shawky, D.M., Abd-El-Hafiz, S.K., El-Sedeek, A.-L.: A Dynamic Approach for the Identification of Object-oriented Design Patterns. In: IASTED Conference on Software Engineering, pp. 138–143 (2005)
24. Shi, N., Olsson, R.A.: Reverse Engineering of Design Patterns from Java Source Code. In: 21st Conference on Automated Software Engineering, pp. 123–134. IEEE Press, Los Alamitos (2006)

25. Smith, J.M.C., Stotts, D.: Elemental Design Patterns: A Formal Semantics for Composition of OO Software Architecture. In: 27th Annual NASA Goddard Software Engineering Workshop, pp. 183 (2002)
26. Smith, J.M.C., Stotts, D.: SPQR: Flexible Automated Design Pattern Extraction From Source Code. In: 2003 IEEE International Conference on Automated Software Engineering, pp. 215-224, IEEE Press (2003)
27. Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S.T.: Design Pattern Detection Using Similarity Scoring. IEEE Transactions on Software Engineering 32(11), 896–909 (2006)
28. Verkamo, A.I., Gustafsson, J., Nenonen, L., Paakki, J.: Design patterns in performance prediction. In: ACM 2nd Workshop on Software and Performance, pp. 143–144 (2000)
29. Wendehals, L.: Improving Design Pattern Instance Recognition by Dynamic Analysis. In: ICSE 2003 Workshop on Dynamic Analysis, pp. 29–32. IEEE Press, Los Alamitos (2003)
30. Wendehals, L., Orso, A.: Recognizing Behavioral Patterns at Runtime using Finite Automata. In: 4th International ICSE Workshop on Dynamic Analysis, pp. 33–39 (2006)
31. Zaidman, A., Hamou-Lhadj, A., Greevy, O.: Program Comprehension through Dynamic Analysis. In: 1st International Workshop on Program Comprehension through Dynamic Analysis, Technical report 2005-12 (2005)

Formalization of the UML Class Diagrams

Janis Osis and Uldis Donins

Department of Applied Computer Science, Institute of Applied Computer Systems
Riga Technical University, Meza iela 1/3, Riga, LV 1048, Latvia
{janis.osis,uldis.donins}@cs.rtu.lv

Abstract. In this paper a system static structure modeling formalization and formalization of static models based on topological functioning model (TFM) is proposed. TFM uses mathematical foundations that holistically represent complete functionality of the problem and application domains. By using TFM within software development process it is possible to do formal analysis of a business system and in a formal way to model the static structure of the system. After construction of the TFM of a system's functioning a problem domain object model is defined by performing transformation of defined TFM. By making further transformations of TFM and by using TFM within software development it is possible to introduce more formalism in the Unified Modeling Language (UML) diagrams and in their construction. In this paper we have introduced topology into the UML class diagrams.

1 Introduction

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components. [3] and [8]

Since the publication of first UML specification, researchers have been working and proposing approaches for the UML formalization. Researches on UML formalization are performed because the meaning of the language, which is mainly described in English, is too informal and unstructured to provide a foundation for developing formal analysis and development techniques, and because of the scope of the model, which is both complex and large [2]. Despite the fact that the latest UML specification [14] which is published by Object Management Group [4] is based on the metamodeling approach, the UML metamodel gives information about abstract syntax of UML but does not deal with semantics which is expressed in natural language.

After the publication of the first UML specification precise UML (pUML) group [13] was found with main goal to bring together international researchers and practitioners who share the aim of developing the UML as a precise modeling language. The aim of pUML group is to work firmly in the context of the existing UML semantics. As a formalization instrument they use several formal notations, for example, Object Constraint Language [12] or the formal language Z [10].

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 180–192, 2010.

There are also other researches on formalization of UML and class diagrams, for example, [11] in which mathematical expressions are used to describe formal semantics of the key elements of the UML static models (class diagrams).

All described researches are provided to formalize only the UML semantics but these approaches does not:

- provide a formal way how to develop system description models in formal way,
- improve system description possibilities (for example, does not define new associations or relations between classes), and
- use topology as a formalization tool to describe system's functioning.

The main idea of the given work is to introduce more formalism into the UML class diagrams and to propose a formal approach for developing formalized class diagrams. To achieve this goal formalism of a Topological Functioning Model (TFM) [6] is used. The TFM holistically represents a complete functionality of the system from the computation-independent viewpoint. It considers problem domain information separate from the application domain information captured in requirements. The TFM is an expressive and powerful instrument for a clear presentation and formal analysis of system functioning and the environment the system works within. We consider that problem domain modeling and understanding should be the primary stage in the software development, especially in the case of embedded and complex business systems, where failure can lead to huge losses. This means that class diagrams must be applied as part of a technique, whose first activity is the construction of a well-defined problem domain model.

This paper is organized as follows. Section 2 describes the suggested solution of formalizing UML class diagrams by using topology which is defined with the help of TFM. Section 3 discusses the use of TFM for problem domain modeling and creation of topological class diagrams. TFM makes it possible to use a formal model as a computation independent one without introducing complex mathematics. Besides that, it allows validation of functional requirements at the beginning of the analysis. By using TFM in the modeling process it is possible to introduce topology in the class diagrams. As a result we have constructed a new type of class diagrams – topological class diagrams. Description of the problem domain modeling is illustrated with an example which clearly shows the process of developing topological class diagrams. Section 4 gives conclusions of our work and discuss future work.

2 Formalization of the Class Diagram

Class diagrams reflect the static structure of the system, and with the help of class diagrams it is possible to model objects and their operations involved in the system. Regardless of the opportunities provided by the class diagrams, it is not possible to reflect the cause and effect relation within a system or to indicate which certain activity accomplishment of an object triggers another object's certain activity accomplishment. By using the idea published in [5] about topological UML diagrams (including topological class diagrams) we have developed method for construction of topological class diagrams and we have developed the topological class diagram.

Before topological class construction it is needed to construct the TFM of the system functioning. After construction of TFM it is possible to transform topology defined in TFM into class diagrams and in such a way introduce more formalism into class diagrams. It is possible to transform topology from TFM into class diagrams because TFM has strong mathematical basis. In this way the formalism of class diagrams means that between classes are precisely defined relations which are identified from the problem domain with help of TFM. In traditional software development scenario relations (mostly associations and generalizations) between classes are defined by the modeler's discretion.

Since the TFM of system functioning relations between objects involved into system define as cause and effect relationships, it is not possible to depict those relations within class diagrams by using existing relations defined in UML specification [14]. To enable depicting cause and effect relationships between objects in class diagrams, we have introduced a new type of relation between classes in class diagram – the topological relation.

TFM has strong mathematical basis and is represented in a form of a topological space (X, Θ), where X is a finite set of functional features of the system under consideration, and Θ is the topology (i.e., cause and effect relations between functional features) that satisfies axioms of topological structures and is represented in a form of a directed graph. The necessary condition for constructing the topological space is a meaningful and exhaustive verbal, graphical, or mathematical system description. The adequacy of a model describing the functioning of a concrete system can be achieved by analyzing mathematical properties of such abstract object [6].

A TFM has topological characteristics: connectedness, closure, neighborhood, and continuous mapping. Despite that any graph is included into combinatorial topology, not every graph is a topological functioning model. A directed graph becomes the TFM only when substantiation of functioning is added to the above mathematical substantiation. The latter is represented by functional characteristics: cause-effect relations, cycle structure, and inputs and outputs. It is acknowledged that every business and technical system is a subsystem of the environment. Besides that a common thing for all system (technical, business, or biological) functioning should be the main feedback, visualization of which is an oriented cycle. Therefore, it is stated that at least one directed closed loop (main functioning cycle) must be present in every topological model of system functioning. It shows the "main" functionality that has a vital importance in the system's life. Usually it is even an expanded hierarchy of cycles. Therefore, a proper cycle analysis is necessary in the TFM construction, because it enables careful analysis of system's operation and communication with the environment [6].

There are two stages at the beginning of the problem analysis: the first one is analysis of the business (or enterprise system) context (the problem domain) and the second one is analysis of the application context (the application domain). These levels should be analyzed separately. The first idea is that the application context constrains the business context, not vice versa. The second idea is that functionality determines the structure of the planned system (Fig. 1). Having knowledge about the complex system that operates in the real world, a TFM of this system can be composed.

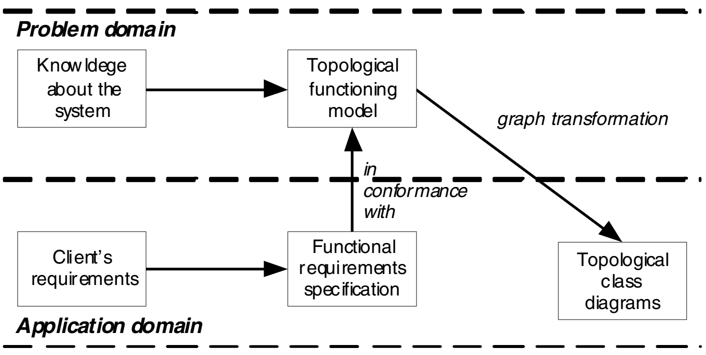

Problem domain
Knowldege about the system
Topological functioning model
graph transformation
in conformance with
Client's requirements
Functional requirements specification
Topological class diagrams
Application domain

Fig. 1. Creation of the software design by using the TFM

In [7] it is suggested that problem domain concepts are selected and described in an UML Class Diagram. In our work we select and describe problem domain concepts by means of topological class diagrams. All these steps are illustrated by the example given in next section.

3 Case Study of the Construction of the Topological Class Diagram

For a better understanding of the construction of the TFM and topological class diagram let us consider small fragment of an informal description from the project, in which a library application is developed:

"All library visitors are registered. Registration is done by the receptionist. Any visitor, who is registered in the library readers' register and who has filled out and filled the reader's card is considered as a reader. If the visitor is not a registered reader yet, the receptionist performs the reader registration. If the visitor does not have the reader's card, the receptionist makes it anew. Registered readers with reader cards have the right to use the library catalogue in order to find the book they need. Only one catalogue is available in the library. To borrow a book from the library, the reader has to complete the request form. Having completed the request form, the reader submits it to the librarian, who counts the number of books already borrowed by the reader. If the number of borrowed books does not exceed the maximum allowed number, the librarian checks, if the reader's chosen book is available from the library repository. If the chosen book is available from the library repository, the librarian hands out the book to the reader. The library has only one book repository. When the reader returns the book, the librarian checks its condition. If the book is damaged, the librarian calculates the fine and issues the fine ticket to the reader. If the book is extremely damaged and cannot be used anymore, the librarian withdraws it and delivers to utilizer, and, if this is the only copy of the book, also removes it form the library catalogue. When the library buys a book, the receptionist checks if the book is not entered in the library catalogue yet, the receptionist registers it in the catalogue. After the book registration, receptionist places it in the book repository and makes it available for borrowing."

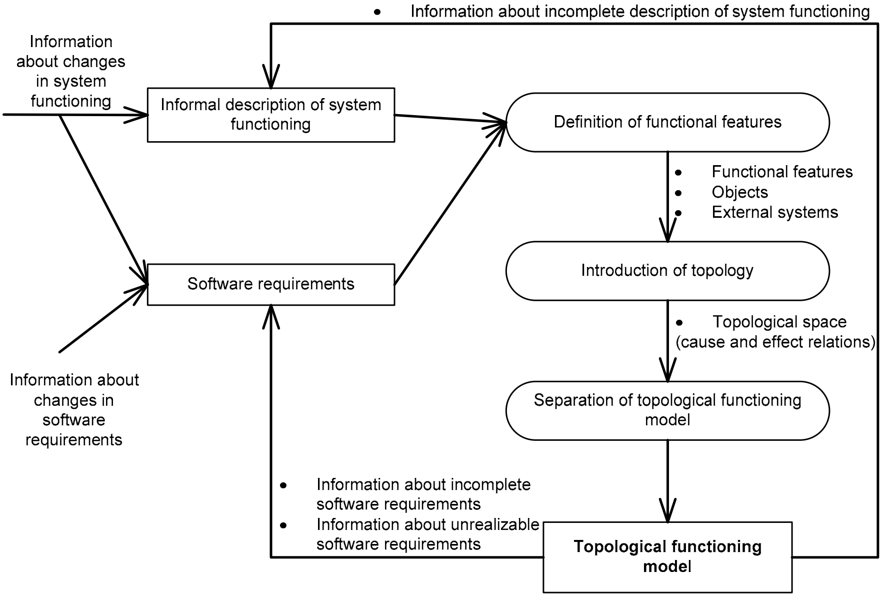

Information about incomplete description of system functioning
Information about changes in system functioning
Informal description of system functioning
Definition of functional features
Functional features
Objects
External systems
Introduction of topology
Topological space (cause and effect relations)
Separation of topological functioning model
Software requirements
Information about changes in software requirements
Information about incomplete software requirements
Information about unrealizable software requirements
Topological functioning model

Fig. 2. The construction of the TFM

3.1 The Construction of the Topological Functioning Model

Construction of the TFM consists of three steps [5] (see Fig. 2).

The steps for the TFM construction are:

Step 1: Definition of physical or business functional characteristics, which consists of the following activities:

1) definition of objects and their properties from the problem domain description;
2) identification of external systems and partially-dependent systems; and
3) definition of functional features using verb analysis in the problem domain description, i.e., by finding meaningful verbs.

Within the [1] it is suggested that each functional feature is a tuple (1),

$$<A, R, O, PrCond, PostCond, E, Cl, Op> \quad (1)$$

where:

- *A* is an object action,
- *R* is a result of this action,
- *O* is an object (objects) that receives the result or that is used in this action (for example, a role, a time period, a catalogue, etc.),
- *PrCond* is a set *PrCond* = {c1, …, ci}, where ci is a precondition or an atomic business rule (it is an optional parameter),
- *PostCond* is a set *PostCond* = {p1, …, pi}, where pi is a postcondition or an atomic business rule (it is an optional parameter),

- *E* is an entity responsible for performing actions,
- *Cl* is a class which will represent in system static model the object which will contain operation for functionality defined by this functional feature (this parameter can be fulfilled when the class diagram is synthesized), and
- *Op* is an operation which will contain functionality defined by functional feature (this parameter can be fulfilled when the class diagram is synthesized).

We have added parameters *Cl* and *Op* to tuple defined in [1] to contain in the tuple all the information about functional feature. If there is a need to store additional information about functional features then it is possible to add more parameters to this tuple.

Each precondition and atomic business rule must be either defined as a functional feature or assigned to an already defined functional feature.

For the library project example we have defined the following 29 functional features (in the form of tuple containing the following parameters: identificator, object action (*A*), precondition (*PrCond*), object (*O*), mark if functional feature is external or internal), where *Rec* denotes Receptionist, *R* – Reader, *L* – Librarian, *In* – Inner, and *Ex* - External:

<**1**, A visitor arriving in the library, Ø, Visitor, Ex>, <**2**, Checking of personal data with the library readers' register, Ø, Rec, In>, <**3**, Reader's registration in the library readers' register, if the person is not registered in the readers' register yet, Rec, In>, <**4**, Reader's card preparation, if the reader does not have the reader's card yet (or) if the reader has lost his/her reader's card, Rec, In>, <**5**, Reader's card issue to the reader, Ø, Rec, In>, <**6**, The reader status authorization, if the reader is registered (and) if the reader has the reader's card, R, In>, <**7**, Searching for a book in the book catalogue, if the reader has the reader's card, R, In>, <**8**, Completion of the book request form, if the reader has found the book he or she needs, R, In>, <**9**, Submission of the book request form, Ø, R, In>, <**10**, Count of books borrowed by the reader, Ø, L, In>, <**11**, Checking of the book availability in the book repository, if the number of books borrowed by the reader does not exceed the maximum allowed, L, In>, <**12**, Taking the book from the book repository, if the book is available in the book repository, L, In>, <**13**, Handing out the book to the reader, Ø, L, In>, <**14**, Borrowing the book, Ø, R, In>, <**15**, Book return, Ø, R, Ex>, <**16**, Checking of the book condition, Ø, L, In>, <**17**, Fine calculation, if the book is damaged, L, In>, <**18**, Handing out the fine ticket, Ø, L, Ex>, <**19**, Fine payment, Ø, R, Ex>, <**20**, Book return/placement into book repository, Ø, L, In>, <**21**, Book withdrawal, if the book is extremely damaged (cannot be used anymore), L, Ex>, <**22**, Book removal from the catalogue, in case of the last copy of the book, L, In>, <**23**, New book purchase, Ø, Library, Ex>, <**24**, Books data entry into catalogue, if the library does not have a copy of this book, Rec, In>, <**25**, Book identification number assignment, Ø, Rec, In>, <**26**, Book utilization, If the book is extremely damaged, Utilizer, Ex>, <**27**, Book repository maintenance, Ø, L, In>, <**28** Completion of the book utilization request form, Ø, L, In>, and <**29**, The fine deletion, if the reader has paid the fine, L, In>.

Step 2: Introduction of topology *Θ*, which means establishing cause and effect relations between functional features. Cause-and-effect relations are represented as arcs of a directed graph that are oriented from a cause vertex to an effect vertex.

The identified cause and effect relations between the functional features are illustrated by the means of the topological space (see Fig. 3).

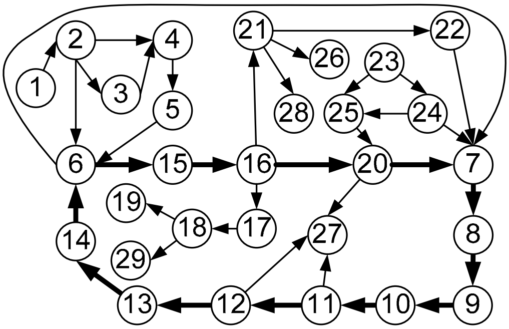


Fig. 3. Topological space of the library functioning

In the Fig. 3 is clearly visible that cause and effect relations form functioning cycles. All cycles and sub-cycles should be carefully analyzed in order to completely identify existing functionality of the system under consideration. The main cycle (or cycles) of system functioning (i.e., functionality that is vital for the system's life – the functionality without which the system can no longer function and exist) must be found and analyzed before starting further analysis. In the case of studying and designing a complex system, the TFM of this system can be divided into a series of subsystems according to the identified cycles.

Step 3: Separation of the topological functioning model, which is performed by applying the closure operation over a set of system's inner functional features [6]: A topological space is a system represented by Equation (2),

$$Z = N \cup M \tag{2}$$

where N is a set of inner system functional features and M is a set of functional features of other systems that interact with the system or of the system itself, which affect the external ones.

A TFM ($X \in \Theta$) is separated from the topological space of a problem domain by the closure operation over the set N as it is shown by Equation (3),

$$X = [N] = \bigcup_{\eta=1}^{n} X_{\eta} \tag{3}$$

where Xη is an adherence point of the set N and capacity of X is the number n of adherence points of N.

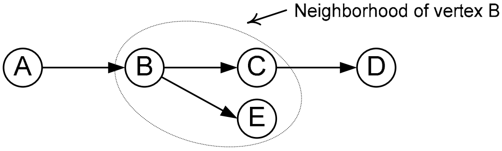


Fig. 4. Neighborhood of element of the set N

An adherence point of the set N is a point, whose each neighborhood includes at least one point from the set N. The neighborhood of a vertex x in a directed graph is the set of all vertices adjacent to x and the vertex x itself. It is assumed here that all vertices adjacent to x lie at the distance d=1 from x on ends of output arcs from x. An illustrative example of vertex's neighborhood is given in Fig. 4.

The example below illustrates how is performed the closuring operation (3) over the set N in order to get all of the system's functionality (the set X):

- The set of the system's inner functional features N = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 17, 20, 21, 22, 24, 25, 27, 28, 29};
- The set of external functional features and system functional features that affect the external environment M = {1, 15, 18, 19, 21, 23, 26};
- The neighborhood of each element of the set N is as follows: X_2 = {2, 3, 4, 6}, X_3 = {3, 4}, X_4 = {4, 5}, X_5 = {5, 6}, X_6 = {6, 15}, X_7 = {7, 8}, X_8 = {8, 9}, X_9 = {9, 10}, X_{10} = {10, 11}, X_{11} = {11, 12, 27}, X_{12} = {12, 13, 27}, X_{13} = {13, 14}, X_{14} = {14, 6}, X_{16} = {16, 17, 20, 21}, X_{17} = {17, 18}, X_{20} = {20, 7, 27}, X_{22} = {22, 7}, X_{24} = {24, 7, 25}, X_{25} = {25, 20}, X_{27} = {27}, X_{28} = {28}, and X_{29} = {29}; and
- The obtained set X (*the TFM*) = {2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 20, 21, 22, 24, 25, 27, 28, 29}.

Obtained TFM of library functioning after performing closuring operation over the set of system inner functional features (the set N) can be seen in Fig. 5.

The example represents the main functional cycle defined by the expert, which includes the following functional features "6-15-16-20-7-8-9-10-11-12-13-14-6" and is denoted by bold lines in Fig. 5. These functional features describe checking out and taking back a book. A cycle that includes the functional features "6-7-8-9-10-11-12-13-14-6" illustrates an example of the first-order sub-cycle. These functional features describe checking out a book.

3.2 Construction of the Topological Class Diagram

In the [7] is offered the conceptual development of class diagrams as the final step of the TFM usage. In this conceptual class diagram relevant information – directions of associations between the classes – is lost. This important information is lost because within approach given in [7] the relations between classes are defined with one of the relations defined in UML – the associations. It is not possible to transform topological (cause and effect) relations between TFM's functional features into associations between classes. It is impossible because:

1) the direction of topological relation is not always the same as direction of association,
2) association also can be bidirected (topological relationship can not be bidirected), and
3) topological relationship only can be binary relation (association can relate more than two classes, for example, ternary association which relates three classes).

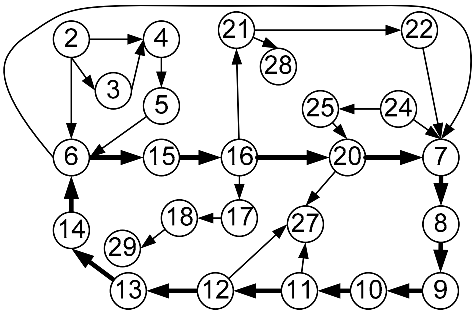


Fig. 5. Topological functioning model of the library functioning

Because of this constraint in [7] it is recommended to define those association directions in further software development, for example, while developing a more detailed software design. But at this point a step back should be taken to review the TFM and its transformation on the conceptual class diagram. To avoid such regression and to save the obtained topology between the classes, by using the idea published in [5] about topological UML (TopUML) diagrams (including topological UML class diagrams), it is possible to develop a topological class diagram where the established TFM topology between classes is retained. The retained topology (cause and effect relations between classes) in class diagrams brings more formalism in these class diagrams. Formalism of class diagrams is improved because between classes now are precisely defined relations. In traditional software development relations (mostly associations and generalizations) between classes are defined by the modeler's discretion (the approach given in the [7] helps to identify associations between classes but the identification of direction for these associations again are defined by the modeler's discretion).

Topological relations between classes throughout this article are marked with directed arcs (this means that within this article notation used for topological relations between classes is similar to notation of associations in UML). The example of topological relations is shown in Fig. 6.

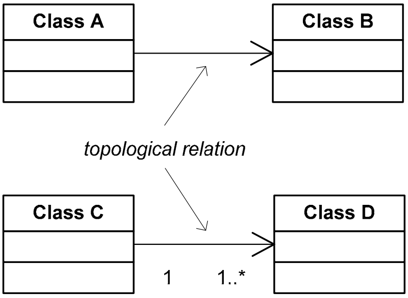


Fig. 6. Topological relations between classes

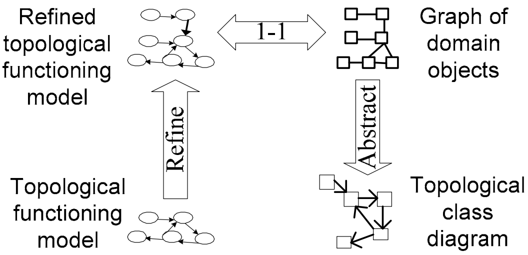

Refined topological functioning model
1-1
Graph of domain objects
Refine
Abstract
Topological functioning model
Topological class diagram

Fig. 7. The process of the development of the topological class diagram

In order to obtain a topological class diagram, first of all a graph of problem domain objects must be developed and afterwards transformed into a class diagram. In order to obtain a problem domain object graph, it is necessary to detail each functional feature of the TFM to a level where it uses only one type of objects.

After construction of detailed TFM this more accurate model must be transformed one-to-one to a problem domain object graph and then the vertices with the same type of objects and operations must be merged, while keeping all relations with other graph vertices. As a result, object graph with direct links is defined. Schematic representation of topological class diagram development is given in Fig. 7.

By using the ideas published in [7] it is possible to obtain from TFM a conceptual class diagram without orientated relations between classes and the classes without operations. Modifying this approach it is possible to develop not only topological class diagrams, where the direction of relations is retained, but also to obtain the possible class operation definitions. In order to define conceptual operations, it is necessary to change not only every functional feature to one kind of object, but also by doing this transformation, to add a operation to the obtained (using a point notation), the description of which shortly describes the defined activity of the functional feature, for example, the functional feature *"The reader's card issue to the reader"* is transformed to the object *"ReaderCard"* and the operation *"GiveOutToReader()"* (when point notation is used the obtained result looks like this: *"ReaderCard.GiveOutToReader()"*).

At this moment it is possible to add additional information to the tuple (fulfil parameters *Cl* and *Op*) which is describing functional feature. After adding two parameters describing class and operation the tuple looks like this: *<5, Reader's card issue to the reader, Ø, Rec, In, ReaderCard, GiveOutToReader>*.

Discussed example skips the step of the TFM refinement, because each functional feature deals only with one type of objects and operations. Fig. 8 shows the transformation of the TFM to the graph of domain objects with conceptual operations.

Fig. 9 presents topological class diagram of the library example after domain object graph is abstracted, i.e., after merging all graph vertices with the same object types.

With the boldest lines in developed topological class diagram is maintained main functional cycle which is defined by the expert within the constructed TFM. This reflects the idea proposed in [5] and [6] that the holistic domain representation by the means of the TFM enables identification of all necessary domain concepts and, even, enables to define their necessity for a successful implementation of the system.

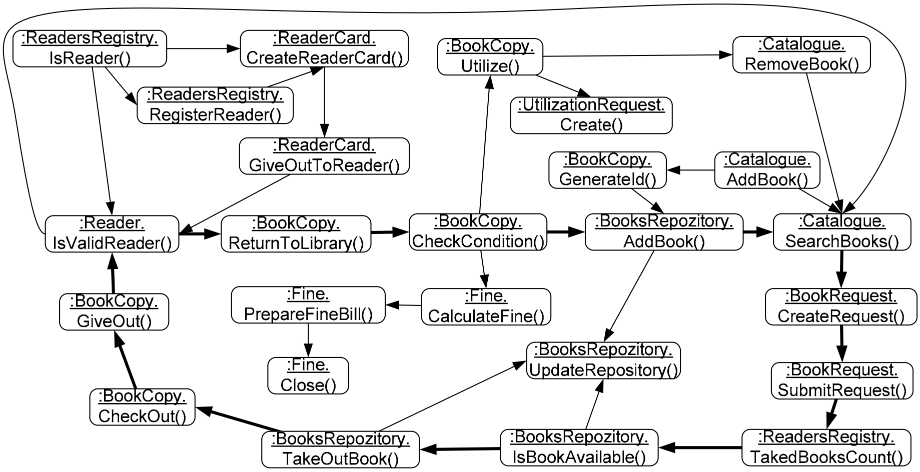


Fig. 8. The graph of domain objects with operations

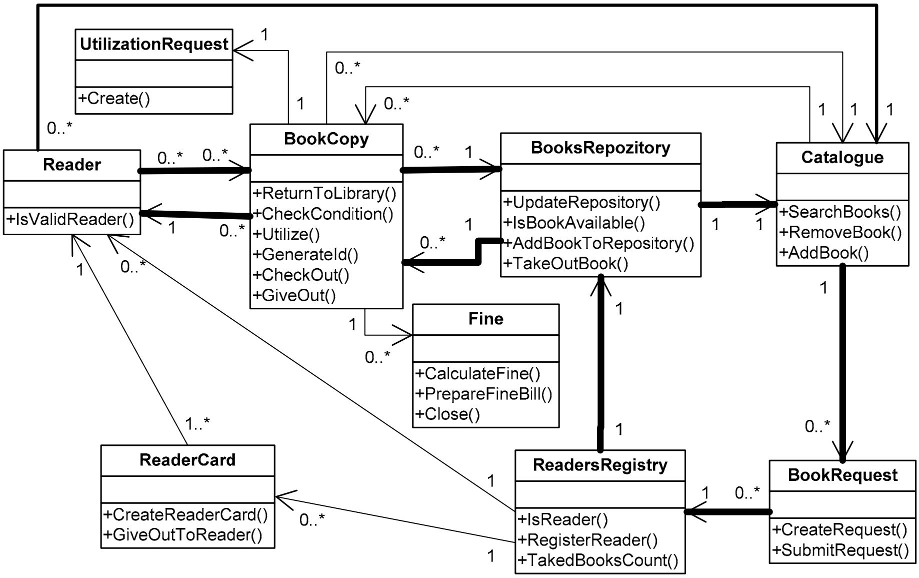


Fig. 9. Topological class diagram

The topological (cause and effect) relationship between classes, which are described with one way directed arc, cannot be compared with none of the UML relationships between the classes given in UML language specification [14]. The UML language specification gives the following relationships between the classes:

- association (including aggregation and composition),
- generalization,

- dependence,
- usage,
- abstraction,
- realization, and
- substitution.

All previously mentioned relationships between classes define only the way in which the classes interact and use each other [9], but the adopted topology in class diagrams allows to keep the cause and effect relationships between objects. The saved topology between classes in class diagram enables more efficient development of the software system class diagram.

By keeping topological relationships between the classes it is recommended to use one-way association, because two mutually opposed associations between two classes can represent various multiplications. If topological class diagram is used to make the non-oriented class diagram, then relations between two classes can be joined into one, and as a multiplicity save the biggest multiplicity of all topological associations between those two classes.

4 Conclusions and Future Work

The application of the TFM has the following advantages:

- With the help of TFM it is possible to introduce more formalism in the UML diagrams and in their construction. In our work we have shown that it is possible to maintain in the class diagrams the topology which is developed using TFM.
- Using TFM for problem domain modeling and application domain definition it is possible to provide traceability between software requirements, functional features and even developed architecture elements.
- By performing TFM transformations it is possible to develop problem domain objects' graphs and topological class diagrams.
- Topological class diagram can also be used as architecture for the new system. With the help of TFM and topological class diagrams it is possible to develop software system's business layer which corresponds to the defined requirements.

To continue working on topological UML diagrams, it is necessary to supplement the description of topological class diagrams, to create the meta-model of the topological class diagram as well as to study the possibilities of topology implementation into other UML diagrams (for example, activity diagrams) and to assess its influence on the software system development.

References

1. Asnina, E.: The Formal Approach to Problem Domain Modelling Within Model Driven Architecture. In: Proceedings of the 9th International Conference "Information Systems Implementation and Modelling" (ISIM 2006), Přerov, Czech Republic. Jan Štefan MARQ, pp. 97–104 (2006)

2. Evans, A., Kent, S.: Core Meta-Modelling Semantics of UML: The pUML Approach. In: France, R.B., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723, pp. 140–155. Springer, Heidelberg (1999)
3. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edn. Addison-Wesley, Reading (2003)
4. Object management group, OMG (2008), `http://www.omg.org`
5. Osis, J.: Extension of Software Development Process for Mechatronic and Embedded Systems. In: Proceeding of the 32nd International Conference on Computer and Industrial Engineering, University of Limerick, Limerick, Ireland, pp. 305–310 (2003)
6. Osis, J.: Formal Computation Independent Model within the MDA Life Cycle. International Transactions on Systems Science and Applications 1(2), 159–166 (2006)
7. Osis, J., Asnina, E.: Enterprise Modeling for Information System Development within MDA. In: Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008), USA, p. 490 (2008)
8. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley, Reading (2004)
9. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
10. Spivey, J.M.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall, Englewood Cliffs (1992)
11. Szlenk, M.: UML Static Models in Formal Approach. In: Meyer, B., Nawrocki, J.R., Walter, B. (eds.) CEE-SET 2007. LNCS, vol. 5082, pp. 129–142. Springer, Heidelberg (2008)
12. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA, 2nd edn. Addison-Wesley, Reading (2003)
13. The Precise UML group (pUML) (2000), `http://www.cs.york.ac.uk/puml/`
14. OMG: Unified Modeling Language Superstructure Specification, version 2.1.2 (2007)

Extended KAOS Method to Model Variability in Requirements

Farida Semmak[1], Christophe Gnaho[1,2], and Régine Laleau[1]

[1] Paris Est University, LACL, France
{semmak,laleau}@u-pec.fr
[2] Paris Descartes University, France
christophe.gnaho@mi.parisdescartes.fr

Abstract. This paper presents an approach to requirements engineering in the Cycab domain. Cycabs are public vehicles with fully automated driving capabilities. So far few studies have dealt with expressing such requirements at the highest level of abstraction. Furthermore, during their building, software systems embedded in Cycabs are subject to frequent changes of requirements. Thus, we need to represent a family of Cycabs that can differ according to different options. The proposed approach tries to address these issues by adopting and extending the KAOS goal oriented method. The main objective is to provide a process to define and adapt specific requirements models from a generic model, according to different situations made available to the stakeholders.

Keywords: Goal-based requirements engineering, Family model, Variability, Land transportation domain.

1 Introduction

The paper sums up a work achieved as part of the Tacos[1] Project [18] whose aim was to define a component-based approach to specify trustworthy systems from the requirements to the specification phase. The application domain of the project is the Cycab, a new mode of urban transportation designed by INRIA [2], [13] in order to offer a new alternative to private vehicles. A Cycab is controlled by an embedded software system which enables fully automated driving capability. This paper focuses on the requirements phase in the Cycab domain. So far, few studies have been concerned with expressing the requirements at a high level of abstraction. The natural language remains the main way for describing such requirements.

Furthermore, during their building, software systems embedded in Cycabs are subject to frequent requirements changes due to effective tests on the prototype. These changes need to be taken into account and integrated in the current specification. Thus, these systems require a rigorous requirements engineering approach that integrates adaptation and tailoring.

[1] The TACOS project (Ref. ANR-06-SETI-017) is supported by the French National Research Agency.

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 193–205, 2010.

This paper tries to address the above-mentioned issues. It proposes a requirements engineering approach whose main objective is to provide the application engineer with a process to define specific requirements models from a generic model, according to different situations selected from a variant model.

The paper is organized as follows. Section 2 presents an overview of the research project. Section 3 describes the variant model and the generic model. Section 4 focuses on the model building process. Related work is given in section 5. Section 6 concludes with some remarks about the results and future works.

2 Overview of the Research Project

This section presents the context of Cycab domain and an overview of the proposed approach.

2.1 Domain Context

The Cycab transportation system has been chosen to be the application domain of the project. Cycabs are small electric vehicles designed for restricted access zones: historic city centers, airports, train stations or university campuses. They must be easy to use by a large population: access control by smart, simple manual control through a joystick, automatic parking and recharging, etc.

A Cycab is controlled by embedded electronic systems which allow automatic driving under computer control. Computer control is achieved by processing information coming from sensors and actuators that regulate the physical devices [22]. Several prototypes have already been built and tested. Two years of work have been necessary to design and implement the first two prototypes [2], [13]. Now that the feasibility of such a concept has been demonstrated, the objective of current Cycab research projects is to enlarge the use of these vehicles by developing specific Cycabs for specific services. We believe that a way to achieve this objective is to provide a family of Cycabs that can differ from the different options made available to the stakeholders, such as the localization mode, the driving mode, etc.

Therefore, Cycab systems represent a diversity of applications for which design efforts could be capitalized. It becomes interesting to identify and express the common and variable elements between these applications in order to reuse them.

2.2 Our Approach

The aim of our approach is to provide a generic model, from which requirements models for specific systems can be derived according to some options selected by the stakeholders. For instance, we can obtain "a Cycab with a GPS localization mode" or "a Cycab with WPS and GPS localization modes" or "a Cycab with automatic driving without doors" and so on. The generic model expresses the requirements at the highest abstraction level.

We believe that Goal-Oriented Requirements Engineering methods like KAOS [3], [4], [9], i* [20], CREWS [15], GBRAM [1] are suitable to specify the generic model. We have adopted the first one because it allows one to express "*goals and their operationalization into specifications of services and constraints (WHAT issues), and* *the assignment of responsibilities to agents such as humans, devices and software pieces available or to be developed (WHO issues)"* [9]. However, in preliminary works [17], we have demonstrated that KAOS concepts are not sufficient for describing variations at goal level. In order to address this issue, we have associated to the generic model a variant model that adds variability concepts to KAOS concepts.

Figure 1 shows an overview of the approach, it distinguishes two steps. In the first step, the model engineer and the domain experts elaborate the generic model and the variant model. In the second step, the application engineer builds and adapts a requirements specific model according to the situations to be satisfied.

As already stated, requirements can change rapidly, therefore, thanks to the variant model, the process should allow the application engineer to modify and adapt the requirements model in order to face new situations.

To summarize, the objective of our approach is threefold. Firstly, it considers requirements at the highest level of abstraction with a goal orientation; secondly, it allows representing all the options and alternatives available in the domain; lastly, it provides a process that helps the application engineer to create, adapt and modify the requirements specific model in a flexible way.

3 Product Models

This section describes the variant model and the generic model, respectively. These models are the product models used as input for the specific model building- process.

3.1 Variant Model

The variant model captures and describes the relevant domain characteristics that present multiple options of realization. An underlying problem is to describe and organize these characteristics in order to better represent the discriminatory elements between applications of the domain. For that, we adopt and extend the concept of facet as defined in [14].

Figure 2 presents the metamodel of *the variant model* as a UML class diagram, focusing on the concepts of *Facet* and *Variant* and their dependencies.

A domain can be characterized by many facets. A *facet* represents a relevant feature that has an interest for the domain. *A facet* is defined by *a name* and *a description*. For instance, the Cycab transportation domain has several facets among which the facet with the *name* "F1: localization mode" and the *description* "to know the (current) vehicle position in order to follow its trajectory". The facet called "F2: Traffic Lane" has for description "to indicate precisely the kind of route the vehicle follows".

A *facet* is composed of one or many *variants*. A *variant* defines a way to realize a facet. For example, in Figure 3, the facet "F1: localization mode" has the following variants: {V1: GPS, V2: wire guiding, V3: magnets plots, etc} meaning that the Cycab vehicle may be localized by using one of the variants or by combining them. The facet "F2: Traffic Lane" presents four variants: {V1: dedicated, V2: semi-protected, V3: pedestrian, V4: urban road}.

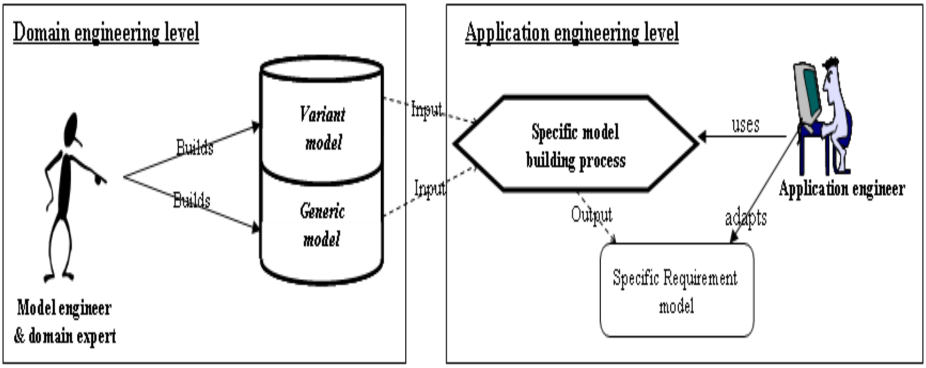

Domain engineering level
Variant model
Generic model
Builds
Builds
Model engineer & domain expert
Application engineering level
Input
Input
Specific model building process
uses
Application engineer
Output
adapts
Specific Requirement model

Fig. 1. Overview of the approach

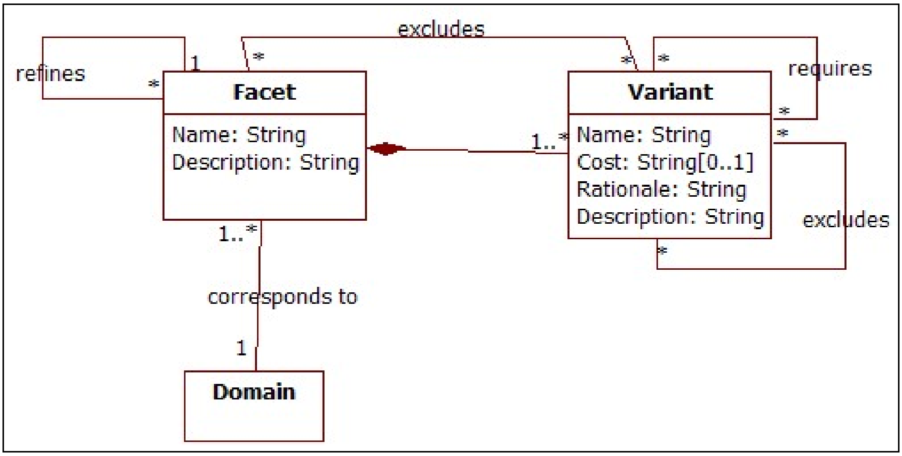


Fig. 2. The Variant Metamodel

A variant is described by *a name*, *a cost*, *a rationale* and *a description*. For instance, one of the variants of facet F1 (Figure 3) has *a name* "GPS", a *description* "localization by measuring signal propagation time from different satellites", *a cost* "Ø" and *a rationale* "efficient if the localized area is not surrounded by high buildings".

The cost and rationale properties not only support decisions taken by the application engineer but also contribute to express non-functional requirements (NFR). Note that in this kind of systems, NFRs such as security or safety are crucial.

A facet may be refined in sub-facets; this is captured by the "*refines"* relationship as shown in Figure 2. For example, the facet "F4: Obstacle detection" can be refined in two sub-facets "F4.1: Collision detection" and "F4.2: Range finder" and each sub-facet has its own variants as illustrated in Figure 4.

The dependencies between facets and variants or between variants are captured by two relationship types. As shown in Figure 2, a variant of a given facet may *require* zero or many variants of another facet; for instance, the variant "V2: Automatic" of the

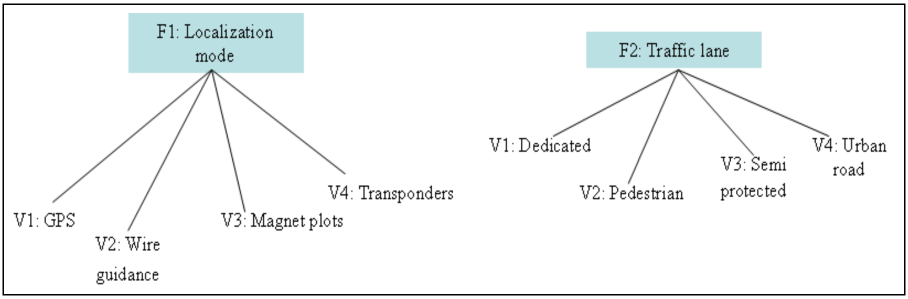


Fig. 3. Two facets and their variants

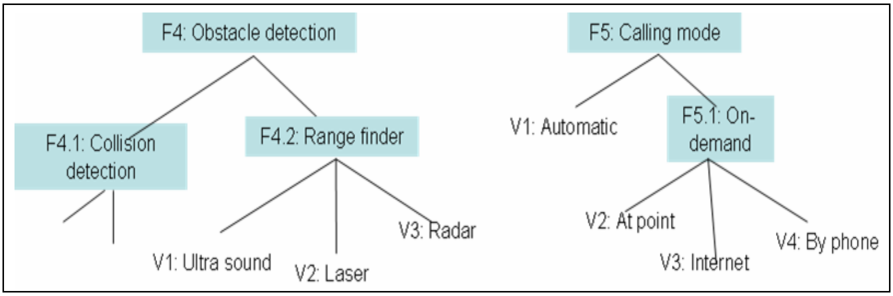


Fig. 4. Two examples of refined facets

facet "F3: Driving Mode" requires the variant "V1: dedicated" of the facet "F2: Traffic Lane". A variant of a given facet may *exclude* zero or many variants of another facet. For instance, the variant "V4: urban road" of facet F2 excludes the variant "V3: magnets plots" of facet F1. A variant of a given facet may *exclude* another facet; for example, the variant "V1: dedicated" of the facet "F2: Traffic Lane" excludes the facet "F4: Obstacle detection".

To sum up, the variant model allows the expression of the relevant domain features and the different ways to realize them. It therefore emphasizes differences between systems of a same family.

3.2 Generic Model

As mentioned in section 2, the aim of the generic model is to capture, in an integrated view, the common and variable requirements of the systems. This model is defined as an instance of the generic metamodel which is obtained by extending the KAOS metamodel with the concepts defined in the variant metamodel.

Figure 5 presents a portion of the generic metamodel. The KAOS concepts are represented on the right side of the figure whereas the extensions made to KAOS are represented by grey boxes on the left side of the figure. A detailed description of the KAOS metamodel can be found in [3], [4], [12].

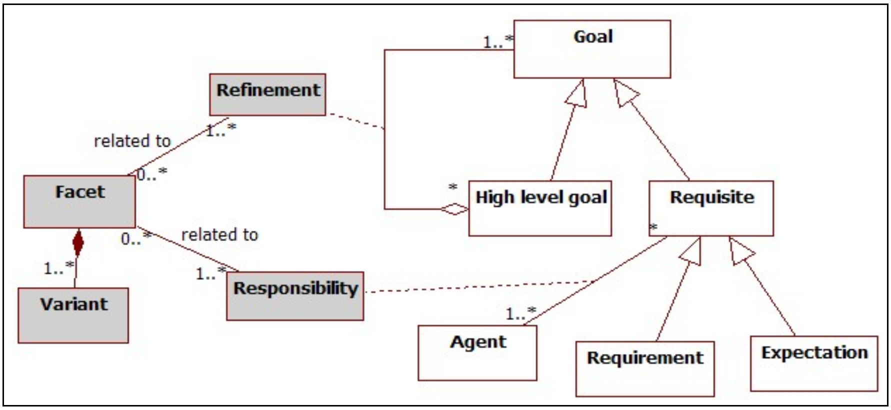


Fig. 5. A portion of the generic metamodel

In the following, we will first present the KAOS concepts and then describe how these concepts are related to the variability concepts.

- Brief overview of KAOS

The KAOS method provides four complementary sub-models that describe the system and its environment: a goal model, a responsibility model, an operation model and an object model. In this paper, we mainly focus on the goal model.

The main concept of the goal model is the concept of *goal*. A *goal* is defined as an objective to be achieved by the system-to-be. A high level goal can be refined into sub-goals, and then, recursively, into low-level sub-goals that lead to the requirements of the system-to-be.

The *refinement* relationship between a high level goal and its sub-goals is an AND/OR meta-relationship. When a goal is *AND-refined* into sub-goals, all of them must be satisfied for the parent goal to be satisfied. When a goal is *OR-refined*, the satisfaction of one of them is sufficient for the satisfaction of the parent goal.

A goal that cannot be refined further and that is assignable to an agent either in the environment or in the system is a *requisite*. A *requisite* that is placed under the responsibility of an agent in the system is a *requirement*, whereas a *requisite* that is placed under the responsibility of an agent in the environment of the system is an *expectation*.

- KAOS with Variability

In our approach, the goal model aims at capturing and expressing all the possible options of the systems-to-be in terms of requirements described at the highest level of abstraction. The concept of alternative in KAOS can represent a kind of variability that is local to a goal. But it is not sufficient to express variability, particularly the one which has an impact on different parts of the goal model. Thus, we propose to extend the KAOS concepts with the concepts of *facet and variant*.

The concept of <*Facet-Variant*> related to a *refinement link*, allows in addition to take into account different situations which could be considered by the application

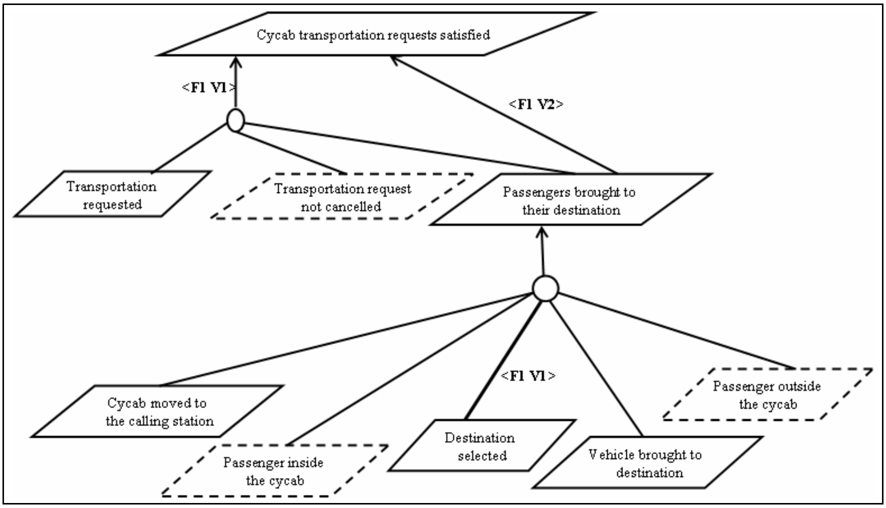

Cycab transportation requests satisfied
<F1 V1>
<F1 V2>
Transportation requested
Transportation request not cancelled
Passengers brought to their destination
<F1 V1>
Cycab moved to the calling station
Passenger inside the cycab
Destination selected
Vehicle brought to destination
Passenger outside the cycab

Fig. 6. Partial Goal Model of the simplified Cycab Case Study

engineer. Thus, a couple<Fi, Vi> may be attached to a refinement link between a goal and its sub goals, meaning that this refinement depends on the variant Vi of the facet Fi. A refinement link that is not attached to a couple <Fi, Vi> is defined as mandatory and is then common to all the applications of the domain.

Let us consider the facet named "F1: Cycab calling mode" with the following two associated variants: "V1: automatic" (a Cycab stopping at each station) and "V2: on demand" (it stops at a station only if there is an external or internal demand). Figure 6 presents an instance of the generic metamodel with the high-level goal "Cycab transportation requests satisfied". So, according to the variant <F1, V1>, this goal is refined into three sub-goals: "transportation requested", "transportation request not cancelled" and "passengers brought to their destination". Note that when the arrow is annotated (see Figure 6 with <F1, V1>), it means that all the links under the circle are annotated.

As previously said, the same variant can have an impact on several parts of the graph. For instance, as shown in Figure 6, the variant <F1, V1> has an impact on two parts of the graph. The sub-goal "passengers brought to their destination" is recursively refined into five low-level sub-goals: the four mandatory sub-goals and the optional sub-goal "Destination selected". With the variant <F1, V2>, the goal "Cycab transportation requests satisfied" is refined into one sub-goal: "Passengers brought to their destination". The latter is then refined into four sub-goals: "Cycab moved to the calling station", "Passenger inside the Cycab", "Vehicle brought to destination" and "Passenger outside the car". So the sub-goal "Destination selected" has not to be taken into account.

Furthermore, the assignment of an agent to a goal (requisite) is captured in the metamodel by the Responsibility relationship (see Figure 5) which can be associated with zero or many facets, which means that the assignment may depend on the selected variants.

In order to illustrate this, let us consider the following facets: "F2: Cycab doors opening/closing mode" with the variants "V1: manual" or "V2: automatic"; and "F3: Cycab driving mode" that can be either "V1: manual" (human driver) or "V2: automatic" (control system driver).

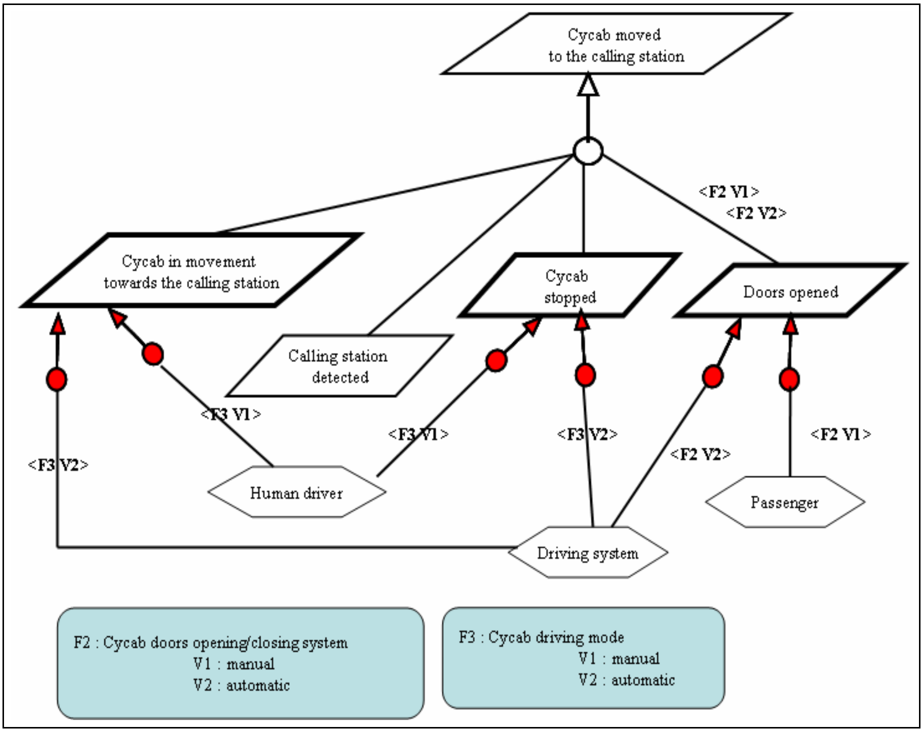


Fig. 7. Refinement of the goal Cycab placed at disposal at the calling station

The impact of these variants is shown in Figure 7. Agents are represented by hexagonal boxes and requisites are represented as thick-bordered parallelograms.

Let us consider the requisite "Cycab in movement towards the calling station"; it may be under the responsibility of either the "driving system" agent or the "human driver" agent.

With KAOS, a requisite is placed under the responsibility of only one agent. However, the extensions in Figure 5 enable the application engineer to make several agents responsible for a given requisite. The decision to consider a requisite either as an expectation or as a requirement will be taken only when building a specific model. For instance, in Figure 7, the requisite "doors opened" can be refined in a requirement or in an expectation depending on the choice of variants. If the variant <F2, V2> ("automatic doors opening/closing mode") is selected, then this requisite will be placed under the responsibility of the system agent "Driving system" and consequently will be specialized in a requirement. On the other hand, if <F2, V1> ("manual doors opening/closing mode") is chosen, this requisite will become an expectation because it will be placed under the responsibility of the "Passenger" which is an agent in the environment of the system.

4 Specific Model Building Process

This section describes the specific model building process, which is provided in order to build a specific requirements model of the system to-be. The proposed process is

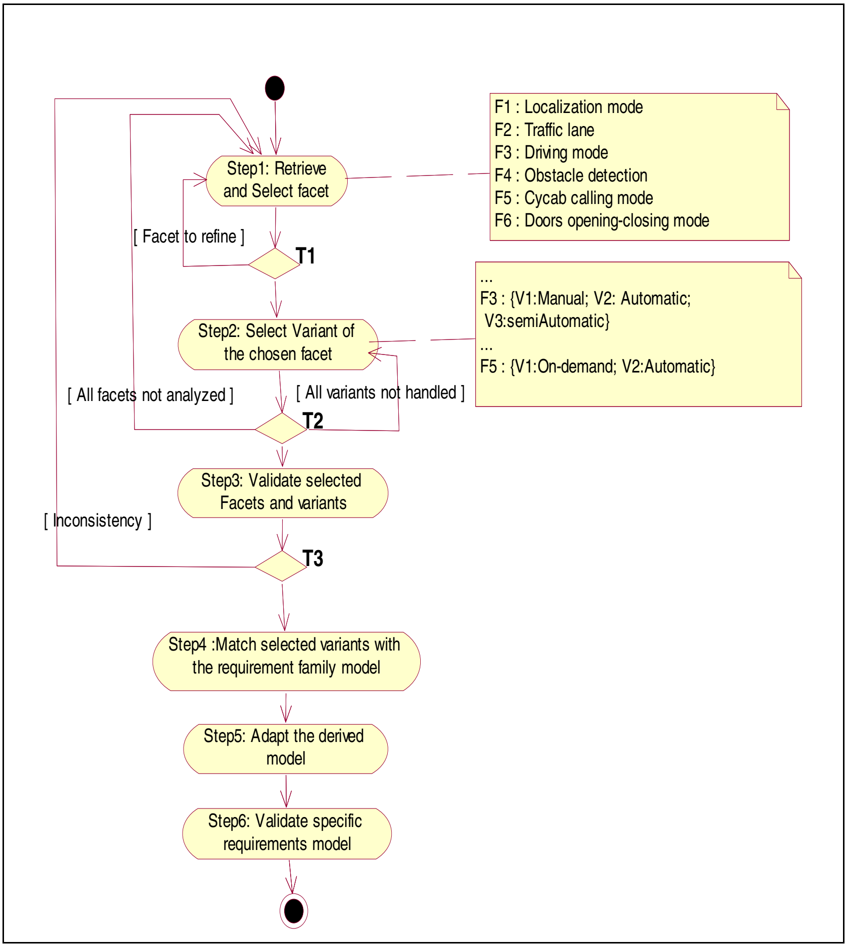

F1 : Localization mode
F2 : Traffic lane
F3 : Driving mode
F4 : Obstacle detection
F5 : Cycab calling mode
F6 : Doors opening-closing mode
Step1: Retrieve and Select facet
[Facet to refine]
T1
...
F3 : {V1:Manual; V2: Automatic; V3:semiAutomatic}
...
F5 : {V1:On-demand; V2:Automatic}
Step2: Select Variant of the chosen facet
[All facets not analyzed]
[All variants not handled]
T2
Step3: Validate selected Facets and variants
[Inconsistency]
T3
Step4 :Match selected variants with the requirement family model
Step5: Adapt the derived model
Step6: Validate specific requirements model

Fig. 8. Overview of the specific model building process

different from a traditional process in which the construction of the model starts from scratch and requirements are identified from documentation and interviews. The main characteristic of this process is that requirements are derived and adapted thanks to both generic and variant models.

Figure 8 represents an overview of the process as a UML activity diagram. It is an iterative and incremental six-step process. The flow among the steps is not sequential but may contain several cycles.

In the following paragraphs, each step involved is briefly explained

Step 1: Retrieve and select facet

This step describes the initial task to be performed by the application engineer. The main objective is to select from the variant model a facet that is relevant to the system to-be. This step may be performed one or many times depending on the conditions involved. For instance, let us consider the choice of the facet "F4: Obstacle detection".

As explained in Figure 4, this facet needs to be refined in "F4.1: Collision detection" and "F4.2: Range finder". To achieve this, step1 will be performed many times.

The transition (T1) from step1 to step 2 occurs if the selected facet does not need to be refined.

Step 2: Select variant of the chosen facet

Having chosen a facet in the previous step, the application engineer needs to select a variant of this facet. This task may be guided by the analysis of the properties of each variant, in particular the *cost* and the *rationale*. The outcome is a couple (*Facet, Variant*) as for instance <F5: Cycab calling mode, V2: Automatic>.

The transition (T2) from this step to the next one is not sequential. This step also may be performed many times if all the variants are not handled. If other facets need to be considered then go back to step 1; if the selection of couples (*Facet, Variant*) is completed, then proceed to step 3.

Step 3: Validate selected Facets and Variants

A number of significant couples (*Facet, Variant*) results from the previous two steps and need to be validated.

One of the main objectives of step 3 is to check the relationships between facets and variants in order to handle interdependencies. For example, as explained in section 3, the variant "V1: dedicated" of the facet "F2: Traffic lane" may exclude the facet "F4: Obstacle detection".

If any inconsistency is detected go back to step 1. Otherwise, continue to step 4 (T3).

Step 4: Match selected variants with the generic model

Once the selected couples (*Facets, Variants*) have been validated in the previous step they can be matched with the generic model. This is the focus of step 4.

To illustrate this step, let us consider the following set of variants (named *situation 1*) : <F5: Cycab calling mode, V1: On demand>, <F6: Doors opening-closing mode, V1: Manual> and <F3: Driving mode, V2: Automatic>. The matching of these variants with the generic model yields the requirements model of the Cycab system presented in Figure 9.

Step 5 and Step 6: Adapt and validate the derived model

The last two steps of the process consist in adapting and validating the specific requirements model resulting from step 4.

For example, the following set of variants defines a new situation named *Situation 2*: <F5: Cycab calling mode, V2: Automatic>, <F6: Doors opening-closing mode, V2: Automatic> and <F3: Driving mode, V2: Automatic>. This new situation leads to the specific requirements model presented in Figure 10 which is adapted to a Cycab without doors.

5 Related Works

The purpose of variability is to be able to identify and express the variable elements between the applications of a given domain. It is considered as a key challenge in building reusable infrastructures. Huge works have notably been done in the domain of software product lines (SPL) [19], [6] where *'Variability'* is modeled by the concepts of *variation point* and of *variant*. A variation point defines an element in the model where a variation occurs, while a variant represents a way to realize this element.

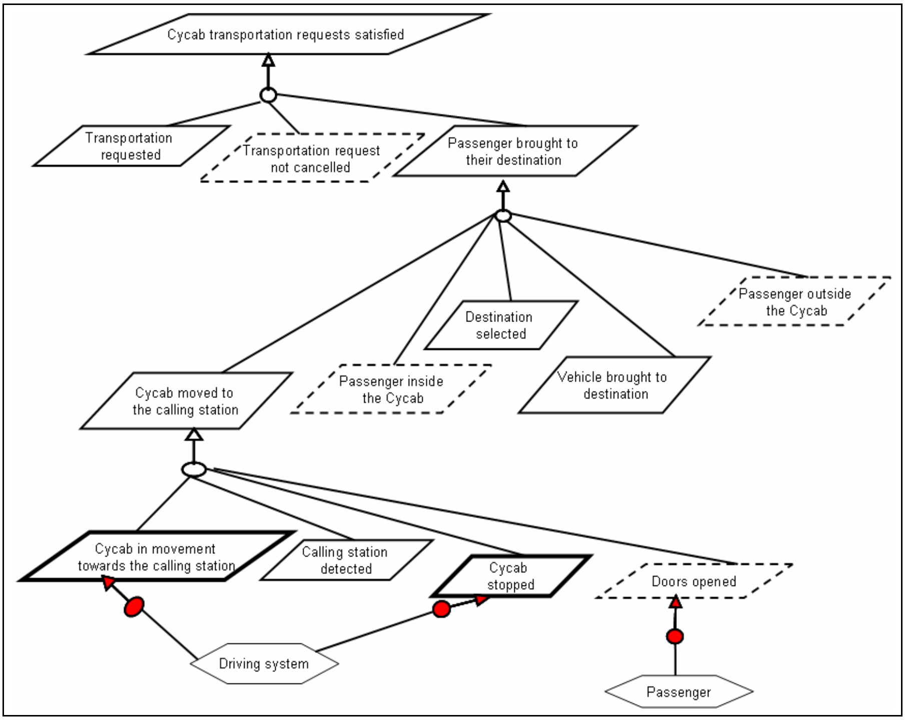

Cycab transportation requests satisfied
Transportation requested
Transportation request not cancelled
Passenger brought to their destination
Destination selected
Passenger outside the Cycab
Cycab moved to the calling station
Passenger inside the Cycab
Vehicle brought to destination
Cycab in movement towards the calling station
Calling station detected
Cycab stopped
Doors opened
Driving system
Passenger

Fig. 9. Requirements model according to Situation 1

Variability has also been studied in domain analysis. Among domain analysis methods, the FODA method [7], [8] has been the first one to propose a model capturing commonalities and variabilities in the form of Features; this model also represents dependencies between features and constraints to combine them. The feature model highlights, in the form of hierarchy sets, the characteristics that discriminate systems in a domain. Many extensions have been proposed to the feature model: For instance, the additional relation in [24] and the cardinality-based features model in [25].

In our work, we are interested in applying the concept of variability at a goal-based requirements level [16], [17]. The approaches such as Foda or SPL do not deal with variability at goal-oriented requirement level. However, other approaches have dealt with variability at requirements engineering stage [10], [11], [21]. The approach proposed in [10] uses an Or-decomposition link of goals to identify all possible options in goal model for a single system. In [21], the authors have explored how goal models and aspect concepts can be applied to deal with variability.

In our approach, we propose to introduce the concept of facet and variant in the goal model (and in any model of KAOS) because we believe that the KAOS OR-Refinement link is not sufficient to acquire variability effectiveness. The concept of facet allows representing viewpoints or dimensions of a domain that help to classify and organize domain knowledge. The notion of facet has been used in library science to define classification methods for the domain [14]. It is important to emphasize that the couple <facet-variant> does not only enable to represent a richer variability while

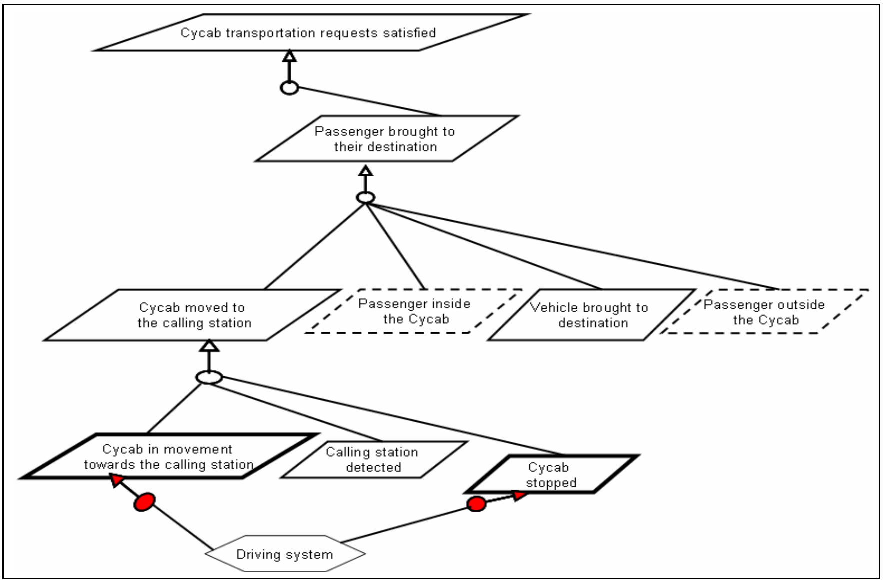

Cycab transportation requests satisfied
Passenger brought to their destination
Cycab moved to the calling station
Passenger inside the Cycab
Vehicle brought to destination
Passenger outside the Cycab
Cycab in movement towards the calling station
Calling station detected
Cycab stopped
Driving system

Fig. 10. Requirements model according to Situation 2

reducing combinatory explosion, but is also an effective support when building a specific requirements model.

6 Conclusions

In this paper, we have presented an approach based on the KAOS method with the intention of introducing variability in goal based requirements. The main element of this approach is a process which enables an application engineer to build a requirements model from a generic model according to some variants selected from a variant model.

The benefit of such an approach is that it permits to elaborate a requirements model in a flexible way and to adapt it according to new situations or to frequent requirements changes.

We are currently validating this approach through a software prototype [5]. The next activity will consist in formally expressing the requirements model in order to make it easier to map the software design from the requirements model.

References

1. Anton, A.I.: Goal based requirements Analysis. In: The 2nd Int. Conf. on RE (1996)
2. Baille, G., et al.: The INRIA Rhônes-Alpes Cycab, Technical Report N°0229 (1999), ISSN 0249-0803
3. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-oriented Requirements Acquisition. Science of Computer (1993)

4. Lamsweerde, A.: Goal-oriented Requirements Engineering: A guided tour. In: Int. Symposium on Requirements Engineering, Toronto (2001)
5. Gnaho, C., Al: A Tool for Modeling Variability at Goal Level. In: Third Int. Workshop on Variability Modelling of Software-intensive Systems, VaMoS (2009)
6. Halmans, G., Pohl, K.: Communicating the variability of a software product family to customers, Software and System Modeling. Springer, Heidelberg (2003)
7. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature-oriented domain analysis (FODA) feasibility study CMU/SEI-90-TR-21, Univ. Pittsburgh, Pennsylvania (1990)
8. Kang, K., Kim, S., Lee, J., et al.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering 5, 143–168 (1998)
9. Lamsweerde, A.: From Systems Goals to Software Architecture. In: Bernardo, M., Inverardi, P. (eds.) SFM 2003. LNCS, vol. 2804, pp. 25–43. Springer, Heidelberg (2003)
10. Liaskos, S., Lapouchnian, A., Yu, Y., Yu, E., Mylopoulos, J.: On Goal-based Variability Acquisition and Analysis. In: 14th IEEE Int. Conf. on Requirements Engineering (2006)
11. Liaskos, S., Jiang, L., Lapouchnian, A., Wang, Y., Yu, Y., Sampaio do Prado Leite, J.C., Mylopoulos, J.: Exploring the Dimensions of Variability: a Requirements Engineering Perspective. In: 1st Int. Workshop on Variability Modelling of Software-intensive Systems, VaMoS (2007)
12. Objectiver Requirement Engineering tool, `http://www.objectiver.com/`
13. Parent, M.: Automated public vehicle: a first step towards the automatic highway. In: The Proc. Of the World Congress on Intelligent transport systems (October 1997)
14. Prieto-Diaz, R.: Implementing Faceted Classification for software reuse. Communications of the ACM 34(5) (1991)
15. Rolland, C., Souveyet, C., Ben Achour, C.: Guiding Goal Modelling Using Scenarios. IEEE Transactions on Software Engineering (1998)
16. Semmak, F., Brunet., J.: Variability in Goal-oriented Domain Requirements. In: Morisio, M. (ed.) ICSR 2006. LNCS, vol. 4039, pp. 390–394. Springer, Heidelberg (2006)
17. Semmak, F., Al: Extended KAOS to support Variability for Goal oriented Requirements reuse. In: Int. Workshop Model Driven Information Systems Engineering with Caise 2008 (2008)
18. TACOS project, ANR-06-SETIN-017, programme SETIN 2006, `http://tacos.loria.fr`
19. Van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in Software Product Lines. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (2001)
20. Yu, E.: Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In: 3rd IEEE International Symposium on Requirements Engineering, pp. 226–235. ACM Press, New York (1997)
21. Gonzales-Baixauli, M.A., Laguna, J.C.: Sampaio do Prado Leite Using Goal-models to analyse variability. In: 1st Workshop on VAMOS, Limerick, Ireland (2007)
22. Broy, M.: Requirements Engineering for Embedded Systems. In: The proceedings of Femsys (1997)
23. Bachmann, F., Bass, L.: Managing variability in software architecture. ACM Press, NY (2001)
24. Griss, M., Favaro, J., d'Alessandro, M.: Integrating feature modeling with the Rseb. In: Proc. Of the 5th int. Conference of Software Reuse (1998)
25. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process Improvement and Practise 10(1), 7–29 (2005)

Orthographic Software Modeling: A Practical Approach to View-Based Development

Colin Atkinson, Dietmar Stoll, and Philipp Bostan

Institute of Computer Science, University of Mannheim
68131 Mannheim, Germany
{Atkinson,Stoll,Bostan}@informatik.uni-mannheim.de

Abstract. Although they are significantly different in how they decompose and conceptualize software systems, one thing that all advanced software engineering paradigms have in common is that they increase the number of different views involved in visualizing a system. Managing these different views can be challenging even when a paradigm is used independently, but when they are used together the number of views and inter-dependencies quickly becomes overwhelming. In this paper we present a novel approach for organizing and generating the different views used in advanced software engineering methods that we call Orthographic Software Modeling (OSM). This provides a simple metaphor for integrating different development paradigms and for leveraging domain specific languages in software engineering. Development environments that support OSM essentially raise the level of abstraction at which developers interact with their tools by hiding the idiosyncrasies of specific editors, storage choices and artifact organization policies. The overall benefit is to significantly simplify the use of advanced software engineering methods.

1 Introduction

In an effort to accommodate the ever growing demand for more complex and feature-rich applications, and to develop software in more cost effective and systematic ways, in recent years the IT industry has experimented with various new paradigms for software engineering. Chief amongst them are model-driven development, component-based development, product line engineering (PLE) and aspect-oriented development. They each use a different combination of abstraction and (de)composition techniques to break a large complex system or family of systems into manageable pieces. However, one thing they all have in common is that they increase the number of artifacts or "views" involved in the software engineering process. Model-driven development introduces views at various levels of platform specificity together with transformations between them, component-based development introduces internal and external views of software components as well as their compositions, PLE introduces family wide and product specific views of systems and the feature choices that relate them, and finally aspect-oriented development introduces a view of functional and cross cutting elements of a software system and how they are woven together.

Even when used alone, therefore, these new methods increase the need to define and manage multiple views, but when two or more of these methods are used together, the number of views quickly explodes out of control. The current view generation and

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 206–219, 2010.

management approaches of most case tools are wholly inadequate to deal with these challenges, however. First, few if any have a concrete idea of what views should be created in a development project, what contents they should contain and how they should be related. Secondly, most have a fixed, hardwired definition of what view types are possible (e.g. UML diagrams, annotated code and aspects etc.). Thirdly, most provide ad-hoc techniques for maintaining consistency between views and do so on a limited, pair wise basis. While this might be feasible for small numbers of views, it does not scale to large numbers.

We believe that one of the next major steps forward in software engineering will be driven by tools and methodologies that provide a systematic and flexible approach to view generation and management. To do this the next generation of tools needs to support:

1. Dynamic View Generation
2. Dimension-based View Navigation
3. View-oriented Methods

We are currently developing an approach for doing this that we refer to as Orthographic Software Modeling (OSM). The name is motivated by the orthographic projection approaches used in CAD tools to visualize physical objects. In this paper[1] we describe the basic idea behind OSM and explain how the tool that supports it meets the three basic requirements outlined above. We then present a small case study that illustrates how OSM might work in practice.

2 View-Based Software Engineering Method

To provide the context for the first two requirements (dynamic view management and dimension based view navigation), in this section we first provide an overview of the view-based method that we are currently trying to support. This is an updated version of the KobrA method [1], KobrA 2.0 [2], enhanced to exploit UML 2 and the latest software implementation technologies such as web services. In this section we will simply refer to this method as KobrA, with the understanding that we are referring to the latest version. This is just one of many possible methods that can be supported by OSM tools, however. In fact, in principle, any method could be supported by an OSM tool, since every method requires some kind of view of the software to be manipulated (e.g. the source code).

2.1 KobrA

KobrA was developed with the goal of integrating model-driven development (MDD), product line engineering (PLE) and component-based development (CBD) [3] in a systematic way. To do this KobrA explicitly identifies three fundamental dimensions, each representing an aspect of a system's description that could vary independently of the others.

[1] This is a modified version of the paper presented at the 4th International Conference on Evaluation on Novel Approaches to Software Engineering (ENASE '09), Milan, Italy, May 9-10, 2009.

The core dimension is the composition dimension in which (de)composition of the system into components is elaborated (CBD). The second most important dimension is the abstraction (or platform specificity) dimension in which the system is described at different levels of platform specificity (MDD). The final dimension is the genericity dimension in which the system is described in both generic (i.e. family level) and specific (i.e. application level) forms (PLE). In principle, each dimension can vary independently, i.e. they are orthogonal to one another. The key idea in KobrA is that these conceptually orthogonal "dimensions" should be made explicit and that different views of the system should be located somewhere in this space. As a UML based method, KobrA also defines strict principles for using UML diagrams to view different aspects of a component from different perspectives.

As shown in Figure 1, the views of a component are separated into two distinct groups – those showing properties that can be seen from the outside by users of the component (i.e. from a black box perspective), and those showing the properties that can be seen from inside by the developer of the component (i.e. from a white box perspective). The former group is known as the specification and the latter as the realization. The black box and white box perspectives of a component have further substructure as also represented in Figure 1. Basically each contains three different views, or projections, which describe different kinds of information about the component. The structural projection shows structural information using UML class diagrams. The operational projection shows information about the functionality of each operation modeled in the form of operation specifications and interaction diagrams. The behavioral projection focuses on the sequencing and algorithmic properties of the component as manifest by state charts and activity diagrams. Although these were not viewed as "dimensions" in the original version of KobrA, during the development of KobrA 2 it was realized that the separation between black box and white box perspectives, and the separation of information into different projections (or different aspects of description) that can vary independently, represent dimensions in the sense used above.

Generic components (for PLE) are described by adding an additional view, known as the decision model, to the views already illustrated in Figure 1. The decision model describes the different variants of the system in terms of decisions that the user can make to decide which features he or she would like. Dependencies between decisions can be specified using OCL constraints [4].

For example, it can be specified that one decision should automatically be resolved when another one is resolved. A decision is further described by its possible *ResolutionSet*. A resolution set represents the range of values (e.g. Boolean, Range or ValueSet) that can be assigned to a decision when defining a particular variant. The effects of each possible resolution value on the other views are defined, such as the removal of model elements like classes, methods or even whole components. With a generic component, every model element that is variable and represents a variation point is marked by the stereotype <<variant>>. To create specific components from the generic component, the associated decision model needs to be resolved by specifying the appropriate values of the available ResolutionSet.

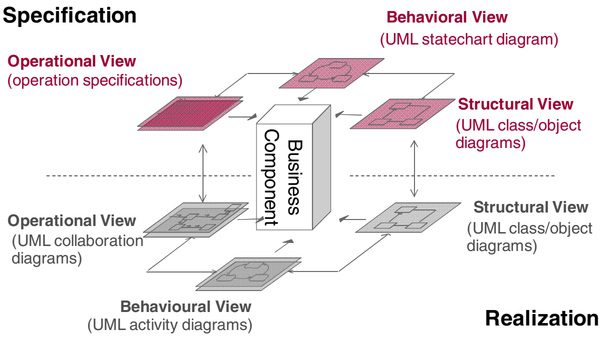


Fig. 1. Component description views

2.2 KobrA Dimensions

Thanks to its principles of separating concerns and defining how different UML/OCL diagrams can be used to portray different views of a system, KobrA is an ideal basis for OSM. Indeed, our vision of an OSM modeling environment was originally motivated by our aim to develop a tool to support KobrA. Although the original version of KobrA did not take the idea of dimensions to its logical conclusion and identify each independently-varying criteria as a true dimension (e.g. the encapsulation levels and the projections), this is an easy step to take. The modeling principles embodied by KobrA naturally suggest the following five dimensions:

Composition. This dimension covers the (de)composition of components into subcomponents. Selecting a point along this dimension corresponds to the specification of the component or subcomponent which is currently being worked on.

Abstraction. This dimension addresses the platform specificity of a view. In other words, selecting a point on the abstraction dimension identifies the level of detail at which the component is being viewed. In principle there can be multiple points along this dimension, but the most important are the platform independent model (PIM), platform specific model (PSM) and implementation. The KobrA approach is mainly concerned with the PIM level.

Encapsulation. The “public” encapsulation option provides a black box view of the component. It describes all externally visible properties of a component and thus serves as its requirements specification. The “private” encapsulation of a component provides the white box view, and thus includes all the information in the black box view.

Projection. This dimension deals with the types of information contained in a view. The projections currently available are the structural, operational, behavioral and variational projections. The latter contains the decisions that determine what aspects of a component’s description are included in a specific variant and which parts are not.

Variant. This dimension enumerates the different variants of a system when following a product line approach (e.g. “Mobile Edition”, “Enterprise Edition”). In addition, the generic variant includes all possible features of the system and a decision model for each component. Each particular variant is then associated with specific decision resolution models, which are resolved in application engineering.

The KobrA method was designed before the notion of OSM was developed as a notion for supporting and characterizing view-based development environments, and is theoretically independent of it. However, we believe that KobrA needs to be supported by such an environment to be used effectively. In the following two sections we explain how the two key ideas for achieving this are realized.

3 Dynamic View Management

Early work on view-based software engineering came to the conclusion that it was not cost effective to derive views of software artifacts dynamically from a single underlying model or representation [5]. However, since this work was carried out, the power of processors and availability of storage has increased tremendously, and new technologies have emerged that specialize in performing transformations between different models – so called model-driven development. Software development is also increasingly becoming a collaborative activity, with multiple developers working concurrently on different aspects of a system on different computers. We believe these developments change the situation, and make the dynamic, on-demand generation of views practicable.

Our approach is based on the idea of creating a Single Underlying Model (SUM) that contains all information about the system currently available, and separate view models that contain the information to be displayed in specific views of the system. When a new view is opened it is generated dynamically from the SUM via the appropriate transformation, and when new information about the system is added to the view this is eventually added to the SUM. This basic principle of “on-demand” generation from a single underlying model is depicted in Figure 2.The boxes and arrows within the ellipse in the center of Figure 2 are meant to represent the data elements making up the SUM. Each of the four shown views is generated dynamically by means of the appropriate transformation whenever the developer wishes to see the view. No effort is needed to keep views consistent on a pairwise basis, because as long as each is consistent with the SUM, it is consistent with all the other views. We also regard traditional source code as a view, just like any other. Since high-level views (e.g. models) can be generated at any time, for example after changes have been added via a code view, this approach provides inherent support for “round trip” engineering.

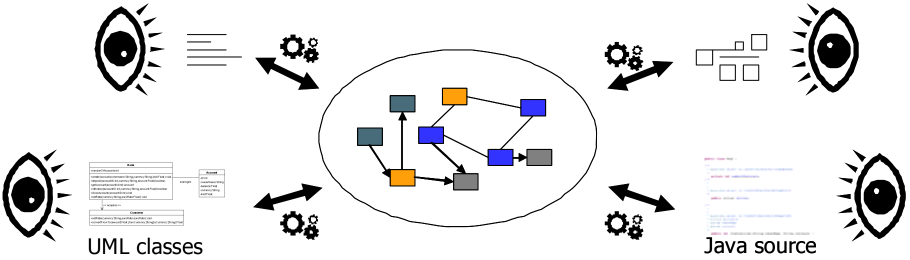


Fig. 2. On-the-fly generation of Views

3.1 SUM and View Metamodels

The metamodel for the SUM defines the concepts used to describe the properties of a software system in the chosen method. The SUM metamodel is thus method specific, and in this case therefore captures the concepts needed by the KobrA method. Since this is a UML centric method, it makes sense for the SUM to be defined as a specialization of the UML. As an example, the UML 2 *Association* metamodel element is reused. It is further specialized into *Creates*, *Acquires* and *Nests*, which describe relationships between KobrA Components and KobrA Classes. *Acquires*, for example, is then constrained using OCL to ensure that only a KobrA Component can acquire another Component (or KobrA Class) and that a KobrA Class is prohibited to acquire anything. Elements from UML 2 that are not needed for our development method are also excluded using OCL constraints. New elements introduced in the metamodel are specialized from UML 2 elements and then further described and constrained.

From the point of view of standard MDD technology, a view is a normal model which just happens to have been generated dynamically for the purpose of allowing a user to see the system from a specific viewpoint. The only other requirement is that the model has to have an associated concrete syntax by which it can be rendered. This can be a graphical syntax, such as the UML, or a textual syntax, such as a programming language like Java. A view can be represented in a general, standardized modeling language like the UML that can be rendered by many tools, or it can be represented in a highly specialized language that is specific to one single tool. Since the transformation technology used to generate and update the views can work with any source and target metamodels, there is no theoretical constraint on what languages and tools can be used. From a tool integration point of view, however, it is more practical to use rendering engines and editors that are part of the same IDE family (e.g. Eclipse [6]) and/or that work on related languages (e.g. EMF or OMG metamodels). Like the SUM, the view metamodels are all accompanied by extensive OCL constraints that define the concrete well-formedness rules that instances of the metamodels must obey.

3.2 “On-the-fly” View Generation

The key technology that makes the dynamic generation of arbitrary views practical are the transformation languages and engines provided by MDD environments. These allow users to add new views to their environment in a straightforward way by defining how a view is generated from the SUM and what well-formedness rules it must adhere to. While the writing of transformations is a non-trivial task, we believe that it will involve far less effort than the consistency checking and verification activities (e.g. inter-view consistency checking) that would otherwise have to be performed manually. Any convenient transformation language can be used. We currently use the ATLAS Transformation Language (ATL) [7].

For “read only” views it is only necessary to define a transformation in one direction – from the SUM to the view. However, when views can also be edited, it is necessary to define reverse transformations as well. The role of the reverse transformations is to add new information about the system entered via a view to the SUM. This takes place whenever the developer working with the view indicates that they would like to “commit” the changes that they have made to the SUM. Before performing the reverse transformation and/or while editing, constraints for the view can be checked to make sure that it is well-formed. After the transformation, checking the SUM against consistency rules can verify that the transformation worked as expected. Changes can be made at various levels of granularity, from very fine-grained changes such as a name-change to very large-grained changes where a large piece of the model is modified. The choice reflects a balance between efficiency and the risk that changes may be inconsistent with those made by another developer working on a different view.

Finally, in order to keep track of the history of changes made to the SUM, a transactional versioning system is needed. Again this can be based on a standard versioning system such as CVS or subversion, or on versioning systems specifically tailored to the SUM.

4 Dimension-Based View Navigation

The Dimension Based View Navigation scheme is perhaps the most novel aspect of the orthographic modeling approach. It aims to mimic the way that users of CAD tools can navigate around the views of a physical object by picking the different perspectives and viewpoints from which they wish to see the object. Each view can be thought of as occupying a single cell in a multi-dimensional cube, which is selected by picking a position in each dimension. Figure 3 shows a schematic picture of such a cube, but only with three dimensions. One dimension has two positions to choose from, one has three positions to choose from and one has four. In general, the cube is multi-dimensional and each dimension can have an unlimited number of positions.

The advantage of such a navigation approach is that it frees the developer from having to work with the navigation tree of each individual tool used to view each type of artifact. With dimension-based navigation each view is identified by its location in the dimension space rather than its location in a specific tool's artifact tree. Different native tools are still used to work with each view, but these are invoked automatically by the OSM tool as needed.

We define two different roles in OSM: Developer and Methodologist. A developer uses the dimension based navigation and manipulates the software system through views, while the methodologist creates a navigation and view configuration for a specific software development method, such as KobrA 2.

4.1 Dimensions

A dimension is any aspect of a software system's description that can vary more or less independently of other aspects, depending on how orthogonal the dimension is in relation to the other dimensions. If all dimensions were fully orthogonal to each other, all combinations of options along all different dimensions would be associated with a view, i.e. all cells of the cube could be shown to the developer as a view. However, dimensions are not always 100% orthogonal – there may be combinations of dimension options that don't make sense and thus are impossible to show to the developer. In other words, some cells might be empty (have no view).

The multi-dimensional cube is manifested in the GUI as a set of separate lists, each holding the different options for a given dimension. To select a cell, the user therefore simply has to select an option from each list. Figure 3 (right-hand side) shows how orthographic navigation around KobrA artifacts might be supported from a GUI perspective. Each list on the left hand side represents the possible choices in each dimension, and the diagram on the right hand side represents the actual view.

4.2 Language and Notation

The top five dimensions in the IDE shown in Figure 3 represent the KobrA-oriented views that were described in section 2. In a sense they define the logical views of the system supported in KobrA, because they characterize the basic nature of the information that can be seen in each view, but they do not deal with how it is presented. This is the job of the last two dimensions. They are an OSM tool's mechanism for dealing with the different language and notation options that can be used to depict a logical view.

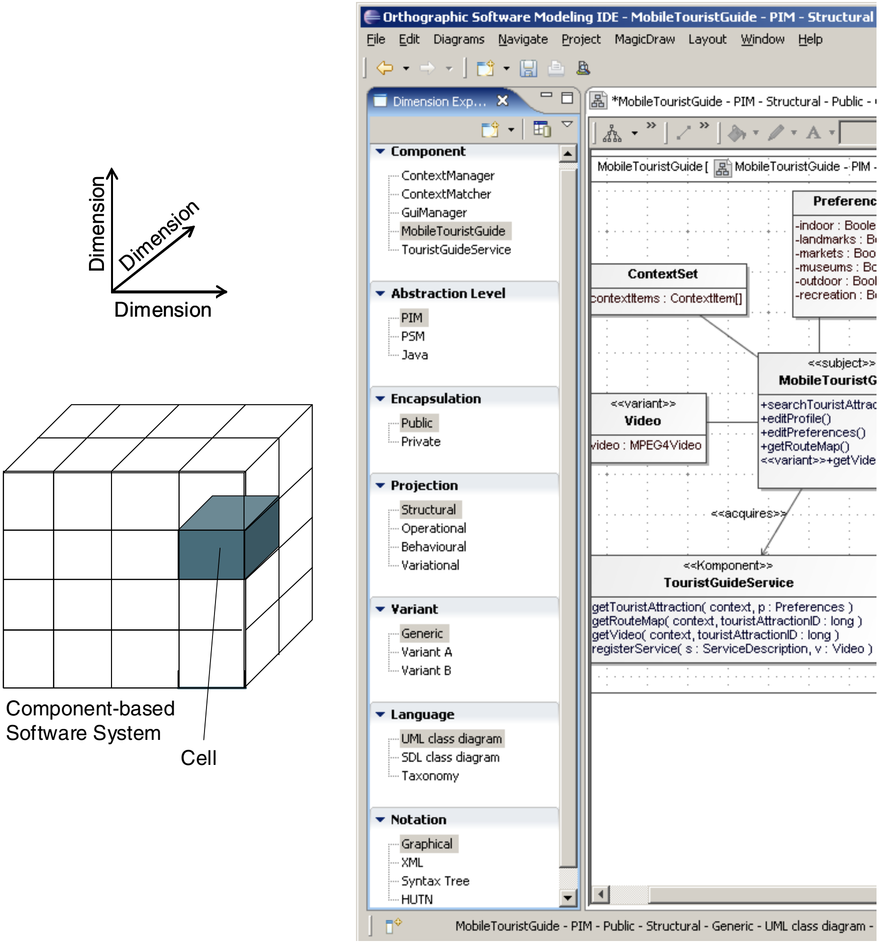


Fig. 3. Component navigation and Orthographic Software Modeling IDE

The language dimension identifies the basic language used to depict a view. We use "language" here in the general sense used in the "domain specific language" field to represent any formal language for representing information. This includes programming languages like Java and modeling languages like UML. Since it was oriented towards the UML, the original KobrA method envisaged that UML classes would be used to represent the structural view. However, in general, any suitable structural modeling language could be used such as SDL or OWL.

Identifying a language still does not provide all the information needed to depict a view because most languages can be rendered using various concrete syntaxes. For example, as well as the well known graphical syntax, UML diagrams can also be represented in a textual form such as XMI or a human readable textual notation such as HUTN. Even programming languages like Java can be rendered in various forms, for example in XML or JavaDoc. The final dimension therefore defines the concrete notation used to depict a view.

Internally, the OSM tool keeps track of which default editor should be used to depict each view, so that once language and notation choices have been made, the system can automatically invoke the editor needed to show the view on the right hand side. In Figure 3, MagicDraw [9] is used for a UML class diagram. Since OSM provides inherent support for identifying languages and notations when working with views, it provides a natural metaphor for integrating domain specific languages into software engineering environments.

4.3 Tailoring

Software companies may tailor existing configurations or create new configurations according to their specific software development method (explained in detail in the next section). A common tailoring scenario is to add new languages and notations to an existing configuration. One could for example add a component descriptor editor which manages general component information for the PIM – Public – Structural view. A new language and notation can be added in the simplest case by adding a new metamodel for the language, a notation and associated transformations for the generation and synchronization of the editor's data. If the language enhances the development approach by adding new concepts, new elements and new consistency checks might also have to be added to the core metamodel of the SUM. Also, whole new dimensions could be added to existing approaches, such as a version dimension, where each element represents a different version of the component-based system.

4.4 OSM Configuration for Specific Software Development Methods

The dimensions and their ordering/dependencies capture the characteristics of a development methodology. A methodologist may create a configuration for software development method (e.g. KobrA 2) by

- defining the dimensions and dimension elements (this corresponds to the lists in the GUI in Figure 3), and
- associating them with views.

A dimension can be static, dynamic, or mixed: A static dimension contains a fixed list of dimension elements (e.g. the Projection dimension in KobrA 2). In contrast the elements for a dynamic dimension (e.g. the Component dimension in KobrA 2) are retrieved from the SUM. An example for a mixed dimension is the Variant dimension, as there is always a "Generic" variant, but other variants depend on the current state of the SUM. The methodologist can also specify the rights of the developer, i.e. whether the developer is allowed to add, rename or delete elements. A developer might be allowed to add new dimension elements (e.g. in the Variant dimension, he might add a "Professional Edition" variant).

Once the methodologist has specified which dimensions and dimension elements exist, he can provide a mapping from dimension element combinations to views. In our current prototype, this is done with a mapping table. Specifying the mapping with a table as described below is only one possibility. In the future, OCL-like constraints might be used.

The table contains a column for each dimension and another one for a specific transformation/view combination. Each row is an association of a dimension combination (or multiple combinations) with a view. A transformation/view combination might be parameterized, e.g. with the currently selected component, so a "*" symbol may be used to express that a mapping is valid for any element in a dimension. A mapping table might look like this:

In a fully orthogonal configuration, there would be no empty cells (for which no views can be generated), but in most configurations, there are. In the case of the mapping table, all combinations that are not listed correspond to cells that don't have a view.

It is a design decision of the GUI how to deal with empty cells. To this end, we allow dependencies to be defined between dimensions, i.e. the selection of elements in a dimension of higher precedence might restrict or change the possible selections in dimensions of lower precedence (e.g. by marking them as non-selectable or only showing selectable elements). The higher a dimension is listed on the left hand side (of the GUI), the higher its precedence.

5 Case Study

The case study is based on a context-aware mobile tourist guide that consists of a mobile client and a server-side tourist guide service. The server stores information about tourist attractions and service descriptions that can be registered at the service. The goal of this section is to show the various kinds of views that can be used to visualize the system, and how they are reached via the dimension-based navigation metaphor.

5.1 Mobile Tourist Guide – Black Box

We start the case study by developing artifacts for the black box model of a component at the PIM level, so we select *Public* from the Encapsulation dimension and *PIM* from the Abstraction dimension.

In the Component dimension we create the top level component *MobileTouristGuide* as a new dimension element. Since we start with the structural description of the publicly visible parts, the Projection dimension is set to *Structural*. In the Variant dimension, we select the *Generic* version of the *MobileTouristGuide*. Once this cell of the conceptual cube has been selected, the system offers an appropriate "editor" for the view – in this case a UML class diagram editor.

Figure 4 shows the UML class diagram. It features the *MobileTouristGuide* which is the "subject" according to KobrA's principle of locality [1]. The specification of a component only includes associations to externally visible components (marked in the association by the stereotype <<acquires>>).

Furthermore, this specification contains an association to a variant class (Video) and a variant method (getVideo) marked by the stereotype <<variant>>. The stereotype can be applied to whole components, to single methods or to variables. These represent variation points that are administrated by the decision model which is also presented in this section.

Every method of the *MobileTouristGuide* can be described further with an *Operation Specification* editor. The operation specification is publicly visible and is part of the *Operational* projection. It is described in tabular form as shown in Table 1. The *Operation Specification Editor* of the IDE offers some additional features like syntax checking for OCL pre- and postconditions (if specified in OCL).

Table 1. Mapping cells of the cube to transformations and views

Comp.	Abstr.	Encaps.	Projection	Variant	Lang.	Not.	Transformation/View
*	PIM	Public	Structural	*	UML	Graph.	...
*	PIM	Public	Operational	*	OpSpec	Tabular	...
...	...	...	...	...	...	...	...

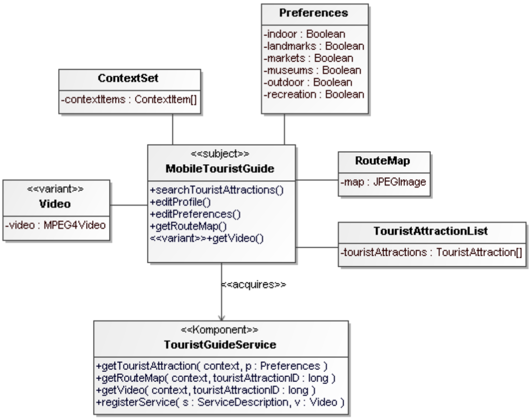


Fig. 4. MobileTouristGuide – PIM – Public – Structural – Generic – UML Class Diagram – Graphical

5.2 Mobile Tourist Guide – White Box

The artifacts of the *public* encapsulation (i.e. black box information hiding) view of a component describe what a component does, i.e. what services it offers to users. The artifacts of the *private* encapsulation describe how the promised functionality is realized, including interactions with (sub-)components. To see the white box views of a component we therefore need to set the *Encapsulation* dimension to *Private*.

Like the black box structural view in the previous subsection, in the white box structural view we can see the UML class diagram of the *MobileTouristGuide*, but this time with additional elements needed for the realization. The dimension Variant is included in the generic model using the stereotype <<variant>> as in the black box view to mark variation points that are used in the decision model for variants of a product family. In addition to the mentioned relationship <<acquires>> which expresses the fact that the subject needs the acquired component to fulfill its own mission, a new relationship, called <<creates>>, is possible in the *Realization* class diagram. This relationship declares that the subject is fully responsible for the sub-component, i.e. its creation and destruction.

The *operational* Projection in the white box views contains UML communication or sequence diagrams which show how a particular function interacts with other artifacts of the system. The behavioral projection shows the algorithms of functions using a UML activity diagrams.

Table 2. MobileTouristGuide – PIM – Public – Operational – Generic – Operation Specification – Tabular

Name	searchTouristAttractions
Description	Searches for tourist attractions depending on the user's preferences and the current context (e.g. location, time, weather)
Receives	-
Returns	TouristAttractionList
Sends	TouristGuideService.getTouristAttraction()
Reads	ContextSet, Preferences
Changes	TouristAttractionList
Body	-
Precondition	Preferences have been set up, Context Sources are available
Postcondition	TouristAttractionList contains suitable attractions

The decision model is associated with the *Variational* projection and, as the decisions are not yet resolved, the *Generic* variant. As mentioned before, the variation points of a component are administrated via the component's local decision model.

In Table 2, the private encapsulation decision model of the *MobileTouristGuide* is shown for the structural projection. It contains questions that have to be resolved in order to create actual product versions. The example shows a decision that is related to two variation points and another decision related to three variation points. For each, the given *ResolutionSet* defines the possible values that can be assigned within a decision. The effects clause specifies which action is performed dependent on the resolution value. In this example, effects are applied to the UML class diagram shown above and the UML communication diagram for *searchTouristAttractions()*.

Table 3. MobileTouristGuide – PIM – Private – Variational – Generic – Decision Model – Tabular

	1	2
Description	Is the mobile client device capable of playing videos?	What visibility should be assignable to context items on the mobile client? (Public context items are transmitted to the server, private context items not)
Component	MobileTouristGuide	MobileTouristGuide
Encapsulation	Private	Private
Projection	Structural	Structural
Constraints	--	--
ResolutionSet	Boolean	ValueSet {Public, Private, PublicAndPrivate}
Effects	*ResolutionValue: **True*** (1) remove stereotype <<variant>> at Class Video (2) remove stereotype <<variant>> on operation MobileTouristGuide.getVideo() *ResolutionValue: **False*** (1) remove Class Video (2) remove operation MobileTouristGuide.getVideo()	*ResolutionValue: **Public*** (1) remove <<Komponent>> PrivateContextMatcher (2) remove association PrivateContextMatcher-MobileTouristGuide *ResolutionValue: **Private*** (1) remove stereotype <<variant>> at Komponent PrivateContextMatcher *ResolutionValue: **PublicAndPrivate*** (1) remove stereotype <<variant>> at Komponent PrivateContextMatcher
Stakeholder	Application Engineer	Application Engineer

6 Conclusions

In this paper we have presented a new paradigm for organizing the many views that need to be manipulated in modern software engineering methods and have outlined the key features of a tool to support it. Known as Orthographic Software Modeling, the approach mimics the orthographic projection principle used in CAD tools to visualize physical engineering artifacts. By doing so, it raises the level of abstraction at which developers interact with tools by hiding the idiosyncrasies of specific editors and tools. We have built a prototype version of this tool in Eclipse, using a heterogeneous mix of well known editors to render and manipulate specific views, such as the Eclipse Java editor for Java views, and MagicDraw for UML-oriented views. We are currently implementing a more generic view generation engine with extended configuration and plug-in capabilities.

To the best of our knowledge, there is currently no approach that combines on-demand view generation with the approach of dimension-based navigation. Glinz et al. [10] feature a systematic, hierarchical modeling approach and a tool with a fisheye zooming algorithm that allows models to be visualized with different levels of detail. Although are some similarities such as the classification in structural and behavioral views, the development approach mainly uses non-UML views for structural and behavioral aspects of a software system while the KobrA method makes heavy use of UML diagrams. Its navigation concept also differs from dimension-based navigation. The importance of view-based modeling is reflected by various approaches, starting from the Zachman Framework [11] to approaches like the Reference Model of Open Distributed Processing (RM-ODP) [12]. However, most only implicitly or informally define consistency rules between the views and don't provide a flexible navigation mechanism. The Zachman Framework gives only short hints about inter-view consistency and while

RM-ODP is one of the few approaches that allow the definition of correspondences between views, their practical realization is the subject of current research [13].

Our approach includes an extensible navigation concept where customized dimensions, dimension elements, languages and notations can be integrated in a systematic and straightforward way. It also allows the users to define a dominance hierarchy between the dimensions such that dimensions near the top of the architecture influence what is available for dimensions lower in the hierarchy. Indeed, it is possible that a choice in a higher level dimension may remove a lower dimension completely (because all the cells for that row are empty). We believe that this definition of dimension dominance relationships and dependencies actually goes a long way to capturing the core ideas that underpin a paradigm. For example, in an MDD focused project, the abstraction dimension would be the most dominant, whereas in a product line engineering oriented project, the variant dimension would dominate. We therefore claim that OSM tools are inherently able to support multiple paradigms, and thus can be used as a vehicle for bringing them together, or using them in different phases of development, whatever best fits the needs of the project in hand.

References

1. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., Zettel, J.: Component-Based Product Line Engineering with UML. Addison-Wesley Publishing Company, Reading (2002)
2. Atkinson, C., Bostan, P., Brenner, D., Falcone, G., Gutheil, M., Hummel, O., Juhasz, M., Stoll, D.: Modeling Components and Component-Based Systems in KobrA. In: Rausch, A., Reussner, R., Mirandola, R., Plášil, F. (eds.) The Common Component Modeling Example. LNCS, vol. 5153, pp. 54–84. Springer, Heidelberg (2008)
3. Szyperksi, C., Gruntz, D., Murer, S.: Component-Software – Beyond Object-oriented Programming, 2nd edn. Addison Wesley / ACM Press (2002)
4. Object Management Group: Object Constraint Language Specification, Version 2.0 (May 2006), http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf
5. Finkelstein, A., Kramer, J., Goedicke, M.: ViewPoint Oriented Software Development. In: Proc. of 3rd Int. Workshop on Software Engineering and its Applications, Toulouse (1990)
6. Eclipse Development Platform (visited May 2008), http://www.eclipse.org
7. The ATLAS Transformation Language (Visited May 2008), http://www.eclipse.org/m2m/atl/
8. Object Management Group, Human-Usable Textual Notation, v1.0 (April 2008), http://www.omg.org/cgi-bin/doc?formal/2004-08-01
9. MagicDraw (Visited May 2008), http://www.magicdraw.com
10. Glinz, M., Berner, S., Joos, S.: Object-oriented Modeling with Adora. Information Systems 27(6), 425–444 (2002), http://www.ifi.unizh.ch/req/ftp/adora.pdf
11. Zachman, J.A.: The Zachman Framework: A Primer for Enterprise Engineering and Manufacturing (Visited September 2009), http://www.zachmaninternational.com
12. ISO/IEC and ITU-T. The Reference Model of Open Distributed Processing. RM-ODP, ITU-T Rec. X.901-X.904 / ISO/IEC 10746 (1998)
13. Romero, J.R., Jaen, J.J., Vallecillo, A.: Realizing Correspondences in MultiViewpoint Specications. In: Proceedings of the Thirteenth IEEE International EDOC Conference, Auckland, New Zealand (September 2009)

Dynamic Management of the Organizational Knowledge Using Case-Based Reasoning

Viviane Santos, Mariela Cortés, and Márcia Brasil

State University of Ceará
Av. Paranjana, 1700, Cep 60.740-000, Fortaleza, CE, Brazil
{viviane.almeida,mariela,marcia.abrasil}@larces.uece.br

Abstract. Software process reuse involves different aspects of the knowledge obtained from generic process models and previous successful projects. The benefit of reuse is reached by the definition of an effective and systematic process to specify, produce, classify, retrieve and adapt software artifacts for utilization in another context. In this work we present a formal approach for software process reuse to assist the definition, adaptation and improvement of the organization's standard process. The Case-Based Reasoning technology is used to manage the collective knowledge of the organization.

1 Introduction

Considering the forward dependency between the development process quality and the product quality, the deep knowledge of the activities involved in the process and their management are critical factors for the organizational success.

In high level, the software development process defines a formal sequence of activities related to a set of artifacts, people, resources, organizational structures and constraints for turning user requirements into software. This knowledge captures the guidelines to drive software development in a specific domain and/or context.

The definition of a process for software development is a complex task since it requires experience and combines the knowledge of diverse technological and social aspects. The utilization of standards for the process definition [1][2][3][4] is recommended in norms, processes and maturity models. However, the process model must be adapted to fit the organization characteristics.

Software process models describe the organization knowledge and, thus, models that enhance successful experiences must be disseminated and recommended for reutilization across the organization [2][5]. The process consolidation is achieved through the systematic reuse and the incremental capture of feedback, looking for the continuous improvement.

The purpose of the process reuse technology is to support the process definition and improving on the basis of standard processes, according to norms and quality models, and learned experiences [6]. Dynamic and context-depending aspects of the knowledge in software development turn the Case-Based Reasoning approach (CBR) [7] useful as it provides a broad support for the dynamic management of the organizational knowledge and continuous incremental learning necessary for the definition and improvement of software development.

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 220–233, 2010.

In this work we describe an approach for definition and reuse of the organizational standard process, on the basis of models, standards, quality norms, and previous experience, in accordance with the organizational reality and characteristics. In addition, on the basis of the reuse process results, an adaptation process is presented. The CBR technology is used for the management of the repository and the retrieval of assets.

This work is organized as follows: in Section 2 the CBR technology is briefly explained. In Section 3 the process reuse using CBR is presented. In Section 4 a case study is illustrated. Finally, final considerations are presented.

2 Case-Based Reasoning

The CBR technology solves problems in a specific situation, through previous similar situations [8]. A case comprises a pair problem that describes the context of an actual case occurrence, and solution that presents the problem solution. Past cases are used to hint strategies to solve new similar problems [9].

A CBR system is composed by 4 basic elements [7]: knowledge representation, similarity measure, adaptation and learning.

The knowledge representation consists on the description of the relevant information for the cases, in order to assess the reuse.

The similarity measure establishes the global similarity degree between a base-case and a new problem under consulting. This measure is based on a heuristic method [9]. The retrieval process results in a set of ranked cases that are based on the global similarity measure.

The utility of base-case to solve a problem is proportionally related to the effort required to adapt it to fit the specific context [9]. This process involves knowledge reuse in problems solutions along the knowledge transference from the previous case to the actual case.

The ability to learn from early experiences is inherent in a CBR system. The continuous learning contributes to increase the system capacity to improve their interpretations to solve new problems. In this sense, feedback about the soundness and effectiveness about their interpretations is required.

3 Process Reuse Approach

The approach for process reuse is presented in Fig. 1 [10]. The main component is the Processes Assets Repository which is designed to store reusable process models and their feature-value representations. This representation involves a set of relevant properties to describe each case, and the values for these properties including numeric, text, pre-defined terms, etc. The utility of a specific case from the repository in the context of a new case under consulting is enabled using this representation.

The Search Engine uses CBR technology to retrieve similar cases through the similarity measurement on the basis of process and project features. Feature-value representations must be defined for the new case, and for the base-cases in the repository.

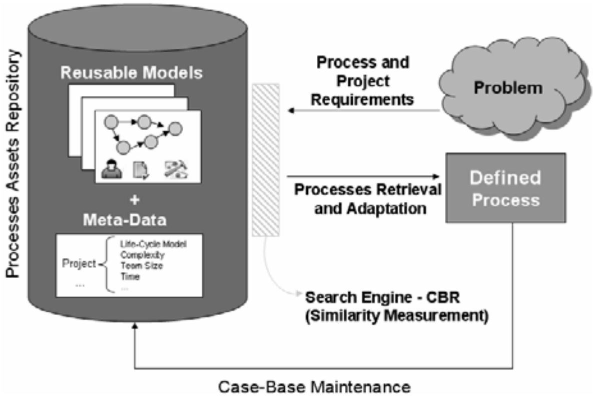

Processes Assets Repository
Reusable Models
+
Meta-Data
Project
Life-Cycle Model
Complexity
Team Size
Time
...
...
Process and Project Requirements
Problem
Processes Retrieval and Adaptation
Defined Process
Search Engine - CBR (Similarity Measurement)
Case-Base Maintenance

Fig. 1. Approach for process reuse

The reutilization involves the adaptation of a previous solution for a similar case, using an appropriate method [9]. After its adaptation and execution in the new project, the new case is evaluated in order to examine its effectiveness and capture the information to make the representation for this new case. Then, the new case can be included into the repository, increasing the feature-value representation.

3.1 Representation of Organizational Assets in the Repository

The reutilization of cases is enabled whenever the cases will be indexed and stored appropriately in the process assets repository, in such a way to make possible its efficient retrieval. The suitable representation of the process assets is a critical factor for the success of the method, since the similarity degree for the correct retrieval of the cases is measured on the basis of this representation. The similarity concept consists of establishing an estimate of the utility of a previous case stored in the repository, in the context of the current case on the basis of the observed similarity among the representations of both cases [7].

The similarity types are restrictions applied to the representation features, to establish its correspondence or co-occurrence among cases [11]. The similarity types used in this work are:

- Numeric (NUM). Positive integer or real numbers
- Qualitative for Fixed Items (QFI). Predefined Terms
- Qualitative for Variable Items (QVI). Registered terms with possibility of new items

The similarity between cases is based on the comparison of the features in the representation and the corresponding values. In this sense, several studies related to the classification of the process assets for reuse in other contexts can be cited [6][11][12] [13][14]. The representation of the assets in the repository proposed in this work is presented in Table 1. The features had been organized in agreement to the target in process and project features.

Table 1. Representation of the assets in the repository

Scope	j	Feature	Description	Similarity Type
Project	1	Life-Cycle Model	Project life-cycle model, such as Cascade, Iterative Incremental, Evolutionary, Spiral.	QVI
	2	Complexity	Project complexity: High (including critical and advanced functionalities), Medium (including feasible functionalities), Low (including simple functionalities).	QFI
	3	Size	Project size regarding the functionalities quantity: Large, Medium or Small.	QFI
	4	Team Size	Project integrant number.	NUM
	5	Time	Project duration in months.	NUM
	6	Software Engineering Knowledge	Knowledge level in Software Engineering: High (theory e practical), Medium (theory only), Low (none knowledge).	QFI
	7	Development Paradigm	Project development paradigm, such as Structured, Object Oriented, etc.).	QVI
Process	8	Development Model	Software development models, like RUP, XP, SCRUM, etc.	QVI
	9	Maturity Model	Maturity model, for example, CMMI, MPS.BR, etc.	QVI
	10	Maturity Level	Specific maturity level related to the maturity model specified previously. It can be, for example, 1 to 5 (CMMI and ISO/IEC 15504) or G to A (MPS.BR).	QVI
	11	Complexity	Process complexity based on the maturity levels: High (advanced levels), Medium (intermediary levels), Low (low levels).	QFI
	12	Process	Specific processes, such as Requirements Management, Project Planning, Quality Assurance, Configuration Management.	QVI
	13	Experience on Process Usage	Team's experience on software process usage: High (process used in more than 15 projects), Medium (process used in a range of 5 to 14 projects), Low (0 to 4 projects).	QFI

3.2 Retrieval Process

The most appropriate solution for the current problem is retrieved from the repository through similarity measurement. The greatest value in this measurement indicates greater similarity between the cases.

In CBR, several techniques can be applied for data retrieval. In [8] the algorithm to calculate the similarity is based on k-NN technique, where the global similarity (SIM) between two cases (*a* and *b*) is defined by the weighted sum of the local similarities (sim_j) for each feature (A_j).

$$SIM(a,b)=\sum_{j=1}^{n} w_j \times sim_j(A_j(a),A_j(b)) \quad (1)$$

The weight (w_j) reflects the relevance of a feature (A_j) concerning the similarity of cases. This factor is determined by the user and is measured by the values: High (100), Medium (50) and Low (10). The features considered more important for the problem resolution from the user's viewpoint, possess higher weights.

The base-cases considered as sufficiently similar can be proposed to the user as reuse candidates. Note that if the same weight is assigned to all the features, the base-case that attends the greater number of features must be the suggested one.

The local similarity is calculated in accordance with the similarity type of each feature. For features of NUM and QFI similarity types, it considers the computation of distance (d_j) between each feature values in the cases *a* and *b*, as presented in the formula **(2)**. Furthermore, for features of QVI similarity type, the local similarity is specially calculated through the formula **(6)** and is detailed later in this section.

$$sim_j = \frac{1}{1 + d_j(a,b)} \tag{2}$$

This measurement must be normalized [15] to avoid over influence of a metric by the great range of values of the features. The normalization process uses smallest and greatest values in the repository to linearly produce values between 0 and 1.

The distance between two features of numeric (NUM) or qualitative for fixed items (QFI) similarity type is calculated on the basis of a proportionality relation between the values, as expressed below:

$$d_j(a,b) = \left| \left(\frac{A_j(a) - \min(A_j)}{\max(A_j) - \min(A_j)} \right) - \left(\frac{A_j(b) - \min(A_j)}{\max(A_j) - \min(A_j)} \right) \right| \tag{3}$$

Finally, to calculate the distance between features of QVI similarity type, a taxonomy is used to hierarchically represent the relationships among the terms (Fig. A1). A hierarchical arrangement implies in a complex similarity relation, which is based on the object position in the hierarchy. In a taxonomy, as deeper the nodes are located in the hierarchy, greater is the similarity value. In the same way, whenever the nodes are closer to the root of the taxonomy the similarity goes to zero. The measurement for a new case may require the inclusion of new terms in the taxonomy.

Considering n_a and n_b different nodes in a taxonomy, the similarity between those nodes, $sim_j(n_a, n_b)$, proposed by [16] consists of:

$$sim_j(n_a, n_b) = \frac{2 \times N(n_{parent}, n_{root})}{N(n_a, n_{parent}) + N(n_b, n_{parent}) + 2 \times N(n_{parent}, n_{root})} \tag{4}$$

where $N(n_a, n_{parent})$ and $N(n_b, n_{parent})$ are the number of edges in the path from the corresponding nodes and their common parent in the hierarchy, and $N(n_{parent}, n_{root})$ is the number of edges among this common parent node and the root of the taxonomy. This measure is interesting because it considers the common parent and the root of the taxonomy o normalize the measurement among the nodes. [17]. If the nodes n_a and n_b are common, the formula **(4)** isn't applicable. In this case, the similarity between those nodes is considered equal to 1.

To illustrate this concept the equation **(5)** presents the similarity measurement between the models *D-CMM* and *RUP-ST* in the taxonomy of *Development Models*

(Fig. A1). The common parent node between those nodes is the *Hybrid* node, therefore the number of edges between them is 1 (one). And the number of edges from this common parent node to the root, *Development Models*, is also 1 (one). Below, there is the calculation of the similarity measurement between those nodes:

$$sim_j(n_a, n_b) = \frac{2\times 1}{1+1+2\times 1} = \frac{2}{4} = 0{,}5 \quad (5)$$

3.3 Adaptation Process

Adaptation involves the process to transform the retrieved results into an appropriated solution for the currently problem. The adaptation process can be realized following different approaches [8]. In this sense two approaches can be suggested: if the similarity measurement of the retrieved process in the top of the ranking is satisfactory[1], a minimal or null adaptation can be required. In other case, when none of the retrieved processes fulfills the requirements for the new case in appropriate manner, a compositional approach is proposed.

In this approach [18], the solution is composed by elements from different cases, on the basis of the most similar base-cases returned from the previous step. In this sense, the maximization of the local similarities of each feature can be used to build a new case in order to match the greatest level of similarity to meet the features of the current case, considering the dependencies and constraints among them. Fig. 2 presents, in general way, the returned base-cases with its features and similarity values against the new case.

Thus, the maximized global similarity of the new case (called *GlobalSIM*) is calculated through the maximization of the local similarity (*LSim*) of each feature from the retrieved base-cases, as presented below:

$$GlobalSIM = \sum_{i=1}^{N} \max\left(LSim(Ai_1), LSim(Ai_2), LSim(Ai_3), ..., LSim(Ai_M)\right) \quad (6)$$

where *N* represents the quantity of features and *M* the quantity of retrieved base-cases. In addition, already dependencies and restrictions between features from the same base-case must be considered in the composition of the new case. Similarly, features from different cases can be incompatible. These restrictions must be considered in the composition process. In this scenario, the following dependencies and constraints were identified:

- Development Model and Maturity Model;
- Maturity Model and Maturity Level;

For example, if the Maturity Model feature value required is SW-CMM or CMMI, the Maturity Level feature must be values from 1 to 5. Similarly, if the required Development Model is XP [19], neither Maturity Model nor Level Maturity can be used. In this sense, a recently published report of the Software Engineering Institute [20] considers the possibility of joint the use of agile development methods and CMMI best practices as a way to improve the performance.

[1] The satisfactory level is determined by the average of the base-case local similarities percent to represent the adherence of a base-case against the current case. The user can restrict the ranking result through specifying a minimum percent of satisfactory level, e.g. 60%.

	Local Similarity of Base-Cases related to the current case				Feature values
Feature	**Base-Case 1**	**Base-Case 2**	**…**	**Base-Case M**	**New Case**
Feature 1	LSim(A_{11})	LSim(A_{12})	…	LSim(A_{1M})	V(A_{1j})
Feature 2	LSim(A_{21})	LSim(A_{22})	…	LSim(A_{2M})	V(A_{2j})
…	…	..	…	…	…
Feature N	LSim(A_{N1})	LSim(A_{N2})	…	LSim(A_{NM})	V(A_{Nj})

Fig. 2. Similarity values of Base-Cases relative to the feature values of the current case

The selection of the features to compose the new case involves the maximization of the global similarity (*GlobalSIM*), and the satisfaction of the dependencies and restrictions between the features to avoid conflicting and incompatible values. In Fig. 3 is presented a preliminary and generic algorithm to describe this approach. Finally, the new case can be instantiated from assets corresponding to the selected features.

3.4 Learning

The learning process in the CBR system [8] is done through the feedback about the performance of the new case, when the project is closed. At this moment, the effectiveness of the new case is evaluated by the user before the storage in the repository.

The case performance evaluation consists of 3 steps, which are global similarity comparison between preliminary and real contexts representation; reuse degree between the new case and the selected base-case; and specification of the base-case success level in the new case. These steps will be detailed as follows.

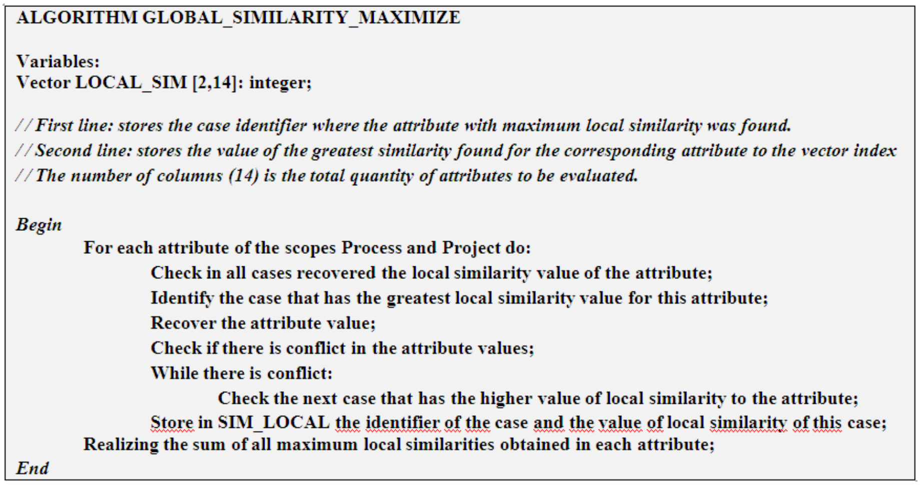
3.4.1 Global Similarity Comparison

When the project is closed, the representation for the executed process can be different from the representation used in the recovery phase. In order to reach the process improvement the comparison between the preliminary and the real representation can be useful to evaluate the adherence level of the adopted process.

The comparison between both representations, called *Global Similarity Comparison* (*GSC*), is based on a proportionality measurement appointing the occurrences of changes in the representation along the project execution. The measurement is obtained on the basis of the global similarity measurement (*SIM),* calculated according to the Section 3.2. The GSC measurement is presented in **(7)** and evaluates the similarities between the selected base-case representation (*a*) against the preliminary representation (*b*) and the representation of the executed process (*b'*).

$$GSC = \left(100 \times \frac{SIM(a,b')}{SIM(a,b)}\right) - 100 \tag{7}$$

When the global similarity values *SIM(a,b)* and *SIM(a,b')* remains the same, the index will return zero, meaning that the similarity values of the contexts stay the same. In this scenario, the selected base-case would stay in the same place of the ranking of similar base-cases. In the other hand, if the *GSC* is a value greater than zero, it means that the real context is more similar to the selected case than the



```
ALGORITHM GLOBAL_SIMILARITY_MAXIMIZE

Variables:
Vector LOCAL_SIM [2,14]: integer;

//First line: stores the case identifier where the attribute with maximum local similarity was found.
//Second line: stores the value of the greatest similarity found for the corresponding attribute to the vector index
//The number of columns (14) is the total quantity of attributes to be evaluated.

Begin
    For each attribute of the scopes Process and Project do:
        Check in all cases recovered the local similarity value of the attribute;
        Identify the case that has the greatest local similarity value for this attribute;
        Recover the attribute value;
        Check if there is conflict in the attribute values;
        While there is conflict:
            Check the next case that has the higher value of local similarity to the attribute;
        Store in SIM_LOCAL the identifier of the case and the value of local similarity of this case;
    Realizing the sum of all maximum local similarities obtained in each attribute;
End
```

Fig. 3. Algorithm to maximize the global similarity

preliminary context, which possibly the user has match the appropriate case to meet the project or organization needs. Otherwise, if the *GSC* is a value less than zero, it means that the real context is less similar to the selected case than the preliminary context, which possibly the user´s choice for the base-case was inappropriate to meet the project or organization needs, and several modifications was required.

3.4.2 Reuse Degree

The *Reuse Degree* (*RD*) is another evaluation metric that appoints the reuse percentage of the selected base-case (*a*), against the new case after the project's end (*b'*). It is obtained by the mapping of all activities components contained in cases *a* and *b'*, such as name, type, artifacts, resources, roles, connections, etc., in order to establish the reuse level. The *RD* formula is presented below:

$$RD = \frac{\sum_{q=1}^{n} N_{SimC_q(a,b')}}{m \times N_C} \tag{8}$$

where n is the number of activities from *b'* and m is the number of activities from *a*. N_{SimC} is a function that returns the number of similar components of a specific activity (q) between *a* and *b'*. To consider a component similar, it is necessary to establish a similarity threshold. Since this metric evaluates the level of reuse, then great variation in the new case should result in low reuse. So, in this research, the similarity threshold is considered a value greater or equal to 90%, because if the activity component is quite different, exceeding the similarity threshold, it is not considered by the function N_{SimC}. The N_C is the number of activity components contained in a case.

Analyses about the result of this metric can be useful to support the organization decision on the new case. With this estimation, the organization may identify how much the selected base-case was reused in the new case. And the user can realize whether the base-case is satisfactory to their specific needs, or may conclude that the adaptation made to the process executed discarded some important components of the original base-case that should be applied to processes and the organization didn't take the advantages of reuse.

3.4.3 Success Level

The success level is a subjective metric fed by the user whenever an executed process is evaluated. This metric is stored as related information about the base-case to register the user feedback about its utility and effectiveness of the process. This evaluation is represented by a value in the range 0 to 10.

This information is useful to the future adoption of the base-case, and contributes in the search for the continuous improvement of the process, since cases with greater success levels will be prioritized in the search engine results.

3.5 Retention

The retention consists in the incorporation process of what is useful in a new problem resolution [7] [8]. Retain continually is fundamental to increment the repository with new solutions. In this research, that phase occurs after the evaluation of the executed process, in such way to extract the knowledge for later use and to integrate cases in the existing representation structure.

Depending on the user evaluation of the executed process, the user may choose to distill the process or not [11]. Distilling a process means to transform it in a base-case, removing its specific project details, like schedule, people, etc., leaving only the suitable information to reuse in other projects and also store its context representation.

4 Case Study

In this section, a case study is presented to illustrate the approach for process reuse. In this sense, the description of a new project is detailed assigning values to the wished features for process and project. Note that the process for the standard process definition and the instantiation for an already defined process is the same. In the table below (Table 2) the definition of the desired features for the new case are presented. The Scope and Feature columns represent the feature's classification as presented in Table 1. The Weight and Value columns refer to properties of the new project, about the relevance and value for each feature, respectively, from the user viewpoint.

To illustrate the retrieval process, the RUP for Small Teams (RUP-ST) model [21] and its respective representation are used. It is important to stand out that the repository of process assets must be wide and diversified in order to take care of the most diverse situations. Table 3 presents the values for each feature for a project based on the RUP-ST.

The *global similarity* is calculated on the basis of their representation in order to determine and retrieve from the repository the most adherent case to fit the new case through minor efforts.

The *local similarity* for *Feature* is calculated in accordance with the similarity type, as referred to Section 3.2, and is described in the *Comparison* column. The product of this value times the *Weight*, presented in Table 2, determines the *Local Similarity (LS)*. Finally, the addition of all local similarities is presented in the line *Global Similarity*, in the current case 330.

Table 2. Feature definition for the case study

Scope	Feature	Weight	Value
Project	Life-Cycle Model	Medium	Spiral
	Complexity	Low	Medium
	Size	Medium	Medium
	Team	Medium	5
	Time	Low	6
	SE Knowledge	Low	Medium
	Development Paradigm	High	O-O
Process	Development Model	Low	-
	Maturity Model	Low	-
	Maturity Level	Low	-
	Complexity	Medium	Low
	Process	Medium	Project Management
	Experience on process usage	Low	Low

A further analysis about the local similarity results can be used to guide the user during the adaptation process. In this sense, the desired features from the retrieved cases can be composed in a new base-case in order to optimize (maximize) the global similarity. To illustrate this approach, the similarities measurements for project instances of ProGer [22] and D-CMM [23] models are used in order to select the features with higher local similarity value (Table 4).

The global similarity result for each base-case appoints the RUP-ST model as the most adherent to the current case, since it presents the greatest measurement value (330).

Table 3. Global Similarity about RUP-ST

Scope	Feature	Base-Case	Comparison	LS
Project	Life-Cycle Model	Iterative/ Incremental	0,5	25
	Complexity	Medium	1	10
	Size	Medium	1	50
	Team	5	1	50
	Time	5	0,5	5
	SE Knowledge	High	0,5	5
	Development Paradigm	O-O	1	100
Process	Development Model	RUP	0	0
	Maturity Model	-	0	0
	Maturity Level	-	0	0
	Complexity	Medium	0,5	25
	Process	Project Management	1	50
	Experience on process usage	Low	1	10
Global Similarity				**330**

Table 4. Global Similarity about ProGer and D-CMM

Scope	Feature	LS ProGer	LS D-CMM
Project	Life-Cycle Model	0	35
	Complexity	10	10
	Size	50	50
	Team	40	25
	Time	8,6	8,6
	SE Knowledge	10	6,6
	Development Paradigm	100	100
Process	Development Model	0	0
	Maturity Model	0	0
	Maturity Level	0	0
	Complexity	50	25
	Process	50	50
	Experience on process usage	10	5
Global Similarity		**328,6**	**315,2**

In another side, using the compositional approach, a new base-case can be obtained on the basis of the maximization algorithm (Fig. 3). The maximized global similarity for the new case, detailed in Table 5, is 373.6. Thus, the new case created through this approach represents the most adherent (similar) model to the current case, involving lower effort for their adaptation and reuse in the new situation.

The existence of features with the same local similarity value is resolved by the selection of the feature from the first case analyzed; however, is still a need for better research to assess whether this is right. Similarly, the feature that did not have values for the current case was disregarded, avoiding their influence in the calculation of similarity.

The process evolution and improvement is realized along its adaptation, reuse, performance evaluation and retention in the repository. Reuse evaluations along diverse projects can guide the adoption of the organization's standard-process.

Table 5. Maximizing the Global Similarity

Feature	Value	Process	LS
Life-Cycle Model	Iterative	D-CMM	35
Complexity	Medium	ProGer	10
Size	Medium	ProGer	50
Team	5	RUP-ST	50
Time	7	ProGer	8,6
SE Knowledge	Medium	ProGer	10
Development Paradigm	O-O	ProGer	100
Complexity	Low	ProGer	50
Process	Project Management	ProGer	50
Experience on process usage	Low	ProGer	10
Global Similarity			**373,6**

5 Final Considerations

The proposed approach promotes the reutilization of process assets as a start point for the elaboration of a standard process to meet the organizational needs. This approach is based on Case-Based Reasoning. It supplies a mechanism for the feature-value representation of cases in the assets repository. The cases are classified according to a set of relevant features to allow an efficient normalized retrieval. An example of similarity measurement was presented.

In addition, an optimization algorithm for the construction of the new case is presented. This model is composed of features from different processes, in order to maximize the global similarity, increasing the adherence of the composed process about the new case, and decreasing the adaptation efforts.

To ensure the learning process, it provides a case evaluation at the project's end, trough two metrics: the global similarity comparison, which presents the context representation variation, and the reuse estimation, which presents the reuse level of the selected base-case through the new case. Also, there's a subjective evaluation where the organization can infer about the new case satisfaction through establishing a success level, this evaluation is an organization-dependent decision and considers its characteristics and needs.

After that, the organization may decide the purpose of the new case. It is strongly suggested to feed the repository with the new case and its feature-value representations in order to provide reuse in future similar projects.

This approach allows the construction of the dynamic organizational knowledge and foresees the continuous improvement of the process through the permanent feedback to the repository involving the incorporation of its successes and failures. The learning capability of CBR systems contribute to the adoption of better and more efficient solutions. Currently, a management tool to support this approach is under development.

Acknowledgements. This work was supported in part by FUNCAP, Brazil.

References

1. The International Organization for Standardization and the International Electrotechnical Commission, Standard for Information Technology—Software Life Cycle Processes. Geneva, Switzerland (2008)
2. The International Organization for Standardization and the International Electrotechnical Commission, ISO/IEC 15504 Information Technology Process Assessment Part 5 (2006)
3. Software Engineering Institute, CMMI for Development, version 1.2 edition. SEI, Carnegie Mellon University, Pittsburg (2006)
4. Softex, Guia Geral MR-MPS (Versão 1.2) (2007), `http://www.softex.br/mpsbr/_guias/guias/MPS.BR_Guia_Geral_V1.2.pdf`
5. PMI Project Management Institute, A Guide to the Project Management Body of Knowledge: PMBOK Guide. PMI, 3rd edn. (2004)

6. Perry, D.: Practical Issues in Process Reuse. In: Baldonado, M., Chang, C., Gravano, L., Paepcke, A. (eds.) ISPW, International Software Process Workshop, Int. J. Digit. Libr, vol. 1, pp. 108–121. IEEE Computer Society Press, France (1997)
7. Kolodner, J.: Case-Based Reasoning. Morgan Kaufmann, San Francisco (1993)
8. Pal, S., Shiu, S.: Foundation of soft case based reasoning, 5th edn. Wiley series in intelligent systems (2004)
9. Mille, A.: From case-based reasoning to traces-based reasoning. Annual Reviews in Control 30(2), 223–232 (2006)
10. Santos, V., Cortés, M.: Software Process Reuse Using Case-Based Reasoning Accepted for publication in the ICAART´2009. In: International Conference on Agents and Artificial Intelligence, Portugal (2009)
11. Reis, R., Reis, C., Nunes, D.J.: Automated Support for Software Process Reuse: Requirements and Early Experiences with the APSEE model. In: 7th International Workshop on Groupware. IEEE Computer Society Press, Darmstadt (2001)
12. Oliveira, K., Gallota, C., Rocha, A., et al.: Defining and Building Domain-Oriented Software Development Environments. In: ICSSEA 1999, 12th International Conference Software & Systems Engineering and their Applications, Paris, France (1999)
13. McManus, J.: How does Software Quality Assurance Fit. In: Handbook of Software Quality Assurance, 3rd edn. Prentice Hall, Englewood Cliffs (1999)
14. Oliveira, S., Vasconcelos, A.: A Continuous Improvement Model in ImPProS. In: 30th Annual International Computer Software and Applications Conference. Proceedings on COMPSAC Fast Abstract Session, Chicago, EUA (2006)
15. Ricci, F., Arslan, B., Mirzadeh, N., Venturini, A.: Detailed Descriptions of CBR Methodologies. Information Society Technologies (2002), `http://dietorecs.itc.it/PubDeliverables/D4.1-V1.pdf`
16. Wu, Z., Palmer, M.: Verb Semantics and Lexical Selection. In: 32nd Annual Meeting of the Association for Computational Linguistic, New Mexico State University, Las Cruces, New Mexico, USA, pp. 133–138 (1994)
17. Cunningham, P.: A Taxonomy of Similarity Mechanisms for Case-Based Reasoning. Technical Report UCD-CSI-2008-01. University College Dublin. Belfield, Ireland (2008)
18. Brasil, M., Cortés, M.: Definição de Processo de Software através da Composição de Atributos de Casos Similares. Hífen, Uruguaiana 32(62) , 91–98 (2008)
19. Beck, K.: Extreme Programming Explained: Embrace Change. Pearson, London (2004)
20. SEI, CMMI® or Agile: Why Not Embrace Both! (2008), `http://www.sei.cmu.edu/pub/documents/08.reports/08tn003.pdf` (Accessed in: 13/11/2008)
21. Pollice, G., Augustine, L., Lowe, C., Madhur, J.: Software development for small teams - a RUP centric approach. Addison-Wesley, Reading (2004)
22. Rouiller, A.: Gerenciamento de Projetos de Software para Empresas de Pequeno Porte, PhD. Thesis, Universidade Federal de Pernambuco (2001)
23. Orci, T., Laryd, A.: Dynamic CMM for small organizations. In: Proceedings of the First Argentine Symposium on Software Engineering (ASSE), Argentina, pp. 133–149 (2000)
24. Kruchten, P., Kroll, P.: The Rational Unified Process Made Easy. Addison-Wesley, Reading (2003)
25. Pressman, R.: Software Engineering, 5th edn. McGraw-Hill, New York (2002)

Appendix

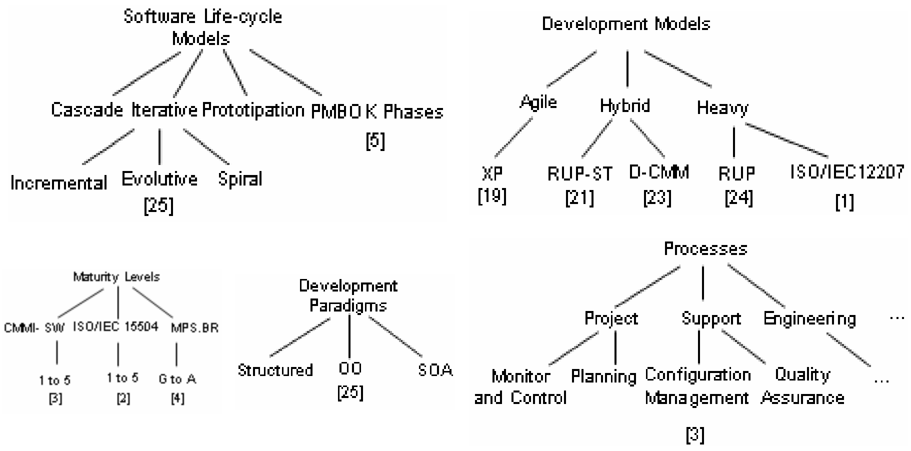


Fig. A1. Taxonomies for QVI features

Mapping Software Acquisition Practices from ISO 12207 and CMMI

Francisco J. Pino[1,3], Maria Teresa Baldassarre[2], Mario Piattini[3], Giuseppe Visaggio[2], and Danilo Caivano[2]

[1] IDIS Research Group, Electronic and Telecommunications Engineering Faculty
University of Cauca, Calle 5 # 4 – 70 Popayán, Colombia
fjpino@unicauca.edu.co
[2] Department of Informatics, University of Bari – SER&Practices SPINOFF
University of Bari, Via E. Orabona 4, 70126, Bari, Italy
{baldassarre,visaggio,caivano}@di.uniba.it
[3] Alarcos Research Group – Institute of Information Technologies & Systems
University of Castilla-La Mancha, Paseo de la Universidad, 4, 13071, Ciudad Real, Spain
Mario.Piattini@uclm.es

Abstract. The CMMI-ACQ and the ISO/IEC 12207:2008 are process reference models that address issues related to the best practices for software product acquisition. With the aim of offering information on how the practices described in these two models are related, and considering that the mapping is one specific strategy for the harmonization of models, we have carried out a mapping of these two reference models for acquisition. We have taken into account the latest versions of the models. Furthermore, to carry out this mapping in a systematic way, we defined a process for this purpose. We consider that the mapping presented in this paper supports the understanding and leveraging of the properties of these reference models, which is the first step towards harmonization of improvement technologies. Furthermore, since a great number of organizations are currently acquiring products and services from suppliers and developing fewer and fewer of these products in-house, this work intends to support organizations which are interested in introducing or improving their practices for acquisition of products and services using these models.

Keywords: Harmonization of improvement technologies, Mapping, Software product acquisition, CMMI-ACQ, ISO/IEC 12207.

1 Introduction

Software process improvement is a planned, managed and controlled effort which aims to enhance the capability of the software development processes of an organization [1]. It is significant to highlight that in a software process improvement effort different types of models are involved. These include the process reference model, the process assessment method and the model that guides the process improvement [2]. According to [3] the purpose of the process reference models is to provide the description of the processes (and their entities) that can be applied during the acquisition, supply, development,

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 234–247, 2010.

operation and maintenance of software. These models describe best practices which should be taken into account by organizations in the acquisition, supply, development, operation and maintenance of software.

Now more than ever, many organizations are increasingly becoming interested in the activities of software acquisition [4]. Currently, a great number of organizations are acquiring products and services from suppliers and developing fewer and fewer of these products inhouse. These organizations may be customers who need to perform good practices to guarantee that the product and service purchased satisfy the defined acceptance criterias. These organizations can also be suppliers that may act as a customer when acquiring a product and service from another supplier.

Regarding the process reference model related to software acquisition, the Software Engineering Institute –SEI– has recently developed the CMMI-ACQ [5], and the International Standardization Organization –ISO– is addressing this issue in the agreement processes category of ISO 12207:2008 [6]. Each model has got its own structure, processes and entities of process for describing best practices in its scope of application.

Given the present need to harmonize different improvement technologies [7] to support organizations which are interested in introducing or improving their practices for acquisition of products and services, it is important to have information on how the practices described in these two models are related. According to [8], the harmonization aims to develop an appropriate solution to meet the individual organizational objectives, which first requires understanding and leveraging of the properties of the technologies of interest.

In this vein, and aiming to support the organizations in the integration, management, and alignment of their activities of software acquisition using these models, in this paper a mapping of CMMI-ACQ and ISO/IEC 12207:2008 is presented. According to [7] mapping is one of the most widely-used specific strategies for the harmonization of models. We have taken into account the following considerations for this mapping: (i) refer to the latest versions of the models, (ii) carry out the mapping with process entities of low level of abstraction, and (iii) guide the mapping through a well defined method.

The paper is structured as follows. Section 2 presents related works, and then the general considerations for mapping are described. Section 4 presents the mapping overview and describes the analysis of results. Lastly, conclusions and future work are set out.

2 Related Work

Literature presents some works that involve mapping and comparisons between different processes models. A mapping is a comparison which: (i) goes beyond the identification of differences and similarities between the improvement technologies, and (ii) connects explicit entities of these technologies. Among these, those related to CMMI V1.1 and ISO 9001 are:

- In [9] a mapping between two models is described.
- In [10] a new model that integrates the content of these two models is introduced.

- In [11] a way for the transition from ISO 9001 to SW-CMM is defined.
- In [12] a comparison and a correspondence between ISO 9001 and SW-CMM are shown.

In the same sense, the following studies regarding the integration of specific assessment frameworks have been conducted:

- An analysis and comparison of ISO/IEC 15504:2004 and CMMI V1.1 for software process assessment is presented in [13].
- An analysis of compatibility between SPICE and CMM is given in [14].
- In [3] the harmonization of CMMI V1.1 and ISO/IEC TR 15504-2:2002 is presented.
- In [15] and [16] a definition of compatibility structures and comparison between CMMI and SPICE is described.

The works that deal with the standards ISO/IEC 15504-2:2004 involve ISO/IEC 12207:2002 directly, because this latter standard is suggested by ISO/IEC 15504 as a process reference model.

As can be seen from the work presented above, the most widely-used models in mapping and comparisons are: ISO 9001, ISO/IEC 15504:2004 and CMMI V1.1. However, in none of these mappings and comparisons are the latest versions of these models involved. Moreover, from the analysis of these studies we have found that the process entities involved in the comparisons or mappings are of high level abstraction (as examples, objectives, outcomes or statements).

We have carried out a mapping between the latest versions of models: ISO/IEC 12207:2008 [6] and CMMI-ACQ V1.2: 2007 [5]. For the development of our explicit comparison we have followed a well defined process, which we also used for other comparisons that we have carried out (ISO 9001 to CMMI-DEV, and ISO 12207 to CMMI-DEV [17]). We might add that the entities involved in the mapping are: (i) activities and tasks for ISO/IEC 12207 and (ii) specific practices for CMMI-ACQ. These process entities are of low level abstraction in the description of the processes or process areas.

A comparison at this abstraction level provides information about what activities and tasks outlined in ISO/IEC 12207 give support to specific practices of CMMI-ACQ. Furthermore, an analysis at this abstraction level can give directions about how a model previously implemented in the organization (ISO 12207) can meet part of the requirements to establish a new model (CMMI-ACQ). This could reduce the effort and costs associated with the implementation of a new model, with reference to a model already used in the organization. The cost reduction in the implementation of a model is a benefit of the harmonization [8].

3 Performing the Mapping

After an analysis of the different related pieces of work mentioned in the previous section, we have observed a constant relationship between some models of the SEI and ISO. Table 1 shows a high-level relationship extracted from the structures of these models and their comparisons.

Table 1. High-level relationship between some ISO and SEI standards

			SEI	
			CMMI-ACQ	
			Generic Goals / Generic Practices	**Process Areas**
ISO	**15504-5** **12207**	**Process**		*Performance of the process*
	15504-2	**Process Attribute**	*Institutionalization of the capability of the process*	

As arises from the table, the process areas of CMMI-ACQ models are closely related to the process reference models described in ISO/IEC 15504-5 [18] and ISO/IEC 12207 [6, 19]. Furthermore, generic goals and practices of CMMI-ACQ models are closely related to the process attributes described in the ISO/IEC 15504-2 standard [20].

Based on the relationship offered in Table 1, the mapping between CMMI-ACQ and ISO 12207:2008 must be carried out at the level of process performance; in other words, this comparison doesn't involve goals and generic practices.

There follows a description of activities carried out to perform the mapping between these two models. These activities are related to the process that we have defined for the comparison and mapping of models. The purpose of this process is to provide a guide with which to perform the comparison and mapping of different models step-by-step. This process defines two roles: the performers and the reviewers of the mapping, along with five tasks: (i) analyzing the models, (ii) designing the mapping, (iii) carrying out the mapping, (iv) presenting the outcomes of the mapping, and (v) analyzing the results of the mapping.

This comparison and mapping process is part of a framework, which we are currently developing, to harmonize improvement technologies. This framework seeks to determine and define an appropriate strategy for connecting two or more improvement technologies to support the achievement of business goals of an organization. A harmonization strategy is a set of methods or techniques defined systematically to implement the connection of diverse improvement technologies. These methods or techniques can be: mapping, comparison, correspondence, synergy, complementation and integration, among others.

3.1 Analyzing the Models

This task involves: (i) acquiring knowledge about the models to compare and (ii) analyze the structure of these models. In this sense, a description of CMMI-ACQ and ISO/IEC 12207 is described in the following lines.

According to [5], the purpose of CMMI-ACQ is to provide guidance for the application of CMMI best practices by the acquirer. Best practices in the model focus on activities for initiating and managing the acquisition of products and services that meet the needs of the customer. Although suppliers may provide artefacts which are useful to the processes addressed in CMMI-ACQ, the focus of the model is on the processes of the acquirer. CMMI-ACQ integrates bodies of knowledge that are essential for an acquirer. It is a collection of best practices that is generated from the CMMI Framework, which is the basic structure that organizes CMMI components and combines them into CMMI constellations and models. Also in the framework is a CMMI model foundation (CMF) which exists within the CMMI Framework, and it is a skeleton model that contains each of the components that must be included in every CMMI model [21].

As regards the CMMI-ACQ's structure, it contains two main sections in its description: (i) generic goals and practices, and (ii) process areas. Each process area is defined in terms of the process entities: purpose, specific goals (required component), specific practices (expected component). A required component describes what an organization must achieve to satisfy a process area, and an expected component describes what an organization may implement to achieve a required component.

On the other hand, according to [6] the purpose of ISO/IEC 12207 standard (Systems and software engineering - Software life cycle processes) is to provide a defined set of processes to facilitate communication among acquirers, suppliers and other stakeholders in the life cycle of a software product.

With respect to the ISO/IEC 12207's structure, the processes are grouped in process groups, and each process is described in terms of the process entities: purpose, outcomes, activities and tasks. The purpose and outcomes are a statement of the goals of the performance of each process. The list of activities and tasks is performed to achieve the outcomes.

3.2 Designing and Carrying Out the Mapping

This task involves: (i) fixing the process entities to be compared, based on the research needs, (ii) defining the comparison scale, (iii) fixing the directionality of the comparison, and (iv) defining a template comparison.

This comparison should find activities which ISO/IEC 12207 and CMMI-ACQ have in common, in order to define goals for a measurement plan using the Multiview Framework [22]. To apply the Multiview Framework, the mapping should be done at the level of: (i) the entities of specific practices for CMMI-ACQ, and (ii) the entities of activity and tasks for ISO/IEC 12207. The Fig. 1 shows these entities involved in the mapping. These entities describe specific practice or activities that should be executed to obtain the intended product or service. Carrying out the mapping using these entities allows us to identify common activities found (from now on called specific activities) in both CMMI-ACQ and ISO/IEC 12207. The above-mentioned specific activities can not be found using entities such as purpose, outcomes or generic goals.

To express the degree of relationship between a Process from ISO/IEC 12207 and a Process area from CMMI-ACQ, we have defined a discrete scale (scale of comparison). Each of the elements of the scale has been associated with a set of numeric values which are described in terms of percentage. This scale is made up of the following elements:

- Strongly related (86% to 100%),
- Largely related (51% to 85%),
- Partially related (16% to 50%),
- Weakly related (1% to 15%), and
- Non-related (0%).

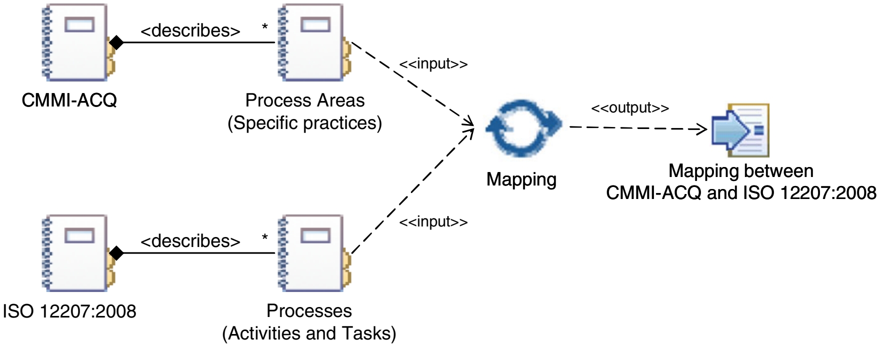


Fig. 1. Structure and elements involved in the mapping

The numeric values can be found by dividing the number of specific practices (from a Process area of CMMI-ACQ) that are related to activities (from a process of ISO/IEC 15504) by the total number of specific practices defined in that Process area.

When a comparison involves process entities of low level abstraction it is relevant to define the direction of the comparison (see a discusion of this issue in section 4.3). The direction of this mapping is from ISO/IEC 12207 to CMMI-ACQ.

The roles were assigned, with two people as performers of the mapping and two reviewers. We then carried out the mapping by means of an iterative and incremental procedure. It is iterative, because the execution (analyze and determine the relationship of the process entities of ISO/IEC 12207 and CMMI-ACQ) of the mapping is carried out completely on one CMMI-ACQ process area first, and then on the others in turn. It is also incremental, in the sense that the template comparison (which is the product) grows and evolves with each iteration until it becomes the definitive one.

4 Presenting and Analyzing the Results of the Mapping

Based on the general considerations of the comparison described in the previous section, the degree of relationship of the CMMI-ACQ process areas from the ISO 12007 process is presented in Table 2.

Each intersection of Table 2 is the overview of a detailed comparison between specific practices of a process area of CMMI-ACQ and activities of a process of ISO 12207.

Table 3 shows a specific comparison between specific practices of Process and Product Quality Assurance (PPQA) of CMMI-ACQ and the activities of Project Assessment and Control Process, Software Quality Assurance Process and Software Review Process of ISO/IEC 12207.

We shall now go on to present a discussion and consideration of several issues which arose during this work, such as: an analysis of the specific issues of acquisition in both models and the lessons learned.

Table 2. Overview of the mapping between ISO/IEC 12207 and CMMI-ACQ

Direction of the comparison: From ISO/IEC 12207 to CMMI-ACQ
Process entities for the comparison:
- For ISO/IEC 12207: Activities and tasks of the standard's processes.
- For CMMI-ACQ: Specific practices.

Research question:
- What activities and tasks of ISO/IEC 12207 can offer support to specific practices of CMMI?
- What ISO/IEC 12207's activities and tasks are strongly related with the support to CMMI's specific practices?

Comparison goal: To determine which activities and tasks of ISO/IEC 12207 have a close relationship with some specific practice of CMMI. The goal is to find out to what degree the specific practices of CMMI based on the activities and tasks described in ISO/IEC 12207 are fulfilled.
Scale of comparison:
- S - Strongly related (86% to 100%)
- L - Largely related (51% to 85%)
- P - Partially related (16% to 50%)
- W - Weakly related (1% to 15%)
- - Non-related (0%)

				CMMI-ACQ																					
				CMMI Framework (16 Process areas)																CMMI-ACQ (6 New Process areas)					
				Causal Analysis and Resolution (CAR)	Configuration Management (CM)	Decision Analysis and Resolution (DAR)	Integrated Project Management (IPM)	Measurement and Analysis (MA)	Organizational Innovation and Deployment (OID)	Organizational Process Definition (OPD)	Organizational Process Focus (OPF)	Organizational Process Performance (OPP)	Organizational Training (OT)	Project Monitoring and Control (PMC)	Project Planning (PP)	Process and Product Quality Assurance (PPQA)	Quantitative Project Management (QPM)	Requirements Management (REQM)	Risk Management (RSKM)	Agreement Management (AM)	Acquisition Requirements Development (ARD)	Acquisition Technical Management (ATM)	Acquisition Validation (AVAL)	Acquisition Verification (AVER)	Solicitation and Supplier Agreement Development (SSAD)
ISO/IEC 12207	System context process	Agreement processes (2 processes)	Acquisition process																	S	L	P			L
			Supply process																						
		Organizational Project-Enabling Processes (5 processes)	Life Cycle Model Management Process	L			W		W	P	P														
			Infrastructure Management Process		P										W										
			Project Portfolio Management Process				W							P											
			Human Resource Management Process				W			P			S		W										
			Quality Management Process														P								
		Project processes (7 processes)	Project Planning Process				W								L			P							
			Project Assessment and Control Process	L			W							L		P			W						
			Decision Management Process			P																			
			Risk Management Process											W	W				S						
			Configuration Management Process		S																				
			Information Management Process												W										
			Measurement Process					S		P															
		Technical processes (11 processes)	Stakeholder Requirements Definition Process															P			P				W
			System Requirements Analysis Process															L			P				
			System Architectural Design Process															P							
			Implementation Process																						
			System Integration Process																						
			System Qualification Testing Process																						
			Software Installation Process																						
			Software Acceptance Support Process																						
			Software Operation Process																						
			Software Maintenance Process																						
			Software Disposal Process																						
	Software specific process	Software Implementation Processes (7 processes)	Software Implementation Process		P																				
			Software Requirements Analysis Process															L							
			Software Architectural Design Process																						
			Software Detailed Design Process																						
			Software Construction Process																						
			Software Integration Process																						
			Software Qualification Testing Process																						
		Software Support Processes (8 processes)	Software Documentation Management Process																						
			Software Configuration Management Process		S																				
			Software Quality Assurance Process													S									
			Software Verification Process																					P	
			Software Validation Process																				L		
			Software Review Process		W									P		S						P		P	
			Software Audit Process																	P		P			
			Software Problem Resolution Process	L																					
		Software reuse processes (3 processes)	Domain Engineering Process																						
			Reuse Asset Management Process							P															
			Reuse Program Management Process																						

Table 3. Specific comparison between specific practices and activities

Process Area	**PROCESS AND PRODUCT QUALITY ASSURANCE (PPQA)**			
Specific practices	SP 1.1 Objectively Evaluate Processes	SP 1.2 Objectively Evaluate Work Products and Services	SP 2.1 Communicate and Ensure the Resolution of Noncompliance Issues	SP 2.2 Establish Records
Degree of relationship	**S (Fuerte)**			
6.3.2 Project Assessment and Control Process				
Activities: 6.3.2.3.1 Project monitoring.				
Activities: 6.3.2.3.2 Project control.				
Activities: 6.3.2.3.3 Project assessment.		▨		
Activities: 6.3.2.3.4 Project closure.				
	P (1 SP of 4)			
7.2.3 Software Quality Assurance Process				
Activities: 7.2.3.3.1 Process implementation.			▨	▨
Activities: 7.2.3.3.2 Product assurance.		▨		
Activities: 7.2.3.3.3 Process assurance.	▨			
Activities: 7.2.3.3.4 Assurance of quality systems.				
	S (4 SP of 4)			
7.2.6 Software Review Process				
Activities: 7.2.6.3.1 Process implementation.	▨	▨	▨	▨
Activities: 7.2.6.3.2 Project Management Reviews.				
Activities: 7.2.6.3.3 Technical Reviews.				
	S (4 SP of 4)			

4.1 The Acquisition in Both Models

The purpose of CMMI-ACQ is to provide guidance for the application of CMMI best practices by the acquirer [5]. This model shows a viewpoint from the side of the acquirer, so the focus of the model is on the processes of the acquirer. Supplier activities are not addressed in this model. It was very important to keep this perspective constantly in mind.

CMMI-ACQ contains 22 process areas. Of those, 16 are CMMI Model Foundation (CMF) process areas. Six process areas focus on practices which are specific to acquisition of both products and services, addressing:

- Agreement management (AM)
- Acquisition requirements development (ARD)
- Acquisition technical management (ATM)
- Acquisition validation (AVAL)
- Acquisition verification (AVER), and
- Solicitation and supplier agreement development (SSAD).

An analysis about the Agreement process group (labelled number 6.1 in the standard) has been carried out. This process group defines the activities necessary to establish an agreement between two organizations, and it defines two processes: Acquisition and Supply. The purpose of the Acquisition Process is to obtain the product and/or service that satisfies/satisfy the need expressed by the acquirer. The purpose of the Supply Process is to provide a product or service to the acquirer that meets the agreed requirements [6].

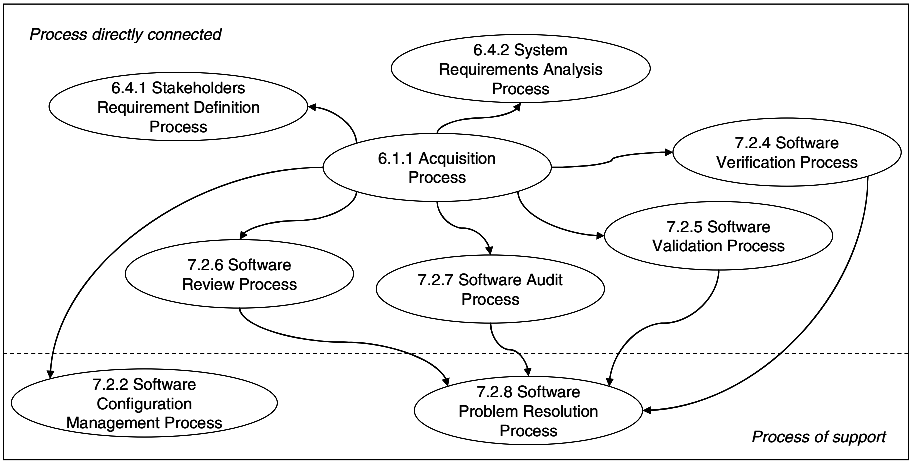

Process directly connected
6.4.1 Stakeholders Requirement Definition Process
6.4.2 System Requirements Analysis Process
6.1.1 Acquisition Process
7.2.4 Software Verification Process
7.2.5 Software Validation Process
7.2.6 Software Review Process
7.2.7 Software Audit Process
7.2.2 Software Configuration Management Process
7.2.8 Software Problem Resolution Process
Process of support

Fig. 2. Relationships among processes of ISO/IEC 12207 for Acquisition

On analyzing the description of the Supply Process, a viewpoint from the supplier is observed. This perspective is opposite to that described by the CMMI-ACQ. Taking into account this consideration, it is observed that the Process Supply is not related to the six process areas which focus on practices specific to acquisition as described by CMMI-ACQ.

Based on the comparison carried out and the description of the Acquisition process from the ISO/IEC 12207 standard, a relationship between these processes is shown in Fig. 2. The goal is to offer an overview to the acquirer, of which processes are involved in the acquisition.

4.2 Detailed View for Acquisition

Table 4 shows a summary of the comparison carried out between the six specific process areas of CMMI-AQC for the acquisition and the processes related to the Agreement processes of ISO 12207. The degree of relationship presented between process areas and process is only described in the direction from ISO 12207 to CMMI. In other words, how the activities of processes of ISO 12207 support the fulfilment of the specific practices of CMMI-ACQ. In Table 5 an example of a detailed comparison between activities and tasks of a process from ISO/IEC 12207 and the Agreement Management process area from CMMI-ACQ is shown.

For each process of ISO/IEC 12207 and process areas of CMMI-ACQ that have some relationship, we have defined a detailed chart like Table 5.

In summary, there are 39 specific practices in 6 process areas (Agreement management - AM, Acquisition requirements development - ARD, Acquisition technical management - ATM, Acquisition validation - AVAL, Acquisition verification - AVER and Solicitation and supplier agreement development - SSAD) of CMMI-ACQ, of which 28 specific practices are related to one or more tasks or activities of ISO 12207. So the degree of general relationship is 72% (28/39).

Table 4. Detailed view of the relationship for acquisition of ISO/IEC 12207 and CMMI-ACQ

		CMMI-ACQ					
		AM	**ARD**	**ATM**	**SSAD**	**AVAL**	**AVER**
ISO 12207	**Acquisition process**	**S** 100%	**L** 63%	**P** 20%	**L** 56%		
	Software Review Process			**P** 20%			**P** 38%
	Software Audit Process	**P** 25%		**P** 20%			
	Stakeholder Req. Definition Process		**P** 50%		**W** 11%		
	System Req. Analysis Process		**P** 25%				
	Validation Process					**L** 80%	
	Verification Process						**P** 50%
Degree of relationship GENERAL		**S** 100%	**L** 75%	**P** 20%	**L** 66%	**L** 80%	**S** 88%

Table 5. Detailed comparison between activities and tasks of Acquisition process of ISO 12207 and specific practices of Agreement management of CMMI-ACQ

		AGREEMENT MANAGEMENT - AM (Specific practices)			
		SP 1.1 Execute the Supplier Agreement.	SP 1.2 Monitor Selected Supplier Process	SP 1.3 Accept the Acquired Product	SP 1.4 Manage Supplier Invoices
6.1.1 Acquisition process (Activities)	6.1.1.3.1 Acquisition preparation.				
	6.1.1.3.2 Acquisition advertisement.				
	6.1.1.3.3 Supplier selection.				
	6.1.1.3.4 Contract agreement.				
	6.1.1.3.5 Agreement monitoring.	Task 6.1.1.3.5.1	Task 6.1.1.3.5.1		
	6.1.1.3.6 Acquirer acceptance.			Task 6.1.1.3.6.2	
	6.1.1.3.7 Closure.				Task 6.1.1.3.7.1
	Degree of relationship (Direction ISO 12207 to CMMI) ***100% (Fulfilment 4 of 4 Specific Practices)***				

The specific practices that are not supported by the activities from ISO 12207 are:

- Acquisition requirements development - ARD, SP 2.2 Allocate Contractual Requirements.
- Acquisition requirements development - ARD, SP 3.3 Analyze Requirements to Achieve Balance
- Acquisition technical management - ATM, SP 1.1 Select Technical Solutions for Analysis
- Acquisition technical management - ATM, SP 1.2 Analyze Selected Technical Solutions
- Acquisition technical management - ATM, SP 2.1 Select Interfaces to Manage
- Acquisition technical management - ATM, SP 2.2 Manage Selected Interfaces
- Solicitation and supplier agreement development - SSAD, SP 1.1 Identify Potential Suppliers
- Solicitation and supplier agreement development - SSAD, SP 2.2 Establish Negotiation Plans

- Solicitation and supplier agreement development - SSAD, SP 3.1 Establish an Understanding of the Agreement
- Acquisition validation - AVAL, SP 1.2 Establish the Validation Environment
- Acquisition verification - AVER, SP 1.2 Establish the Verification Environment

4.3 Lessons Learned

In a comparison at low level abstraction, the degree of relationship between a Process from ISO/IEC 12207 and a Process area from CMMI-ACQ, depends on the direction of the relationship. In other words, this relationship is not bi-directional. For instance, Table 5 shows the next degree of relationship:

- Direction from ISO 12207 to CMMI: The degree of relationship is 100% (S) (Fulfilment 4 of 4 Specific Practices). As shown in the chart: 3 activities of this process of ISO 12207 meet 4 specific practices of the 4 that this process area of CMMI-ACQ has defined.
- Direction from CMMI to ISO 12007: The degree of relationship is 43% (P) (Fulfilment of 3 out of 7 Activities). As shown in chart 4, specific practices of this process area meet 3 activities of the 7 that this process of ISO 12207 has defined.

With the early detailed comparisons between a process and a process area, an analysis of the degrees of relationship in the comparison was conducted. According to this analysis, we conclude that this degree is not always possible to establish in both directions. In some cases, in a given direction it loses meaning. An example is shown in Table 6. In this table it does not make sense to establish a degree of relationship in the direction from CMMI to ISO 12007, because it is not correct to say that the Software Audit Process has 100% of fulfilment if only the specific practices SP 1.3 of ATM process area have been carried out. In these cases this row is labelled as Not Applicable.

Regarding the processes of ISO 12207 and their relationship with the sixteen process areas of the CMMI-ACQ (which are part of the CMMI Framework), we can observe that there is:

- Strong support level to the process areas: Configuration Management, Measurement and Analysis, Project Monitoring and Control, Process and Product Quality Assurance, Requirements Management, Organizational Training, Risk Management, and Causal Analysis and Resolution.
- Large support level to the process area of Project Planning.
- Partial support level to the process areas: Decision Analysis and Resolution, Integrated Project Management, Organizational Process Definition, Organizational Process Focus and Quantitative Project Management.
- Weak support level to the process areas: Organizational Innovation and Deployment, and Organizational Process Performance.

As regards the value of the degree of relationship is important to highlight that a strong degree of relationship does not mean that a process area of CMMI-ACQ is satisfied. It only indicates that most of the specific practices of this process area are connected to the processes of ISO 12207. To determine the degree of implementation of a specific practice from the processes of ISO 12207 it is necessary to conduct a detailed analysis of the relationships presented in this mapping.

Table 6. Degree of relationship "Not Applicable" between activities from ISO 12207 and specific practices from CMMI-ACQ

		ATM (Specific practices)					
		SP 1.1 Select Technical Solutions for Analysis	SP 1.2 Analyze Selected Technical Solutions	SP 1.3 Conduct Technical Reviews	SP 2.1 Select Interfaces to Manage	SP 2.2 Manage Selected Interfaces	***Degree of relationship (Direction CMMI to ISO 12207)*** ***Not Applicable***
7.2.7 Software Audit Process (Activities)	7.2.7.3.1 Process implementation.			▨			
	7.2.7.3.2 Software audit.			▨			
	Degree of relationship (Direction ISO 12207-CMMI) ***20% (Fulfilment 1 of 5 Specific Practices)***						

On the topic of the comparison process, to follow an iterative and incremental procedure to perform the mapping brought some advantages, for example:

- The performing of the mapping starts with a process area, to reduce the complexity and scope of each iteration.
- Each iteration of mapping is short and provides feedback for the next iteration.
- There is an integration of the results of each iteration into the mapping final report.
- With the design of the mapping the iterations can be carried out both independently and in parallel.
- The complexity of each iteration is easier to manage.

5 Conclusions

In this work we have presented a mapping between two reference models: CMMI-ACQ and ISO/IEC 12207:2008. To carry out this comparison in a systematic way, we defined a process for that purpose. Following this process has helped us to organize and manage the work performed for comparison, with the aim of reducing the two types of error in the comparisons described by Yoo in [10]. To increase the reliability of results, this process proposes using pair review by the performers of the mapping in the task which carries out the comparison. Furthermore, the reviewer of the mapping validates the result and this resolves the divergences between the performers.

Taking into account the activities and tasks described by the processes of ISO/IEC 12207 for Acquisition and their relationship with six process areas focused on practices which are specific to acquisition of CMMI-ACQ, we can observe that there is a suitable support level for these process areas: Agreement Management, Acquisition Verification, Acquisition Validation, Acquisition Requirements Development and even to the process area of Solicitation and Supplier Agreement Development. However, the support level to the Acquisition Technical Management is low.

We will use specific activities for the definition of goals for a measurement plan in a software enterprise, following the Multiview Framework. On the other hand, we are currently working on the definition of a methodology which would offer companies a strategy for harmonization of process entities described by different reference models. The mapping process described in this paper is a component of that methodology of harmonization.

Acknowledgements. This work has been funded by the projects: INGENIO (PAC08-0154-9262, JCCM of Spain), ARMONIAS (PII2109-0223-7948 of JCCM of Spain) and PEGASO/MAGO (TIN2009-13718-C02-01 of Ministerio de Ciencia e Innovacion – MICINN, and Fondo Europeo de Desarrollo Regional - FEDER). By the first author to the research fellowships granted by JCCM and funded by Fondo Europeo de Desarrollo Regional- FEDER.

References

1. Krasner, H.: Accumulating the Body of Evidence for the Payoff of Software Process Improvement. In: Hunter, R.B., Thayer, R.H. (eds.) Software Process Improvement, pp. 519–540. Wiley-IEEE Computer Society (2001)
2. Pino, F., Garcia, F., Piattini, M.: Software Process Improvement in Small and Medium Software Enterprises: A Systematic Review. Software Quality Journal 16(2), 237–261 (2008)
3. Rout, T., Tuffley, A.: Harmonizing ISO/IEC 15504 and CMMI. Software Process: Improvement and Practice 12(4), 361–371 (2007)
4. Weber, C., De Araújo, E.E.R., Scalet, D., De Andrade, E.L.P., Da Rocha, A.R.C., Montoni, M.A.: MPS model-based software acquisition process improvement in Brazil. In: QUATIC 2007 - 6th International Conference on the Quality of Information and Communications Technology, Lisboa, Portugal, pp. 110–119 (2007)
5. SEI, CMMI for Acquisition, Version 1.2. Technical Report CMU/SEI-2007-TR-017, Software Engineering Institute (SEI): Pittsburgh (2007)
6. ISO, ISO/IEC 12207: Systems and software engineering - Software life cycle processes. International Organization for Standardization: Geneva (2008)
7. SEI. Process Improvement in Multimodel Environments (PrIME Project). 2008 [cited 2008 May], `http://www.sei.cmu.edu/prime/primedesc.html`
8. Siviy, J., Kirwan, P.: Process Improvement in Multimodel Environments. In: Past, Present, Future, p. 45. Software Ingineering Institute, Carnegie Mellon (2008)
9. Mutafelija, B., Stromber, H.: ISO 9001:2000 - CMMI V1.1 Mappings. Software Engineering Institute - SEI, 1–31 (2003)

10. Yoo, C., Yoon, J., Lee, B., Lee, C., Lee, J., Hyun, S., Wu, C.: unified model for the implementation of both ISO 9001:2000 and CMMI by ISO-certified organizations. Journal of Systems and Software 79(7), 954–961 (2006)
11. Jalote, P.: CMM in Practice: Processes for Executing Software Projects, in Infosys. Addison-Wesley, Reading (1999)
12. Paulk, M.C.: A Comparison of ISO 9001 and the capability maturity model for software (CMU/SEI-94-TR-12), Software Engineering Institute (1994)
13. Wangenheim, C.G.v., Thiry, M.: Analysing the Integration of ISO/IEC 15504 and CMMI-SE/SW Technical Report LPQS001.05E, Universidade do Vale do Itajaí - UNIVALI: Sao José/SC, Brazil. p. 28 (2005)
14. Rout, T.: SPICE and the CMM: is the CMM compatible with ISO/IEC 15504? In: AquIS, Venecia, Italy, p. 12 (1998)
15. Lepasaar, M., Mäkinen, T., Varkoi, T.: Structural comparison of SPICE and continuos CMMI. In: SPICE 2002, Venicia, Italia, pp. 223–234 (2002)
16. Foegen, M., Richter, J.: CMM, CMMI and ISO 15504 (SPICE). IT Maturity Services, 52 (2003)
17. Pino, F., Baldassarre, M.T., Piattini, M., Visaggio, G.: Relationship between maturity levels of ISO/IEC 15504-7 and CMMI-DEV v1.2. In: Software Process Improvement and Capability Determination Conference (SPICE 2009), Turku, Finland, pp. 69–76 (2009)
18. ISO, ISO/IEC 15504-5:2006(E). Information technology - Process assessment - Part 5: An exemplar Process Assessment Model. International Organization for Standardization, Geneva (2006)
19. ISO, ISO/IEC 12207:2002. Information technology - Software life cycle processes. International Organization for Standardization, Geneva (2002)
20. ISO, ISO/IEC 15504-2:2003/Cor.1:2004(E). Information technology - Process assessment - Part 2: Performing an assessment. International Organization for Standardization, Geneva (2004)
21. SEI, Introduction to the Architecture of the CMMI® Framework. TECHNICAL NOTE CMU/SEI-2007-TN-009. Software Engineering Institute (SEI), Pittsburgh (2007)
22. Ardimento, P., Baldassarre, M.T., Caivano, D., Visaggio, G.: Multiview framework for goal oriented measurement plan design. In: Bomarius, F., Iida, H. (eds.) PROFES 2004. LNCS, vol. 3009, pp. 159–173. Springer, Heidelberg (2004)

Concept Management: Identification and Storage of Concepts in the Focus of Formal Z Specifications

Daniela Pohl and Andreas Bollin

Alpen-Adria Universität Klagenfurt, 9020 Klagenfurt, Austria
{daniela,andi}@isys.uni-klu.ac.at
http://www.uni-klu.ac.at/tewi/inf/isys/sesc/index.html

Abstract. Concept location is a necessary but all too often laborious task during maintenance phases. Part of the reasons is that repeatedly the same or similar concepts have to be reconstructed, which is a resource and time-consuming process. This contribution investigates the situation and suggests a framework that persistently stores conceptual elements and their dependencies in an *SQL* database. On the example of formal Z specifications it demonstrates that concept location is alleviated by simple queries that automatically identify concepts based on the database entries.

Keywords: Concept location, Formal Z specifications, Maintenance.

1 Introduction

"*People like to write code, but they do not like to read somebody else's code.*" This statement becomes increasingly apparent as the number of software systems in use is growing – and have to be maintained. Why might this be the case?

In [1, p.242] it is postulated that it is easier to express ones owns concepts and ideas into the tight formality of a (programming) language than to reconstruct the concepts the original developer had in mind from the formal expressions formulated in low level code. This observation is above all true when the text or code expresses a concept previously unknown to the reader – which is often the case in maintenance situations. In the lucky case there are at least high-level specification documents around, but, without supporting tools, the identification of the relevant information stays a hard business.

Maintenance activities are often formulated in terms of adding/changing/deleting features or concepts [2], and concept location techniques play an important role, in software as well as in specification maintenance. Formal specification frameworks provide excellent support for editing and verification, but they do not provide concept location facilities. A (semi-) automatic identification of concepts is missing and, for formal specifications, also the possibility to store the, often cumbersome, reconstructed concepts to be found in the documents.

The objective of this contribution is to demonstrate that not so much has to be done in order to identify *and* store concepts. We introduce a generic model that is able to deal with documents and concepts of different types. As a proof of concept a prototype for formal Z specifications [3] has been implemented. But the basic ideas also apply to other artifacts ... from natural language descriptions to program code.

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 248–261, 2010.

The paper is structured as follows. Sec. 2 discusses the related work. Sec. 3 derives the requirements for a framework that is able to persistently store identified concepts. With the necessary basics in concept location of Sec. 4 in mind, Sec. 5 introduces the key ideas behind our suggested framework. Sec. 6 evaluates its functionality in respect to correctness and time complexity. The contribution closes with a short summary and an outlook of work to be done.

2 Related Work

Due to the size of today's systems, maintenance and reverse engineering activities are usually supported by tools and frameworks.

At first, there are SW-Engineering frameworks that can also be used for reverse engineering activities [4,5,6,7,8]. Usually, they enable the reconstruction or extractions of diagrams from code, and thus establish links between different representations supporting round-trip engineering. Their disadvantage is that they are limited to a very specific notation (e.g. UML) or assume that the code has already been written within the framework.

Another group is that of explicit reverse engineering tools. There, the input is the code or an abstract representation of it, and they sustain the process of creating extractions or views onto the source. Popular representatives are RIGI [9], going back to the work of Müller, Tilley, and Wong in the early 90s, or Bauhaus [10]. In the meantime there are a lot of extensions and, especially for C++ and Java programs, similar frameworks [11,12,13,14,15,16].

Finally, there are frameworks that focus explicitly on concept location [17,18,19]. They make use of techniques similar to those of reverse engineering environments, but provide additional support for storing and retrieving previously identified concepts.

Environments for formal specifications have their focus on writing down a syntactically correct specification and providing verification support. They are not meant to be used for reverse engineering. However, one tool-set that permits looking at a specification from different angles is *VDMTools* with its *RoseLink* feature [20]. It can be used to generate UML diagrams from VDM specifications. Tools for *B* also focus on the creation of the specification. Some representatives, e.g. Atelier-B [21], at least provide views onto dependencies between the components. In the case of Z the situation is almost the same. One exception is the *ViZ* toolkit [22], where the emphasis was laid on concept location. But *ViZ* also has its limitations, first and foremost the inability to persistently store identified concepts.

3 Maintenance Support

The motivation for this work goes back to a project where we tried to improve maintenance and re-engineering activities of formal Z specifications. A big advantage of formal specifications is their semantic density. One can express his or her thoughts precisely and on a high level of abstraction. But with that, the complexity (and density) of even small specifications is quite high. As shown in [23], specifications might have thousands of dependencies between their elements, and comprehension aids are definitely necessary.

3.1 RE of Formal Z Specifications

At first sight approaches from the traditional field of software comprehension are not suitable. Z (among others) is a state-based, declarative specification language, with no explicit control and data flow – dependencies that are typically utilized when looking for concepts. But there is a solution to this problem.

In [24, p.60–63] a syntactical approximation to the identification of dependencies was described, which then enabled the identification of concepts like slices, chunks, and clusters within formal specifications. *ViZ* (for *Vi*zualization of *Z* Specifications) implements these algorithms and supports typical comprehension activities. But with its employment the following issues have been observed:

- The same comprehension steps are often carried out more than once - even if there is additional documentation. So dependencies and concepts identified once have to be reconstructed again.
- The calculation and identification of concepts is still a time consuming task. Moreover, these calculations and findings are lost when the framework is closed and the state is not saved for later use.
- It is not sufficient to look at a concept in isolation. Depending on the problem at hand more than just a single view onto the artifact has to be generated.

To summarize, a framework sustaining comprehension tasks should not ignore the above observations. It should support the identification of new concepts at different levels, enable the linkage between them, and be able to store the findings persistently in a database. Please note that the observations above are not only limited to the field of Z specifications. They apply to other artifacts, too. Due to the resource-consuming calculations necessary for dense and complex formal specifications, the storage of concepts is of major importance in our case.

3.2 Multi-dimensional Problem

The reconstruction of concepts within an artifact is not trivial. In order to reconstruct (or better approximate) the concepts a former developer had in mind, one has to take into account that different facets led to the writing of the artifact:

- *The environment and context of the problem.* There are maybe several assumptions the developer had to consider and that are not fully documented. So, an artifact only makes sense when put into the right context.
- *The concepts inherent in the language.* Different (programming) languages are differently suitable for describing problems. In fact, the semantics behind a language is often used to reduce writing effort. E.g. dependencies do not have to be made explicit, names are declared once, and it is clear when they are usable and when not. Those concepts, let us call them *"behind the scenes"*, are important for grasping the whole meaning.

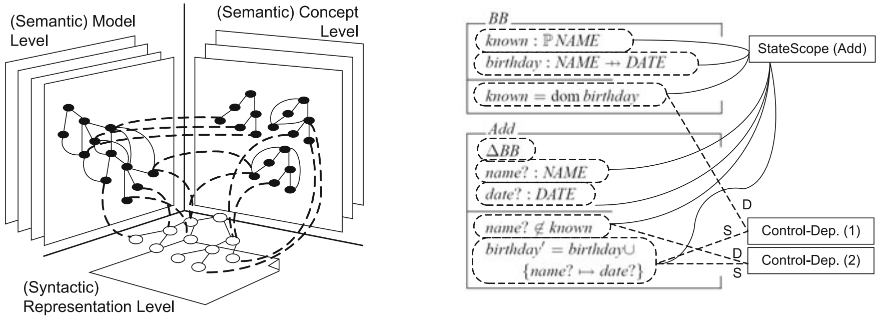


Fig. 1. (Left) An artifact contains concepts in three different dimensions and at different levels. (Right) Specifications are dismantled into syntactical elements (primes) and then extended by dependency and scope annotations.

- *Concepts made explicit in the source.* The concepts mentioned above are problem- and language-inherent. What is left are those concepts that are visible in the artifact. E.g. a case-statement represents an n-ary decision, and its meaning is clearly defined. Such concepts are called *”before the scenes”*.

When one is at least familiar with the problem field and the environment, the identification of the concepts behind and before the scenes might be seen as a multidimensional problem. Fig. 1 (left side) demonstrates this viewpoint.

The **(Syntactic) Representation Level.** At first, there is the artifact itself. It has been written in some pre-defined language, with clearly defined rules for its syntax. It is assumed that it can be divided into a structured set of basic elements (primes, statements, paragraphs ...). This sequence of elements, its nouns and verbs, and the structure make up a lot of the underlying concept(s). Thus, the representation level deals with the source and the structure of the artifact.

The **(Semantic) Model Level.** In general, a language comes along with a clearly defined semantics (e.g. statements have to be put into some order). This implies a specific meaning and leads to several dependencies and relations between the elements (see Fig. 1). These rules are not written down in the artifact, but belong to it and make up another part of the underlying concept(s). They can be seen as named concepts, going back to the semantic possibilities of the language at hand.

The **(Semantic) Concept Level.** What is left are the concepts the developer expresses unconsciously. They describe specific aspects of the problem and are recognizable when looking at the artifact from some distance. These mental macros, as Baxter et. al. [25] call them, express higher-level concepts, and program comprehension techniques are typically used to carve them out. To this dimension belong concepts like slices [26], chunks [27], clichés [28], and different types of clusters [29].

To exemplify the situation, the calculation of a specification or program slice (stored in the concept level) depends on the concept of dependencies (model level), the concept of scope (model level), and the basic elements in the source (representation level). When these concepts (at different levels) are calculated they can be stored in a database for later use.

4 Formal Specification Concepts

The model presented above has been mapped to a database schema and forms the basis for the concept location process. Though the strength of the framework is to deal with different types of documents, our experiences arise from the scope of maintaining formal Z specifications – which also was the starting point of the requirements' considerations. The specification concepts we are interested in are those as described in more detail in [22]: slices, chunks, and clusters.

4.1 Conceptual Elements

A *formal specification concept* is a coherent, abstract (or generic) pattern of specification text that is generalized from particular instances of the specification. It can be understood and recognized as a whole even when standing alone.

As explained in more detail in [22, p.81], the basic elements such concepts are built upon are called specification primes. Such *formal specification primes* (also called prime-objects) also represent the basic entities of a specification. They are built from literals of the specification and form logic, syntactic, or semantic units. A prime is a syntactically coherent sequence of literals within a specification, forming semantic entities that can be paraphrased by a short sentence in natural language.

With programming languages, primes would be programming statements. In the case of formal Z specifications these primes are declarations, definitions, and predicates. Fig. 1 (to the right) marks the primes by dashed ellipses, e.g. the prime "$name? \notin known$" (the second prime from below).

When primes are combined they do form so-called *higher-level primes*. The *Add* schema operation in Fig. 1 is an example of such a higher-level prime, telling the user about the things happening when the operation applies.

4.2 Specification Concepts

Concepts within formal specifications are identified in an iterative manner [22, p.83]. Starting with a domain-level request, one forms a mental model of the problem in mind and concept location is about to begin. Concept candidates are identified and matched against the model of the problem. When the candidates match, the concept is (very likely) identified and the elements of the related candidates are tagged. The concept location process makes use of the following steps: pattern matching, slicing and chunking, and cluster identification.

As explained in [30], experienced users first browse the text and try to identify relevant parts by *grep*-ing for keywords. When this is not successful, more complicated methods are used. Structures are especially of interest, and clustering is a feasible way in identifying related regions. The selection might be based on the use of identifiers, or on the number of dependencies that glue the primes together. Similarly, slices and chunks can be generated for a point of interest by just looking at specific primes and by following different types of dependencies.

Specification concepts are identified by looking at dependencies among primes. For the calculation of slices, chunks, and clusters, control- and data-dependencies are needed, and though these dependencies are not explicitly available, they can be reconstructed by looking at pre- and post- conditions. The approach goes back to the work of Oda and Araki [31] and has been refined in [32,24]. The basic idea is that, within a specific scope, primes that are part of post-conditions are dependent on primes that contribute to pre-conditions. In order to ease their identification, all primes get tagged with annotations. For every identifier used in a prime the following meta-information is assigned to the prime: CI (channel input) when it is an input identifier which is decorated by a $\boxed{?}$, CO (channel output) for an result identifier decorated by an $\boxed{!}$, D (declaration) for an identifier that denotes the identifier's after-state and which is decorated by a $\boxed{'}$, T (type declaration) for identifiers that are declared, and U (used) otherwise. So, the two Z primes (of the *Add* schema in Fig. 1)

$$\begin{array}{ll} P1: & name? \notin known \\ P2: & birthday' = birthday \cup \{name? \mapsto date?\} \end{array}$$

would be tagged as follows. Prime $P1$ is annotated by $\{CI \mapsto \{name\}, U \mapsto \{known\}\}$, and prime $P2$ is annotated by $\{D \mapsto \{birthday\}, U \mapsto \{birthday\}, CI \mapsto \{name, date\}\}$. Post-condition primes are those primes that have a tag containing a D or CO set. In our case $P2$ would be a post-condition prime, prime $P1$ is a so-called pre-condition prime.

The identification of dependencies is explained in more details in [24, pp.126–132]. However, when the meta-information is stored in the database (and assigned to the prime objects), the queries are quite simple. Our agents, as introduced in Sec. 5.2, make use of this meta-information in form of *SQL* queries.

5 Concept Location Framework

The framework for identifying the different concepts in Z specifications is designed to cope with different types of artifacts. It implements a traditional client-server architecture pattern based on the *EJB* Technology. The client is responsible for visualizing the results and for triggering the concept extraction. On the server side it is designed to handle different types of artifacts. Whenever a document is stored, different analysis tasks are started by a scheduler extracting concepts, and the findings are stored in the database again (see Sec. 5.2).

5.1 Database

The database forms the basis for the management of conceptual elements and their dependencies, sustaining the concept location process of formal Z specification documents.

There are four areas covered by the database. Three of these areas are related to the multi-dimensional view as described in Sec. 3.2. The forth area is used for the management of artifacts within the software engineering life-cycle.

Management/Project Pane. Based on the software engineering process, the *Management/Project* section deals with the management of artifacts within different phases of

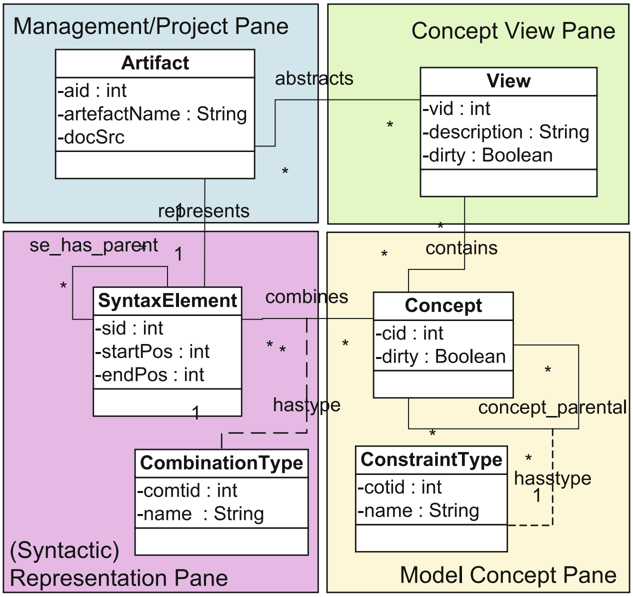


Fig. 2. The four different panes of the database model (Please note that for reasons of space only the major entities are shown. See [33] for more details.)

the project. There, a *Project* consists of different *Phases*. Within each phase *Artifacts* are created, most of them depending on each other. Different artifact versions might exist. Hence, the database schema takes this into account by assigning the *ArtifactMetaData* information to an artifact.

(Syntactic) Representation Pane. Every artifact, independently of its nature, consists of a certain structure. This structure is built upon so called *SyntaxElement*s. *SyntaxElements* are characterized by their *ElementType*s:

- *Content:* It represents a pure structural element (so-called basic elements like sentences, expressions, or statements).
- *Presentation:* Text is often decorated (e.g. by boxes). As sometimes this decoration carries information, it is also stored.
- *Aggregation:* It provides the possibility to explicitly mark higher-level concepts that have been created by the aggregation of basic elements.

Syntax elements carry a lot of information. E.g. they refer to identifiers, define labels, or describe some input operations. A set of meta-data is introduced to store them. Every data entry of *ElementMetaData* belongs to a specific *AnnotationType*, and so different (but consistent) categorizations get possible.

Model Concept Pane. A *Concept* corresponds to one or several syntactical elements (*SyntaxElements*). For different types of concepts also different relationships are possible. This is done by the *CombinationType* entity. Besides, it is possible to express some kind of direction or ordering between the related elements. The characterization of a concept is made up by the *ConceptType* entity. Concepts also form hierarchies, and to express these relations, an n-to-m recursive relationship is introduced.

Concept View Pane. The database schema allows for different views onto the concepts, be them explicitly or implicitly available. The main purpose of the *View* entity is to cluster related concepts (concepts of the same type) or to form different views onto the current artifact. This information is, besides the creation date, stored in the *ViewMetaData* entity. Every view belongs to a certain category. This classification is stored in the *ViewType*.

It is also possible to annotate a view with *ViewData* entries of specific *ViewDataTypes*. Those entries are, e.g., used to store metrics of clusters or other characteristics relevant for concepts within the view. These steps are carried out by agents like those introduced in the following section.

5.2 Agents

Our prototype provides different agents: scope agents that regard scope rules of Z, dependency agents for reconstructing dependencies, and, based on them, slice/chunk/ cluster agents for carving out higher-level concepts.

For the creation of slices or chunks typically two different types of dependencies (data-, and control-dependencies) and the syntactical environment are necessary. So, at first, these dependencies have to be extracted, but the extraction is complicated due to language-specific scope rules. The first task for our framework is therefore to reconstruct the concepts representing the scope.

In the context of formal Z specifications three types of scopes can be identified (and are calculated by three agents in our framework). The *Declaration Scope* represents all visible declarations for a prime in the specification. The scope is also needed to derive the syntactical dependencies and thus for building syntactically correct partial specification. The *State Scope* deals with schema inclusions within a specification document and aggregates the primes of the inclusion and the primes of the including schemata (see Fig. 1, $Add_{(State)}$ for an example). Finally, the *Connectivity Scope* merges all primes of two or more schemata combined via schema operations.

In our framework, at first, the agents launch queries to identify the correct scopes, then they start reconstructing control- and data dependencies. Sec. 5.3 demonstrates the simplicity on the example of control dependency identification.

After scope and dependency calculation the *Slice* and *Chunk* agents can be activated by the user. Beginning with a ”point of interest” (a set of primes), the agents calculate slices and chunks by following the stored control- and data dependencies. The results are again stored in the database for later use.

The last agent presented here is called *Clustering Agent*. It is responsible for the generation of clusters of a specification document. In order to ease deciding about the most useful number of clusters to be generated, the agent pre-calculates and stores all variants of them. Every cluster view is then extended by meta-information. This meta-data describes different types of cluster-metrics, like the partition entropy or the partition coefficient measure. This information can later help the user to decide about the usefulness of the clusters.

Some of the agents are executed in parallel; other agents have to wait. Therefore all agents are registered in an agent scheduler, which is responsible for the right execution order.

5.3 Queries for Concept Location

Our framework makes use of a simple idea: the calculation logic is moved from traditional program code to SQL queries disposed by the agents. The extraction is done by expressions which are based on the annotations of the primes in the database. For Z documents the necessary queries are already implemented.

To demonstrate the elegance of the queries we look at the steps necessary to carve out control dependencies from Z specifications. The relevant primes in the database are the syntax elements tagged by the *Content* element type. The extraction-process then uses the *State* and *Connectivity Scope* for the calculation.

$$\begin{aligned}&\Pi_{sid}\ \sigma_{AnnotationType.name=''D''}\\&((\sigma_{Concept.id=act}Concept \bowtie\\&(\sigma_{ConceptType.name=''State''}\ ConceptType))\ \_\\&\bowtie SyntaxElement \bowtie\\&ElementMetaData \bowtie AnnotationType)\end{aligned} \tag{1}$$

$$\begin{aligned}&\Pi_{sid}(\sigma_{Concept.id=act}Concept \bowtie\\&(\sigma_{ConceptType.name=''State''}\ ConceptType))\\&[sid \neq sid]\\&\Pi_{sid}\\&(\sigma_{\substack{AnnotationType.name\neq''T''\ or\\ AnnotationType.name\neq''C''\ orAnnotationType.name\neq''D''}}\\&SyntaxElement \bowtie ElementMetaData\\&\bowtie AnnotationType)\end{aligned} \tag{2}$$

The queries (1) and (2) above extract control–dependencies of the *Add* schema operation of the *'Birthday Book'*-Specification (where *act* represents the identifier of the current *State Scope*). The queries lead to the source (*S*) and destination (*D*) primes for the dependency arcs. In fact, the results of the queries are elements of the *SyntaxElement* entity. The agent takes all elements resulting from the first query and connects them to the resulting elements of the second query. Every pair forms a *Concept* within the database. This information is stored in the database and results in the concept entries *Control-Dep. (1)* and *Control-Dep. (2)*) as exemplified in Fig. 1.

The identification of data–dependencies is similar to that of control–dependencies. Its only difference is related to the *U* tag, and the consideration of the name of an identifier. A detailed description of the queries for scope and dependency calculation can be found in [33, p.102-106].

Table 1. Complexity attributes and calculation time (in seconds)

Specification	Lines	Pages A4	Primes	CD	DD	ViZ [s]	EJB/DB [s]	Concepts
BB	40	2	34	10	5	4.6	7.0	36
Cinema	95	4	74	121	43	75.3	43.2	114
Petrol	88	3	65	192	177	152.9	51.9	219
Elevator	193	6	185	1,628	992	1,223.4	709.3	984

6 Evaluation

The evaluation of the framework was carried out in two steps. First, the correctness of the identified concepts were checked, and, secondly, the usefulness in respect to performance explored. In fact, both steps also hearken back to results we gained from the existing *ViZ* framework.

6.1 Setting and Correctness

The first step was the validation of the concepts that have been identified by the agents and stored in the database. The evaluation involves specifications of raising sizes, known as Birthday Book [3], Petrol Station [24], and Elevator [32]. Additionally, a student's specification (Cinema) was added to the set. Tab. 1 (left part) summarizes the key attributes in order to assess the complexity of the specifications. It exemplifies the number of lines, pages (when printed), primes, control- (CD), and data dependencies (DD).

The correctness of the identified concepts was checked in two steps. At first, the new framework was used to identify dependencies, slices, and chunks. The results were then exported to a structured file. In a second step these entries have been compared to the concepts and dependencies identified by the *ViZ* framework. As Tab. 1 (right column) demonstrates, this involved 1353 concepts (consisting of slices and chunks for every prime occurring in the predicate part of every schema) and 3168 dependencies (CD and DD).

6.2 Performance Considerations

As every dependency and concept has been detected correctly, we were also eager to see whether the framework scales and improves operating speed. In fact, in our case

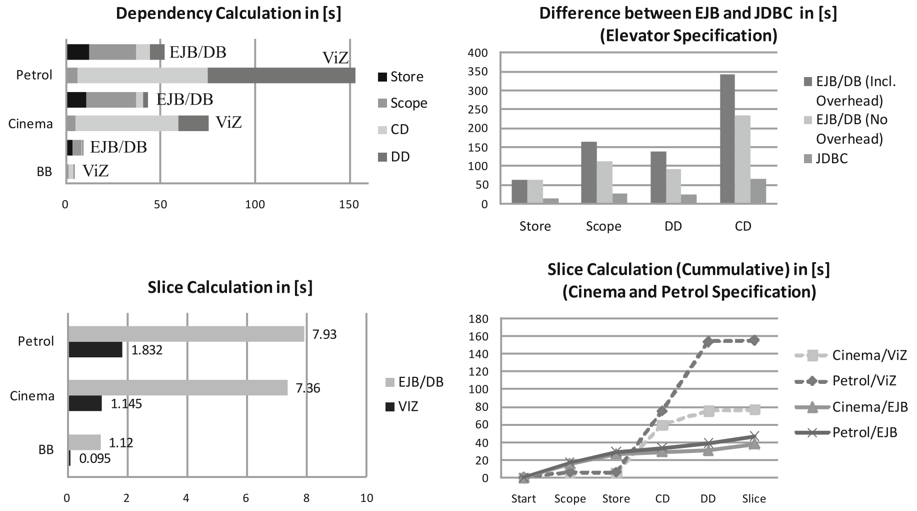


Fig. 3. Performance considerations between the ViZ and the EJB framework

operation time it up to (a) dependency calculation, (b) storage and retrieval, and (c) concept identification.

In the case of *ViZ* the calculation of dependencies (and thereinafter slices or chunks) is time-consuming. *ViZ* uses an annotated graph to store primes and its connections, and dependency calculation is based on reachability considerations. It has a runtime performance of $O(n*(m+n*log*n))$ (with n related to the number of primes and m related to the number of dependencies in the specification). The new framework considers Def-Use equations based on scope relations (that are stored in the database), and its runtime complexity is in $O(n^2)$. Tab. 1 (center part) presents the time needed to calculate all dependencies for the *ViZ* environment and the *EJB* based framework (where the system consisted of a Windows XP Professional OS, Intel Core2 CPU, 2.00GHz, 2 GB RAM). This difference can also be seen in Fig. 3 (top left). Though *ViZ* does not store the elements in a database, the total time is much higher due to the extra time needed for control and data dependency calculation.

The type of the database access is also crucial for the performance. The most complex artifact in our considerations is the *Elevator* specification, and it takes about ten minutes till all dependencies are analyzed (and about 2,600 data-sets are stored for later use). As a few thousand data-sets are not so much for a database, we were eager to know why it took so long.

It turned out that a lot of time is lost due to *EJB*'s synchronization between the database and Java's objects. The overhead is about *one-third* of the time. Furthermore, there is very high execution time latency between *EJB* and its corresponding *JDBC* queries. As explained in more details in [34, p.234], in our setting *JDBC* scales about *six times* better than *EJB*. Fig. 3 (top right) demonstrates this time-differences on the example of the elevator specification.

The calculation of concepts like slices or chunks implies looking at a specific prime and following the relevant dependencies. Fig. 3 (bottom left) shows that *ViZ* is definitively faster than the new environment. There, all possible slices for three different specifications have been generated once and the total time measured. *ViZ* is much faster, which is not surprising as it's internal graph already contains the dependencies as arcs and they do not have to be read from a database. However, the new framework stores the slice as a view for later use, and calculated once, it does not have to be (re-)calculated again.

Considering the above observations, the new framework seems to be an improvement in the case of concept location environments for Z specifications. Fig. 3 (bottom right) demonstrates this by accumulating the time till all possible slices have been calculated once. *ViZ* is faster at the beginning, as it does not store the elements in a database, but the new framework invests in storing the syntactical elements in the database and assigns scope information to it. This investment pays back when dependencies are to be calculated, and it outpaces *ViZ*. Retrieving the concepts then is slower, but merely depends on the number of elements to be retrieved by a select operator. In addition to that, they have only to be retrieved once, as after retrieval they are stored as a view in the database. Here the strengths of a relational database pay off.

Though the new framework is faster, we conclude from the analysis above that the use of *EJB* is less suitable. It brings maintenance advantages, but, as also addressed in

[34], one has to expect performance loss that should not be neglected. For this reason we are currently working on a new release of the framework that replaces the middleware technology by *JDBC*.

7 Conclusions

This paper introduces the problem of concept location within state-based specifications and motivates for a framework that persistently stores concepts for later use and fast access. Starting with a thorough analysis of concept location aspects, a database schema has been developed which, thereinafter, forms the basis for our concept location framework for formal Z specifications.

The paper then introduces the key ideas behind our prototype. Besides storing concepts, it is based upon the idea of individual agents that quickly identify different relations among syntactical elements (of our specification) stored in the *MySQL* database. Their elegance originates from the fact that an *SQL* database is very efficient in looking for specific relations between elements, and thus most of the calculation logic could be put into slim *SQL* queries.

The evaluation is based on a comparison with *ViZ*, an already existing concept location framework for Z specifications. The evaluation shows that it produces the same results than *ViZ*, but calculation times varied. The performance of the framework was strongly influenced by *EJB*. The analysis of *JDBC* and *EJB* shows a high factor of performance loss when using *EJB*. *JDBC* scales about six times better than *EJB* in terms of runtime. Additionally, *EJB* implements an intermediate layer and, therefore, runs into performance latencies. It is going to be replaced by *JDBC* in the next release of our framework.

References

1. Mittermeir, R.T., Bollin, A.: Demand-driven Specification Partitioning. In: Proceedings of the 5th Joint Modular Languages Conference (2003)
2. Wilde, N., Scully, M.C.: Software Reconnaissance: Mapping Program Features to Code. Journal of Software Maintenance: Research and Practice 7, 49–62 (1995)
3. Spivey, J.: The Z Notation: A Reference Manual, 2nd edn. Prentice Hall International, Englewood Cliffs (1992)
4. Rational-XDE: IBM Rational XDE DeveloperWorks Home Page, www.ibm.com/developerworks/rational/products/xde/ (Page last visited: March 2009)
5. Eclipse-GMT: Homepage, http://www.eclipse.org/gmt/ (Page last visited: March 2009)
6. Nickel, U., Niere, J., Wadsack, J., Zündorf, A.: Roundtrip Engineering with FUJABA. In: Ebert, J., Kullbach, B., Lehner, F. (eds.) Proceedings of 2nd Workshop on Software-Reengineering (WSR), Bad Honnef, Germany (August 2000)
7. Jouault, F.: Loosely Coupled Traceability for ATL. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA 2005), Workshop on Traceability (2005)
8. MetaEdit+: Homepage, www.metacase.com (Page last visited: March 2009)

9. Müller, H.A., Tilley, S.R., Wong, K.: Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project. In: CASCON 1993, October 1993, pp. 217–226 (1993)
10. Koschke, R.: Software Visualization for Reverse Engineering. In: Diehl, S. (ed.) Dagstuhl Seminar 2001. LNCS, vol. 2269, pp. 524–527. Springer, Heidelberg (2002)
11. Ferenc, R., Beszedes, A., Tarkiainen, M., Gyimothy, T.: Columbus – Reverse Engineering Tool and Schema for C++. In: IEEE International Conference on Software Maintenance, Montreal, Canada, pp. 172–181 (2002)
12. Korshunova, E., Petkovic, M., van den Brand, M.G.J., Mousavi, M.R.: CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code (Tool Paper). In: Working Conference on Reverse Engineering (WCRE 2006), Benevento, Italy (2006)
13. Computer Human Interaction and Software Engineering Lab (CHISEL): SHriMP Homepage, www.thechiselgroup.org/shrimp (Page last visited: October 2008)
14. Holt, R.: PBS – The Portable Bookshelf Homepage, http://www.swag.uwaterloo.ca/pbs/intro.html (Page last visited: October 2008)
15. Ebert, J., Kullbach, B., Riediger, V., Winter, A.: GUPRO – Generic Understanding of Programs – An Overview. Electronic Notes in Theoretical Computer Science 72(2) (2002)
16. Holt, R., Schürr, A., Sim, S.E., Winter, A.: GXL Graph Exchange Library Homepage, http://www.gupro.de/GXL/ (Page last visited: April 2008)
17. Chen, K., Rajlich, V.: RIPPLES: Tool for Change in Legacy Software. In: IEEE International Conference on Software Maintenance, p. 230. IEEE Computer Society, Los Alamitos (2001)
18. Xie, X., Poshyvanyk, D., Marcus, A.: 3D Visualization for Concept Location in Source Code. In: Proceedings of 28th IEEE/ACM International Conference on Software Engineering (ICSE 2006), May 20–28, pp. 839–842 (2006)
19. Poshyvanyk, D., Marcus, A.: Combining Formal Concept Analysis with Information Retrieval for Concept Location in Source Code. In: Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC 2007), June 26–29, pp. 37–48 (2007)
20. Agerholm, S., Larsen, P.G.: Applied Formal Methods – FM-Trends 98. In: Hutter, D., Traverso, P. (eds.) FM-Trends 1998. LNCS, vol. 1641, pp. 326–339. Springer, Heidelberg (1999)
21. Engineering, C.S.: The Atelier-B Homepage, http://www.atelierb.eu/index-en.php (Page last visited: June 2009)
22. Bollin, A.: Concept Location in Formal Specifications. Journal of Software Maintenance and Evolution: Research and Practice 20(2), 77–104 (2008)
23. Bollin, A.: The Efficiency of Specification Fragments. In: Proceedings of the 11th IEEE Working Conference on Reverse Engineering (2004)
24. Bollin, A.: Specification Comprehension – Reducing the Complexity of Specifications. PhD thesis, Universität Klagenfurt (April 2004)
25. Baxter, I.D., Yahin, A., Moura, L., SantAnna, M., Bier, L.: Clone Detection Using Abstract Syntax Trees. In: Proceedings of the International Conference on Software Maintenance, pp. 368–377. IEEE Computer Society, Los Alamitos (1998)
26. Weiser, M.: Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. PhD thesis, University of Michigan (1979)
27. Burnstein, I., Roberson, K., Saner, F., Mirza, A., Tubaishat, A.: A Role for Chunking and Fuzzy Reasoning in a Program Comprehension and Debugging Tool. In: TAI 1997, 9^{th} International Conference on Tools with Artificial Intelligence, November 1997. IEEE press, Los Alamitos (1997)

28. Broad, A., Filer, N.: Applying Case-Based Reasoning to Code Understanding and Generation. In: Proceedings of the Fourth United Kingdom Case-Based Reasoning Workshop (UKCBR4), University of Salford, Salford, England, September 1999, pp. 35–48 (1999)
29. Wiggerts, T.: Using Clustering Algorithms in Legacy System Remodularization. In: Proceedings of the 4th Working Conference on Reverse Engineering (WCRE 1997). IEEE Press, Los Alamitos (1997)
30. Rajlich, V., Wilde, N.: The Role of Concepts in Program Comprehension. In: International Workshop on Program Comprehension, pp. 271–278. IEEE Computer Society, Los Alamitos (2002)
31. Oda, T., Araki, K.: Specification slicing in a formal methods software development. In: Seventeenth Annual International Computer Software and Applications Conference, November 1993, pp. 313–319. IEEE Computer Socienty Press, Los Alamitos (1993)
32. Chang, J., Richardson, D.: Static and Dynamic Specification Slicing. In: Proceedings of the Fourth Irvine Software Symposium, Irvine, CA (April 1994)
33. Pohl, D.: Specification Comprehension – Konzeptverwaltung am Beispiel zustandsbasierter Spezifikationen (in German). Master's thesis, University of Klagenfurt, Software Engineering and Soft Computing (Juli 2008)
34. Pohl, D., Bollin, A.: Database-Driven Concept Management – Lessons Learned from Using EJB Technologies. In: 4th International Conference on Evaluation of Novel Approaches to Software Engineering (May 2009)

A Model Driven Approach to Upgrade Package-Based Software Systems*

Antonio Cicchetti[1], Davide Di Ruscio[1], Patrizio Pelliccione[1],
Alfonso Pierantonio[1], and Stefano Zacchiroli[2]

[1] Università degli Studi dell'Aquila, Dipartimento di Informatica, Italy
{cicchetti,diruscio,pellicci,alfonso}@di.univaq.it
[2] Université Paris Diderot, PPS, UMR 7126, France
zack@pps.jussieu.fr

Abstract. Complex software systems are often based on the abstraction of *package*, brought to popularity by Free and Open Source Software (FOSS) *distributions*. While helpful as an encapsulation layer, packages do not solve all problems of deployment, and more generally of management, of large software collections. In particular *upgrades*, which can affect several packages at once due to interpackage dependencies, often fail and do not hold good transactional properties. This paper shows how to apply *model driven* techniques to describe and manage software upgrades of FOSS distributions. It is discussed how to model static and dynamic aspects of package upgrades—the latter being the more challenging to deal with—in order to be able to *predict* common causes of upgrade failures and *undo* residual effects of failed or undesired upgrades.

Keywords: Model-driven engineering, Software change and configuration management, Metamodeling, Open source, Package.

1 Introduction

Increasingly, software systems are designed to routinely accommodate new features before and after the deployment stage. The deriving evolutionary pressure requires the system design and architecture to have enhanced quality factors: in particular, they have to retain the (user perceived as well as system-intrinsic) dependability at a satisfactory level and make component installation/removal operations less haphazard [2].

Free and Open Source Software (FOSS) distributions are among the most complex software systems known, being made of tens of thousands components evolving rapidly without centralized coordination. Similarly to other software distribution infrastructures, FOSS components are provided in "packaged" form by distribution editors. Packages define the granularity at which components are managed (installed, removed, upgraded to newer version, etc.) using *package manager* applications, such as *APT*[1] or *Apache maven*[2]. Furthermore, the system openness affords an anarchic array of dependency

[*] Preliminary results appeared in [1].

[1] APT howto: http://www.debian.org/doc/manuals/apt-howto

[2] Apache Maven Project: http://maven.apache.org

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 262–276, 2010.

modalities between the adopted packages. These usually contain *maintainer scripts*, which are executed during the upgrade process to finalize component configuration. The adopted scripting languages have rarely been formally investigated, thus posing additional difficulties in understanding their side-effects which can spread throughout the system. In other words, even though a package might be viewed as a software unit, it lives without a proper component model usually defining standards (e.g., how a component interface has to be specified and how components communicate) [3,4] which facilitate integration and assure that components can be upgraded in isolation.

The problem of maintaining FOSS installations, or similarly structured software distributions, is intrinsically difficult and a satisfactory solution is missing. Today's available package managers lack several important features such as complete dependency resolution and roll-back of failed upgrades [5]. Moreover, there is no support to simulate upgrades taking the behavior of maintainer scripts into account. In fact, current tools consider only inter-package relationships which are not enough to predict side-effects and system inconsistencies encountered during upgrades.

This work is part of the MANCOOSI[3] project which aims at improving the management of complex software systems built of composable units evolving independently. In particular, this paper describes a model-driven approach to specify system configurations and available packages. Maintainer scripts are described in terms of models which abstract from the real system, but are expressive enough to predict several of their effects on package upgrades. Intuitively, we provide a more abstract interpretation of scripts, which focuses on the relevant aspects to predict the operation effects on the software distribution. To this end, models can be used to drive roll-back operations to recover previous configurations according to user decisions or after upgrade failures.

The paper is structured as follows: Section 2 describes the upgrade process of FOSS packages and, briefly, the MANCOOSI project. Section 3 describes a model driven approach to (i) specify system configurations and packages, (ii) simulate the installation of software packages, (iii) assist roll-backs. Section 4 analyzes the FOSS domain and introduces the required modeling constructs which are captured in different metamodels. Finally, Sections 5 and 6 present related and future work, respectively.

2 Packages, Upgrades and Failures

Overall, the architectures of all FOSS distributions are similar. Each user machine has a local *package status* recording which packages are currently installed and which are available from remote repositories. Package managers are used to manipulate the package status and can be classified in two categories [6]: *installers*, which deploy individual packages on the filesystem (possibly aborting the operation if problems are encountered) and *meta-installers*, which act at the inter-package level, solving dependencies and conflicts, and retrieving packages from remote repositories as needed. In an *upgrade scenario*, a user request (install, remove, upgrade to a newer version, etc.) is submitted to a meta-installer to change the system configuration status; the aim of the meta-installer is then to find a suitable *upgrade plan*, where one exists. In the rest of the

[3] MANCOOSI project: http://www.mancoosi.org

section we give a brief description of packages (as they can be found in current distributions), their role in the upgrade process, and the failures that can impact on upgrade deployment.

Packages. Abstracting over format-specific details, a *package* is a bundle of three main parts: (1) set of files, (2) meta-information, (3) maintainer scripts. The core of a package is the set of *files* (1) that ships: executable binaries, data, documentation, etc. *Configuration files* are a distinguished subset of shipped files, which are tagged as affecting the runtime behavior of the package and meant to be customized by local administrators. Configuration files need to be present in the bundle (e.g., to provide sane defaults or documentation), but need of special treatment: during upgrades they cannot be simply overwritten by newer versions, as they may contain local changes which should not be thrown away.

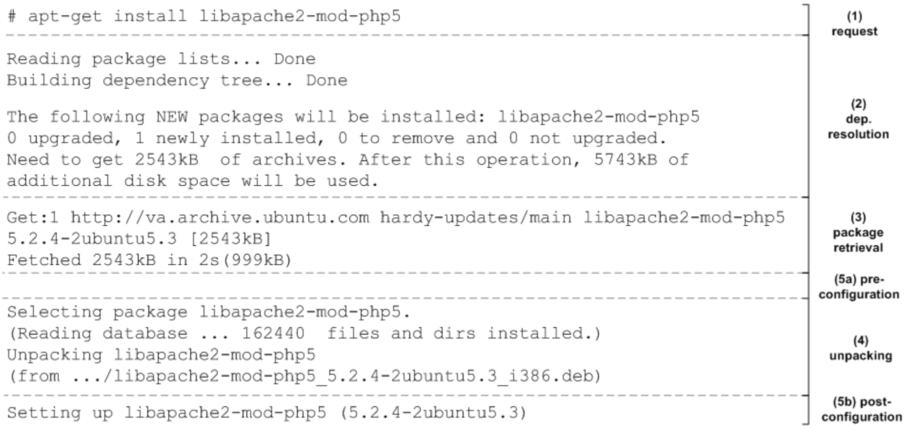
Package *meta-information* (2) is used by meta-installers to design upgrade plans. Details change from distribution to distribution, but a common core of meta-information consists of: a unique identifier (the name), software version, maintainer and package description, *inter-package relationships*. These relationships represent the most valuable information for dependency resolution and usually include: dependencies (the need of other packages to work properly), conflicts (the inability of being co-installed with other packages), feature provisions (the ability to declare named features as provided by a given package, so that other packages can depend on them), and boolean combinations of them [6].

Packages come with a set of executable *maintainer* (also known as "configuration") *scripts* (3). Their purpose is to attach actions to hooks invoked by the installer. The most common use case for maintainer scripts is to update some cache, blending together data shipped by the package, with data installed on the system, possibly by other packages. Three facets of maintainer scripts are noteworthy:
(a) maintainer scripts are full-fledged programs, written in Turing-complete programming languages. They can do anything permitted to the installer, which is usually run with system administrator rights;
(b) the functionality of maintainer scripts can not be substituted by just shipping extra files: the scripts often rely on data which are available only in the target installation machine, and not in the package itself;
(c) maintainer scripts are required to work "properly": upgrade runs, in which they fail, trigger upgrade failures and are usually detected via inspection of script exit code.

Upgrades. Table 1 summarizes the different phases of what we call the *upgrade process*, using as an example the popular `APT` meta-installer. The process starts in phase (1) with the user requesting to alter the local package status. The expressiveness of the requests varies with the meta-installer, but the aforementioned actions (install, remove, etc.) are ubiquitously supported.

Phase (2) checks whether a package satisfying dependencies and conflicts exists (the problem is at least NP-complete [6]). If this is the case one is chosen in this phase. Deploying the new status consists of package retrieval, phase (3), and unpacking, phase (4). Unpacking is the first phase actually changing both the package status (to keep track of installed packages) and the filesystem (to add or remove the involved files).

Table 1. The package upgrade process



```
# apt-get install libapache2-mod-php5
---------------------------------------------------------------------
Reading package lists... Done
Building dependency tree... Done

The following NEW packages will be installed: libapache2-mod-php5
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 2543kB  of archives. After this operation, 5743kB of
additional disk space will be used.
---------------------------------------------------------------------
Get:1 http://va.archive.ubuntu.com hardy-updates/main libapache2-mod-php5
5.2.4-2ubuntu5.3 [2543kB]
Fetched 2543kB in 2s(999kB)
---------------------------------------------------------------------
---------------------------------------------------------------------
Selecting package libapache2-mod-php5.
(Reading database ... 162440  files and dirs installed.)
Unpacking libapache2-mod-php5
(from .../libapache2-mod-php5_5.2.4-2ubuntu5.3_i386.deb)
---------------------------------------------------------------------
Setting up libapache2-mod-php5 (5.2.4-2ubuntu5.3)
```

Intertwined with package retrieval and unpacking, there can be several configuration phases, (exemplified by phases (5a) and (5b) in Table 1), where maintainer scripts get executed. The details depend on the available hooks; dpkg offers: pre/post-installation, pre/post-removal, and upgrade to some version [7].

Exemple 1. PHP5 (a scripting language integrated with the Apache web server) executes as its postinst (post-installation) script the following snippet, on the left hand-side:

```
#!/bin/sh
if [-e /etc/apache2/apache2.conf];
  then
    a2enmod php5 >/dev/null || true
    reload_apache
fi
```

```
#!/bin/sh
if [-e /etc/apache2/apache2.conf];
  then
    a2dismod php5 || true
fi
```

Note that prerm is executed *before* removing files from disk, which is necessary to avoid reaching an inconsistent configuration where the Apache server is configured to rely on no longer existing files. The expressiveness of inter-package dependencies is not enough to encode this kind of dependencies: Apache does not depend on php5 (and should not, because it is useful also without it), but while php5 is installed, Apache needs specific configuration to work in harmony with it; at the same time, such configuration would inhibit Apache to work properly once php5 gets removed. The bookkeeping of such configuration intricacies is delegated to maintainer scripts.

Failures. Each phase of the upgrade process can fail. Dependency resolution can fail either because the user request is unsatisfiable (e.g., user error or inconsistent distributions [8]) or because the meta-installer is unable to find a solution. Completeness—the guarantee that a solution will be found whenever one exists—is a desirable meta-installer property unfortunately missing in most meta-installers, with too few claimed exceptions [9].

Package deployment can fail as well. Trivial failures, e.g., network or disk failures, can be easily dealt with when considered in isolation from the other upgrade phases: the whole upgrade process can be aborted and unpack can be undone, since all the involved files are known. Maintainer script failures can not be as easily undone or prevented, since all non-trivial properties about scripts are undecidable, including determining *a priori* which parts of file-system they affect to revert them *a posteriori*.

The Mancoosi project is working to improve upgrade support in complex software systems such as FOSS distributions. On one hand the project is working on algorithms for finding optimal upgrade paths addressing complex preferences, on the other is working on models and tools to *(i)* simulate the execution of maintainer scripts, *(ii)* predict side-effects and system inconsistences which might be raised by package upgrades, and *(iii)* instruct roll-back operations to recover previous configurations according to user decisions or after upgrade failures. In the rest of this paper we introduce the approach we are using, in the context of Mancoosi, to attack the problems of package upgrades, namely: modeling of the involved entities and upgrade simulation.

3 Proposed Approach

As discussed, the problem of maintaining FOSS installations is far from trivial and has not yet been addressed properly [5]. In particular, current package managers are neither able to *predict* nor to *counter* vast classes of upgrade failures. The reason is that package managers rely on package meta-information only (in particular on inter-package relationships), which is not expressive enough. Our proposal consists in maintaining a model-based description of the system and simulate upgrades in advance on top of it, to detect predictable upgrade failures and notify the user before the system is affected. More generally, the models are expressive enough to isolate *inconsistent configurations* (e.g., situations in which installed components rely on the presence of disappeared subcomponents), which are currently not expressible as inter-package relationships.

The adoption of model-driven techniques presents several advantages: *a)* models can be given at any level of abstraction depending on the analysis and operations one would like to perform as opposed to package dependency information whose granularity is fixed and often too coarse; *b)* complex and powerful analysis techniques are already available to detect model conflicts and inconsistencies [10,11]. In particular, contradictory patterns can be specified in a structural way by referring to the domain underlying semantics in contrast with text-based tools like version control systems where conflicts are defined at a much lower level of abstraction as diverging modifications of the same lexical element.

Figure 1 depicts the proposed approach. Basically, to simulate an upgrade run, two models are taken into account: the *System Model* and the *Package Model* (see the arrow ⓐ). The former describes the state of a given system in terms of installed packages, running services, configuration files, etc. The latter provides information about the packages involved in the upgrade, in terms of inter-package relationships. Moreover, since a trustworthy simulation has to consider the behavior of the maintainer scripts which are executed during the package upgrades, the package model specifies also an abstraction of the behaviors of such scripts. There are two possible simulation outcomes: *not valid* and *valid* (see the arrows ⓒ and ⓓ, respectively). In the former case

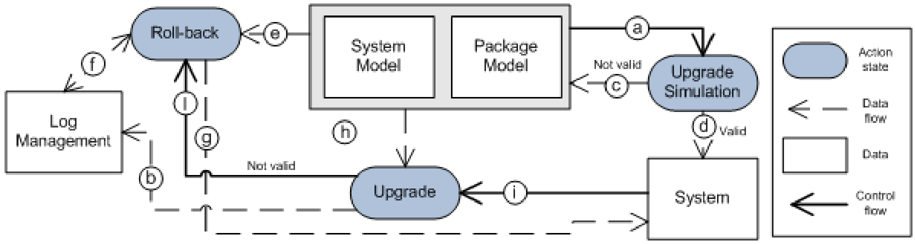


Fig. 1. Overall approach

it is granted that the upgrade on the real system will fail. Thus, before proceeding with it the problem spotted by the simulation should be fixed. In the latter case—*valid*—the upgrade on the real system can be operated (see the arrow ⓘ. However, since the models are an abstraction of the reality, upgrades failures might occur.

During package upgrades *Log models* are produced to store all the transitions between configurations (see arrow ⓑ). The information contained in the system, package, and log models (arrows ⓔ and ⓕ) are used in case of failures (arrow ⓛ) when the performed changes have to be undone to bring the system back to the previous valid configuration (arrow ⓖ). Since it is not possible to specify in detail every single part of systems and packages, trade-offs between model completeness and usefulness have been evaluated; the result of such a study has been formalized in terms of metamodels (see next section) which can be considered one of the constituting concepts of Model Driven Engineering (MDE) [12]. They are the formal definition of well-formed models, constituting the languages by which a given reality can be described in some abstract sense [13] defining an abstract interpretation of the system.

Even though the proposed approach is expressed in terms of simulations, the entailed metamodels do not mandate a simulator. Hybrid architectures composed by a package manager and metamodel implementations can be more lightweight than the simulator, yet being helpful to spot inconsistent configurations not detectable without metamodel guidance.

4 Modeling System and Packages

The simulation approach outlined in the previous section is based on a set of coordinated metamodels which have been defined by analyzing the domain of FOSS systems. In general, a metamodel specifies the modeling constructs that can be used to define models which are said to conform to a given metamodel like a program conforms to the grammar of the programming language in which it is written [13].

In this work, we have considered two complex FOSS distributions (the Debian[4] and Mandriva[5] distributions). Their analysis has induced the definition of three metamodels (see Figure 2) which describe the concepts making up a system configuration and a

[4] http://www.debian.org

[5] http://www.mandriva.com

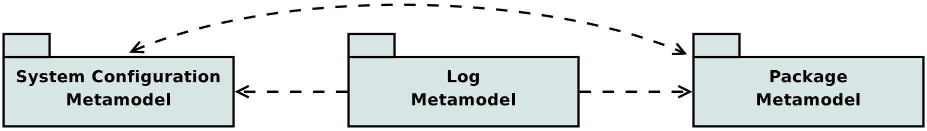


Fig. 2. Metamodels and their inter-dependencies

software package, and how to maintain the log of all upgrades. The metamodels have been defined according to an iterative process concisting of two main steps *a)* elicitation of new concepts from the domain to the metamodel *b)* validation of the formalization of the concepts by describing part of the real systems. In particular, the analysis has been performed considering the official packages released by the distributions with the aim of identifying elements that must be considered as part of the metamodels. Due to space constraints we report here only the results of the analysis, i.e., the metamodels themselves:

- the *System Configuration metamodel*, which contains all the modeling constructs necessary to make the FOSS system able to perform its intended functions. In particular it specifies installed packages, configuration files, services, filesystem state, loaded modules, shared libraries, running processes, etc. The system configuration metamodel takes into account the possible dependency between the configuration of an installed package and other package configurations. The ability to express such fine-grained and installation-specific dependencies is a significant advantage offered by the proposed metamodels which embody domain concepts which are not taken into account by current package manager tools;
- the *Package metamodel*, which describes the relevant elements making up a software package. The metamodel also gives the possibility to specify the maintainer script behaviors which are currently ignored—beside mere execution—by existing package managers. In order to describe the scripts behavior, the package metamodel contains the `Statement` metaclass, see Figure. 5, that represents an abstraction of the commands that can be executed by a given script to affect the environment, the file system or the package settings of a given configuration;
- the *Log metamodel*, which is based on the concept of transactions that represent a set of statements that change the system configurations. Transitions can be considered as model transformations [13] which let a configuration C_1 evolve into a configuration C_2.

As depicted in Figure 2, *System Configuration* and *Package* metamodels have mutual dependencies, whereas the *Log* metamodel has only direct relations with both *System Configuration* and *Package* metamodels.

4.1 Modeling Maintainer Scripts

The most challenging part of the conducted modeling process has been modeling maintainer scripts. The reason is twofold, on one hand maintainer scripts are the entities which contribute the most to difficulties in dealing with upgrade failures; on the other hand maintainer scripts are also particularly challenging to model, due to their implementation language. Here we briefly report about the analysis (further details can be found in [14,15]) which has enabled to discover recurrent patterns in the huge amount of scripts to consider (e.g., about 25·000 on Debian Lenny). We tried to collect scripts in clusters to be able to concentrate the analysis on representatives of the equivalence classes identified. The adopted procedure for clustering has been as follows.

1. *Collect all maintainer scripts* of a given distribution. By not choosing a random subset we are sure to have collected the most representative set of scripts.
2. *Identify scripts generated from helper tools*. A large number of scripts or part of them is generated by means of "helper" tools that provide a collection of small, simple and easily understood tools that are used to automate various common aspects of building a package. Since (part of) maintainer scripts are automatically generated using these helpers and their boiler plates, we can concentrate the analysis on the helpers themselves, rather than on the result of their usage.
3. *Ignore inert script parts*. As all scripting languages, shell scripts contain parts which do not affect their computational state such as blank lines of comments. Intertwined with the removal of generated parts (to be analyzed later on) we have systematically ignored inert script parts, possibly leading upon removal to empty scripts that have been therefore ignored as a whole.
4. *Study of scripts written "by hand"*. The remaining scripts need to be more carefully studied, as they have been written from scratch by package maintainers to address a specific need, most likely not covered by any helper tool. Actually we worked on identifying recurrent templates that maintainers use when writing the scripts.

In the rest of the section, the conceived metamodels are described in more details and some explanatory models conforming to them are also provided.

4.2 Configuration Metamodel

A system configuration is the composition of artifacts necessary to make computer systems perform their intended functions [16]. In this respect, the metamodel depicted in Figure 3 specifies the main concepts which make up the configuration of a FOSS system. In particular, the `Environment` metaclass enables the specification of loaded modules, shared libraries, and running process as in the sample configuration reported in Figure 4. In such a model the reported environment is composed of the services www, and `sendmail` (see the instances `s1` and `s2`) corresponding to the running web and mail servers, respectively.

All the services provided by a system can be used once the corresponding packages have been installed (see the association between the `Configuration` and `Package` metaclasses in Figure 3) and properly configured (`PackageSetting`). Moreover, the configuration of an installed package might depend on other package configurations. For example, considering the PHP5 upgrade described in Section 2, the instances `ps1` and `ps2` of the `PackageSetting` metaclass in Figure 4 represents the settings of the installed packages `apache2`, and `libapache-mod-php5`, respectively. The former depends on the latter (see the value of the attribute `depends` of `ps1` in Figure 4) and both are also associated with the corresponding files which store their configurations.

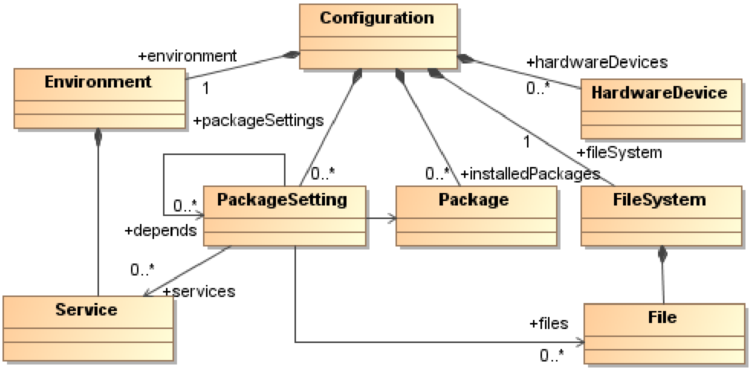


Fig. 3. Fragment of the Configuration metamodel

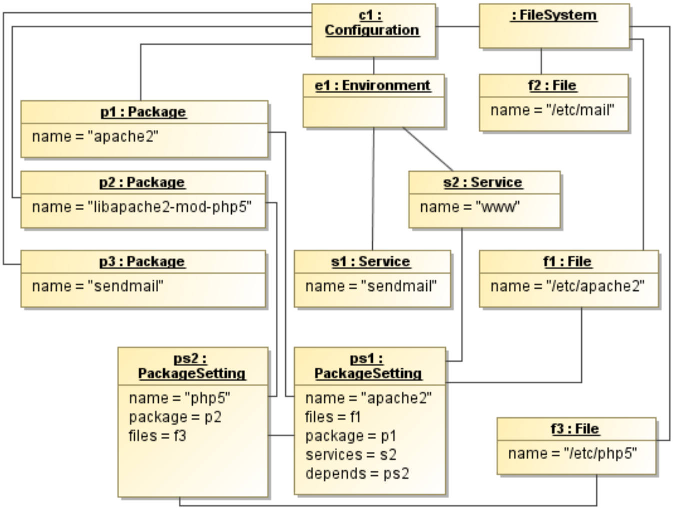


Fig. 4. Sample Configuration model

Note that at the level of package meta-information such a dependency should not be expressed, in spite of actually occurring on real systems. The ability to express such fine-grained and installation-specific dependencies is a significant advantage offered by metamodeling.

The configuration metamodel gives also the possibility to specify the hardware devices of a system by means of the `HardwareDevice` metaclass. Due to space constraint the usage of such a metaclass is omitted; for more information the interested reader can found the complete metamodels on the Web.[6]

[6] http://www.di.univaq.it/diruscio/mancoosi

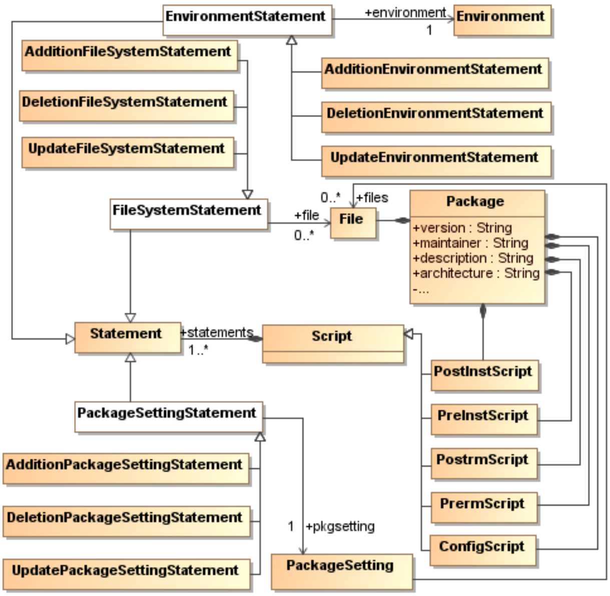


Fig. 5. Fragment of the Package metamodel

The packages which are installed on a given system are specified by means of the modeling constructs provided by the *Package metamodel* described in the next section.

4.3 Package Metamodel

The metamodel reported in Figure 5 plays a key role in the overall simulation. In fact, in addition to the information already available in current package descriptions, the concepts captured by the metamodel enable the specification of the behavior of maintainer scripts. In this respect, the metaclass Statement in Figure 5 represents an abstraction of the commands that can be executed by a given script to affect the environment, the file system or the package settings of a given configuration (EnvironmentStatement, FileSystemStatement, and PackageSettingStatement, respectively). For instance, the sample package model in Figure 6 reports the scripts contained in the package libapache-mod-php5 introduced in Section 2. Due to space constraints, Figure 6 contains only the relevant elements of the *postinst* and *prerm* scripts which are represented by the elements pis1 and prs1, respectively.

According to the model in Figure 6 the represented scripts update the configuration of the package apache2 (see the element ps1) which depends on libapache-mod-php5. In particular, the element upss2 corresponds to the statement a2dismod which disables the PHP5 module in the Apache configuration before removing the package libapache-mod-php5 from the filesystem. This statement is necessary, otherwise inconsistent configurations can be reached like the one shown in Figure 7. The figure reports the sample Configuration2 which has been reached by

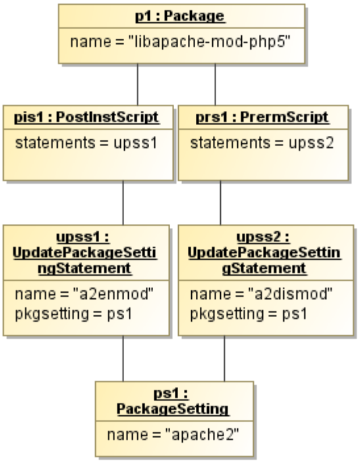


Fig. 6. Sample Package model

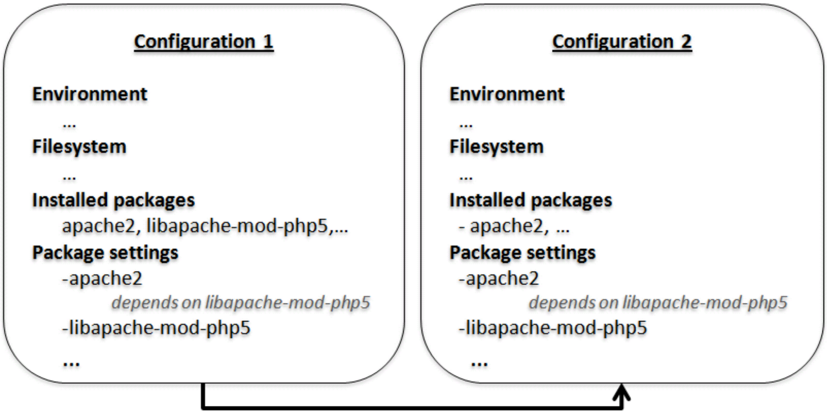


Fig. 7. Incorrect package removal

removing `libapache-mod-php5` without changing the configuration of `apache2`. Such a configuration is broken since it contains a dependency between the `apache2` and `libapache-mod-php5` package settings, whereas only the package `apache2` is installed.

Currently, the package managers are not able to predict inconsistencies like the one in Figure 7 since they take into account only information about package dependencies and conflicts. The metamodel reported in Figure 5 gives the possibility to specify an abstraction of the involved maintainer scripts which are executed during the package upgrades. This way, consistence checking possibilities are increased and trustworthy simulations of package upgrades can be operated.

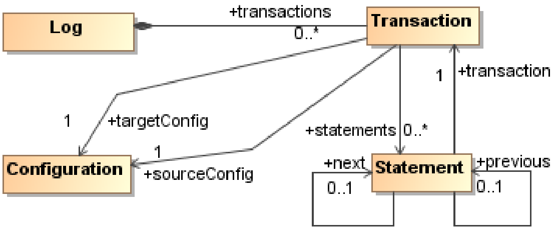


Fig. 8. Fragment of the Log metamodel

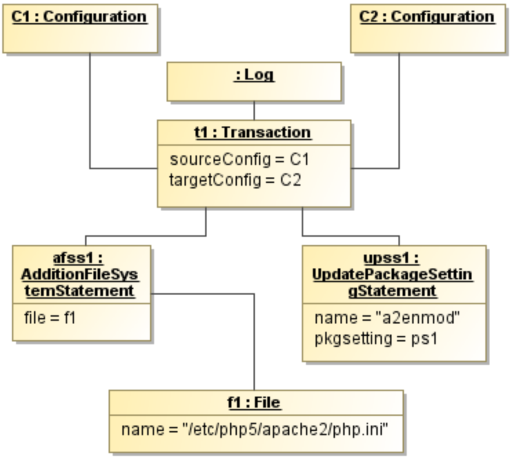


Fig. 9. Sample Log model

4.4 Log Metamodel

The metamodel depicted in Figure 8 is a step towards the development of a transactional model of upgradeability that will allow us to roll-back long upgrade history, restoring previous configurations. In particular, the metaclass `Transaction` in Figure 8 refers to the set of statements which have been executed from a source configuration leading to a target one. For instance, according to the sample log model in Figure 9, the installation of the package `libapache-mod-php5` modifies the file system (see the statement `afss1` which represents the addition of the file `f1`) and updates the Apache configuration (see the element `upss1`).

The usefulness of log models like the one in Figure 9 is manyfold and accounts for several roll-back needs:

(a) *Preference roll-back:* the user wants to recover a previous configuration, for whatever reason. For instance, the user is not in need of PHP5 anymore and wants to remove the installed package `libapache-mod-php5`. In this case, the configuration `C1` can be recovered by executing the dual operation of each statement in the transaction

between `C1` and `C2`. Note that the log models have all the information necessary to roll-back to any previous valid configuration not necessary a contiguous one;
(b) *Compensate model incompleteness*: as already discussed, upgrade simulation is not complete with respect to upgrades, and undetected failures can be encountered while deploying upgrades on the real system. For instance, the addition of the file `php.ini` during the installation of the package `libapache-mod-php5` can raise faults because of disk errors. In this case we can exploit the information stored in the log model to retrieve the fallacious statements and to roll-back to the configuration from which the broken transaction has started.
(c) *"Live" failures:* the proposed approach does not mandate to pre-simulate upgrades. In fact, it is possible as well to avoid simulation and have metamodeling supervise upgrades to detect invalid configurations as soon as they are reached. At that point, if any, log models comes into play and enable rolling back deployed changes to bring the system back to a previous valid configuration.

5 Related Works

The main difficulties related to the management of software packages depend on the existence of maintainer scripts which can have system-wide effects, and hence can not be narrowed to the involved packages only. In this respect, proposals like [17,18] represent a first step toward roll-back management. In fact, they support the re-creation of removed packages on-the-fly, so that they can be re-installed to undo an upgrade. However, such approaches can track only files which are under package manager control. Therefore, differently from us, none of such approaches can undo maintainer script side effects.

An interesting proposal to support the upgrade of a system, called NixOS, is presented in [19]. It is a purely functional distribution meaning that all static parts of a system (such as software packages, configuration files and system startup scripts) are built by pure functions. Among the main limitations of NixOS there is the fact that some actions related to upgrade deployment can not be made purely functional (e.g., user database management). Moreover, since NixOS implements a sort of "package garbage collection" (package versions are not removed as long as some other package need them) security upgrades get intrinsically more difficult due to the need of finding all versions that need upgrades. [20] proposes an attempt to monitor the upgrade process with the aim to discover what is actually being touched by an upgrade. Unfortunately, it is not sufficient to know which files have been involved in the maintainer scripts execution but we have also to consider system configuration, running services etc., as taken into account by our metamodels. Even focusing only on touched files, it is not always possible to undo an upgrade by simply recopying the old file[7]. Finally, this work can be related with techniques for static analysis of (shell) scripts. Some previous work [21] deals with SQL injection detection for PHP scripts, but it did not consider the most dynamic parts of the PHP language, quite common in scripting languages. Whereas, [22] presents an "arity" bug detection in shell scripts, but once more only considers a tiny fragment of the shell language. Both works hence are far even from the minimal

[7] This argument goes far beyond the scope of this work. For more information see [5].

requirement of determining a priori the set of files touched by script execution, letting aside how restricted were the considered shell language subsets. Given these premises, we are skeptical that static analysis can fully solve the problem illustrated in our work.

6 Conclusions and Future Works

In this paper we presented a model-driven approach to manage the upgrade of FOSS distributions and similarly structured complex, package based software systems. This approach represents an important advance with respect to the state of the art in the following directions: it provides the base on which developing features to (*i*) complete resolve packages dependencies, (*ii*) support the roll-back of failed or unwanted upgrades, and (*iii*) simulate the execution of maintainer scripts that we described in terms of models. A running example showed how the proposed models allow a reasonable description of the state of the system and representation of its evolution over time.

As future work we plan to implement these results and to develop a transactional update engine in the real context of Debian and Mandriva distributions. Moreover, the metamodels proposed in this paper will be the foundation to define a new Domain Specific Language (DSL) for maintainer script specifications.

Acknowledgements. This work is partially supported by they European Community's 7th Framework Programme (FP7/2007–2013), http://www.mancoosi.org MANCOOSI project, grant agreement n°214898.

References

1. Cicchetti, A., Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Towards a model driven approach to upgrade complex software systems. In: Proceedings of ENASE (2009)
2. Spinellis, D., Szyperski, C.: How is open source affecting software development. IEEE Computer 21(1), 28–33 (2004)
3. Szyperski, C.: Component Software. Beyond Object-Oriented Programming. Addison-Wesley, Reading (1998)
4. Szyperski, C.: Component technology: what, where, and how? In: Proceedings of ICSE 2003. ACM, New York (2003)
5. Di Cosmo, R., Zacchiroli, S., Trezentos, P.: Package upgrades in FOSS distributions: details and challenges. In: HotSWUp 2008, pp. 1–5. ACM, New York (2008)
6. EDOS Project: Report on formal management of software dependencies. EDOS Project Deliverable D2.1 and D2.2 (March 2006)
7. Jackson, I., Schwarz, C.: Debian policy manual (2008), http://www.debian.org/doc/debian-policy/
8. Mancinelli, F., Boender, J., Cosmo, R.D., Vouillon, J., Durak, B., Leroy, X., Treinen, R.: Managing the complexity of large free and open source package-based software distributions. In: ASE 2006, Tokyo, Japan, September 2006, pp. 199–208. IEEE CS Press, Los Alamitos (2006)
9. Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: Opium: Optimal package install/uninstall manager. In: ICSE 2007, pp. 178–188. IEEE Computer Society, Los Alamitos (2007)
10. Mens, T., Straeten, R.V.D., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)

11. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing model conflicts in distributed development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
12. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer 39(2), 25–31 (2006)
13. Bézivin, J.: On the Unification Power of Models. SOSYM 4(2), 171–188 (2005)
14. Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Towards maintainer script modernization in foss distributions. In: IWOCE 2009, pp. 11–20. ACM, New York (2009)
15. Di Ruscio, D., Pelliccione, P., Pierantonio, A., Zacchiroli, S.: Metamodel for describing system structure and state. Mancoosi Project deliverable D2.1 (January 2009), http://www.mancoosi.org/deliverables/d2.1.pdf
16. Dolstra, E., Hemel, A.: Purely functional system configuration management. In: USENIX 2007, San Diego, CA, pp. 1–6 (2007)
17. Olin Oden, J.: Transactions and rollback with rpm. Linux Journal 121, 1 (2004)
18. Trezentos, P., Di Cosmo, R., Lauriere, S., Morgado, M., Abecasis, J., Mancinelli, F., Oliveira, A.: New Generation of Linux Meta-installers. In: Research Track of FOSDEM (2007)
19. Dolstra, E., Löh, A.: NixOS: A purely functional linux distribution. In: ICFP (2008) (to appear)
20. McQueen, R.: Creating, reverting & manipulating filesystem changesets on Linux. Part II Dissertation, Computer Laboratory, University of Cambridge (May 2005)
21. Xie, Y., Aiken, A.: Static detection of security vulnerabilities in scripting languages. In: USENIX-SS 2006, pp. 179–192 (2006)
22. Mazurak, K., Zdancewic, S.: Abash: finding bugs in bash scripts. In: PLAS 2007, pp. 105–114. ACM, New York (2007)

Coupling Metrics for Aspect-Oriented Programming: A Systematic Review of Maintainability Studies

Rachel Burrows[1], Alessandro Garcia[2], and François Taïani[1]

[1] Computing Department, Lancaster University, U.K.
{rachel.burrows,francois.taiani}@comp.lancs.ac.uk
[2] Informatics Department, Pontifical Catholic University of Rio de Janeiro, Brazil
afgarcia@inf.puc-rio.br

Abstract. Over the last few years, a growing number of studies have explored how Aspect-Oriented Programming (AOP) might impact software maintainability. Most of the studies use coupling metrics to assess the impact of AOP mechanisms on maintainability attributes such as design stability. Unfortunately, the use of such metrics is fraught with dangers, which have so far not been thoroughly investigated. To clarify this problem, this paper presents a systematic review of recent AOP maintainability studies. We look at attributes most frequently used as indicators of maintainability in current aspect-oriented (AO) programs; we investigate whether coupling metrics are an effective surrogate to measure theses attributes; we study the extent to which AOP abstractions and mechanisms are covered by used coupling metrics; and we analyse whether AO coupling metrics meet popular theoretical validation criteria. Our review consolidates data from recent research results, highlights circumstances when the applied coupling measures are suitable to AO programs and draws attention to deficiencies where coupling metrics need to be improved.

Keywords: Coupling, Aspect-oriented programming, Systematic review, Maintainability.

1 Introduction

Aspect-oriented programming (AOP)[2] is now well established in both academic and industrial circles, and is increasingly being adopted by designers of mainstream implementation frameworks (e.g. *JBoss* and *Spring*). AOP aims at improving the modularity and maintainability of crosscutting concerns (e.g. security, exception handling, caching) in complex software systems. It does so by allowing programmers to factor out these concerns into well-modularised entities (e.g. aspects and advices) that are then *woven* into the rest of the system using a range of composition mechanisms, from *pointcuts* and *advices*, to *inter-type declarations*[27], and *aspect collaboration interfaces*[8].

Unfortunately, and in spite of AOP's claims to modularity, it is widely acknowledged that AOP mechanisms introduce new intricate forms of coupling[33], which in turn might jeopardise maintainability[1,4]. To explore this, a growing number of exploratory studies have recently investigated how maintainability might be impacted by the new forms of coupling introduced by AOP mechanisms[e.g 19,20,26].

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 277–290, 2010.

The metrics used by these studies are typically taken from the literature[10,11,33,37,39] and are assumed to effectively capture coupling phenomenon in AOP software. However, the use of coupling metrics is fraught with dangers, which as far as AOP maintainability is concerned have not yet been thoroughly investigated. In order to measure coupling effectively a metrics suite should fulfill a number of key requirements. For instance: the suite should take into account all the composition mechanisms offered by the targeted paradigm[29,31]; the metrics definitions should be formalised according to well-accepted validation frameworks, e.g. Kitchenham's validation framework[30]; and they should take into account important coupling dimensions, such as coupling type or strength. If these criteria are not fully satisfied, maintainability studies of AOP might draw artificial or inaccurate conclusion and, worse, might mislead programmers about the potential benefits and dangers of AOP mechanisms regarding software maintenance.

Unfortunately, the validity and reliability of coupling metrics as indicators of maintainability in AOP systems remains predominantly untested. In particular, there has been no *systematic review* on the use of coupling metrics in AOP maintainability studies. Inspired from medical research, a systematic review is a fundamental empirical instrument based on a literature analysis that seeks to identify flaws and research gaps in existing work by focusing on explicit research questions[29]. This paper proposes such a systematic review with the aim to pinpoint situations where existing coupling metrics have been (or not) effective as surrogate measures for key maintainability attributes. Our systematic review consolidates data from a range of relavent AOP studies, highlights circumstances when the applied coupling measures are suitable to AO programs and draws attention to deficiencies where coupling metrics needs to be improved.

The remainder of this paper provides some background on AOP and coupling measurement (Section 2). We then discuss the design of our systematic review and present its results (Section 3 and 4). Finally, we discuss our findings (Sections 5) and conclude (Section 6).

2 AOP and Coupling Measurement

This section gives a brief discussion on three representative AOP languages and also gives a background on coupling metrics for AOP.

2.1 AOP Languages and Constructs

One of the reasons why the impact of AOP on maintainability is difficult to study pertains to the inherent heterogeneity of aspect-oriented mechanisms and languages. Different AOP languages tend to incarnate distinct blends of AOP and use different encapsulation and composition mechanisms. They might also borrow abstractions and composition mechanisms from other programming paradigms, such as collaboration languages (CaesarJ), feature-oriented programming (CaesarJ), and subject-oriented programming (HyperJ). Most AOP languages tend to encompass conventional AOP properties such as joinpoint models, advice and aspects, or their equivalent, but each possesses unique features that make cross-language assessment difficult.

Table 1. AO abstractions and mechanisms unique to three main AOP languages

AO Language	Abstraction / Mechanism
AspectJ	Intertype Declaration
	Dynamic Pointcut Designators
	Aspect
CaesarJ	Aspect Collaboration Interface
	Weavlet
	Virtual Class
HyperJ	Hyperspace
	Concern Mapping
	Hypermodule
	Composition Relationship

Table 1 lists ten such features for AspectJ[2], HyperJ[23] and CaesarJ[8], three of the most popular AOP languages. For instance, AspectJ supports advanced dynamic pointcut designators, such as "cflow". HyperJ uses hyperspace modules to modularise crosscutting behaviour as well as non-crosscutting behaviour. HyperJ thus does not distinguish explicitly between aspects and classes in the way AspectJ does. Other abstractions unique to HyperJ include Compositions Relationships. These use merge-like operators to define how surrounding modules should be assembled. Finally, CaesarJ supports the use of virtual classes to implement a more pluggable crosscutting behaviour. This pluggable behaviour is connected with the base code through Aspect Collaboration Interfaces.

2.2 Existing AO Coupling Metrics

Coupling metrics aim to measure the level of interdependency between modules within a program[12], thus assessing a code's modularisation, and indirectly maintainability. Unfortunately, each language's unique features introduce new forms of coupling, which cannot always readily be mapped onto existing concepts (Table 1). This creates a challenge when designing coupling metrics for AOP, as these metrics should ideally take into account each language's unique features, while still providing a fair basis for comparison multiple AOP languages. This is particularly difficult.

A number of coupling metrics have so far been proposed for AO programs. Some are adapted from object-orientation, and transposed to account for AO mechanisms. For instance, both Ceccato and Tonella[10] and Sant'Anna et al[36] have proposed coupling metrics adapted from an object-oriented (OO) metrics suite by Chidamber and Kemerer [11]. These metrics can be applied to both OO and AO programs. This is especially useful in empirical studies that perform aspect-aware refactoring. Unfortunately, because these metrics are not specific to AOP, they might overlook the unique intricate forms of coupling described in Table 1.

Zhao[39] uses dependency graphs to measure some AO mechanisms that are not measured individually in either Ceccato and Tonella or Sant'Anna's suites. Zhao's suite contains metrics that measure coupling sourced from AO abstractions and mechanisms independently of OO abstractions and mechanisms.

Coupling metrics are however rarely used as a direct representation of maintainability, but instead are typically contrasted against a particular maintainability attribute, such as code stability. The choice of this attribute (or attributes) might in turn influence which coupling metrics is the most suitable.

3 Systematic Review

This section describes the objectives and questions (Section 3.1) as well as the strategical steps carried out in the systematic review.

3.1 Objectives and Questions

The *aim* of our systematic review is to analyse the effectiveness of coupling metrics in existing AO empirical studies as a predictor of maintainability, and in particular focus on the following four *research questions:*

a) Which external attributes are most frequently used to indicate maintainability in current AO programs?
b) Are used coupling metrics effective surrogate measures for software maintainability?
c) Are all AOP abstractions and mechanisms covered in the design of the used coupling metrics?
d) Do AO coupling metrics meet well-established theoretical validation criteria?

3.2 Review Strategy

Searches for papers took place in 14 renowned online journal banks or were those published in recognised conference papers such as AOSD(Aspect-Oriented Software Development) and ECOOP (European Conference on Object-Oriented Programming). We gave priorities to publications in conferences with an acceptance rate below 30%. Relavent papers were found from ACM, SpringerLink, IEEE, Google Scholar, Lancaster University Online Library, and two were collected from other sources.

Sampling Criteria. From this base we sampled papers that met the following criteria. Each selected paper had to:

- use an empirical study to measure maintainability attributes in AOP;
- and use coupling metrics within the study.

Due to low retrieval rate from journal banks, alternative approaches were also used. This included both consulting references on already-found papers and searching specifically for papers we knew met our criteria (from previous knowledge). The distribution of collected research is recorded in the review results (Section 4).

Exracted Data. We recorded which independent / dependent variables where measured, the goals of measurement, the type of study, measurement results, which coupling metrics were used, their origin, and whether the applied metrics were specifically for AOP or adapted from another programming technique (e.g. OOP).

Table 2. Distribution of Studies

Electronic Journal	# Retrieved	# Rejected	# Used
ACM	4	0	4
IEEE	2	1	1
SpringerLink	3	1	2
L.U. Online Library	5	2	3
Other	4	2	2
Total	**18**	**6**	**12**

4 Results

A final set of 12 papers was finally obtained (Table 2), which is a typical sample size for systematic reviews in software engineering[28].

4.1 Assessed Maintainability Attributes

It is difficult to select coupling metrics to assess maintainability as definitions are often open to interpretations. For instance in[24], maintainability is "the ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment".

There is also no consensus about the external and internal attributes are the most significant indicators of maintainability. This is apparent in the empirical studies from the diverse selection of metrics used. Two main processes were recorded to select suitable coupling metrics. Firstly, many studies used coupling metrics previously selected in similar AOP empirical studies. Secondly, results showed the Goal-Question-Metric (GQM) [6] style approach is a common technique used to select appropriate metrics in empirical studies. This approach guides researchers to: (i) define the goal of measuring maintainability, then (ii) derive external attributes that are possible indicators of maintainability, then (iii) derive from these a set of internal measurable attributes, and finally (iv) derive a set of metrics to measure the internal measurable attributes. Unfortunately, using GQM still leaves a large degree of interpretation to its users, who might independently reach divergent conclusions. One further problem with this uncertainty is that the metric selection process can become circular, especially when measuring maintainability, as external quality attributes are interconnected. For instance, stability indicates maintainability, yet maintainability can be seen as an indicator of stability.

Similar techniques for selecting appropriate metrics in empirical studies have been used in [33]. This study decided to measure attributes such as maintainability, reusability and reliability as indicators of maintainability. From this list, internal attributes such as separation of concerns, coupling, complexity, cohesion and size were selected. The final set of selected coupling metrics was then defined based upon these internal attributes. We can therefore see that uncertainty on key external attributes has great impact on the remainder of the metric selection process.

This lack of conformity on these attributes has unsurprisingly affected the selected coupling metrics. For instance, maintainability is measured in studies[7,15,17] through the application of 9 metrics to measure size, coupling, cohesion and separation of concerns metrics. In[10,33] complexity is in addition derived as an external attribute contributing to maintainability. We return to this topic in Section 5.

Similar problems have been observed in maintainability studies of object-oriented programming (OOP) this has been highlighted in a survey of existing OO empirical studies and their methodologies to predict external quality attributes[5].

Many studies acknowledge that modularity, coupling, cohesion and complexity are internal attributes that affect maintainability. Interestingly, error-proneness was the attribute that was not explicitly derived as an indicator of maintainability.

In short, different interpretations of maintainability and its subsequent derived attributes influence the coupling metrics chosen or defined within the context of an empirical study. This may explain the wide range of coupling metrics observed in AOP empirical studies, which we review in the next subsections.

4.2 Coupling Metrics Used to Measure Maintainability

We identified 27 coupling metrics in our sample set of studies. A representative subset of these metrics is shown in Table 3. For each metric, the table lists it's name, description, and six characteristics.

Generally, the most frequent metrics were adapted from object orientation (OO). Among them, the most common were Coupling Between Components (CBC) and Depth of Inheritance Tree (DIT), appearing in 66% of the studies. Adapted metrics hold the advantage of being based upon OO metrics that are widely used, and can be assumed reliable. The (implicit) reasoning is that adapting OO metrics to AOP maintains their usefulness. This however might no hold: DIT for instance combines both the implicit AO inheritance with the traditional OO inheritance. It thus considers two very different coupling sources together. These sources may have different affects upon maintainability and it may be beneficial to consider these seperately.

In contrast, some of the studies also use coupling metrics developed for AOP, such as Coupling on Advice Execution (CAE) and Number of degree Diffusion Pointcuts (dPC). These metrics enable a more in-depth analysis of the system coupling behaviour, as they consider finer-grained langauge constructs. However, they are more likely to behave unexpectedly, being underdeveloped.

No AO coupling metrics were found to be interchangeable, i.e. none were found to be applicable to different AO languages without any ambiguity. This is probably due to the heterogeneity of AO programming abstractions and mechanisms that makes it very hard to define metrics accurately across multiple AO languages.

The majority of metrics found in our study assess outgoing coupling connections (indicated as “Fan Out” in Table 3). This can be seen as a weakness, as both incoming and outgoing coupling connections help refactoring decisions, as discussed in[31].

4.3 Measured AOP Mechanisms

OO coupling metrics can be adapted to take into account AO mechanisms, producing a seemingly equivalent measure. However, this approach might miss some of specific needs of AO programs. We now review how the mechanisms of the AOP languages most commonly used in maintainability studies of AOP were accounted for in coupling measures, and draw attention to mechanisms that are frequently overlooked.

Table 3. Properties of used coupling metrics

Metric	Description
(DIT) Depth of Inheritance Tree [10]	Longest path from class / aspect to hierarchial root.
(RFM) Response for a Module [10]	Methods and advices potentially executed in response to a message received by a given module.
(NOC) No. of Children [10]	Immediate sub-classes / aspects of a module.
(CBC) Coupling Between Components [10]	Number of classes / aspects to which a class / aspect is coupled.
(CAE) Coupling on Advice Execution [10]	No. of aspects containing advice possibly triggered by execution of operations in a given module.
(dPC) No. of Degree Diffusion Pointcuts [31]	No. of modules depending on pointcuts defined in the module.
(InC) No. of In-Different Concerns [31]	No. of different concerns to which a module is participating.

Metric	Measurement Granularity	Measurement Entity	Measurement Type	Fan In / Fan Out	Inter-changeable	AO / Adapted
(DIT)	class / aspect	Class / aspect	inheritance	n/a	no	adapted
(RFM)	module	method / advice	environmental	fan out	no	adapted
(NOC)	module	Class / aspect	inheritance	n/a	no	adapted
(CBC)	class / aspect	Class / aspect	environmental	fan out	no	adapted
(CAE)	module	aspect	environmental	fan out	no	AO
(dPC)	module	module	environmental	fan in	no	AO
(InC)	module	concern	environmental	n/a	no	AO

Table 4 lists the mechanisms and abstractions used in the coupling metrics of our study. One first challenge arises from the ambiguity of many notions. For instance, seven metrics use "modules" as their level of granularity, but what is module might vary across languages. In AspectJ an aspect may be considered a module – containing advice, pointcuts and intertype declarations, yet in CaesarJ, each advice forms its own module. More generally, many coupling metrics use ambiguously terms ("module", "concern", or "component") which might be mapped to widely varying constructs in different languages. This hampers the ability of the metrics to draw cross-language comparisons[20].

Another challenge comes from the fact that certain phenomenon are best analysed by looking at the base and aspect codes separately. For instance, as a program evolves, it may lose its original structure. However, in AO programs, the base level and aspect level often evolve independently and have different structures. Understanding how each evolution impacts structure thus requires that each be investigated separately. This is not done in most of the empirical studies we found.

We also noted that the majority of used AO metric suites did not focus on interface complexity. This is a problem as AO systems are at risk of creating complex interfaces by extracting code which is heavily dependent on the surrounding base code, and metrics are needed to identify problematic situations[33].

Table 4. AO mechanisms and abstractions accounted for in used coupling metrics

Abstraction / Mechanism	No. of Metrics	Measurement Entity	Measurement Granularity	Singular Entity Metric
Module	15	8	12	5
Component	1	0	0	0
Concern	7	5	7	2
Pointcut	3	0	0	0
Joinpoint	2	2	0	2
Intertype Declaration	1	1	0	0
Aspect	7	4	4	1
Advice	3	1	0	0

More generally, few studies look at the connection between maintainability and specific AO mechanisms. For instance Response for a Module (RFM) measures connections from a module to methods / advices. This is useful in analysing coupling on a "per module" basis, but does not distinguish between individual AO language constructs. For instance, it adds up intertype declarations jointly with advice as they both provide functionality that insert extra code into the normal execution flow of the system. However intertype declarations differ from other types of advice as they inject new members (e.g. attributes) into the base code. Coupling metrics have been proposed to address this problem and measure singular mechanisms, such as advice, pointcuts, joinpoints and some intertype declarations[10,26,36], but have rarely been used in maintainability studies.

To sum up, no study used metrics to measure constructs unique to AO programming languages, and very few measured finer-grained language constructs. Although this depends on the particular goals of each maintainability study, this is generally problematic as each mechanism within a particular language has the potential to affect maintainability differently, and should therefore be analysed in its own right.

4.4 Validation of Coupling Metrics

Metrics are useful indicators only if they have been validated. There are two complementary approaches to validate software metrics, *empirical validation* and *theoretical validation*[30]. We will focus on the latter. In our context, theoretical validation tests that a coupling metric is accurately measuring coupling and there is evidence that the metric can be an indirect measure of maintainability.

Here we consider the 8 validity properties suggested by Kitchenham[30]. The theoretical criteria are split into two categories: *(i)* properties to be addressed by all metrics; and *(ii)* properties to be satisfied by metrics used as indirect measures. [3] has already used the first criteria on coupling metrics for AO programs. We offer some alternative viewpoints here, and also evaluate the coupling metrics against properties that indirect measures should possess. When we applied this framework to the 27 coupling metrics found in our review, we identified three potential violations of these criteria, discussed below.

A Valid Measure Must Obey the 'Representation Condition'. This criterion states that there should be a homomorphism between the numerical relation system and the measureable entities. In other words a coupling metric should accurately express the relationship between the parts of the system that it claims to measure. It also implies that coupling metrics should be intuitive of our understanding of program coupling[30]. For instance, a program with a CBC value of 6 should be more coupled than a program with a CBC value of 5. This metric holds true to it's definition, however if a study is using CBC as a representation of coupling within a system this validation criteria becomes questionable. When measuring coupling we often do not perceive each connection as equal. There are different types and strengths of coupling. If we consider two AO systems; the first with 5 coupling connections via inter-type declarations, and the second with 5 coupling connections via advice. Even though both systems contain 5 coupling connections, they are not equivalent, and are not equally interdependent. Various sources and types of coupling may influence the interdependency of a system in multiple ways. We found no metrics in the studies that took this finer differences into account.

Each Unit of an Attribute Contributing to a Valid Measure is Equivalent. We are assuming that units (modules) that are measured alongside each other are equivalent. There are some AO coupling metrics that only consider coupling from one language 'unit'. For example, the CAE metric satisfies this property as each connection counted by metric value involves an advice method. However, many metrics used in empirical studies of AOP assume that counting coupling connections between AO modules is equivalent to coupling connections between OO modules. As mentioned b, classes and aspects are often measured together as equivalent modules (e.g in DIT), yet we do not have evidence that they have the same effect upon maintainability, thus violating this criteria.

There should be an Underlying Model to Justify its Construction. To give good reason for the creation of coupling metrics, there should be underlying evidence that the metric will be an effective indicator of maintainability. Unfortunately, this criterion definition is somewhat circular in the case of maintainability; metrics are often already constructed and applied before supporting this underlying theory and justifying their construction. In OOP it is widely accepted that there is a relationship between coupling and external quality attributes. Because AOP and OOP share similarities, we could infer that metrics that measure a specific form of coupling in OOP hold a similar potential when adapted to AOP (such as DIT, CBC). This however needs to be validated. This need is even more acute for metrics specific to AOP (e.g. CAE), as we have less information on how coupling induced by AOP-specific mechanisms correlate with maintainability.

5 Discussion

We first discuss the potential threats to the validity of our study (5.1), and then revisit our original research questions (5.2) in the light of the results we have just discussed.

5.1 Threats to Validity

Our study raises both internal and external validity issues. In terms of internal validity, our study is based on 12 papers that matched our criteria (Section 4). This number is not high however this is in line with systematic reviews in software engineering, which often rely on approximately 10 target papers[28]. The size of the sample should however be kept in mind when assessing the generality of our results.

In terms of external validity, we identified a number strengths and liabilities in the state-of-the-art of AO coupling metrics. However, this list is certainly not exhaustive, and does no cover a number of broader issues about AO metrics and maintainability.

For instance, there are certain forms of (semantic) module dependencies that cannot be quantified by conventional coupling metrics, such as those captured by network analysis[40]. The same argument applies to Net Option Values and Design Structure Matrices[9,32]. Finally, AO empirical studies often rely on multiple metrics suites to measure module complexity, module cohesion, and concern properties. Considering coupling in isolation thus limits our horizon, a broader review would be complementary to this work.

5.2 Analysis and Implications

The design and use of AO coupling metrics needs to be improved. Analysis of findings revealed problems corresponding to each of the four original research questions.

The selection of metrics to measure maintainability in AO studies is ambiguous. Many issues contribute to this. Some studies specified key external attributes that contribute to the maintainability of a program. The subjectivity and variations of these external attributes(Section 4.1) causes uncertainty of the most effective metrics to select to measure them. Also, deriving attributes that influence maintainability has shown to be a circular process e.g. stability affects maintainability, and maintainability affect stability. Empirical validation may aid researchers to converge on a smaller set of validated coupling metrics. Better-defined metrics will help this process as well (Section 4.4).

Adapted OO metrics are useful to cross-compare AO and OO programs. Naturally, OO coupling metrics that successfully served as valid indicators of maintainability are likely to be re-used. In fact, this assumption applies to many studies that refactor OO programs with aspects. However, it is important that adapted metrics are not the only ones used. Adapted metrics (such as CBC) overlook characteristics that are unique to a particular AOP language as discussed in Section 4.3. For instance, this metric cannot be used to pinpoint the coupling caused by particular AOP constructs, yet specific AOP constructs may impact unexpectedly upon the maintainability of a program. Also, coupling introduced by unique AOP constructs should be also measured as single entities. Otherwise, we are unable to gain in-depth knowledge about the impact of AOP on maintainability.

Some AO metrics provide initial means to measure coupling introduced by specific AO language constructs[10,33,39]. These fine-grained metrics make it easier to locate program elements that are responsible for positive (or negative) results. For example, if we can correlate a high CAE coupling value with poor maintainability, we may infer that specific advice types in AOP languages are harmful.

Also, results from fine-grained AO coupling metrics may facilitate the identification of solutions for classical problems in AOP, such as *pointcut fragility* [26]. Pointcut fragility is the phenomenon associated with instabilities observed in poincut specifications in the presence of changes. It is commonly assumed the syntax-based nature of most pointcut designators is the cause of their fragilities [26]. There are speculations that certain pointcut designators, such as cflow (Section 2.1), cause more instability. These hypotheses re-enforce the need for metrics that quantify specialised types of coupling links between aspectual code and base code. Such envisaged fine-grained metrics would enable us to better understand the effects of particular AOP mechanisms upon maintainability. We need to know which specific mechanisms are typically the cause of high coupling, and does coupling via different mechanisms have the same impact upon maintainability.

There are other important dimensions of coupling beyond granularity, such as direction, or strength of coupling [4]. We identified that the analysed 27 metrics for AOP do overlook important coupling dimensions. This might be misleading conclusions, as different coupling dimensions may affect maintainability in different ways.

Most AO coupling metrics are created with AspectJ as the target language[10,33,36]. However, alternative languages, such as HyperJ and CaesarJ, support AOP based on different mechanisms (Table 1). Section 4.3 discussed the need for coupling metrics tailored to these unique mechanisms of alternative AOP languages. It is also required to define coupling metrics that are interchangeable across these multiple AOP languages.

Not all coupling metrics meet popular validation criteria (Section4.4). Without theoretical validation there is the risk of using metrics that are providing inaccurate results. Even subtle adaptations to widely accepted OO metrics need to be validated. A recurring point in this review was that certain metric definitions assume different language constructs can be measured together as equivalent entities. For instance, coupling via class inheritance in OO programs might not demonstrate equivalent maintainability effects as aspect inheritance in AO programs. Similar effects might also be overlooked in other forms of module specialisations in AOP, such as intertype declarations. Therefore it might not be appropriate to quantify together heterogeneous specialisation forms in AOP.

Liabilities of AO coupling metrics are not restricted to unsatisfactory theoretical validation. Their empirical validation is also limited, and the statistical relevance of coupling metrics' results is compromised. For example, metrics adapted from OOP remain invalidated within the context of AOP. It would be wrong to assume that such adapted metrics can be similarly interpreted in the context of AO software maintainability.

6 Conclusions

Conducting the systematic review has presented valuable insights into current trends on coupling measurement for AOP. This has consequently highlighted the need for fine-grained metrics that consider specific AOP constructs. Existing metrics that are frequently used are therefore in danger of overlooking key contributors to maintainability.

For this reason, there is a niche in current maintainability studies of AOP to use coupling metrics that: (i) take specific language constructs into account, (ii) distinguish between the various dimensions of coupling, and (iii) can be applied unambiguously to a variety of AOP languages.

We have also noticed that the maintainability studies of AOP overly concentrate on static coupling metrics. Dynamic coupling metrics [1] for AOP have not been applied in all the analysed studies. This came as a surprise as many AO composition mechanisms rely on the behavioural program semantics. Also, key maintainability attributes, such as error proneness (Section 4.1), are never explicitly derived as an assessment goal.

Validating new metrics is a non-trivial matter. Kitchenham raised the problem that validating metrics solely with predictive models can be problematic [29]. Without theoretical validation, metrics might not be suitable indirect measures of maintainability. It is important to consider the context that a metric is being applied and whether it is an accurate representation of maintainability in AO systems. Therefore, even AO metrics adapted from empirically-validated OO metrics, can fail to be theoretically sound predictors of maintainability. In fact, our systematic review found that some AO metrics do not obey the representation condition and other criteria.

However, the above goals are difficult to address in one approach. For instance, defining fine-grained metrics to analyse language specific mechanisms is conflicting with the goal of having course-grained metrics that can be applied across multiple AOP languages. Unfortunately, all these goals are crucial for an in-depth comparison of AOP mechanisms. As part of our future works we aim to undertake empirical studies to explore how the goals we have identified may be reconciled in a unified approach.

References

1. Arisholm, E., Briand, L., Foyen, A.: Dynamic Coupling Measurement for Object-Oriented Software. IEEE Trans. Soft. Eng. 30(8), 491–506 (2004)
2. The AspectJ Prog. Guide, `http://eclipse.org/aspectj`
3. Bartsch, M., Harrison, R.: An Evaluation of Coupling Measures for AOP. In: LATE Workshop AOSD (2006)
4. Briand, L., Daly, J., Wüst, J.: A Unified Framework for Coupling Measurement in Object-Oriented Systems. IEEE Trans. Software Eng. 25(1), 91–121 (1999)
5. Briand, L., Wüst, J.: Empirical Studies of Quality Models in Object-Oriented Systems. In: Advances in Computers. Academic Press, London (2002)
6. Basili, V., et al.: GQM Paradigm. Comp. Encyclopedia of Soft. Eng. JW&S 1, 528–532 (1994)
7. Cacho, N., et al.: Composing design patterns: a scalability study of aspect-oriented programming. In: AOSD 2006, pp. 109–121 (2006)
8. CaesarJ homepage, `http://caesarj.org`
9. Cai, Y., Sullivan, K.J.: Modularity Analysis of Logical Design Models. ASE 21, 91–102 (2006)
10. Ceccato, M., Tonella, P.: Measuring the Effects of Software Aspectization. WARE cd-rom (2004)
11. Chidamber, S., Kemerer, C.: A Metrics Suite for OO Design. IEEE Trans. Soft. Eng. 20(6), 476–493 (1994)

12. Fenton, N.E., Pfleeger, S.L.: Software Metrics: a Rigorous and Practical Approach, 2nd edn. PWS Publishing Co., Boston (1998)
13. Figueiredo, E., et al.: Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. In: ECOOP (2005)
14. Filho, F.C., et al.: Exceptions and aspects: the devil is in the details. FSE 14, 152–156 (2006)
15. Filho, F.C., Garcia, A., Rubira, C.M.F.: A quantitative study on the aspectization of exception handling. In: Proc. ECOOP (2005)
16. Garcia, A., et al.: On the modular representation of architectural aspects. In: Gruhn, V., Oquendo, F., et al. (eds.) EWSA 2006. LNCS, vol. 4344, pp. 82–97. Springer, Heidelberg (2006)
17. Garcia, A., et al.: Separation of Concerns in Multi-Agent Systems: An Empirical Study. Software Engineering for Multi-Agent Systems with Aspects and Patterns. J. Brazilian Comp. Soc. 1(8), 57–72 (2004)
18. Garcia, A., et al.: Aspectizing Multi-Agent Systems: From Architecture to Implementation. In: Choren, R., Garcia, A., Lucena, C., Romanovsky, A. (eds.) SELMAS 2004. LNCS, vol. 3390, pp. 121–143. Springer, Heidelberg (2005)
19. Garcia, A., et al.: Modularizing Design Patterns with Aspects: A Quantitative Study. In: Proc. AOSD, pp. 3–14 (2005)
20. Greenwood, P., et al.: On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 176–200. Springer, Heidelberg (2007)
21. Harrison, R., Counsell, S., Nithi., R.: An Overview of Object-Oriented Design Metrics. In: Proc. STEP, pp. 230–234 (1997)
22. Hitz, M., Montezeri, B.: Measuring Coupling and Cohesion in Object-Oriented Systems. In: Proc. Int. Symposium on Applied Corporate Computing (1995)
23. Hyper/J home page, http://www.research.ibm.com/hyperspace/HyperJ.htm
24. IEEE Glossaries, http://www.computer.org/portal/site/seportal/index.jsp
25. JBoss AOP, http://labs.jboss.com/jbossaop
26. Kastner, C., Apel, S., Batory, D.: Case Study Implementing Features Using AspectJ. In: Proc. SPLC, pp. 223–232 (2007)
27. Kiczales, G., et al.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S., et al. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
28. Kitchenham, B., et al.: Systematic Literature Reviews in Software Engineering – A Systematic Literature Review. Information and Software Technology (2008)
29. Kitchenham, B.: Procedures for Performing Systematic Reviews. Joint Tech. Rep. S.E.G. (2004)
30. Kitchenham, B., Pfleeger, S.L., Fenton, N.: Towards a Framework for Software Validation Measures. IEEE TSE 21(12), 929–944 (1995)
31. Kulesza, U., et al.: Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In: Proc. ICSM, pp. 223–233 (2006)
32. Lopes, C.V., Bajracharya, S.K.: An analysis of modularity in aspect oriented design. In: AOSD, pp. 15–26 (2005)
33. Marchetto, A.: A Concerns-based Metrics Suite for Web Applications. INFOCOMP journal of computer science 4(3) (2004)
34. Pressman, R.S.: Software Engineering: a Practitioner's Approach. McGraw Hill, NY (1987)

35. Sant'Anna, C., et al.: On the Modularity of Software Architectures: A Concern-Driven Measurement Framework. In: Oquendo, F. (ed.) ECSA 2007. LNCS, vol. 4758, pp. 207–224. Springer, Heidelberg (2007)
36. Sant'Anna, C., et al.: On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In: Proc. SBES, pp. 19–34 (2003)
37. Sant'Anna, C., et al.: On the Modularity of Software Architectures: A Concern-Driven Measurement Framework. In: Proc. ECSA (2008)
38. Spring AOP, http://www.springframework.org
39. Zhao, J.: Measuring Coupling in Aspect-Oriented Systems. In: Int. Soft. Metrics Symp. (2004)
40. Zimmermann, T., Nagappan, N.: Predicting defects using network analysis on dependency graphs. In: ICSE, pp. 531–540 (2008)

Revealing Commonalities Concerning Maintenance of Software Product Line Platform Components

Martin Assmann[1], Gregor Engels[1], Thomas von der Massen[2], and Andreas Wübbeke[1,2]

[1] Dept. of Computer Science, University of Paderborn
Warburger Straße 100, 33098 Paderborn, Germany
{martin.assmann,engels,andreas.wuebbeke}@upb.de
[2] DC Application Development, arvato services, An der Autobahn, 33310 Gütersloh
Thomas.vonderMassen@bertelsmann.de

Abstract. Software Product Line (SPL) development provides the possibility of reusing common parts in similar software products. However the SPL approach does not centrally improve the maintenance of software products of a Software Product Line. This paper presents an approach for reducing maintenance costs of SPL products by using the concept Software as a Service (SaaS) by revealing exploitable commonalities concerning the maintenance of SPL platform components. This SPL-SaaS approach was developed with the experiences of arvato services integrating the software product line concept since years. It shows up the advantageous and disadvantageous characteristics of platform components that play a role for the concept combination. Main goal is to enable an IT-architect to identify platform components to be adequate for a common maintenance. Therefore criteria for the identification of platform components suitable for the approach are derived from these characteristics. Furthermore the requirements of the potential service users are examined and categorized concerning their effects on the system architecture. Special requirements of customers often lead to architectural constraints that are not compatible with the approach. If both, the criteria are met and the architectural constraints are compatible, the SPL-SaaS approach can be applied to a component. The whole approach is applied on an example of arvato services.

Keywords: Software product lines, Software as a service, Service-oriented computing.

1 Introduction

Software Product Lines were becoming an important development paradigm over the past years. The idea is to develop similar software products on a common basis, called platform. Arvato services has gathered experience with the SPL approach and always tries to exploit this approach in ever new ways. This paper presents the idea to extend the SPL development process with the maintenance process and illustrates this with the example of an address validation service. The goal is to reduce costs for maintaining software products by using the concepts of *Service-oriented Computing (SoC)* and

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 291–304, 2010.

Software as a Service (SaaS). In [1] and [8] it is already pointed out that the combination of SPL, Service Oriented Architecture (SOA) and Component Frameworks can benefit from each other. We have pursued a similar idea by combining SPL and SoC, which is a subordinate concept of SOA. But our approach distinguishes from [1] and [8] by not regarding everything as a service. The domain we are developing software for, deals with customer specific solutions in heterogeneous system and user landscapes. We learned from it, that it is hardly possible to satisfy the different customer needs and at the same time convert our SPL approach to a completely service oriented SPL. Often the customers are against giving away the sovereignty of their systems. Further on, things like non functional requirements (e. g. performance and security) and the increased complexity of the systems and their infrastructure impede the realization of a fully service oriented approach. Although, we try to identify single (functional) components, which can be exposed as a service without provoking the above mentioned problems.

At first, this contribution presents the idea of reducing maintenance costs by deploying platform software components as central, common services that are used for different software products at the same time. We point out the important characteristics of platform components, which make them reusable in the way our approach proposes. Thus, criteria for identifying platform components to be reused in the maintenance process are developed by us. In addition a categorization of the requirements of different service users is given. Any of these categories affect the system architecture in its own way. Certain effects respectively constraints are not compatible with the SPL-SaaS approach. Therefore it is revealed which requirement categories these are.

The remainder of this paper is structured as follows: In the 1st section foundations on software Product Lines and the concept Software as a Service are introduced. In the 2nd section the potential of extending the SPL development process is described. In the 3rd section the characteristics of our approach with its advantages and disadvantages are figured out. In section 4 we present an exemplary address validation service focusing its basic architecture. Related to the characteristics from section three and with the experience from the address validation service we present criteria to find fitting software components for our SPL-SaaS approach in section 5. Section 6 provides the related work in this area. The last section summarizes the results and points out further research topics in this field.

1.1 Software Product Lines

The Software Product Line approach deals with the development of similar Software Products based on a common platform. Thereby in all phases of the development process reuse of different artifacts is the main aim. In this context the platform provides different types of artifacts: Artifacts can be common to all products developed within the SPL. Artifacts can contain variation points to which variants are bind while developing a concrete Software Product. Furthermore each Product can have individual parts. The SPL development process proposed in literature contains the phases requirements engineering, design, implementation (realization) and testing. In these phases different techniques (e. g. variability modeling) provide the possibility of reusing parts of the platform in different products of the SPL. Further information about SPL and its development process can be found in [2] and [3].

1.2 Service-Oriented Computing and Software as a Service

The second major concept that is part of this paper is *Service-oriented Computing (SoC)*. A relatively similar concept is S*oftware as a Service (SaaS)*. Both utilize the service notion and both relate to software that is offered as a service. The vision of SaaS is to change the basic paradigm for development and maintenance of software systems, which is discussed in detail in [4] by Turner. He proposes to deliver software as a service rather than a licensed product. The main idea of the service is that it is not deployed where it is used but somewhere centrally. Users bind services needed at compile time or as preferred by Turner at runtime. Therefore the software services must have adequate descriptions, must be discoverable and should be composable, i.e. services can be created by combining other services. Service-oriented Computing, as described by Papazoglou in [5] and [6], builds the foundation for Service Oriented Architectures (SOA). SOA additionally addresses aspects like governance (like finances, employee training) and Business Process Management. Software services are used to build composite applications. Again services are understood as distributed components that need descriptions, discovery methods etc.

While SaaS aims on providing complete applications as services, SoC wants to provide business functions of finer granularity. Though, both approaches aim at advantages like lowered maintenance costs, higher degree of reuse as well as a different business model. It allows paying software per usage instead of buying licenses inflicting high fix costs. In the following we will point out how the software product line approach can benefit from these advantages.

2 Exploiting Potentials in the Whole SPL Lifecycle

As already pointed out in section 1.1 the SPL approach tries to increase the efficiency of software development by identifying common components and exploiting their similarities. But software costs money during its whole lifecycle. Regarding to software development processes like the Rational Unified Process (RUP) or the waterfall model the software lifecycle usually contains a phase for maintenance and operations. Within the RUP this phase is called transition [7]. SPL covers mainly the first three phases of the RUP. We believe that there are potential cost savings in the maintenance and operations phase being not utilized yet.

The SPL approach primarily decreases the development effort for software products but hardly addresses reduction of their maintenance and operation costs. Deploying and maintaining products separately at customer sites, makes it hard to exploit commonalities of the products. This means that the common components of a platform are only reused until software product assembly. Afterwards, deployed common components exist in separate system environments and are maintained individually. Therefore their maintenance is as expensive as the maintenance of single software products. As similarities between components exist but are not exploited at all regarding maintenance we think that there is a high cost saving potential in the transition phase. However, SPL domains that do not allow remote communication within their software components cannot benefit from our idea, for example SPLs for embedded systems like cell phone software.

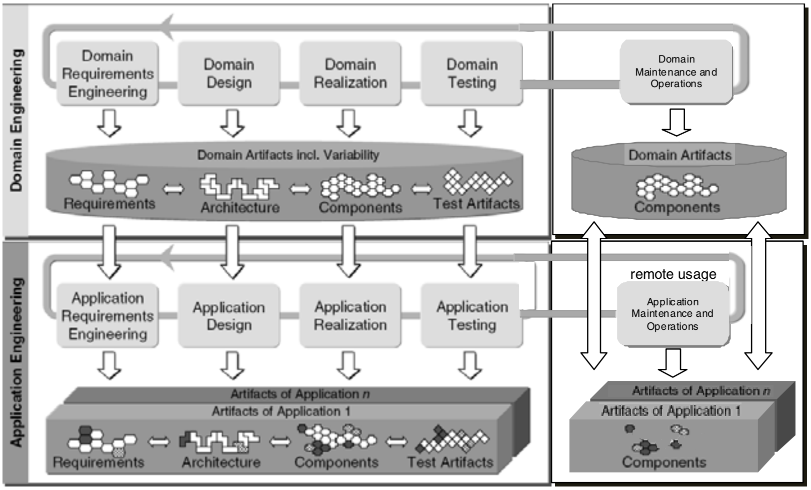


Fig. 1. Extension of the SPL development process modified from [2]

Figure 1 shows the extension of Pohl's SPL development process [2] by a new pair of sub-processes called domain maintenance and operations, and application maintenance and operations. As the figure illustrates, the common components are held on the domain level (maintained and operated by the SPL platform provider). Only the product specific components are maintained and operated by the customer. The communication between the two levels is realized in a remote way. As shown, the lifecycle process is extended to the maintenance sub-processes, because change requests concerning the components take effect on the maintenance.

Figure 2 illustrates the changes from the usual situation in SPL architectures (upper part) to the situation that our approach suggests (lower part). The upper part of the picture depicts the architecture of two customers with individual applications. In the realization phase both have been assigned a bonus system component and a customer management component. The blue background of the bonus system component indicates that this is a custom component that usually has some customer specific modifications. The customer management component is deployed with little or no modifications.

The transformation addresses the deployment of the customer management component. Instead of deploying it once for every customer it is deployed as part of the SPL Platform (platform component). The customer application now remotely uses the component as a software service. This means that the component is reused and has to be maintained and deployed only once. A major question arising when examining figure 2 is: “What makes a common component in a software product line that is suitable for the deployment as a service?” Probably just as important is to know: “Are there any other advantages and which disadvantages come with the approach?” We will answer both questions in the sections 3 and 5 in reversed order.

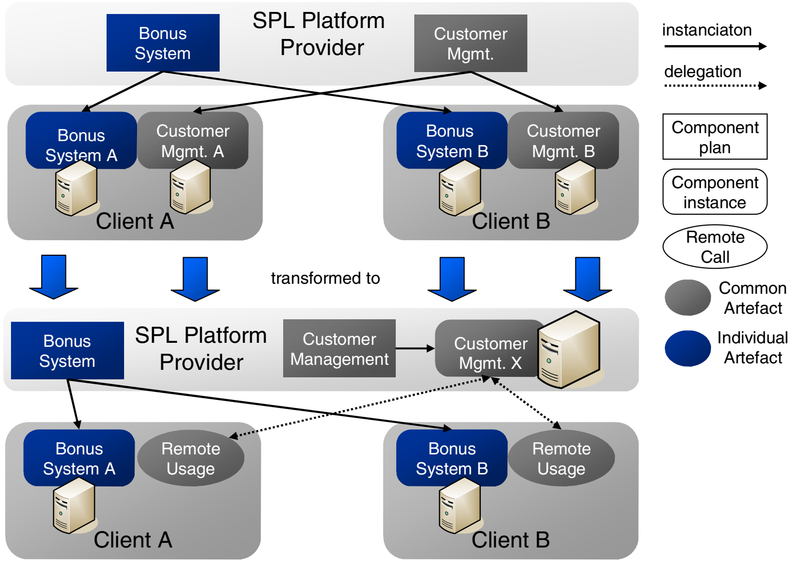


Fig. 2. Changes in the component deployment

3 Characteristics of the Combined Approach

For our approach we want to answer two important questions. Firstly, what are the advantages/ disadvantages? Secondly, when is a software component suitable for the approach? In the following we list the characteristics starting with positive ones. With each characteristic we try to identify influencing factors of software components that can intensify/weaken the advantage/disadvantage. In section 5 these influencing factors are carried together. With this information the software architect can determine software components with characteristics that minimize disadvantages and maximize advantages of the approach. Regarding software product lines there are always two interesting aspects, the variation points and their binding time. There are two major categories of variation points. Firstly, the deployment location of a service has to be settled during design time. Secondly, there are minor variations for centrally deployed services. These variations reflect the range of the customers' requirements for the service. After the range is anticipated the service is designed accordingly. By this the variability can be bound at design time. The variation points are described in section 5.

3.1 Advantageous Characteristics

The central benefit of our approach will be effort savings in software product maintenance and operations. In the classical SPL development process the reuse of common parts of the platform is limit to the phases Domain Engineering, Domain Design, Domain Realization and Domain Testing. The idea of sharing common parts in maintenance is based on the concept of Service-oriented Computing (SoC). Advantages in detail are the following:

First of all several issues can be consolidated. The first four points address this topic. Firstly, hardware (e. g. a server system) can be used for several products. This means to deploy a common platform component only once on a central server system and share it via remote connection.

Secondly, by sharing a server system providing services of platform components deployed on it, several maintenance and operation aspects can be improved: availability and backup solution only has to be treated once, i.e. before we had several servers and every of them had to have availability and backup mechanisms like idle stand-by servers and mirrored hard disks. Functional extensions and updates to a common component have to be made only once. The same applies to the correction of faults. These changes always concern one component and thus the distribution effort is reduced, because the platform component is deployed only once on the shared server system.

Thirdly, the overall operation costs for hardware resources are reduced because as a single instance of a component requires less resources than several instances. The mentioned advantages all lead to less effort concerning maintenance and hence reduce the costs for it. According to this advantage suitable components should underlie frequent changes for updates. Additionally they should have high availability and backup requirements.

Fourthly, consolidation comes with an additional advantage concerning load-balancing. Usually a customer with its own systems can hardly afford to cover peak loads so his systems will just cover average load. Even if the centralized service is only able to cover average load of all its consumers, a single consumer causing a performance peak will not cause heavy performance losses on the central system, because his peak load has to be put into relation with computing power of the system designed for several customers. For the suitability of a component we can derive that it is increased if the component causes critical peak loads with relatively low average load.

Furthermore our approach on treating software product line platform components as services opens up the possibility for new cost models, which are addressed by the next two paragraphs. In this context the maintenance and operations costs for the common parts of all products of the SPL can be shifted from a model with high fix investment cost (e. g. license, hardware) combined with effort for human resources to a usage cost model. Until now the customer usually buys a license of the component (predictable fix costs) and takes care of the maintenance and operations himself (usually predictable fix costs). Now there are mainly two new options for payment. Firstly, the customer can buy the service, its maintenance, and operations for a fixed amount of money per period. The second possibility derives from the SaaS approach. It means that the maintained and operated service is paid per usage only (variable costs). The latter kind of cost model has definitely less fix costs, which reduces the financial risks of the customer. Hence suitable platform components are expensive with a high investment risk for the customer.

Furthermore, the new cost model provides the possibility of outsourcing the maintenance of the common parts of the product line. Firstly, this leads to the possibility for the customer to save money, because the SPL platform provider is able to be more efficient in maintenance and thus cheaper (even if the provider makes profit with it). Secondly, it provides the chance for the product owner to concentrate on the product specific core business functions. In the end this approach can lead to a win-win situation for the customer and the SPL platform provider.

In addition the customer can outsource functions that do not belong to his core business. Especially in combination with the pay-per-usage cost model he gains flexibility, i.e. he can dispose of the product respectively the service easily. From this we can conclude that suitability is influenced in a positive way if platform components do not provide core business functions of the customer. We suppose that customers do not want to source out their core components, as they are too confidential and specialized.

3.2 Disadvantageous Characteristics

With the combined approach several drawbacks arise. These have to be concerned with future research work on this topic. In the following we point out disadvantages and as in the section before also the factors regarding software components that influence the disadvantages.

The first three points address drawbacks that arise from the centralization of components. Firstly, all shared software services have to be client-capable unless they are stateless. That means customer data is stored by the component this has to be taken into account designing the component. If every customer runs its own instance, then data and also access rights are divided without extra effort. Suitability of software components is increased by statelessness and public access. Otherwise the implementation of client-capability is inevitable.

The second point is closely related to the previous one. Data sovereignty is transferred from the customer to the service provider. This may lead to acceptance problems if the data is of high confidentiality, e.g. bank transfer data. The requester has to trust the provider that handles the data with adequate security mechanisms. Often client-capability is assumed to be less secure as data of several requesters is stored in the same main memory, providing no physical barrier. We think that a physical barrier is not needed, no matter how confidential the data may be. Though, we cannot prove this at the moment. Suitability of software components will be decreased if the component stores confidential data and the customer does not trust the service provider at the same time.

Thirdly, virtual reuse means that a single server solution does the work, which has some disadvantages compared to a solution with several completely independent servers. A single point of failure is created by this. If the central component crashes, then every customer is affected. Nowadays, this should only be a question of costs as high-availability server solutions are on the market. The single point of failure is a performance bottleneck at the same time. The load that was distributed over many systems is now concentrated on one. Therefore a high performance system may be required. On the one hand, in peak load situations again all customers are affected, even those that are not responsible for the peak load situation. On the other hand a load balancing between all the requesters is given now. Peak loads of single requesters do not lead to performance problems as their single server would have encountered. High availability and high performance server solutions are required unless performance and availability of the centralized components are not critical. For the suitability of a platform component this means that high average loads or simultaneous peak loads of several users are negative.

Fourthly, communication via internet is sheer unavoidable. This has several consequences: Secure transmission has to be implemented. This might not have been necessary in decentralized solutions with data transfer in secure subnets only. Secure communication usually leads to higher communication overhead. Furthermore, the reliability of internet connections usually cannot be guaranteed. Last but not least it is more difficult to implement communication that can be initiated by both, remote and local, components. Suitability of components is decreased by data confidentiality as well as high communication effort and the ability to initiate communication. Higher reliability demands than the internet can offer prohibit the service approach.

The fifth point concerns change requests. As we pointed out an advantage in maintenance updates, we must also point out that individual change requests have a negative effect on the approach. Too much individuality caused by change requests leads to software that is hardly maintainable and can decrease all the benefits of the approach. However, any change request leads to a down time if the software component is not runtime reconfigurable. All customers are affected from the downtime but all except from one have no benefit from the down time. Individual change requests are a serious thread to the approach as they are to SPL in general. Only platform components with little individual change requests are suitable, but these should be identified in the SPL platform anyways.

4 The Address Validation Service

Arvato services is an enterprise specialized on customized software solutions and services for outsourcing solutions. To achieve a high degree of efficiency by using synergies among different projects, arvato services is on the way to implement a software product line approach as depicted in figure 1.

The address validation service (AVS) is a service allowing access on an address database. In general the requester sends an address to the service and the result is whether the address is valid or not. The validity check of an address contains several sub checks that are partly optional. For example the validity check if city and postal code match is obligatory. The comparison with blacklists is optional and charged separately.

Address validation is useful for many applications that can be (and some also were) produced with a software product line approach. Therefore it was recognized as a reusable component and as this it is used in different products.

If deployed at the customers IT site this component causes high maintenance costs. First of all it has to be regularly updated with new customer data and secondly optimization of algorithms is steadily done due to the high and still growing number of data entries. Furthermore, the component requires relatively expensive hardware, so that performance can be guaranteed. These are the main reasons why the address validation component was designed to be a centrally hosted service. It is remotely connected to customer systems that are produced with the product line and can also be used as a stand alone service.

The cost model is realized as a pay per usage cost model, which means that for each customer service calls are counted and charged accordingly.

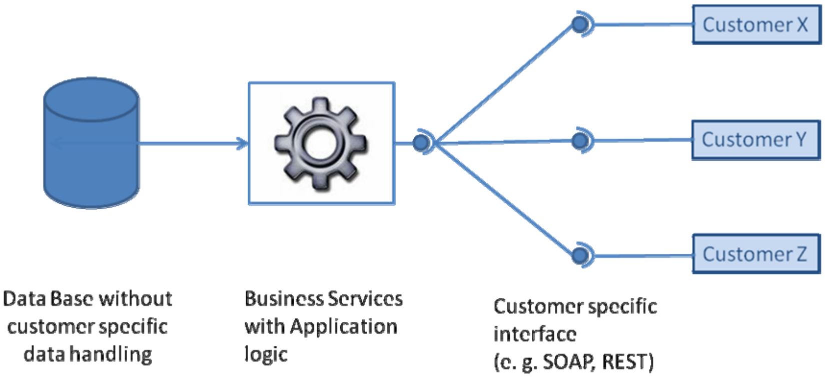

Customer X
Customer Y
Customer Z
Data Base without customer specific data handling
Business Services with Application logic
Customer specific interface (e. g. SOAP, REST)

Fig. 3. Basic service architecture

The address validation is an example that can realize many of the advantageous characteristics while avoiding many of the disadvantageous characteristics of the SPL-SaaS approach. For example the high cost savings that can be achieved because the deployment on expensive customer hardware is avoided as well as the unproblematic trust of customers for not having to store address validity data on their own systems.

The very basic system architecture of the address validation service is depicted in figure 3. In the back end there is a data base containing all the relevant data for the validity checks of the addresses. Its content is steadily kept up-to-date. Furthermore there is a business service that contains all the business logic to serve customers' validation requests. It has exactly one well defined interface offering all possible operations.

As customers may have their requirements concerning the interface, adapters offering different interface technologies like SOAP and REST are created. In addition the adapters may hide parts of the functionality, reducing the complexity for the customer as well as strictly enforcing access rights.

The address validation service has become very successful and is now often used as a stand alone product. But still we have great cost-reducing effects from embedding the service in customized software solutions that are created with the help of our software product line approach.

5 Characteristics of Suitable Software Components

To identify suitable components we derive two categories of characteristics. Firstly, the high level characteristics delivering service candidates that are worth the effort of being analyzed on the architectural level. If the analysis on the architectural level is also positive, the realization of the platform component as a centrally deployed service is assumed as economically recommendable.

5.1 High Level Characteristics

As already mentioned our SPL-SaaS approach has several advantages and drawbacks that are dependant from the choice of the platform components centrally maintained and delivered as a service. With the experience gathered from the address validation service in mind, we analyzed advantages and drawbacks mentioned in section three. We derived adequate high level criteria to evaluate suitability of components for our SPL-SaaS approach. We identified positive as well as negative criteria. If components mainly fulfill the positive and mainly avoid fulfilling the negative criteria, they are adequate service candidates having enough potential for achieving a reduction on maintenance costs. The positive criteria maximizing the benefit of the approach are:

- High usage degree provides high reuse potential
- Frequent changes (functional extensions, fault corrections)
- High availability and backup requirements
- Components cause critical peak loads but have relatively low average load
- A high price which means a high investment risk (due to fix costs) for the customer

On the other hand there are negative criteria that should be minimized with the choice of suitable components to increase benefit:

- Providing core business functions of the customer
- Statefulness of the service and non-public access
- Storing confidential data with the component while the customer does not trust the service provider
- High average loads or simultaneous peak loads of several users
- High communication effort
- Bi-directional initiation of communication
- Higher reliability demands (higher than the internet can provide)
- Individual change requests
- High performance requirements

5.2 Architectural Level Characteristics

If there is a service that fits to the criteria listed in the previous subsection, it is not yet clear if it is adequate for our approach. This is because of the architectural constraints that will arise due to the requesters' requirements. In general the abstract system architecture would be as depicted in figure 3. There is a database that is accessed by a service and a business service implementing the business logic used by the database. In addition there are several customer specific interfaces, which access the business service interface.

But there can be differences in the architecture according to varying customers' requirements. We have depicted these differences that have to be covered by the service in a feature diagram in figure 4. To any given service candidate the range of possible customer requirements being supported has to be anticipated. This results in a set of variation paths that have to be covered by the service solution. Every path indicates a certain complexity concerning the service realization and operation. Within the figure the complexity is generally decreasing for leaves from left to right. The variation points from the feature diagram are application logic, data base schema, data base content and interface. Any of them can be required to be customer specific or not, but not every combination makes sense. The feature diagram lists the five reasonable variation possibilities.

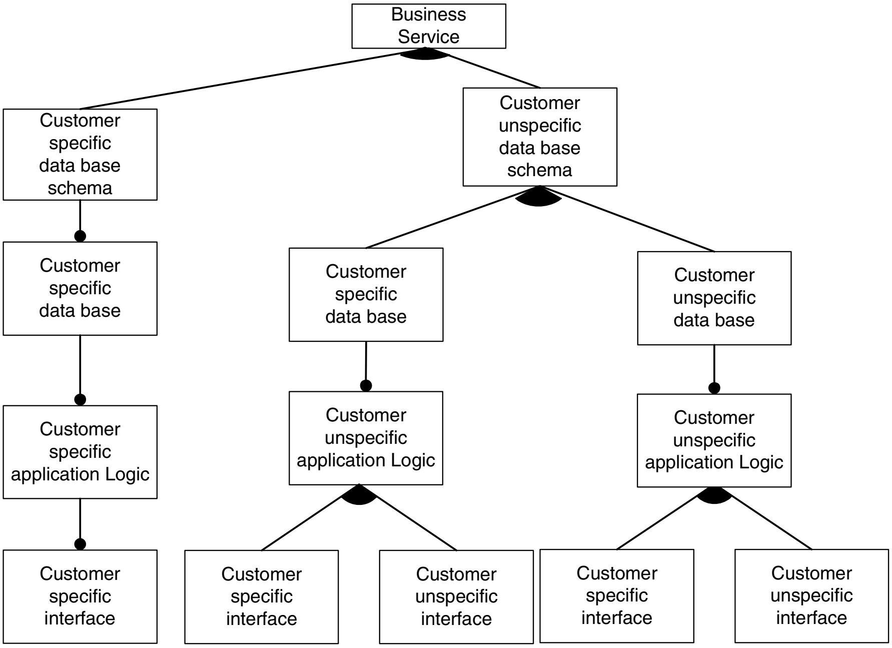

Business Service
Customer specific data base schema
Customer unspecific data base schema
Customer specific data base
Customer specific application Logic
Customer specific interface
Customer specific data base
Customer unspecific application Logic
Customer specific interface
Customer unspecific interface
Customer unspecific data base
Customer unspecific application Logic
Customer specific interface
Customer unspecific interface

Fig. 4. Service architecture feature diagram

First of the variation points is the data base schema. It may happen that customers need to have different data base schemas within a service. This requirement makes it quite hard to realize the service economically as it implies that each of the following presented variation points is forced to the customer specific choice. An example is a master data management service. Its purpose is to allow customers to store, update and request their master data. This is useful as else wise master data would be scattered over several databases. But for our case this would imply that we have to maintain a data base schema for each customer. Customer specific business logic and interfaces would also be required. For our address validation service, there exists only one data base schema serving all customers.

Second variation point is the data base content. If customers can be served with a service that has only one data base schema it still may occur that each customer has its own content for the data base. Furthermore the customers do not want that their data is accessible by other service users. An example for customer specific data base content is a schedule service. It offers a calendar and participants known to the system can be invited to meetings. The data base schema is the same for every customer, client, but every customer wants to be sure that only his employees can see their own appointments. To this end, data records could be marked with their owners name so that access control can be guaranteed. In the end it means less effort to maintain a database with a single schema instead of maintaining multiple schemas. In the case of our address validation service all customers access the same data base content – the addresses plus some extra validation content.

Third variation point is the application logic. It is possible that not all customers want exactly the same operations on the data. If that is the case, usually the data base schema and the thus also the data base content will be customer specific. This is the worst case scenario because it means that every customer has its own specific service. This makes it hard to realize cost efficiency with our SPL-SaaS approach and is generally not encouraged. If there is only customer unspecific application logic, then we assume that there are only customer unspecific data base schemas. Executing the same operations on different data schemas is generally not advisable. The address validation service offers the same application logic for all customers.

Fourth variation point is the interface towards the customer. Even if the service is always part of products built with the SPL and the customer does not access it directly, it is helpful to have different interfaces. For example, if variability within the SPL platform allows products based on different technologies, several interfaces differing in technology are useful. The different interfaces are only adapters to the interface offered by the business component. Adapters could also be located on the client side, but if other customers want to access the service directly, it is an advantage being able to offer a plethora of interfaces to the service. A customer specific interface can also hide some functionality if the complexity of functions shall be reduced towards the customer. Additionally it can be used to restrict the access to certain functions by simply not adapting them. The adapters can restrict the functionality of the original interface but do not change it in other ways. Different point-to-point security variations, like REST over https, can also be covered with adapters. The address validation service offers different interfaces like SOAP over JML and REST over http.

We have seen that not all combinations of features make sense. According to figure four there are five combinations. Each combination implies a different complexity for the centralized service.

The left most leaf shows the most inadequate case. Customer specific data base schemas require higher development, test and maintenance efforts. Therefore the realization of a service that has only customer specific properties is discouraged.

The right most leaf has only customer unspecific variants. This means that there is no variability that has to be bound. The service provider is in the lucky position to design one service and to have multiple service users. In this case efforts for design, test and maintenance do not scale with the number of users. This is the good situation as the savings with the centralized service approach are very high.

Customer specific interfaces are relatively easy to develop and maintain. Testing new customer specific interfaces can be reduced to testing the sole interface instead of testing the whole service. Our address validation service is an example for a service categorized like the second leaf from the right. It has different interfaces and we have experienced that each interface causes only little effort overall. Furthermore an existing interface might be reused for a new customer. The more interfaces there are the higher is the probability to be able to reuse an interface.

The two remaining leaves in figure 4 cause more design, test and maintenance effort, but are still considered as suitable service candidates.

6 Related Work

The idea of combining the SPL and SaaS approach is addressed by several contributions. The importance of the topic is brought to life by [1] and [8].

In [9] the authors describe the idea of a web based product line. In this case technological issues on building such product lines are discussed. Compared to our approach the contribution concentrates on building product lines completely from web services.

[10] concentrates on variability in web service flows. Some of the described variability points like *protocol* are also interesting for our approach, but the aim of our contribution is not centrally to handle variability in the flow of web services.

Chang and Kim also recognized the common reuse potentials in SPL and SOA [11], but they consider everything as a service. Thus, they identify variation points on process level, which is not applicable for our domain, because of the previously mentions drawbacks (see also section 1).

An interesting approach combining SOA and SPL concepts for creating business process families is given in [12]. Though, this approach does not cover the deployment phase.

In [13] the authors describe how to manage variability in service centric systems with technologies from the SPL approach. Our approach works the other way round and provides service technology for SPLs.

7 Conclusions and Future Work

In this paper we presented an approach to evaluate commonalities in platform components to be suitable to reduce maintenance costs of software products, developed from a SPL. Therefore the SPL process is extended with the maintenance process. This means to identify and deploy common platform components centrally and offering them as a service. The service then is used for different products of the SPL. This concept is called Software as a Service as the common platform components can be seen as a service, provided for different products. The presented approach holds several advantageous and disadvantageous characteristics ready, which have to be taken into account for selecting adequate candidates for common services. From these characteristics several criteria for identifying suitable platform components have been derived. Additionally we have developed a service architecture feature diagram, which provides the possibility to evaluate components concerning their adequacy for our SPL-SaaS approach. Suitable components have a high reuse potential while causing little service development costs. We presented an example of such a component from the arvato environment to show the practical need of our idea.

The presented criteria are a first step for a detailed evaluation catalogue for assessing common components concerning to their reuse potential. In future research we are going to built up this evaluation catalogue to evaluate the reuse potential in reference to maintainability. Afterwards the catalogue is going to be used and evaluated by analyzing the arvato SPL to show its workability.

References

1. Helferich, A., Herzwurm, G., Jesse, S., Mikusz, M.: Software Product Lines, Service-Oriented Architecture and Frameworks: Worlds Apart or Ideal Partners? LNCS, pp. 187–201. Springer, Heidelberg (2006)
2. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
3. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley Professional, Reading (2001)
4. Turner, M., Budgen, D., Brereton, P.: Turning Software into a Service. Computer 36(10), 38–44 (2003)
5. Papazoglou, M.P., Georgakopoulos, D.: Service Oriented Computing. Comm. ACM 46(10), 25–28 (2003)
6. Papazoglou, M.P.: Service-Oriented Computing: Concepts, Characteristics and Directions. In: 4th International Conference on Web Information Systems Engineering (WISE 2003), Rome, Italy (2003)
7. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley Professional, Reading (1999)
8. Wienands, C.: Synergies Between Service-Oriented Architecture and Software Product Lines. In: 2006. Sie-mens Corporate Research, Princeton, NJ (2006)
9. Sillitti, A., Vernazza, T., Succi, G.: Service based Product Lines. In: Proceedings of the 3rd International Workshop on Software Product Lines: Economics, Architectures, and Implications, ICSE 2002 (2002)
10. Segura, S., Benavides, D., Ruiz-Cortés, A.: A Taxonomy of Variability in Web Service Flows Service Oriented Architectures and Product Lines. In: SOAPL, SPLC 2007, Kyoto, Japan (2007)
11. Chang, S.H., Kim, S.D.: A Variability Modeling Method for Adaptable Services in Service-Oriented Computing. In: Software Product Line Conference, SPLC 2007, pp. 261–268 (2007)
12. Ye, E., Moon, M., Kim, Y., Yeom, K.: An Approach to Designing Service-Oriented Product-Line Architecture for Business Process Families. In: Proceedings of the 9th International Conference on Advanced Communication Technology, Phoenix Park, Republic of Korea, pp. 999–1002 (2007)
13. Lee, J., Muthig, D., Kim, M., Park, S.: Identifying and Specifying Reusable Services of Service Centric Systems Through Product Line Technology. In: Proceedings of the First Workshop on Service-Oriented Architectures and Product Lines (SOAPL 2007), pp. 57–67

Service Based Development of a Cross Domain Reference Architecture

Liliana Dobrica[1] and Eila Ovaska[2]

[1] University Politehnica of Bucharest, Faculty of Automatic Control and Computers
Spl. Independentei 313, Romania
liliana@aii.pub.ro
[2] VTT Technical Research Centre of Finland, Oulu, Kaitoyvala 1, Finland
eila.ovaska@vtt.fi

Abstract. An important trend of software engineering is that systems are in transition from component based architectures towards service centric ones. However, software product line engineering techniques can help in a quality based and systematic reuse. The content of this paper addresses the issue of how to perform design and quality analysis of cross domain reference architecture. The reference architecture is designed based on the domains requirements and features modelling. We propose a service based approach for cross-domain reference architecture development. Throughout the sections we try to introduce an innovative way of thinking founded on bridging concepts from software architecture, service orientation, software product lines, and quality analysis with the purpose to initiate and evolve software systems.

Keywords: Software, Cross domain reference architecture, Service, Design, Analysis, Quality, Scenarios.

1 Introduction

In software development, the systems of yesterday become components of today systems. The fundamental principle stating that "any system consists of components" is common for any technical system and it is sometimes called "a law of nature" [6]. Among the requirements and constraints that have to be satisfied we can mention a higher diversity and complexity of systems and components, increased quality, productivity and reuse content, standardization, and stricter time-to-market. The domain technology causes exponential growth of the designed systems.

Nowadays many systems are used as subsystems in a variety of domains such as enterprise systems, embedded systems, and so on. In these domains there is a variety of functions; however, they might be composed of a limited number of common software/hardware components. In various industries it has been recognized a significant duplication of development effort for hardware, software and services [1]. Due to the escalating complexity level, the technology trends and increased competition in the world market, a coherent and integrated development strategy is required. It becomes a research priority the creation of a generic platform and a suite of abstract components with which new developments in different application domains can be engineered with

L.A. Maciaszek, C. González-Pérez, and S. Jablonski (Eds.): ENASE 2008/2009, CCIS 69, pp. 305–318, 2010.

minimal effort. Generic platforms, or reference designs, can be based on a common architectural style that supports the composition of systems out of independently developed subsystems that meet the requirements of the different application domains. Given a core architectural style, different components are created for different application domains, while retaining the capability of component reuse across these domains.

Reference architecture (RA) serves several purposes, of which the most important are knowledge base, starting point and reuse. Knowledge base represents a common terminology for software system architects. The shared terminology enables architects to communicate experiences more efficiently. Starting point means that architectural documentation can be used as a root for an iterative development process, reducing in this way the effort for designing architectures for new products. Reuse in the sense that the RA describes the generic structure and behaviour of the services. This makes integrating existing compliant software components easier, and thus increases the reuse potential of those services. The RA components, interfaces and constraints are abstract and complex. Not all the development organizations will understand them well. Not knowing RA capabilities may lead to the architecture not being fully used. However, the aim is that all products should fit into the provided RA and benefit from it. Requirements that have already been considered might be re-implemented for various products. An impact of multi-implemented requirements is an unstable RA.

In this paper we propose a coherent and integrated development strategy for complex systems that considers the architecture the main design artefact. We argue with our experiences in the software architectures design and analysis for various domains [4][5] and other researchers' recent studies that will be revealed during the paper. Our contribution is in the synthesis of the most important issues that can be applied in a cross domain development strategy based on quality. We propose a service based approach for cross-domain RA development.

2 Background

The focus of this section is to discuss about those elements directly connected to the paper's contribution. Since our approach combines a couple of matured concepts from software and service architecture, software product line and quality evaluation at architectural level in a new and synergetic manner, these will be introduced.

2.1 Software and Service Architecture

Software architecture (SA) provides design-level models and guidelines for composing software systems. The SA is defined as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" [12]. The SA description is designed to address the different perspectives one could have on the architecture. Each perspective is a view. The information relevant to one view is different from that of others and should be described using the most appropriate technique. Several models have been proposed including a number of views that should be described in the SA. The view models have something in common. They address the static structure, the dynamic aspect, the physical layout and the development of the system. In general, it is the responsibility of the architect to decide which view to use for describing the SA. Architectural styles are recurring patterns of system organization. Their application results in systems with known, desirable properties. In practice, a style consists of rules and guidelines for the partitioning of a system into subsystems and for the designing of the interactions among subsystems. The subsystems must comply with a style to avoid a property mismatch at the interfaces between them.

Service architecture is a set of concepts and principles for specification, design, implementation and management of software services [7]. This definition is similar to SA that also includes the principles for guiding its design and evolution and has a strong influence over the lifecycle of a system [10]. Service architecture refers mostly to the software architecture of applications and middleware which is the software that is located between applications and network layer. A middleware layer hide the underlying network environment complexity insulating applications from explicit protocol handling, disjoint memories, data replication and parallelism. Furthermore, the middleware layer masks the heterogeneity of operating systems, programming languages and networking technologies to facilitate application programming and management [8]. A service based approach provides support for adaptability and flexibility of components and frameworks [9]. A design approach of services at the architecture level has to consider quality attributes and standards.

2.2 The Software Product Line Development

In general the software product line development consists of two stages which are domain engineering and application engineering (Fig. 1) [15].

Domain engineering is divided in: Domain Analysis, Domain Design and Domain Implementation. The domain analysis consists of capturing information and organizing it as a model. Some methods, such as FODA (Feature-Oriented Domain Analysis) [3] propose a set of notations for the domain modelling using the notion of features to refer to products properties. The input represents domain knowledge and outputs are domain requirements. The domain design consists of establishing the product line architecture. The domain implementation consists of implementing the architecture as software components. The results represent core assets such as domain requirements, product-line architecture and components. The application engineering stage consists of building products based on the results of domain engineering and users needs. During application analysis of a new system, the requirements of the existing domain model, which matches the user's needs, are selected. Applications are assembled from the existing reusable components. Variability management is a key issue in the success of product line engineering.

2.3 Quality Evaluation Techniques at the Architectural Level. Scenarios

Evaluation techniques are categorized in questioning and measuring techniques [12]. The first category generates qualitative questions to ask about a SA and they are applied to evaluate SA for any given quality. Questioning techniques include scenarios, questionnaires and checklists. Measuring techniques suggest quantitative measurements to be made on SA. They are used to answer specific questions and to address specific software qualities, and therefore, they are not as broadly applicable as

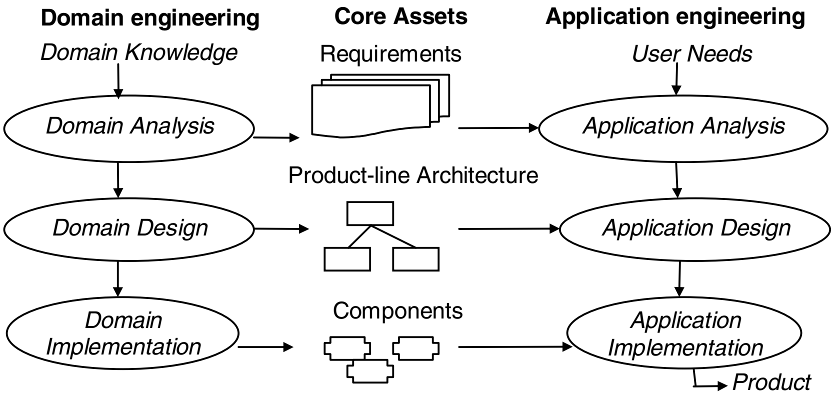


Fig. 1. Software product-line development

questioning techniques. This category includes metrics, simulations, prototypes and experiences. In terms of quantitative and qualitative aspects, both classes of techniques are needed for evaluating SA. Various analyzing models expressed in formal methods are included in quantitative techniques. Qualitative techniques illustrate SA evaluation with scenarios. Scenarios are rough, qualitative evaluations of architecture. Scenarios are necessary but not sufficient to predict and control quality attributes and have to be supplemented with other evaluation techniques. Including questions about quality indicators in the scenarios enriches SA evaluation.

The existing practices with scenarios are systematized in [12]. The usage of scenarios is motivated by the consensus it brings to understanding of what a particular software quality really means. Scenarios are a good way of synthesizing individual interpretations of a software quality into a common view. This view is more concrete than the general software quality definition and it also incorporates the specifics of a system to be developed, i.e. it is more context sensitive. Scenarios are a postulated set of uses or modifications of the system, they are typically one sentence long, and modifications reflected in scenarios could be a change to how one or more components perform an assigned activity, the addition of a component to perform some activity, the addition of a connection between existing components, or a combination of these factors. The scenario development is based on the system requirements that are considered in the architecture. Scenarios have to be sufficiently concrete to ensure the expressiveness of the analysis.

3 Our Approach

In an innovative way of thinking our work bridges the mature concepts from software architecture, service orientation, software product lines and quality analysis with the purpose to initiate and evolve complex software systems. We consider that one of the next major steps forward in SA development will be driven by methodologies and tools that give systematic and flexible means of reaching a goal. In our view a service based approach for development of a cross domain RA integrates iteratively specific design and quality analysis techniques. This section gives a big picture of these techniques.

3.1 Architecture Design

We define a cross domain approach that extends to three levels of the architecture development of a software system (Fig. 2.). We consider the system as a collection of cooperating services that deliver required functionality. These services may be executed in a networked environment and may be recomposed dynamically. The RA level includes core services and focuses on commonality analysis. Also the RA includes rules or constraints on how core services should be combined to realize a particular functional goal. The domain architecture level includes domain specific services and it requires variability management concerns. The last level is reserved for the set of product architectures, where rules for product derivation and configuration are included. A feature model is a prerequisite of our approach [2]. This model is essential for both variability management and product derivation, because it describes the requirements in terms of commonality and variability, as well as it defines dependencies. We have built an UML meta-model for features modelling (Fig. 3.). The features model specifies dependencies called composition rules. The *requires rule* expresses the presence implication of two features and the *mutually exclusive rule* captures the mutual exclusion constraint on combinations of features.

RA defines quality attributes, architectural styles and patterns and abstract architectural models (Fig. 4.). *Quality attributes* clarify their meaning and importance for core service components. The interest of the quality attributes for the RA is how the quality attribute interacts with and constrains the achievement of other quality attributes. Services have to meet many quality attributes. Modifiability of a service is divided into the ability to support new features, simplify the functionality of an existing system, adapt to new operating environments, or restructure system services. Integrability measures the ability of the parts of a system to work together. It depends on the external complexity of the components, their interaction mechanisms and protocols, and the degree to which responsibilities have been clearly partitioned.

The *styles and patterns* are the starting point for architecture development (Fig. 4). Architectural styles and patterns are means to achieve qualities. A style defines a class of architectures and it is an abstraction for a set of architectures that meet it. An architectural pattern is a documented description of a style or a set of styles that expresses a fundamental structural organization schema applied to high-level system subdivision, distribution, interaction, and adaptation [13].

Design patterns, on the other hand, are on a detailed level. They refine single components and their relationships in a particular context [14]. In this way, the RA creates the framework from which the architecture of new products is developed. It provides generic architectural services and imposes an architectural style for constraining specific domain services in such a way that the final product is understandable, maintainable, extensible, and it can be built cost-effectively. Potential reusability is highest on the RA level. Core services and the architectural style of the RA are reused in every domain architecture. The RA is build based on a *service taxonomy*. We adopted the idea from WISA [11] of an existing knowledge on software engineering that is integrated and adapted to service engineering. The standards related to each domain, applicable styles and patterns and existing concepts of services and components are the driving forces in the development. A service taxonomy defines the main categories called domains. Typical features that have been abstracted from requirements characterize services. The reason for a

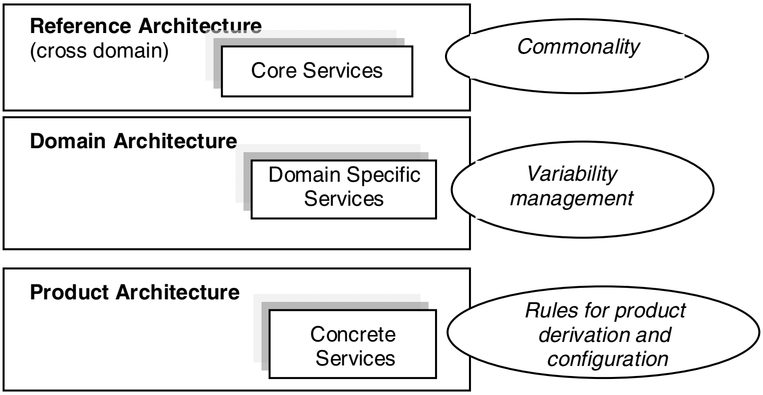


Fig. 2. Three-level architecture development approach

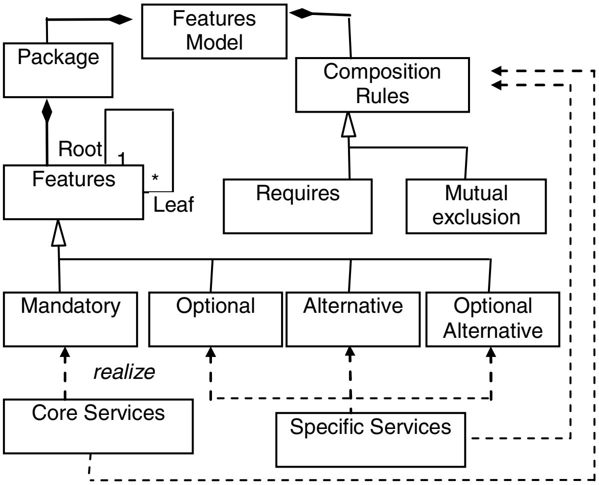


Fig. 3. Features - UML metamodel

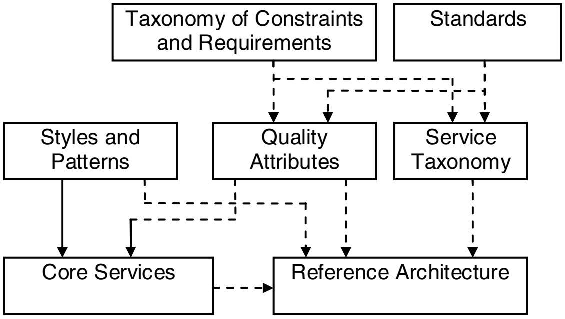


Fig. 4. RA realization

service taxonomy is guiding the developers on a certain domain and getting assistance in identifying the required supporting services and features of the services.

Domain architecture describes ready made building blocks that assist application/products developers in using specific domains services. When the RA has been defined, the existing components and services are considered as building blocks of the architecture of the set of products. The domain services provide variable assets repository. Variability appears in functional and non-functional requirements (including quality attributes). A structured domain architecture repository may be provided at this level. A schema for this repository has to be defined in a form such as relationships between services. In this way, we are mapping domain specific services to core abstract services. The specialization relation is a solution to be used for variability management. Run-time quality attributes variability requires tool support for its modelling. This tool must provide monitoring mechanisms, measuring techniques and decision models for making tradeoffs [3].

The product architecture level consists of concrete services that are derived and configured based on rules. The goal of product derivation is to reach a configuration in which necessary variabilities have been bound. The decision model for bounding specific services of a domain to a product may be in a tabular form or a more comprehensive tool based on the feature types and composition rules. By selecting a consistent set of features that are asked for an individual product, the corresponding domain specific services that realize those features are selected from the domain architecture repository to constitute the product.

3.2 Architecture Analysis

We have applied an analysis method that consists of the following five steps:

1. Deriving of change categories from the problem domain. Fig. 5 presents five categories of change scenarios derived from the problem domains. A change scenario related to one of these categories may require other changes in the other categories. It is recommended to consider this possibility in the scenario development process. Usually it is easy to identify the main roots and add subsequent features to the domain when the problem domain is well-organized.

2. Scenarios identification. Possible changes may happen in the life of the system based on the derived categories. Scenarios should illustrate the kinds of anticipated changes that will be made to the system. A common problem of the scenario development is when to stop generating scenarios. Using a set of *standard quality attribute-specific questions* we ensure proper coverage of an attribute by the scenarios. The boundary conditions should be covered. A standard set of quality-specific questions allows the possibility of extracting the information needed to analyze that quality in a predictable, repeatable fashion. It is assumed that the architecture is a good one and it is not necessary to generate scenarios to verify the functional requirements. Otherwise these should also be considered. When analyzing the modifiability we must look for possible changes in the problem domain.

3. Architecture Description could be performed in parallel with the previous step. Architecture description may use multiple views. For a common level of understanding a small and simple lexicon could be used in describing structures.

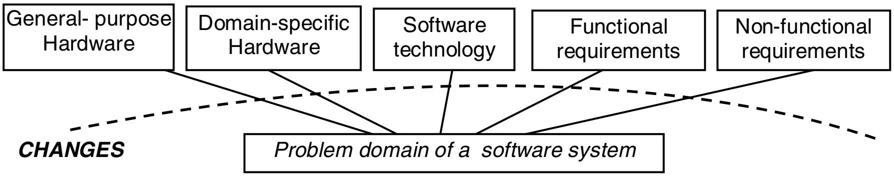

General- purpose Hardware
Domain-specific Hardware
Software technology
Functional requirements
Non-functional requirements
CHANGES
Problem domain of a software system

Fig. 5. Categories of scenarios

4. Evaluate the effect of the scenarios on the architecture elements. The effect is estimated by investigating which services are affected by that scenario. The cost of the modifications associated with each change scenario is predicted by listing the services that are affected and counting the number of changes. The objective is to get a measurement of the quality of the core and domain services with respect to the anticipated variability in functional or non-functional characteristics.

5. Scenario interaction. The result of the effects evaluation represents the input for this step. The activity is to determine which scenarios affect the same service. High interactions of unrelated scenarios indicate a poor separation of concerns. If any of the scenarios affects a core service this is no more part of the RA, but a domain specific.

4 Example

We illustrate the service based development of a cross domain reference architecture with a simple example of embedded software systems. Our approach is applied to design and quality analysis of a measurement controller software architecture.

4.1 Example Description

Our example is abstracted from our experiences with the architecture design of a scientific on-board silicon X-ray array (SIXA) spectrometer control software. SIXA is a multi-element X-ray photon counting spectrometer. It consists of specific domain hardware elements. The SIXA measurement activity consists of observations of time-resolved X-ray spectra for a variety of astronomical objects. Fig. 6 introduces the context view of SIXA considering it a measurement controller. External elements are a command interface and physical devices, i.e. sensors and actuators. The system is programmed and it operates using a set of commands sent from a command interface.

The role of the spectrometer controller is to control the following modes: (a) Energy Spectrum (EGY), which consists of three energy-spectrum observing modes:

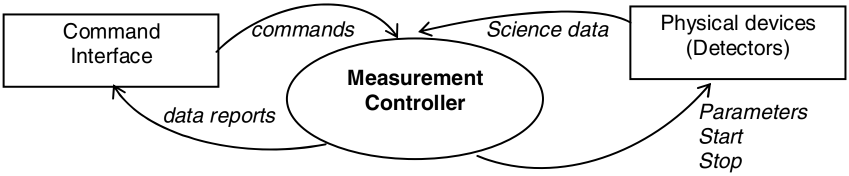


Fig. 6. Context view of the system

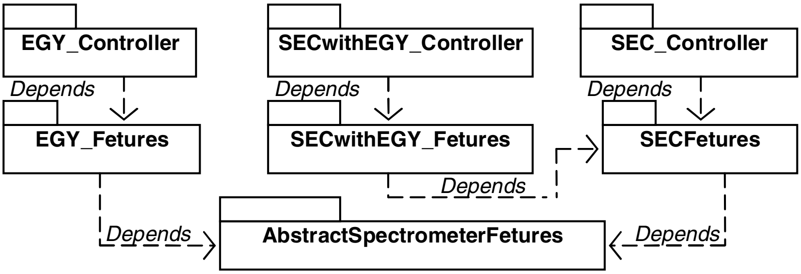


Fig. 7. Mapping features into packages

Energy-Spectrum Mode (ESM), Window Counting Mode (WCM) and Time-Interval Mode (TIM). (b) SEC, which consists of single event characterization observing modes: SEC1, SEC2 and SEC3. Each mode could be controlled individually. A coordinated control of the analogue electronics is required when both measurement modes are on.

The analysis result of the requirements for domain engineering is the features model. This has been structured in packages (Fig. 7). The *with reuse* aspect of reusability is described in the architecture by the abstract features. The abstract features are encapsulated in three main abstract domains MeasurementController, Data Management and DataAcquisition. They are completely reused in all the derived products. The AbstractSpectrometerFeatures package has the highest degree of reusability but also the highest degree of dependability. The abstract features depend on the commonality between EGY and SEC features. A change in the domain of a product is reflected in the degree of reusability of the abstract domain features.

The sets of products that could be derived from the domain specific services during application engineering are: (1) P1 – EGYController, which includes specific services of a standalone control of EGY mode; (2) P2 – SECController, which includes specific services of a standalone control of SEC mode; (3) P3 – SECwith EGYController, which includes specific services of coordinated control.

4.2 Example Architectural Design

The architecture model is documented around multiple views describing conceptual and concrete levels, for each view a static and dynamic perspective being offered. Architecture documentation addresses specific concerns about measurement control, data acquisition control and data management. The views are illustrated with diagrams expressed in UML-RT, a real-time extension of UML. The conceptual level considers a functional decomposition of the architecture into domains. The relationships between architectural elements are based on “pass control-to” and “pass data-to” or “uses”. Functional decomposition is useful for understanding the interactions between entities in the problem space, for understanding the cross-domain perspective, and hence thereafter, the possibilities for creating a system of systems. It includes: (1) Measurement Controller Subsystem (MCS), which has the main role in controlling acquisition and dumping science data. (2) Housekeeping (HK), which forms the reports and sends them to command interface when requested by the command interface subsystem. Also it uses services provided by PMS. (3) Command Interface Subsystem (CIS), which hides the hardware buses' interfaces from the rest of the software. (4) On-board clock (OBC), which maintains an on-board clock used for time-stamping spectra in data files. Also it includes services for timing the start/stop of spectra and targets and other timing related services. (5). Memory Management Subsystem (MMS), which provides services for handling the storages in RAM and EEPROM areas. (6) Parameter Management Subsystem (PMS), which provides services for initiating, changing and reading the on-board parameters in EEPROM. (7) StartUp, which implements the power up and watchdog timer start-up. (8) Communication buffer management (BUFMAN), which provides services for allocating/deallocating transmit buffers. (9) CPU specific services, which provide optimized high speed assembly language services (word copy, interrupt enable/disable). (10) Hardware encapsulation modules, which control specific hardware (analog electronics, watchdog timer). The concrete level considers a more detailed functional description, where the main architectural elements are packages, capsules, ports and protocols. The relationships are association, specialization, generalization, etc. Considering the dynamic aspect state-chart diagrams and message-sequence charts are also part of this description level. Abstract components are modelled based on a recursive control architecture style [16].

Fig. 8 presents the spectrometer controller cross domain architecture design approach. The RA encapsulated in the Measurement «Domain» is composed of three core abstract «Domain»s Measurement Control, DataAcquisitionControl and DataManagement. In each core «Domain» abstract features are collected. The MeasurementControl is responsible for services of starting and stopping the operating mode for data acquisition according to the commands received from the command interface and according to the events generated in other parts of the software. DataAcquisitionControl service collects events (science data) to the spectra data file during observation of a target. This abstract service includes as well as hides data acquisition details. DataManagement abstract services provide interfaces for storing science data, opening/closing/writing the data files, hiding storing details and controlling transmission of the stored data to command interface.

Domain architecture. Domain architecture consists of domain specific services and variability management services. Each of the three core services is specialized in domain specific services. For example, MeasurementControl is specialized in StandAloneControl (SAC) and CoordinatedControl (CC), DataAcquisitionControl (DAC) is specialized in EGY_DataAcquisitionControl (EGY_DAC) and SEC_DataAcquisitionControl (SEC_DAC), Data Management (DM) is specialized in EGY_Data Management (EGY_DM) and SEC_DataManagement (SEC_DM). This architecture includes services associated to variability management.

Product architecture. Product architecture of the sets of products includes rules for product derivation and configuration. Table 1 presents domain specific services and products derivation. Products are horizontally distributed and the domain services are dispersed vertically. Each cell t_{ij} of the table is marked if product P_j uses component C_{i}. For example, two products, P1 and P2, include a SAC service of the measurement control domain.

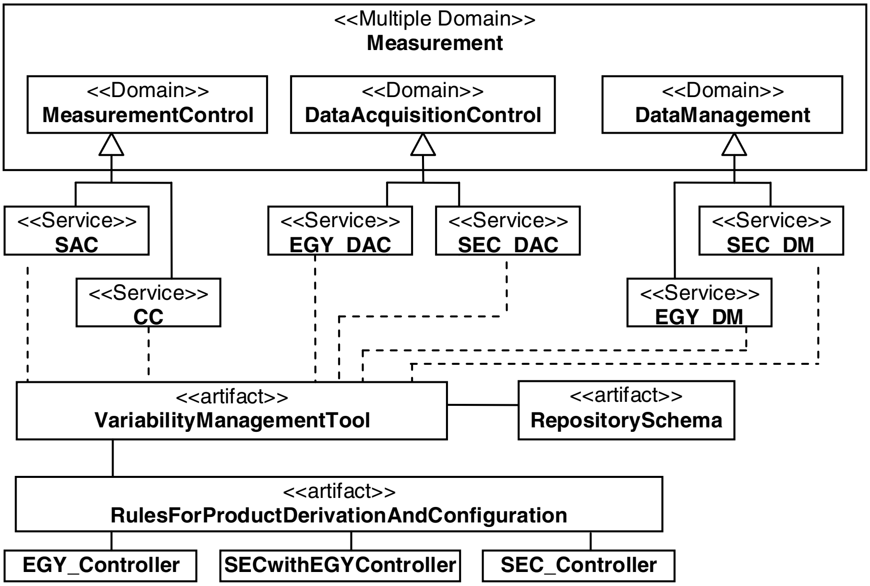


Fig. 8. Spectrometer controller cross domain architecture design approach

Table 1. Domain specific services and products

Domain	Specific Service	Products		
		P1	P2	P3
MeasurementControl	SAC	x	x	
	CC			x
DataAcquisitionControl	EGY_DAC	x		x
	SEC_DAC		x	x
DataManagement	EGY_DM	x		x
	SEC_DM		x	x

4.3 Example Architecture Analysis

We have defined change scenarios for changes in general purpose hardware, domain specific hardware, technology, functionality, non-functional requirements and other changes. For each category we'll exemplify with a scenario in the following. The effect of the scenarios on the service components and the required views of the architecture are discussed.

1. General purpose hardware changes scenario: "Change the central processing unit (CPU)".

Effect on the architecture: CPU specific services provide highly optimized high speed assembly language services (high speed word copy, interrupt enable/disable, etc.) The services are not applicable at the level of description.
Result: Not applicable to the available views.

2. Domain-specific hardware changes scenario: "Add a hard disk for a SEC product".

Effect on the architecture: The SEC_controller and SECwithEGY_controller contain a hard disk for data storage. This scenario requires a lot at the architectural level, most of them related to the DataManagement domain.
Result: Multiple changes in detailed functional decomposition, localized in the SEC_DM specific domain service.

3. Technology changes scenario: "Change the generator polynomial (different from CCITT polynomial) for 16 bit CRC sum of errors handlers"

Effect on the architecture: MMS consists of service functions for managing the storage RAM and EEPROM. It also includes a state for refreshing RAM and the memory error exceptions handlers (double and single bit).
Result: Modification of one component in the conceptual view.

4. Functional requirements changes scenario: "How is the architecture affected when the operation mode is changed?"

Effect on the architecture: The operation modes are part of the variability among domains. These are encapsulated into DataAcquisition and DataFileManagement. The measurement control domain is decoupled from the operation modes of different products.
Result: No change to the RA – abstract concrete or features of measurement control.

5. Non-functional requirements changes: "How is the average SRG-bus speed of 744kbit/sec on reading data from disk, which is time critical, maintained? "

Alternative solutions: (1) Change the hard disk: Use a Fast disk: Optimal disk interleaving factor and storing the data file in sequential sectors on the disk. (2) Send filler blocks to the bus while waiting for the disk – a sufficient number of filler blocks could be reserved in the vector word sent in advance to BIUS. (3) Use a busy bit of SRG-bus. (4) Optimize disk driver – If the disk drive has been changed, the software has to be tuned separately for the new disk.
Result: Not applicable to the available views.

A good architecture design must provide a good localization of changes. Most of the changes required by scenarios were applied to one service component, which indicates a good decoupling of concerns. The most important change was the addition of the hard disk, a variability among domains. This scenario required changes to the domain specific services. By structuring the RA in abstract services, which encapsulate abstract features of the domains and concrete components, which in turn represent specialization of the variable features, the effects of the change scenarios are minimized and localized. Changes did not affect the core services of the cross domain RA, which confirms the stability of the architecture across domains. The results of the analysis depend on the description of the architecture. By using only the decomposition view on the conceptual level the effects of the change scenarios are reduced because not all the details are included. On the concrete level are the views developed with the help of a CASE tool and the effect of change scenarios is more relevant. This is an argument for that the evaluation method should be applied iteratively while the architecture design becomes more detailed. The purpose of the evaluation is to analyze the architecture to identify potential risks by predicting the quality of the products before they have been built. Iterative methods promote analysis at multiple resolutions as a means of minimizing risk at the acceptable levels of time and effort. Areas of high risk are analyzed more profoundly (simulated, modelled or prototyped) than the rest of the architecture. Each iteration determines where to analyze more deeply in the next iteration.

The measurement controller domain also requires run-time qualities such as performance, safety and reliability. These are mandatory root features for the domain. However, variants could include variability in these aspects. These variable features must be considered from the cross domain design perspective in order to minimize the risk that the final software products do not conform to these quality attributes. For architectural evaluation of these aspects several progresses have been identified in the literature that will be analyzed in our future work. It is important to estimate what is the degree of reuse at architectural level and what are the reusable assets when the variability of these run-time qualities is considered.

5 Conclusions and Future Work

We have proposed an approach for software development based on a cross-domain RA. We have provided an integrated strategy with an incremental design and analysis approach based on services, which is more practical, easy to follow and benefits of advantages provided by service engineering. Our approach has been validated by a simple example. The problem dimension for the development of a cross-domain RA increases due to the larger number of requirements and constraints that may be specified by the complex systems domains. Building the features model may require a tool in order to manage the analysis and structuring the abstract features in domains. The cross domain RA contains core services of the domains included in the abstract features package. The appropriate architectural style is provided by a knowledge base through service taxonomy. A domain architecture repository is a solution for variability management of specific services. A decision support tool is proposed for product derivation. The role of this tool is to bound variabilities in order to get a service configuration for a product architecture. In our example we developed a tabular form for the decision model. When the complexity increases a more elaborated tool is required and is a subject of our future research. The analysis strategy based on scenarios has been used to verify architecture against anticipated changes in domain knowledge. From the commonality viewpoint analysis results should consider if scenarios affect core services of the RA. If these core services are affected they should be domain specific.

Future research work is needed to develop systematic ways of bridging requirements taxonomy of each domain to a cross domain RA. However this paper presented the main concepts and justified why these concepts are required. When several domains adopt a service oriented approach it is possible to develop products which address functions from across two or more domains and consume services from multiple domains. Seeking engagement of communities of practice across domains is a more challenging but worthwhile goal. It remains to be seen how relevant international bodies foster such engagement. An essential prerequisite, however, is to have in place a coherent core services for all specific domain that can be used as a point of reference in establishing cross domain exchanges.

Acknowledgements. We wish to thank the anonymous referees for their valuable suggestions and comments. This work has been partially supported by the Romanian scientific research grant CNCSIS IDEI no. 627/2009.

References

1. Kopetz, H.: The ARTEMIS Cross-Domain Architecture for Embedded Systems (2007)
2. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis Feasibility Study, SEI Technical Report CMU/SEI- 90-TR-21 (1990)
3. Niemelä, E., Evesti, A., Savolainen, P.: Modeling Quality Attribute Variability. In: Procs. of the 3rd Int. Conf. ENASE, pp. 169–176. INSTICC Press (2008)
4. Dobrica, L., Niemelä, E.: A survey on software architecture analysis methods. IEEE Trans. on Soft. Eng. Journal 28(7), 638–653 (2002)
5. Dobrica, L., Niemelä, E.: Modeling Variability in the Software Product Line Architecture of Distributed Services. In: Procs of SERP 2007, pp. 269–275 (2007)
6. Szypersky, C.: Component Software Beyond Object-Oriented Programming. Addison-Wesley, Reading (1999)
7. TINA, Service Architecture Specification (1997), http://www.tinac.com
8. Dobrica, L., Niemelä, E.: Adaptive middleware services. In: Procs. IASTED Applied Informatics, Int. Symp. on Soft. Eng., Databases and Applications, pp. 137–142. ACTA Press (2002)
9. Costa, E., Blair, G., Coulson, G.: Experiments with reflexive middleware. In: Procs. ECOOP 1998 Workshop Reflexive Object Oriented Programming and Systems (1998)
10. IEEE Recommended Practice for Architectural descriptions of Software Intensive Systems, Std 1417-2000, (2000)
11. Niemelä, E., Kalaoja, J., Lago, P.: Towards an architectural knowledge base for wireless service engineering. IEEE Trans. on Soft. Eng. 31(5), 361–379 (2005)
12. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley, Reading (1998)
13. Buschmann, F., Meunier, R., Rohnert, H.: Pattern-Oriented Software Architecture:A System of Patterns. John Wiley and Sons, Chichester (1996)
14. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1994)
15. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
16. Selic, B.: Recursive control. In: Martin, R., et al. (eds.) Patterns Languages of Program Design, vol. 3, pp. 147–162. Addison-Wesley, Reading (1998)

Author Index