View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by KITopen

Model-Based Software Reuse

- Proceedings -

ECOOP 2002 Workshop #12
http://research.intershop.com/workshop/ECOOP2002/
or
http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ECOOP2002/

In Association with the
16th European Conference on Object-Oriented Programming

Malaga, Spain -- June 10, 2002
http://www.ecoop.org

Andreas Speck, Intershop Research, Germany
Elke Pulvermüller, Universität Karlsruhe, Germany
Matthias Clauß, Solutionline CSS GmbH, Germany
Ragnhild Van Der Straeten, Vrije Universiteit Brussel, Belgium
Ralf Reussner, DSTC, Monash University, Australia
(Eds.)

Universität Karlsruhe
Fakultät für Informatik / Institut für Programmstrukturen und Datenorganisation (IPD)
Adenauerring 20a
76128 Karlsruhe, Germany

Universität Karlsruhe
Fakultät für Informatik
Interner Bericht (Internal Report)
Technical Report No. 2002-4
September 2002

Preface

This proceedings contains the contributions to the Workshop on Model-based Software Reuse, held in conjunction with the 16th European Conference on Object-Oriented Programming (ECOOP) Malaga, Spain June 10, 2002.
The workshop was motivated by the observation that convenient models are essential to understand the mechanisms of reuse.
Models may help to define the interoperability between components, to detect feature interaction and to increase the traceability. They have the potential to define the essential aspects of the compositionality of the assets (i.e., components, aspects, views, etc.).

11 contributions give an overview about current research directions in the field of model-based software reuse. The topics discussed in the contributions to this workshop embrace reasoning, verification, stability issues as well as support for reuse and modeling. Discussion groups during the workshop have explored the areas „Software Architectures as Composed Components“, „Automated Analysis and Verification“ and „Modelling and Formalising“.
Results from the discussions during the workshop may be found in the ECOOP 2002 workshop reader LNCS 2548.

The web page of the workshop as well as the contributions of this proceedings may be found at URL:
`http://research.intershop.com/workshop/ECOOP2002/`

or at URL (mirror):
`http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ECOOP2002/`

A related workshop about feature interaction has been held at ECOOP 2001; the contributions are published as technical report No. 2001-14 at Universität Karlsruhe, Fakultät für Informatik
URL:
`http://www.info.uni-karlsruhe.de/~pulvermu/workshops/ecoop2001/`

We would like to thank the program committee for their support as well as the authors and participants for their high-quality submissions and engaged contributions during the workshop.

The Workshop Organisers
Andreas Speck, Elke Pulvermüller, Matthias Clauß, Ragnhild Van Der Straeten, Ralf Reussner

Program Committee

Lynne Blair, Lancaster University, UK
Hans de Bruin, Vrije Universiteit Amsterdam, Netherlands
Jim Coplien, University of Manchester, UK
Gerhard Goos, Universität Karlsruhe, Germany
Jilles van Gurp, University of Groningen, Netherlands
Wilhelm Hasselbring, Carl v. Ossietzky Universität, Germany
Heinrich Hußmann, Dresden University of Technology, Germany
Bernd Kraemer, Fern-Universität Hagen, Germany
Kim Mens, Université catholique de Louvain (UCL), Belgium
Silva Robak, University of Zielona Gora, Poland
Heinz W. Schmidt, Monash University, Australia
Judith Stafford, Carnegie-Mellon-University, USA
Liping Zhao, University of Manchester, UK

Organisation

Andreas Speck
Intershop Research, Germany
Email: a.speck@intershop.com

Elke Pulvermüller
IPD, Universität Karlsruhe, Germany
Email: pulvermueller@acm.org
WWW: http://www.info.uni-karlsruhe.de/~pulvermu/

Matthias Clauß
Solutionline CSS GmbH, Germany
Email: matthias.clauss@gmx.de

Ragnhild Van Der Straeten
Vrije Universiteit Brussel
Email: rvdstrae@vub.ac.be

Ralf Reussner
DSTC, Monash University, Australia
Email: reussner@dstc.com

Table of Contents

Reasoning and Verification

ARIFS: Reusing Formal Verification Efforts in a Requirements
Specifications Stage 1
Rebeca P. Díaz Redondo (University of Vigo, Spain) ,
José J. Pazos Arias (University of Vigo, Spain) ,
Ana Fernández Vilas (University of Vigo, Spain) and
Belén Barragáns Martínez (University of Vigo, Spain)

Feature Description Logic: A Knowledge-Based Modeling Approach to
Component Semantics 9
Yu Jia (Chinese Academy of Science, China) and
Yuqing Gu (Chinese Academy of Science, China)

Version-based Approach for Modeling Software Systems 15
Andreas Speck (Intershop Research, Germany),
Silva Robak (University of Zielona Gora, Poland),
Elke Pulvermüller(Universität Karlsruhe, Germany) and
Matthias Clauß (Intershop Research, Germany)

Stability

Stable and Reusable Model-Based Architectures 23
Ahmed Mahdy (University of Nebraska-Lincoln, USA),
Mohamed E. Fayad (University of Nebraska-Lincoln, USA),
Haitham Hamza (University of Nebraska-Lincoln, USA) and
Peeyush Tugnawat (University of Nebraska-Lincoln, USA)

Stable Model-Based Software Reuse 29
Mohamed E. Fayad (University of Nebraska-Lincoln, USA),
Shasha Wu (University of Nebraska-Lincoln, USA) and
Majid Nabavi (University of Nebraska-Lincoln, USA)

Model-based Software Reuse Using Stable Analysis Patterns 41
Haitham Hamza (University of Nebraska-Lincoln, USA) and
Mohamed E. Fayad (University of Nebraska-Lincoln, USA)

Supporting Reuse / Modelling

Modelling Component Libraries for Reuse and Evolution 49
Miro Casanova (Vrije Universiteit Brussel, Belgium) and
Ragnhild Van Der Straeten (Vrije Universiteit Brussel, Belgium)

Modelling With Components - Towards a Unified Component Meta-Model 57
Uwe Rastofer (method park Software AG, Erlangen, Germany and
University of Erlangen-Nuremberg, Germany)

Describing and Reusing Software Design Assets for System Family Engineering 63
Alexander Fried (University Linz, Austria) and
Herbert Prähofer (University Linz, Austria)

A Preliminary Analysis in Mapping UML Use Cases to State Machines 71
Luca Pazzi (University of Modena and Reggio Emilia, Italy)

Coupling MDA and Parlay to increase reuse in telecommunication application
development 77
Babak A. Farshchian (Telenor Research and Development, Norway),
Sune Jakobsson (Telenor Research and Development, Norway) and
Erik Berg (Telenor Research and Development, Norway)

ARIFS: Reusing Formal Verification Efforts in a Requirements Specifications Stage

Rebeca P. Díaz Redondo, José J. Pazos Arias, Ana Fernández Vilas and Belén Barragáns Martínez
Departamento de Enxeñería Telemática. University of Vigo. 36200 Vigo. Spain
{rebeca, jose, avilas, belen}@det.uvigo.es

Abstract

Even though verifying systems during any phase of the development process is a remarkable advantage of using formal techniques in software engineering practice, the great computing resources needed to verify medium-large and large systems entails an efficiency problem in incremental and iterative life cycles, where each iteration implies identifying new requirements, verifying them and, in many cases, modifying the current release of the system to satisfy the new functional specifications. Reusing formal verification efforts is our proposal to reduce formal verification costs in this kind of life cycles, and in this paper we describe how ARIFS tool[1] *(*Approximate *R*etrieval *of I*ncomplete *and F*ormal *S*pecifications*) provides a suitable environment to achieve it. This reusing environment offers an approximate and efficient retrieval, without formal proofs, which enables classifying, retrieving and adapting formal and incomplete requirements specifications and the formal verification results linked to them.*

Keywords: software reuse, component-based requirements engineering, reuse of formal requirement specifications, iterative and incremental software processes.

1. Introduction

Reusing at early stages of the development process —like at the requirements specification stage— is accepted by many within the community as a desirable aim, because of the possibility of increasing the reuse benefits [6]. However, there is little evidence in the literature to suggest that software reuse at requirements specification stage is widely practiced.

Our proposal [4, 3] deals with this concern, offering a methodology to reuse high abstract level components: incomplete specifications —obtained from transient phase of an iterative and incremental requirements specification process—; and their verification results —obtained from a model checking algorithm. This methodology has come to fruition in ARIFS tool, which provides a friendly environment to classify, retrieve and adapt reusable components in the requirements specification phase of the SCTL-MUS methodology (section 2). The main characteristics of the retrieval process can be summarized as follows:

- Although it is based on formal descriptions of the components, these components are not low-level ones (like code). Therefore, the formal description of the functionality of a component is simultaneously index and objective of the retrieval, having a a **content-oriented retrieval**, which allows reusing high abstract level components in a natural way.
- Instead of having an exact retrieval based on formal proofs, we propose an **approximate components retrieval** based on the concept of *unspecification*, inherent to incomplete systems —which are obtained from a transient phase of the iterative and incremental development process—, that is, not everything is true or false, maybe non specified yet.
- Because of efficiency reasons, the retrieval is made in two steps, a **layered retrieval process**: in the first phase, *rough search*, a small set of suitable components is retrieved; and in the second one, *refined search*, these components are ordered depending on the prediction of the efforts needed to adapt each one to the functionality required by the query.

2. Formal basis

In this section we briefly describe the software development process, SCTL-MUS methodology [7], where the reusing environment is going to be included. As figure 1 shows, this methodology joins both a totally formalization of the process, and an incremental and iterative point

[1]Partially supported by PGIDT01PXI32203PR project (Xunta de Galicia)

of view. In the first phase (*Initial goals*), a complete and consistent functional specification of the system is obtained from user's specifications. In every iteration, the user specifies a set of functional requirements which lead to a growth in the system functionality. These requirements are verified in the current model or prototype to check: if the model already satisfies the requirements; if it is not able to provide these functional requirements nor in the current iteration neither in future ones (inconsistency); or, if the system does not satisfy the requirements, but it is able to do it (incompleteness).

Functional requirements are formally specified by using the many-valued logic SCTL [7] (S*imple* C*ausal* T*emporal* L*ogic*), and a generic SCTL requirement follows this pattern:

$$\textbf{Premise} \Rightarrow\otimes \textbf{ Consequence},$$

which establishes a causing condition (premise); a temporal operator determining the applicability of the cause ($\Rightarrow\otimes$); and a condition which is the effect (consequence). Temporal operator $\Rightarrow\otimes \in \{\Rightarrow, \Rightarrow\bigodot, \Rightarrow\bigcirc\}$ —referred to as *simultaneously*, *previously* and *next*— is used to reason about transition successors and predecessors of a given state by determining the order pattern between the state in which premise is formulated, and the states in the scope of the consequence. Apart from causation, SCTL is a six-valued logic, even though it is only possible specifying three different values: possible or *true* (1), non possible or *false* (0) and *unspecified* ($\frac{1}{2}$). This concept of unspecification is specially useful to deal with both incomplete and inconsistent information obtained by requirements capture, because although events will be *true* or *false* at the final stage, in intermediate phases of the specification process it is possible that users do not have enough information about them yet, being *unspecified* in these phases.

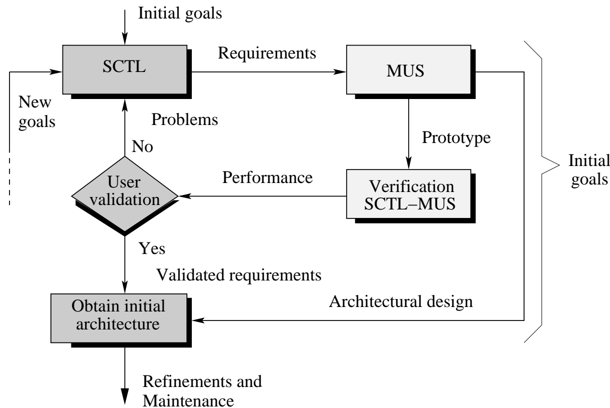


Figure 1. SCTL-MUS methodology

SCTL requirements are synthesized to obtain a model of the system by using MUS [7] (M*odel of* U*nspecified* S*tates*). This state-transition formalism allows prototyping and feedback with users, and supports the consistency checking by using a model checking algorithm. MUS graphs are based on typical labeled-transitions graph, but including another facility: unspecification of its elements.

The degree of satisfaction of an SCTL requirement is based on causal propositions: "*an SCTL requirement is satisfied iff its premise is satisfied and its consequence is satisfied according to its temporal operator*". As SCTL-MUS methodology adds unspecification concept, this degree of satisfaction must not be *false* (nor *true*), just as the Boolean logic. In fact, it must have a degree of satisfaction related to its unspecification (totally or partially unspecified on the MUS model), because it can become *true* or *false* requirement, depending on how it is specified in future. Consequently, this methodology defines six different degrees of satisfaction, $\phi \in \Phi = \{0, \frac{1}{4}, \frac{1}{2}, \widehat{\frac{1}{2}}, \frac{3}{4}, 1\}$, which can be partially ordered according to a *knowledge level* ($\leq_c$) (figure 2) as follows:

- $\{1, \widehat{\frac{1}{2}}, 0\}$ are the highest knowledge levels. We know at the current stage of the system the final degree of satisfaction of the property. The meaning of this verification results are the following ones: 1 or *true* means the requirement is satisfied; 0 or *false* implies the requirement is not satisfied; and $\widehat{\frac{1}{2}}$ or *contradictory* means the requirement cannot become *true* or *false*.

- $\{\frac{1}{4}, \frac{3}{4}\}$ are the middle knowledge levels. Although at the current stage of the system, the property is partially unspecified, we know its satisfaction tendency. That is, in a subsequent stage of specification, the degree of satisfaction will be $\frac{1}{4} \leq_c \phi'$ (respectively $\frac{3}{4} \leq_c \phi'$) for the current value $\frac{1}{4}$ (respectively $\frac{3}{4}$).

- $\{\frac{1}{2}\}$ is the lowest knowledge level. The property is totally unspecified at the current system's stage and we do not know any information about its future behaviour.

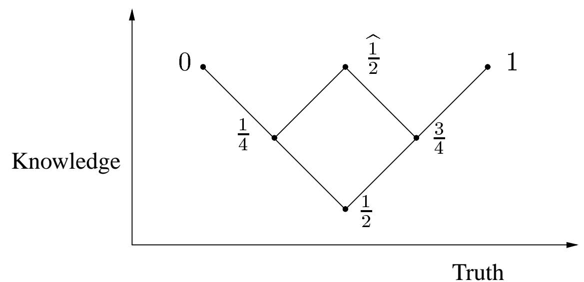


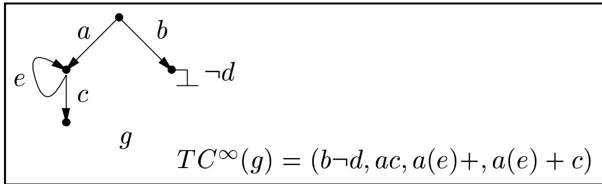
Figure 2. Knowledge and Truth partial orderings among degrees of satisfaction.

In short, the degree of satisfaction of an SCTL requirement varies according to its closeness to the *true* (or *false*) degree of satisfaction —partial order according to *truth level* whose Hasse diagram is in figure 2. According to this *truth ordering*, Φ is a quasi-boolean lattice with the least upper bound $\vee$ operator, the greatest lower bound $\wedge$ operator, and the unary operation $\neg$ defined by horizontal symmetry. The 4-tuple $(\Phi, \vee, \wedge, \neg)$ has the structure of the De Morgan algebra called algebra of MPU [7] (M*iddle* P*oint* U*ncertainty*). As it is shown in figure 2 —according to the *truth level* Hasse diagram— 0 is the smallest truth degree, whereas 1 is the greatest; so $\widehat{\frac{1}{2}}$ and $\frac{1}{2}$ are middle points in this partial ordering. Besides that, $\widehat{\frac{1}{2}}$ is far form the two ends (0 and 1), but it cannot get to them, whereas $\frac{1}{2}$ is near them, and it can get to either, and this is the reason why it is called algebra of Middle Point Uncertainty.

3. Functional relationships among reusable components and SCTL properties

We have defined four partial ordering relations among components to define component hierarchies or lattices to classify and retrieve them properly. As this paper focus on reusing verification efforts, we only describe here one of them because the verification reuse process is based on it.

Function TC^{∞} associates with every MUS graph $g \in \mathbb{G}$ a set $TC^{\infty}(g)$, which is based on complete trace semantics [1]. Main difference with traditional ones are that TC^{∞} takes into account both *true* and *false* events, in order to differentiate *false* events from *unspecified* ones; and it includes infinite traces in $TC^{\infty}(g)$. An example of $TC^{\infty}(g)$ obtaining is shown in figure below.



As it is shown in this example, $TC^{\infty}(g)$ provides all possible evolutions of the system, that is, it can evolve from initial state to a final one, where event d is non possible (denoted by $\neg d$), through event b, from initial state to a final one through events a and c; from initial state to a final one through events a, a number non determined of events e and, finally, event c; and from initial state through events a and a number infinite of events e. Hence $TC^{\infty}(g)$ provides a good approximation of graph's functionality.

$TC^{\infty}(g)$ constitutes the observable behaviour of g according to TC^{∞}-criteria and it allows defining the equivalence relation $=^{\infty}_{\mathrm{TC}} \in \mathbb{G} \times \mathbb{G}$ given by $g =^{\infty}_{\mathrm{TC}} g' \Leftrightarrow TC^{\infty}(g) = TC^{\infty}(g')$, and the preorder $\sqsubseteq^{\infty}_{\mathrm{TC}} \in \mathbb{G} \times \mathbb{G}$ by $g \sqsubseteq^{\infty}_{\mathrm{TC}} g' \Leftrightarrow TC^{\infty}(g) \sqsubseteq TC^{\infty}(g')$. $\sqsubseteq^{\infty}_{\mathrm{TC}}$ provides a partial order between equivalence classes, that is, graph sets indistinguishable using TC^{∞}-observations, so $(\mathbb{G}, \sqsubseteq^{\infty}_{\mathrm{TC}})$ is a *partially ordered set*, or *poset*. A subset $G_1 \in \mathbb{G}$ is called a *chain* if every two graphs in G_1 are TC^{∞}-related.

Every reusable component (C) gathers both its functional specification, which is expressed by the set of SCTL requirements and modeled by the temporal evolution MUS graph, and an interface or *profile* information, which is automatically obtained from its functional characteristics to classify and retrieve it from the repository. Besides this, every reusable component stores verification information, that is, the set of properties which had been verified on the MUS graph and their verification results (section 4).

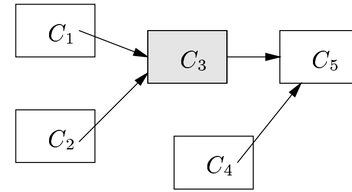


Figure 3. Chain of reusable components

Each reusable component (C) is classified in the repository after finding its *correct place* in the lattice defined by TC^{∞} relation. That is, it is necessary looking for those components TC^{∞}-related to C[2] such as C is TC^{∞}-included on them, and those components TC^{∞}-related to C such as they are TC^{∞}-included on C. In order to eliminate superfluous reusable components connections, anti-symmetric property is applied (figure 3[3]).

TC^{∞} can be also applied to SCTL properties in order to obtain the sequence of events which is specified by functional requirements. For instance, the requirement

$$R_1 \equiv (((d \wedge b) \Rightarrow \bigcirc c) \wedge (true \Rightarrow a)$$

expresses that after being possible events d and b, it must be possible event c, and in the same state where events d and b are possible, event a must be also possible, so, $TC^{\infty}(R_1) = (a, bc, dc)$. These results allow classifying SCTL properties in functional equivalent classes which express the same behaviour, and, consequently, share the same verification information. For instance, the requirement

$$R_2 \equiv ((a \Rightarrow (b \Rightarrow \bigcirc c)) \wedge (d \Rightarrow \bigcirc c)$$

shares the same TC^{∞} information than R_1: $TC^{\infty}(R_2) = (a, bc, dc)$.

[2] Two components C and C' are TC^{∞}-related ($C \sqsubseteq^{\infty}_{\mathrm{TC}} C'$ or $C' \sqsubseteq^{\infty}_{\mathrm{TC}} C$) iff their MUS graphs g and g' are TC^{∞}-related ($g \sqsubseteq^{\infty}_{\mathrm{TC}} g'$ or $g' \sqsubseteq^{\infty}_{\mathrm{TC}} g$).

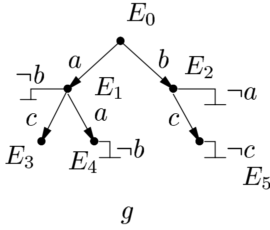
[3] In this figure, $C_i \sqsubseteq^{\infty}_{\mathrm{TC}} C_j$ is represented by $C_i \rightarrow C_j$

4. Reusable verification information

In order to store interesting verification information linked to each reusable component, we define four properties which summarize the degrees of satisfaction of an SCTL property R in the states of a MUS graph g:

- $\exists \lozenge R$ expresses that "*some trace of the system satisfies eventually* R" and its degree of satisfaction is denoted $\models (\exists \lozenge R, g)$.
- $\exists \square R$ expresses that "*some trace of the system satisfies invariantly* R" and its degree of satisfaction is denoted $\models (\exists \square R, g)$.
- $\forall \lozenge R$ expresses that "*every trace of the system satisfies eventually* R" and its degree of satisfaction is denoted $\models (\forall \lozenge R, g)$.
- $\forall \square R$ expresses that "*every trace of the system satisfies invariantly* R" and its degree of satisfaction is denoted $\models (\forall \square R, g)$.

To sum up, for each property verified in the MUS graph, we will have four derived properties whose degrees of satisfaction make up the degree of satisfaction of an SCTL property R in a MUS graph g, denoted $\models (R, g) = (\models (\exists \lozenge R, g), \models (\forall \lozenge R, g), \models (\exists \square R, g), \models (\forall \square R, g))$. This verification information is stored in the reusable component whose MUS graph is g, ready to be recovered whenever it is necessary.



(a) g

$\models (R, E_j)$	a	b
$\models (R, E_0) = 1$	1	1
$\models (R, E_1) = 0$	1	0
$\models (R, E_2) = \widehat{\frac{1}{2}}$	0	$\frac{1}{2}$
$\models (R, E_3) = \frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\models (R, E_4) = \frac{1}{4}$	$\frac{1}{2}$	0
$\models (R, E_5) = \frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$

(b) $\models (R, E_j) \quad \forall E_j \in g$

Figure 4. Degrees of satisfaction of R in every state of g

In order to obtain the degree of satisfaction of an SCTL property R in a MUS graph g, it is necessary studying the traces of states of the graph, that is, those sequences of states through which the system can evolve. For instance, in figure 4(a), the traces of states of g are the following ones:

$$(\{E_0, E_1, E_3\}, \{E_0, E_1, E_4\}, \{E_0, E_2, E_5\})$$

As each component of $\models (R, g)$ is the result of analyzing the degrees of satisfaction of R in each trace, having the degrees of satisfaction of R in every state of g is essential. For instance, in figure 4(b), these verification results are shown for property $R \equiv (a \Rightarrow b)$, which expresses that in the same state where event a is possible, event b must be also possible.

Starting from the degrees of satisfaction of the property in every state of the prototype and having the different traces of g, we can obtain the following information:

- $\models (\lozenge R, E(\pi_i)) = \models (R, E_i^1) \vee \ldots \vee \models (R, E_i^n)$ where $E(\pi_i)$ is one of the traces of states of the system, and $\{E_i^j\}_{j=1}^n$ its set of states. This information expresses if "$E(\pi_i)$ *satisfies eventually* R". The logic connective $\vee$ is the least upper bound operator in the *truth level* partial ordering of the figure 2.
- $\models (\square R, E(\pi_i)) = \models (R, E_i^1) \wedge \ldots \wedge \models (R, E_i^n)$ where $E(\pi_i)$ is one of the traces of states of the system, and $\{E_i^j\}_{j=1}^n$ its set of states. In this case, this information expresses if "$E(\pi_i)$ *satisfies invariantly* R". The logic connective $\wedge$ is the greatest lower bound operator in the *truth level* partial ordering of the figure 2.

For instance, we can obtain these two degrees of satisfaction for the trace of states $E(\pi_1) = \{E_0, E_2, E_5\}$ of the graph in figure 4(a) for the property $R \equiv (a \Rightarrow b)$. As

$$\begin{aligned} \models (\lozenge R, E(\pi_1)) &= \models (R, E_0) \vee \models (R, E_2) \vee \models (R, E_5) \\ &= 1 \vee \widehat{\frac{1}{2}} \vee \frac{1}{2} = 1 \end{aligned}$$

we known that R is satisfied at least for one of the states of the trace, and because of the following result

$$\begin{aligned} \models (\square R, E(\pi_1)) &= \models (R, E_0) \wedge \models (R, E_2) \wedge \models (R, E_5) \\ &= 1 \wedge \widehat{\frac{1}{2}} \wedge \frac{1}{2} = \frac{1}{4} \end{aligned}$$

we known that R is partially specified on $E(\pi_1)$, but, regardless of future iterations, R will not be satisfied in every state on the trace.

Finally, knowing $\models (\square R, E(\pi_i))$ and $\models (\lozenge R, E(\pi_i))$ in every trace of states of the graph, we can conclude the degree of satisfaction of R in g as follows:

$$\models (\exists \lozenge R, g) = \models (\lozenge R, E(\pi_1)) \vee \ldots \vee \models (\lozenge R, E(\pi_m))$$

$$\models (\forall \lozenge R, g) = \models (\lozenge R, E(\pi_1)) \wedge \ldots \wedge \models (\lozenge R, E(\pi_m))$$

$$\models (\exists\, \Box R, g) \;=\; \models (\Box R, E(\pi_1)) \vee \ldots \vee \models (\Box R, E(\pi_m))$$

$$\models (\forall\, \Box R, g) \;=\; \models (\Box R, E(\pi_1)) \wedge \ldots \wedge \models (\Box R, E(\pi_m))$$

For instance, in the graph of figure 4(a), where the degrees of satisfaction of $R \equiv (a \Rightarrow b)$ in every state are reflected in the table 4(b), we can deduce that $\models (R, g) = (1, 1, \frac{1}{4}, 0)$. This verification information entails the following conclusions: because of $\models (\forall\, \Diamond R, g) = 1$, every trace of g satisfies eventually R, that is, R is a *liveness property* in g; since $\models (\exists\, \Box R, g) = \frac{1}{4}$, R is partially specified in g, but regardless of future iterations, any trace of g does not satisfy invariantly R, that is, R is *not a safety property* in g.

5. How to reuse verification efforts?

The defined classification scheme (section 3) implies that, for instance in figure 3, C_1 and C_2 are *functional parts* of C_3, being the last one a *functional part* of C_5. Main question in this situation is: how to know the degree of satisfaction of an SCTL property R in C_3, if we know the degrees of satisfaction of R in C_1, C_2 and C_5?. In this section we resolve this question after studying some mathematical aspects related to the ordering of degrees of satisfaction, and by applying these results to the proposed practical environment.

5.1. Mathematical aspects

Let $\sqsubseteq_{\mathrm{e}}$ be a simulation relation between two states E_1, and E_2, denoted by $E_1 \sqsubseteq_{\mathrm{e}} E_2$, satisfying:

$$\forall E_1{}' \mid E_1 \overset{\omega}{\to} E_1{}' \text{ then } \exists\, E_2{}' \mid E_2 \overset{\omega}{\to} E_2{}' \text{ and } E_1{}' \sqsubseteq_{\mathrm{e}} E_2{}'$$

$$\text{and if } E_1 \overset{\omega}{\not\to} \text{ then } E_2 \overset{\omega}{\not\to}$$

where $E_1 \overset{\omega}{\to} E_1{}'$ means the system can evolve from state E_1 to E_2 through event ω, that is it has been characterized as possible or *true* in this state; and $E_1 \overset{\omega}{\not\to}$ implies the system cannot evolve from the state E_1 through event ω, that is, this event has been characterized as non possible or *false* in this state.

Let g and $g\,'$ two MUS graphs, then $g\,'$ simulates g, denoted $g \sqsubseteq_{\mathrm{e}} g\,'$, iff $E_0 \sqsubseteq_{\mathrm{e}} E_0\,'$, where E_0 is the initial state of g and $E_0\,'$ the initial state of $g\,'$.

Property 1. *Let E and $E\,'$ be two states satisfying $E \sqsubseteq_{\mathrm{e}} E\,'$, then $\models (R, E) \leq_{\mathrm{c}} \models (R, E\,')$. That is, the degree of satisfaction of a property R in E has a lower* knowledge level *than its degree of satisfaction in $E\,'$.* [4]

As consequence of property 1:

- it is possible to obtain verification information about the degree of satisfaction of one SCTL property R in a MUS graph g, $\models (R, g)$, knowing the degree of satisfaction of R in $g\,'$, $\models (R, g\,')$, where $g \sqsubseteq_{\mathrm{e}} g\,'$. The verification information which can be obtained is shown in tables 2.(a) and 2.(b).
- and it is possible to obtain verification information about the degree of satisfaction of R in $g\,'$, $\models (R, g\,')$, knowing the degree of satisfaction of R in g, $\models (R, g)$, where $g \sqsubseteq_{\mathrm{e}} g\,'$. The verification information which can be obtained is shown in tables 1.(a), 1.(b), 1.(c) and 1.(d).

Apart from the verification results shown in these tables, we can deduce more verification information taking into account that the initial state of a MUS graph is contained in every single trace of states of the graph:

Property 2. *Let R be an SCTL property which is satisfied in the initial state of a MUS graph g, that is, $\models (R, E_0|_g) = 1$, then we know that $\models (\exists\, \Diamond R, g) = 1$, and we can deduce that $\models (\forall\, \Diamond R, g) = 1$, and $\models (\forall\, \Diamond R, g\,') = 1,\ \forall g\,' \mid g \sqsubseteq_{\mathrm{e}} g\,'$.*

This property gives more information than table 1.(a), because this table shows that if a MUS graph satisfies $\models (\exists\, \Diamond R, g) = 1$, we can only deduce that $\models (\exists\, \Diamond R, g\,') = 1$ in every MUS graph $g\,' \mid g \sqsubseteq_{\mathrm{e}} g\,'$. Taking into account property 2, if R is *true* in the initial state of g, we can also conclude that $\models (\forall\, \Diamond R, g\,') = 1,\ \forall g\,' \mid g \sqsubseteq_{\mathrm{e}} g\,'$.

5.2. Practical aspects

The main problem of the solution proposed in the previous section is comparing MUS graphs using the $\sqsubseteq_{\mathrm{e}}$ relationship in an efficient way. The following property offers a solution to this problem:

Property 3. *$\sqsubseteq_{\mathrm{e}}$ defines a partial order between MUS graphs, but, for deterministic graphs, it can be demonstrate that $\sqsubseteq_{\mathrm{e}}$ is totally equivalent to $\sqsubseteq^{\infty}_{\mathrm{TC}}$.*

because comparing components according to TC^{∞} relationship is much more efficient and equal effective (property 3) than comparing components according to $\sqsubseteq_{\mathrm{e}}$.

So, box labeled as *Verification SCTL-MUS* in figure 1, where a property R_i is formally verified on a MUS prototype g, may be replaced by the following steps, which are shown in figure 5:

1. Obtain the $TC^{\infty}(g)$ information to be able to locate in the repository the reusable components which are TC^{∞}-related to g.

[4] This property's demonstration is based on the structure of an SCTL requirement.

<table>
<tr><td>$\models (\exists \Diamond R, g) = 0$</td><td>$\models (\forall \Box R, g') = 0$</td></tr>
<tr><td>$\models (\exists \Diamond R, g) = 1$</td><td>$\models (\exists \Diamond R, g') = 1$</td></tr>
<tr><td>$\models (\exists \Diamond R, g) = \widehat{\frac{1}{2}}$</td><td rowspan="2">$\models (\forall \Box R, g') \geq_c \frac{1}{4}$</td></tr>
<tr><td>$\models (\exists \Diamond R, g) = \frac{1}{4}$</td></tr>
</table>

(a) Results obtained from $\models (\exists \Diamond R, g)$

<table>
<tr><td>$\models (\forall \Diamond R, g) = 0$</td><td>$\models (\forall \Box R, g') = 0$</td></tr>
<tr><td>$\models (\forall \Diamond R, g) = 1$</td><td>$\models (\exists \Diamond R, g') = 1$</td></tr>
<tr><td>$\models (\forall \Diamond R, g) = \widehat{\frac{1}{2}}$</td><td rowspan="2">$\models (\exists \Diamond R, g') \geq_c \frac{3}{4}$</td></tr>
<tr><td>$\models (\forall \Diamond R, g) = \frac{3}{4}$</td></tr>
</table>

(b) Results obtained from $\models (\forall \Diamond R, g)$

<table>
<tr><td>$\models (\exists \Box R, g) = 0$</td><td>$\models (\forall \Box R, g') = 0$</td></tr>
<tr><td>$\models (\exists \Box R, g) = 1$</td><td>$\models (\exists \Diamond R, g') = 1$</td></tr>
<tr><td>$\models (\exists \Box R, g) = \widehat{\frac{1}{2}}$</td><td rowspan="2">$\models (\forall \Box R, g') \geq_c \frac{1}{4}$</td></tr>
<tr><td>$\models (\exists \Box R, g) = \frac{1}{4}$</td></tr>
</table>

(c) Results obtained from $\models (\exists \Box R, g)$

<table>
<tr><td>$\models (\forall \Box R, g) = 0$</td><td>$\models (\forall \Box R, g') = 0$</td></tr>
<tr><td>$\models (\forall \Box R, g) = 1$</td><td>$\models (\exists \Diamond R, g') = 1$</td></tr>
<tr><td>$\models (\forall \Box R, g) = \widehat{\frac{1}{2}}$</td><td rowspan="2">$\models (\exists \Diamond R, g') \geq_c \frac{3}{4}$</td></tr>
<tr><td>$\models (\forall \Box R, g) = \frac{3}{4}$</td></tr>
</table>

(d) Results obtained from $\models (\forall \Box R, g)$

Table 1. Reuse of verification results obtained from $\models (R, g)$

<table>
<tr><td rowspan="4">$\models (\forall \Box R, g') \leq_c 1$</td><td>$\models (\exists \Diamond R, g) \leq_c 1$</td></tr>
<tr><td>$\models (\forall \Diamond R, g) \leq_c 1$</td></tr>
<tr><td>$\models (\exists \Box R, g) \leq_c 1$</td></tr>
<tr><td>$\models (\forall \Box R, g) \leq_c 1$</td></tr>
<tr><td rowspan="4">$\models (\forall \Box R, g') = \widehat{\frac{1}{2}}$ or
$\models (\forall \Box R, g') = \frac{1}{4}$</td><td>$\models (\exists \Diamond R, g) \in \Phi - \{0\}$</td></tr>
<tr><td>$\models (\forall \Diamond R, g) \in \Phi - \{0\}$</td></tr>
<tr><td>$\models (\exists \Box R, g) \in \Phi - \{0\}$</td></tr>
<tr><td>$\models (\forall \Box R, g) \in \Phi - \{0\}$</td></tr>
</table>

(a) Results obtained from $\models (\forall \Box R, g')$

<table>
<tr><td rowspan="4">$\models (\exists \Diamond R, g') \leq_c 0$</td><td>$\models (\exists \Diamond R, g) \leq_c 0$</td></tr>
<tr><td>$\models (\forall \Diamond R, g) \leq_c 0$</td></tr>
<tr><td>$\models (\exists \Box R, g) \leq_c 0$</td></tr>
<tr><td>$\models (\forall \Box R, g) \leq_c 0$</td></tr>
<tr><td rowspan="4">$\models (\exists \Diamond R, g') = \widehat{\frac{1}{2}}$ or
$\models (\exists \Diamond R, g') = \frac{3}{4}$</td><td>$\models (\exists \Diamond R, g) \in \Phi - \{1\}$</td></tr>
<tr><td>$\models (\forall \Diamond R, g) \in \Phi - \{1\}$</td></tr>
<tr><td>$\models (\exists \Box R, g) \in \Phi - \{1\}$</td></tr>
<tr><td>$\models (\forall \Box R, g) \in \Phi - \{1\}$</td></tr>
</table>

(b) Results obtained from $\models (\exists \Diamond R, g')$

Table 2. Reuse of verification results obtained from $\models (R, g')$

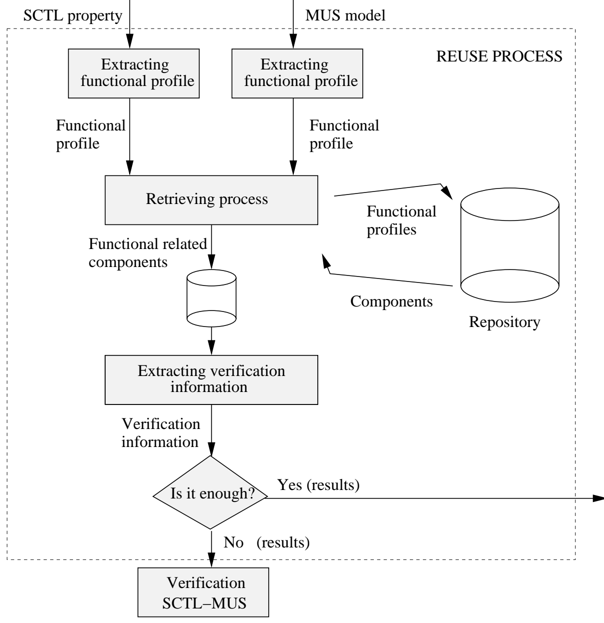


Figure 5. Verification information reuse

2. Obtain the $TC^{\infty}(R_i)$ information in order to locate those functional requirements which are functionally equivalents to each R_i.
3. Retrieve those components whose classification distance to g is as little as possible and where verification information about functionally equivalent to R_i properties are stored.
4. Extract verification information about $\models (R_i, g)$ from the recovered components.
5. If the verification information obtained is not enough to know the required verification results, it is necessary to run the model checking algorithm, but this execution can be reduced depending on the available verification information.

6. Example of application

In this section, an example of verification reuse is outlined. The situation is as follows: we want to know the degree of satisfaction of the property $R \equiv (c \Rightarrow a)$ in the prototype g (figure 6).

Following the steps detailed in figure 5, we firstly obtain $TC^{\infty}(g)$ in order to find in the repository those reusable components which are functionally related to g, and we obtain $TC^{\infty}(R)$ in order to locate those requirements which are functionally equivalents to R. Applying these functional profiles ($TC^{\infty}(g)$ and $TC^{\infty}(R)$), we are able to recover from the repository the more suitable reusable components

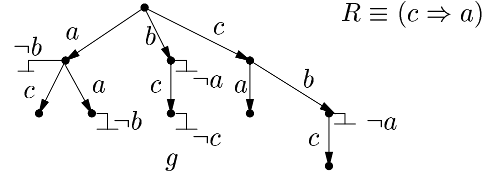
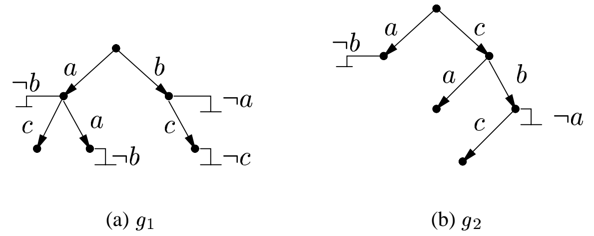


Figure 6. MUS graph g and requirement R

—for this example, figures 7(a) and 7(b) show the retrieved components from a given repository. Both of them are functional parts of g because of $g_1 \sqsubseteq^{\infty}_{\mathrm{TC}} g$ and $g_2 \sqsubseteq^{\infty}_{\mathrm{TC}} g$, and they also have verification information about a functional requirement equivalent to R (figure 8).



(a) g_1 (b) g_2

Figure 7. MUS graphs related to g

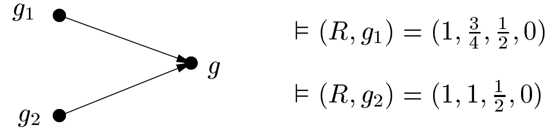


Figure 8. TC^{∞}-ordering among g_1, g_2 and g.

In order to obtain the degree of satisfaction of R in g, that is $\models (R, g)$, we study the verification information retrieved from g_1 and g_2. Starting from $\models (R, g_1) = (1, \frac{3}{4}, \frac{1}{2}, 0)$, the following verification information can be deduced:

– As $\models (\exists \Diamond R, g_1) = 1$ and because of the information stored in table 1.(a), $\models (\exists \Diamond R, g) = 1$ is obtained.

– Neither $\models (\forall \Diamond R, g_1) = \frac{3}{4}$ (table 1.(b)) nor $\models (\exists \Box R, g_1) = \frac{1}{2}$ (table 1.(c)) offer useful information about $\models (R, g)$.

– And, finally, $\models (\forall \Box R, g_1) = 0$ implies $\models (\forall \Box R, g) = 0$ (table 1.(d)).

As we do not still have enough information about $\models (R, g)$, $\models (R, g_2) = (1, 1, \frac{1}{2}, 0)$ is studied:

– As $\models (R, E_0|_{g_2}) = 1$, then $\models (\forall \Diamond R, g) = 1$ (property 2).

– And $\models (\exists \Box R, g_2) = \frac{1}{2}$ don not allow us to deduce any information about $\models (\exists \Box R, g)$ (table 1.(c)).

To sum up, we have obtained that the degree of satisfaction of R in g is $\models (R, g) = (1, 1, \phi, 0)$, where ϕ can be any degree of satisfaction of Φ. So, R is a *liveness property* in g, that is, any trace of the model satisfies eventually R ($\models (\forall \Diamond R, g) = 1$); and R is *not a safety property* in g, that is, there are at least one state which does not satisfy R ($\models (\forall \Box R, g) = 0$). These conclusions have been obtained without running the model checking algorithm by using the verification information about R in two functional parts of g.

7. Summary and future work

The work introduced in this paper focuses on reusing verification information linked to incomplete systems in a totally formalized, incremental and iterative software development process with the aim of minimizing its formal verification costs. That is, we propose reusing verification information obtained from the requirements specification stage, as difference to other approaches like [5] where although reusing verification results is also proposed, they are less formalized proofs (simulation proofs) over code components (algorithms).

After studying different relationships among incomplete specifications, we have identified a criteria to compare functional specifications which is based on trace semantics and takes advance of unspecification inherent to incomplete models. Applying this criteria, we build a lattice of reusable components which allows avoiding formal verification tasks in the retrieval process. This entails a fast retrieval which is accurate enough to reuse verification information and it makes a difference between other proposals [2, 8, 9] where specification matching is based on theorem proving. We have also identified what verification information can be reused and, consequently, how to reduce formal verification tasks.

In order to continue this proposal, we are working on reusing verification results of *functional similar* properties with the given one; and with the possibility of dividing the given property into several properties. Both lines share the same goal: increasing the possibility of finding interesting verification information in the repository.

References

[1] *Handbook of Process Algebra*, chapter The Linerar Time - Branching Time Spectrum I: The Semantics of Concrete, Sequential Processes. Elsevier Science.

[2] B. H. C. Cheng and J. J. Jeng. Reusing Analogous Components. *IEEE Trans. on Knowledge and Data Engineering*, 9(2), Mar. 1997.

[3] R. P. Díaz-Redondo. *Reutilización de Requisitos Funcionales de Sistemas Distribuidos utilizando Técnicas de Descripción Formal*. PhD thesis, Departamento de Enxeñería Telemática - Universidade de Vigo, 2002.

[4] R. P. Díaz-Redondo and J. J. Pazos-Arias. Reuse of Verification Efforts and Incomplete Specifications in a Formalized, Iterative and Incremental Software Process. In *Proceedings of International Conference on Software Engineering (ICSE) Doctoral Symposium*, 2001.

[5] I. Keidar, R. Khazan, N. Lynch, and A. Shvartsman. An Inheritance-Based Technique for Building Simulation Proofs Incrementally. In *22nd International Conference on Software Engineering (ICSE)*, pages 478–487, 2000.

[6] W. Lam, J. A. McDermid, and A. J. Vickers. Ten Steps Towards Systematic Requirements Reuse. *Requirements Engineering*, 2:102–113, 1997. Springer Verlag.

[7] J. J. Pazos-Arias and J. García-Duque. SCTL-MUS: A Formal Methodology for Software Development of Distributed Systems. A Case Study. *Formal Aspects of Computing*, 13:50–91, 2001.

[8] J. Schumann and Fischer. NORA/HAMMR: Making Deduction-Based Software Component Retrieval Practical. In *Proceedings of the 12th International Conference Automated Software Engineering*, 1997.

[9] A. M. Zaremski and J. M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, Oct. 1997.

Feature Description Logic: A Knowledge-Based Modeling Approach to Component Semantics

Yu Jia[†] Yuqing Gu[‡]
Institute of Software, Chinese Academy of Science
Beijing(100080), China.
Email: [†]jia_yu@263.net [‡] guyq@sinosoftgroup.com

Abstract: *In this paper a knowledge-based modeling approach is suggested to represent and reason the software component semantics. The core of this approach is a logical tool called Feature Description Logic ($\mathcal{FDL}$), where the Description Logics (DLs) are applied to the Feature-Oriented Modeling technique, as well as the a set of Feature Spaces expressed in DLs is taken as the formal semantics to describe the component properties. The goal of this paper is to find a way to reason about the properties of component -based system from the properties of individual component.*

Keywords: *Knowledge-Based Modeling, Description Logic, Component Semantics, Feature-Oriented.*

1. Introduction

Software Reuse is the process of implementing or updating software systems using existing software assets, which offers a great deal of potential in terms of software productivity and software quality with a long-term decrease of costs for software development and maintenance [5, 6]. In a broad sense, the reusable assets encompass all the resources that are used and produced during the development of software and have potential value to reusers, such as features, subsystems, components, aspects, etc.

Modeling assets is a critical issue in Software Reuse. To date there exist many methods to model assets, however most of them cannot provide a principle of analyzing the compositional properties of asset assembly [11]. This is an inevitable problem to be solved in Software Reuse, because the reuse philosophy is gradually transferring from inheritance as in object-oriented technique to composition as in Component-Based Development (CBD) [2].

In this paper, a knowledge-based modeling approach with logical tool is suggested to represent and reason the properties of one kind of most important reusable assets - software components. Here we particularly regard the component properties as the component *semantics* that refers to the meaning and usage of the components in the specific business domain. In addition, our approach originates from and improves the Feature-Oriented Modeling technique [4], which is taken as an engineering method, making use of its practicability and success in Domain Engineering.

This paper is organized as follows: Section 2 analyzes the necessity of using knowledge-based approach in assets modeling; the aim is to present a general reason why we adopt the knowledge-based tool such as DLs in our approach. Then in section 3 a feature-oriented component semantic model is provided and is further represented by the Feature Description Logic ($\mathcal{FDL}$) in section 4. Finally the basic reasoning tasks are presented to indicate the applicable power of the logic tool in composing reusable components.

2. The Necessity of Knowledge-Based Reusable Assets Modeling

What is the essence of the Software Reuse? Obviously, the answer is mostly dependent upon the profound awareness of reusable assets. We should not be bewildered by the various exterior forms of assets, Whether they are the features, subsystems, components or aspects, the reusable assets are in fact the carriers of mankind intelligence. Undoubtedly we believe: Software reuse is reuse of knowledge, not only reuse of software assets themselves.

The Software Reuse Initiative of U. S. Department of Defense (DoD) is one of the supporters of this opinion. One of their literatures writes “The ability of an engineer to reuse software is a direct consequence of the engineer's knowledge of an asset's functions and other characteristics, and the engineer's knowledge of how that asset `fits' into a newly developed application and its architecture” [6]. It is reasonable to think that the research on the *mental model* of creating, understanding and reusing assets will greatly benefit the upcoming era of

Software Reuse.

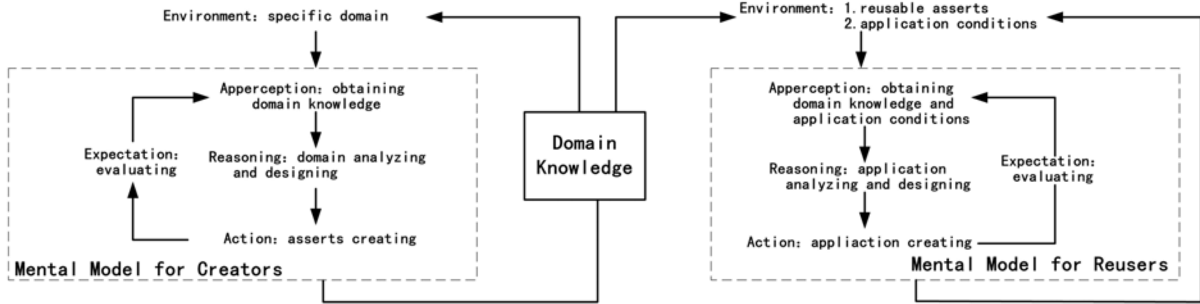


Figure 1. Domain Knowledge: The Connection between the Creators and Reusers

Analyzing from the relationship between the mankind behavior in mind and the environment of real world, the function and role of mental model for both asset creators and reusers can be achieved. As depicted in figure 1, the creators first apperceive the specific domain, which covers a limited range of problems in the environment, to obtain and represent business knowledge by some means or other. Then they make decisions and design operations as the result of reasoning to the knowledge. Finally the decisions and operations are coded as a family of assets (viz. the solution domain) to act on the problem domain, as well as a kind of expectation is preset to evaluate actual results of these assets to affect the environment and feed back the results to the initial stage. On the other hand, the reusers apperceive, obtain and produce the application related knowledge from the existing assets and application conditions. The solution of application system is designed based on reasoning to the knowledge; and the system building is under the guide of the solution and from the existing assets.

Above analysis indicates that in the Software Reuse process there are two mental models for reusable assets: the mental model of creators when assets are originally built; and the mental model of reusers for understanding and using the existing assets. Ideally, if expressed in knowledge-based approach, the same problem should have the equivalent intension for the two type mental models. Unfortunately, it is not always the case. The insufficient and non-rigorous modeling methods frequently lead to misunderstanding between the creators and reusers. The lack of reasoning tools makes the reusers impossible to check the reusability and consistency of the assembly composed from the existing assets. In the rest of paper a knowledge-base approach is suggested to represent the mental model of software components; and a logic tools is provided to check the properties of the component assembly. Although the components are taken as the research objects, what is discussed in this paper is universal to all kinds of reusable assets based on composition principle.

3. The Feature-Oriented Component Semantics Model

A *component* is an identifiable software unit in an explicit context with contractually specified semantic interfaces that are reasonable in a domain as well as syntactic interfaces that are supported by component frameworks [2]. The component *semantics* is the meaning and use of components in perspective of domain-specific service in the real world [1,7].

Features are the constructing units of component semantics, as well as the ontology of domain knowledge in real world. A *Feature Space* is the architecture of component semantics formed by features and feature relations.

The feature-oriented component semantics model is defined as follow [7]:

$$CSemantics = (\Omega_{dom}, \Omega_{def}, \Omega_{con})$$

Where,

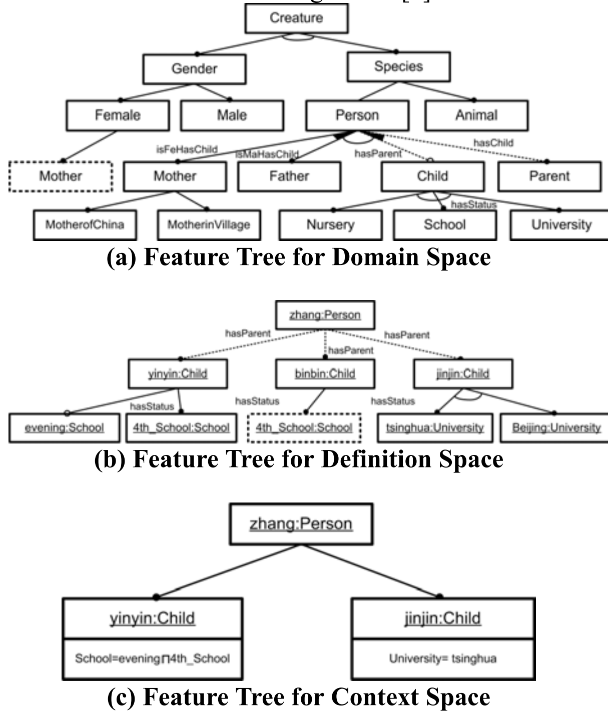
The *Domain Space* Ω_{dom} is a sound and complete Feature Space that expresses the knowledge for a specific domain. Domain Space is the product of Domain Engineering, which represents the commonality and variability in Feature Space to specify the Domain-Specific Software Architecture (DSSA) of the software families.

The *Definition Space* Ω_{def} is an instance set of Ω_{dom} that expresses the service provided by a component. Definition Space specifies the semantics for an individual component. The feature-oriented Method is a kind of descriptive semantics, which declares the intension of the functional and extra-functional properties of a component without concerning the implementation and the state transition.

The *Context Space* Ω_{con} is a collection of configurable features and feature relations that represent the variable parts of the component semantics. They are set by context. The component semantics is possibly influenced by the context when an individual component is integrated into an application. The Context Space is what expresses the

variability of an individual component when adapting to the context.

From the perspective of semantics-driven development, the process of CBD is a sequential of operations to compose, decompose and modify the Feature Spaces. The Feature Space can be visually represented by the Feature Diagram which consists of a set of nodes (denoting features), a set of directed edges (denoting feature relations), and a set of edge decorations. As the example depicted in figure 2, the Feature Diagram is organized as a tree structure; the three feature trees respectively describing the three aspects of the component semantics model. In figure 2 the tree notations are modified and extended to the Feature Diagram in [8]. Complete details about feature tree notations are given in [9].



(a) Feature Tree for Domain Space

(b) Feature Tree for Definition Space

(c) Feature Tree for Context Space

Figure 2. An Example of Feature Diagram for Component Semantics

According to their functions and logic relationships, the features can be categorized into *mandatory* feature (denoted by a rectangle with a simple edge ending with a filled circle. e.g. "Gender"), *optional* feature (denoted by a rectangle with a simple edge ending with an empty circle. e.g. "Child" in (a)), *alternative* feature (denoted by a rectangle with edges connected by an arc. e.g. the group of "Nursery", "School" and "University") and *or-feature* (denoted by a rectangle with edges connected by a filled arc. e.g. the group of "Mother" and "Father"). Each type of feature can have a set of instances called feature items (e.g. "yinyin:Child" means feature item "yinyin" is a instance of feature "Child"); and the feature relationships include two types: the *aggregation* relation (denoted by the solid edges) and the *instantiation* relation (denoted by the dotted edges).

Figure 2 shows each of the three aspects of the component semantics model has a set of specified notations. They are listed in Table 2,3 and 4.

4. Component Semantics Modeling In Feature Description Logic

Description Logics (DLs) are knowledge representation languages for expressing knowledge about concepts and concept hierarchies [10]. In DLs the domain of interest is modeled by means of *individuals*, *concepts*, *roles* and *knowledge base* exactly corresponding to the *feature items*, *features*, *feature relationships* and *Feature Space* respectively. A knowledge representation system based on DLs is able to perform specific kinds of reasoning. The main reasoning tasks are *classification* and *satisfiability*, *subsumption* and *instance checking*.

The purpose of using DLs in our modeling approach includes three points:

- A *formal language* for component semantics description. The formal semantics is the highest level of semantic awareness for CBD society to pursue [1].
- A *knowledge-based component reuse mechanism*, which is taken as a practical technique mirroring the knowledge-based reuse theory stated in section 1.
- A *reasoning tool* supported by the basic reasoning services in DLs. The reasoning ability is the premise for CBD automation that is regarded as the only way for the large-scale component reuse to turn into practice.

Table1 Syntax and semantics of $\mathcal{FDL}$ concept and role constructs

Features (Concepts) C	Syntax	Semantics
atomic feature	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
universal feature	$\top$	$\Delta^{\mathcal{I}}$
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$
conjunction	$C_1 \sqcup C_2$	$C_1^{\mathcal{I}} \cup C_2^{\mathcal{I}}$
alternation	$C_1 \veebar C_2$	$(C_1^{\mathcal{I}} \cap \neg C_2^{\mathcal{I}}) \cup (\neg C_1^{\mathcal{I}} \cap C_2^{\mathcal{I}})$
universal role quantification	$\forall R.C$	$\{o \mid \forall o' : (o, o') \in R^{\mathcal{I}} \rightarrow o' \in C^{\mathcal{I}}\}$
quantified number restriction	$(\leq n\ P.C)$	$\{o \mid \#\{o' \mid (o, o') \in P^{\mathcal{I}} \wedge o' \in C^{\mathcal{I}}\} \leq n\}$
collection of individuals	$\{a_1, \ldots, a_2\}$	$\{a_1^{\mathcal{I}}, \ldots, a_2^{\mathcal{I}}\}$
Feature Relations (Roles) R	**Syntax**	**Semantics**
atomic feature relations	P	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
disjunction	$R_1 \sqcup R_2$	$R_1^{\mathcal{I}} \cup R_2^{\mathcal{I}}$
reverse	R^-	$\{(o, o') \in \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}} \mid (o', o) \in R^{\mathcal{I}}\}$
concatenation	$R_1 \circ R_2$	$R_1^{\mathcal{I}} \circ R_2^{\mathcal{I}}$
reflect transitive closure	R^*	$(R^{\mathcal{I}})^*$

Table 1 shows a variant of DLs called $\mathcal{FDL}$, especially designed for expressing Feature Space. In $\mathcal{FDL}$, starting from a set of *atomic features* and *atomic feature relations*, complex features and relations can be built by applying certain *constructs*. Atomic features are denoted by $\boldsymbol{A}$, arbitrary features by $\boldsymbol{C}$ and $\boldsymbol{D}$, atomic relations by $\boldsymbol{P}$, and arbitrary relations by $\boldsymbol{R}$, all possibly with subscripts. The

following abbreviations are used to increase readability: $\bot$ for $\neg\top$, $C_1 \sqcap C_2$ for $\neg(\neg C_1 \sqcup \neg C_2)$, and $\exists R.\ C$ for $\neg\forall R.\neg C$ (means $\{o \mid \exists o' : (o, o') \in R^{\mathcal{I}}\}$. The constructs of $\mathcal{FDL}$ shown in Table 1 are almost standard ones except for the alternative (the notation is $\curlyvee$) which is interpreted as those features from which exactly one feature is included in the description.

In DLs the formal semantics is specified through the notion of interpretation. An *interpretation* $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a set $\Delta^{\mathcal{I}}$ (the domain of $\mathcal{I}$) and a function $\cdot^{\mathcal{I}}$ (the interpretation function of $\mathcal{I}$) that maps every feature to a subset of $\Delta^{\mathcal{I}}$(i.e. $C^{\mathcal{I}}$ to concept C); and every relation to a subset of $\Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ (i.e. $R^{\mathcal{I}}$ to role R), respecting the specific conditions imposed by the structure of the feature or relation.

A $\mathcal{FDL}$ knowledge base is formed by two constituent parts: The intensional one, called TBox, and the extensional one, called ABox. The TBox is a set of assertions of the forms:

$C_1 \sqsubseteq C_2$ Feature Inclusion

$C_1 \equiv C_2$ Feature Equality

Where C_1 and C_2 are arbitrary concepts. $C_1 \equiv C_2$ is an abbreviation for the pair of assertions $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. An interpretation $\mathcal{I}$ *satisfies* the assertion $C_1 \sqsubseteq C_2$ if $C^{\mathcal{I}}_1 \subseteq C^{\mathcal{I}}_2$. An interpretation is a *model* of a knowledge base if it satisfies all assertions in it. In addition, the cyclic statements are not allowed in $\mathcal{FDL}$.

The ABox has one of the forms:

$C(a)$ Feature Membership Assertion

$R(a, b)$ Relation Membership Assertion

where C is a feature, R is a relation and a , b are individuals. If $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ is an interpretation, $C(a)$ is satisfied by $\mathcal{I}$ if $a^{\mathcal{I}} \in C^{\mathcal{I}}$,and $R(a, b)$ is satisfied by $\mathcal{I}$ if $(a^{\mathcal{I}}, b^{\mathcal{I}}) \in R^{\mathcal{I}}$.

Table 2 Description of Domain Space in $\mathcal{FDL}$

Notation Names	Expressions in $\mathcal{FDL}$	Examples
Feature Item	$C(a),D(b_1),D(b_2)$	Person (zhang), Child(yinyin)
Item Relation	$R(a, b)$	hasChild(zhang,yinyin)
Composite item	$C(a),D(b_1),D(b_2)$ $R(a, b_1)$, $R(a, b_2)$ or $C(a) \equiv R.D(b_1) \sqcup R.D(b_1)$	Person (zhang) Child(yinyin), Child(jinjin) hasChild(zhang,yinyin) hasChild(zhang,jinjin)
Control Item	$C(a)$ $\forall R^{-}.\bot(a)$	Person (zhang)
Basic Item	$C(a)$ $\forall R.\bot(a)$	University(tsinghua)
Shared Item	$C(a_1)$, $C(a_2)$, $D(b)$ $R(a_1, b)$, $R(a_2, b)$	hasStatus(yinyin, 4th_School) hasStatus(binbin, 4th_School)
Alternative Relation	$C(a)$, $D(b_1)$, $D(b_2)$ $C(a) \equiv D(b_1) \curlyvee D(b_2)$	Child(jinjin),University(tsinghua), University(Beijing) Child(jinjin)≡University(tsinghua), ⋎University(Beijing)
Mandatory Relation	$C(a)$, $D(b)$, $R(a, b)$	Person (zhang), Child(yinyin) hasChild(zhang,yinyin)
Optional Relation	$C(a)$, $D(b)$ $C(a) \sqsubseteq (\leq 1 R.D(b))$	Child(yinyin)， School(evening) Child(yinyin)⊑(≤1 hasStatus.(evening))

Table 3 Description of Definition Space in $\mathcal{FDL}$

Notation Names	Expressions in $\mathcal{FDL}$	Examples
Feature	A, C, D, D_1, D_2	Person, Female
Feature Relation	P,R,Q	hasChild
Composite Feature	$C \sqsubseteq \exists R.\top$; or $C \equiv D_1 \sqcup D_2$	Parent≡Mother⊔Father
Root Feature	$C \sqsubseteq \forall R^{-}.\bot$	Creature
Atomic Feature	$A \sqsubseteq \forall R.\bot$	White, Male
Not Feature	$C \equiv \neg D$	Mother≡¬Father
Shared Feature	$C \sqsubseteq R_1.D_1$， $C \sqsubseteq R_2.D_2$	Mother⊑Female, Mother⊑Person
Aggregative Relation	$C \equiv D_1 \sqcup D_2$	Parent≡Mother⊔Father
Instantial Relation	$C \sqsubseteq R.D$ (R is has-a relation)	Mother⊑hasChild.Person
Crossover Relation	$C \equiv D_1 \sqcap D_2$; or $D_1 \sqsubseteq R.C$, $D_2 \sqsubseteq R.C$	Mother≡Woman⊓hasChild.Person
Alternative Relation	$C \equiv D_1 \curlyvee D_2$	Child≡ Nursery⋎School⋎University
OR-Relation	$C \equiv D_1 \sqcup D_2$	Parent≡Mother⊔Father
Mandatory Relation	$C \sqsubseteq R.D$	Mother⊑hasChild.Person
Optional Relation	$C \sqsubseteq (\leq 1 R.\top)$	MotherofChina⊑(≤1hasChild.Person)
Num-restrict Relation	$C \sqsubseteq (\leq n R.\top)$	MotherinVillage⊑(≤2hasChild.Person)
Composite Relation	$R \equiv Q_1 \sqcup Q_2$	hasChild≡isFehasChild⊔isMahasChild

Table 4 Description of Context Space in $\mathcal{FDL}$

Notation Names	Expressions in $\mathcal{FDL}$	Examples
Feature Parameter	$R.(C(a))(b)$	hasStatus.Child(yinyin)(evening)
Parameter Relation	$R(a, b)$	hasChild(zhang,yinyin)
Composite Parameter	$C(a)$, $D(b_1)$, $D(b_2)$ $R.(C(a))(b_1)$, $R.(C(a))(b_2)$	hasStatus.Child(yinyin)(evening) hasStatus.Child(yinyin)(4th_School)
Independent Parameter	$R.(C(a))(b)$ $\forall R^{-}.\bot(a)$	Person (zhang)
Basic Parameter	$C(a)$, $\forall R.\bot(a)$	University(tsinghua)
Shared Parameter	$C(a_1)$, $C(a_2)$, $D(b)$ $R.(C(a_1))(b)$, $R.(C(a_2))(b)$	hasStatus.Child(yinyin)(evening) hasStatus.Child(binbin)(evening)
Mandatory Relation	$C(a)$,$D(b)$, $C(a) \sqsubseteq (\leq 1 R.D(b))$ $R.(C(a))(b)$	Person (zhang), Child(yinyin) hasChild(zhang,yinyin)

As discussed above, the feature-oriented component semantics model includes three aspects: the Domain Space Ω_{dom}, the Definition Space Ω_{def} and the Context Space Ω_{con}. All these Spaces are the objects for $\mathcal{FDL}$ to describe. Considering the different properties of the three aspects, Table 2,3 and 4 respectively list the names of notation desired to construct each feature tree; and the corresponding $\mathcal{FDL}$ expressions or asserts are given, demonstrating that $\mathcal{FDL}$ is sufficient to describe the component semantics model. In column 3 of each Table, examples selected from Figure 2 are given to simply explain the meaning of notations and $\mathcal{FDL}$ expressions.

5. Model Checking and Reasoning in $\mathcal{FDL}$

The basic reasoning service in $\mathcal{FDL}$ is *satisfiability* of a feature C in a knowledge base Σ, written as $\Sigma \not\models C \equiv \bot$. It checks whether there exists a model $\mathcal{I}$ of Σ such that $C^{\mathcal{I}} \neq 0$. Other reasoning services such as *Subsumption*, which is to check whether a feature is subsumed by another, and *Consistency*, which is to check whether a knowledge base is satisfiable, can be reduced to feature satisfiablilty [10].

Suppose there exist two arbitrary features C, D, C_1 and

C_2 , an atom feature A, and an relation R. Following is the procedure to reason about feature satisfiablilty [3, 10].

STEP 1 Transforming C into *negation normal form* D which contains only complements of the atomic feature by following ten rules:

(1) $\neg\top\rightarrow\bot$;
(2) $\neg\bot\rightarrow\top$;
(3) $\neg(C_1 \sqcap C_2) \rightarrow \neg C_1 \sqcup \neg C_2$;
(4) $\neg(C_1 \sqcup C_2) \rightarrow \neg C_1 \sqcap \neg C_2$;
(5) $\neg\neg C\rightarrow C$;
(6) $\neg(\forall R.C) \rightarrow \exists R.\neg C$;
(7) $\neg(\exists R.C) \rightarrow \forall R.\neg C$;
(8) $(\leq n\ R) \rightarrow \neg(\geq n+1\ R)$;
(9) $\neg\ (\geq n\ R) \rightarrow \forall R.\ \bot$ if $n = 1$;
(10) $\neg\ (\geq n\ R) \rightarrow (\leq n - 1\ R)$ if $n > 1$.

STEP 2 Generating all *complete constraint systems* deriving from $\{x{:}D\}$.

First we introduce what are the forms of constraint. Assuming x, y, z are variable symbols, α is a function maps every variable to an element of $\Delta^{\mathcal{I}}$, then a constraint is a syntactic object of one of the forms:

$$x : F, \text{ if } \alpha(x) \in F^{\mathcal{I}};$$
$$x\ R\ y, \text{ if } (\alpha(x), \alpha(y)) \in R^{\mathcal{I}};$$
$$x \neq y, \text{ if } \alpha(x) \neq \alpha(y)$$

A *constraint system* S is a finite, nonempty set of constrains. Following six quasi-completion rules are given to generate constraint systems:

(1) Intersection: $S \rightarrow_{\sqcap} \{x{:}\ D_1\ ,\ x{:}D_2\} \cup S$, if x: $D_1 \sqcap D_2$ is in S, and x: D_1 and x: D_2 are not both in S.

(2) Union: $S \rightarrow_{\sqcup} \{x{:}\ D\ \} \cup S$, if x: $D_1 \sqcup D_2$ is in S, neither x: F_1 nor $x{:}F_2$ is in S, and $D = D_1$ or $D = D_2$.

(3) Existential Quantification: $S \rightarrow_{\exists}\{xRy,\ y{:}\ D\ \} \cup S$, if $x{:}\exists R.D$ is in S, there is no z such that z is successor of x and z: F is in S, and y is a new variable.

(4) Universal Quantification: $S \rightarrow_{\forall} \{\ y{:}\ D\ \} \cup S$, if $x{:}\forall R.D$ is in S, y is successor of x in S, and y: D is not in S.

(5) At-least Restriction: $S \rightarrow_{\geq} \{x\ R\ y\} \cup S$, if no other completion rule applies to S, x: $(\geq n\ R\ )$ is in S, x does not have a R-successor in S, and y is a new variable.

(6) At-most Restriction: $S \rightarrow_{\leq} S\ [y/z]$, if x: $(\leq n\ R)$ is in S, x has more than n successor in S, and y, z are two R-successors of x that are not separated. ($S[y/z]$ denotes the constraint system obtained from S by replacing each occurrence of y by z.)

STEP 3 Checking whether all constrain systems contain a clash. If any of them contains a clash, then C is satisfiable; otherwise C is not satisfiable.

A *clash* is a constraint system having one of following forms:

H. $\{x{:}\bot\}$;
(2) $\{x{:}\ A,\ x{:}\neg A\}$;
(3) $\{x{:}\ (\geq m\ R),\ x{:}\ (\leq n\ R)\}$ where $m>n$.

The $\mathcal{FDL}$ is a kind of $\mathcal{ALCN}$. According to [10], the satisfiability of such features can be decided in nondeterministic polynomial time.

The above checking procedure can be used to reason about the semantic consistency of the component-based system. Supporting system SYS is the assembly of components $CP_1, CP_2, \ldots, CP_n$. Firstly, we use feature-oriented modeling techniques to draw the feature trees for each $CP_i (0 \leq i \leq n)$, and express these trees in $\mathcal{FDL}$ to produce a group of feature spaces Ω^i_{dom} and Ω^i_{def} . Note that the context space is instantiated by configuration. Secondly, we combine all Feature Spaces to form a knowledge base for SYS: $\Sigma_{dom} = \bigcup_{i=1}^{n} \Omega^i_{dom}$ and $\Sigma_{def} = \bigcup_{i=1}^{n} \Omega^i_{def}$. Finally, we check the satisfiability of each feature in Σ_{dom} and each feature item in Σ_{def} . If all the features and feature items are satisfiable, then the component-based system is consistency.

6. Conclusions

The radical source of difficulty in component reuse may be the comprehension gap between the component creators and reusers in different contexts. $\mathcal{FDL}$ addresses this problem through a practical knowledge-based modeling approach. One of its advantages is the checking and reasoning mechanism can be applied in the modeling process. However, there still exist lots of problems for further investigation. For example, we should find the solution to decrease the complexities of time and space about Feature Space.

References

[1] Martin Blom, Eivind J. Nordby. “Semantic Integrity in Component Based Development”. Project Report, Mälardalen University, Sweden, March 2000.

[2] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau. Technical Concepts of Component-Based Software Engineering (Volume II), TECHNICAL REPORT CMU/SEI-2000-TR-008May 2000.

[3] Yu Jia, Yuqing Gu. “Representing and Reasoning on Feature Architecture: A Description Logic Approach”. *Workshop on “Feature Interaction in Composed Systems”*, ECOOP 2001.

[4] Kang, K.; Kim, S.; Lee, J.; Shin, E.; & Huh, M. “FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures”. *Annals of Software Engineering* 5, 5 (September 1998): 143-168.

[5] H. Milli, F.Milli, A.Milli. "Reusing software: Issues and research directions". IEEE Trans. On software Engineering, 21(6):529-561,1995.

[6] DoD, "Software Reuse Primer", DoD Software Reuse Initiative, http://dii-sw.ncr.disa.mil/reuseic/ pol-hist/ primer, April 15, 1996

[7] Yu Jia, Yuqing Gu. "The Representation of Component Semantics: A Feature-Oriented Approach". Processing of Workshop On Component-Based Software Engineering: composing systems from components. 2002

[8] K. Czarnecki and U. Eisenecker. "Generative Programming: Methods, Tools, and Applications". Addison-Wesley, New York, 2000.

[9] Yu Jia. "The Evolutionary Component-based Software Reuse Approach". PH.D. Dissertation. 2002

[10] F. M. Donin, M. Lenzerini, D. Nardi, and W. Nutt. "The Complexity of Concept Languages. Information and Computation": pp.1-58, Vol. 34. 1997

[11] Ivica Crnkovic, Heinz Schmidt, Judith Stafford, Kurt Wallnau. Anatomy of a Research Project in Predictable Assembly. Fifth ICSE Workshop on Component-Based Software Engineering White paper. http://www.sei.cmu.edu/pacc/CBSE5/CBSE5-CFP.html,2002.

Version-based Approach for Modeling Software Systems

A. Speck [1] S. Robak [2] E. Pulvermüller [3] M. Clauss [1]

[1]Intershop Research
Intershop Tower
D-07740 Jena, Germany
a.speck@intershop.com
matthias.clauss@gmx.de

[3]Instytut Informatyki i Zarzadzania
Uniwersytet Zielonogórski
ul. Podgórna 50
PL-65-246 Zielona Góra, Poland
s.robak@iiz.uz.zgora.pl

[3]Fakultät für Informatik, IPD
Universität Karlsruhe
D-76128 Karlsruhe, Germany
pulvermueller@acm.org

Abstract

Since the first discussion about the software crisis in 1968 many concepts in order to improve the software development and reuse have been introduced. Some of them like frameworks or components aim on the reuse of code others capture experiences with system architecture, design or coding recommendations.

This paper focuses on a way to express the reuse of pieces of software and design reasoning. We apply versions in order to describe sets of features we want to have in a system. Conditions are used to formulate these requirements.

Then these conditions are integrated into UML models using the existing mechanisms for extensions. This enables a tight integration of versions ands modelling constructs and enhances integrity of models.

The use of version is demonstrated by an example from the eCommerce domain. This paper also discusses the effects of the integration of versions into UML models.

1 Introduction

Software developers always have the generic problem to structure their systems, to deal with the size and to master the complexity of large systems. This is the reason for the development of programming languages and compilers and it is the motivation for the development of different paradigms and methods for the software development. Additionally more and more improved means to divide and structure systems are introduced.

For the development of software systems, especially large systems, modelling notations are used to provide an abstract view on the system. The Unified Modelling Language [16] provides a standardised modelling notation and includes mechanisms to introduce user-defined extensions and adaptations for specific needs [21, 19].

In this paper we present an approach to define versions of systems and subsystems. These versions are composed by logical operations and serve as a model for the verification of real systems. The versions are structured hierarchically which means that versions of lower levels may be part of higher level versions.

The advantage of the definition of software systems and

their variability in versions is that versions capture the static dependencies within systems in the same way like other meta models (e.g. UML). Moreover version systems may be validated automatically which means that the verification of a specific system may be tool-based. The version definition of components to be built in a system may be checked against the needs defined in the version specification of the system. This eases the reuse of existing systems and system components.

A first step towards such an integration of versions into modelling tools is done by defining versions with UML models. The version information is attached to its referring elements and can be extracted from the model to get a view of the version model.

First we introduce the version concept in section 2. In section 3 we demonstrate the use of version on an example based on the features of a simple messaging system and discuss the effects of modelling versions in UML.

2 Versions and Conditions

Software modules [17], the object-oriented paradigm [4] and components [22] may be regarded as the base of the proposed versioning approach. However they do not provide solutions for the problems caused by the existence of multiple systems derived from one common base. Component systems are typical examples for this problem: There are a lot of technologies to combine software components (plug them together) like CORBA or COM+ [22]. Means to assure that component systems are providing exactly the services they are requested to perform are quite rare. In this paper we will focus on the existence of different realizations or implementations of systems built from the same and / or similar components. How could the combinations and variability within these different systems be handled and how could means to deal with these problems be used to support the system development?

The concept of versions is well-known in the domain of software development supported by version or revision control systems like RCS or CVS [7, 14, 1]. These systems are used to administrate and store the different states or versions of systems code during the implementation phase. However such versioning systems are not the main issue in our approach but may be applied to administer the versions of our approach.

2.1 Version Model

We propose to apply versions for the architecture of component systems. Our version model integrates consistency and dependency management in a natural way at design time. It allows to describe the dependencies between components as well as the internal structure of components (consisting of other components). A version determines a software core which may contain other versions and has to consist of a valid set of conditions.

Definition: (*Version*)
The symbol reflecting a version is V_i^k where k represents the granularity (level 0 defines the most low-level granularity) and i gives the index distinguishing between versions on the same level of granularity.

Now we can inductively define the construction of versions:

$$Version\ V_i^0 \ = \ true \ \wedge \ Cond_i^0$$

where $Cond_i^0$ represents the condition [1] that has to be true for one version V_i^0 on level 0. $V_i^0 \in V^0$ where

$$V^0 \ = \ \{\bigcup_{j=1}^{\infty} V_j^0\}$$

is the set of all versions on level 0.

In the same way we can define the induction step:

$$Version\ V_i^{(n+1)} \ = \ \bigwedge_{j=l}^{m} V_j^n \ \wedge \ Cond_i^{(n+1)}$$

with

$$Cond_i^{(n+1)}\ is\ true, \quad 1 \leq l \leq m \leq |V^n|, \quad V_j^n \in V^n$$

In other words: a version is a set of conditions (a unification of the particular conditions of a certain version and all conditions of the sub-versions contained in the version). A condition is expressed as boolean expression.

The operands in such an expression are conjunction, disjunction and negation.

[1] Several conditions may be unified in one condition.

The use of the boolean operations conjunction, disjunction and negation allows to apply all techniques to minimise the terms (the number of components and their relations) which are known from the domain of digital electronics. Additionally simple verification tools may provide checks to verify whether the set of components is valid, e.g. all required components are included or forbidden components do not exist in the systems. One example for such a tool is introduced in [13].

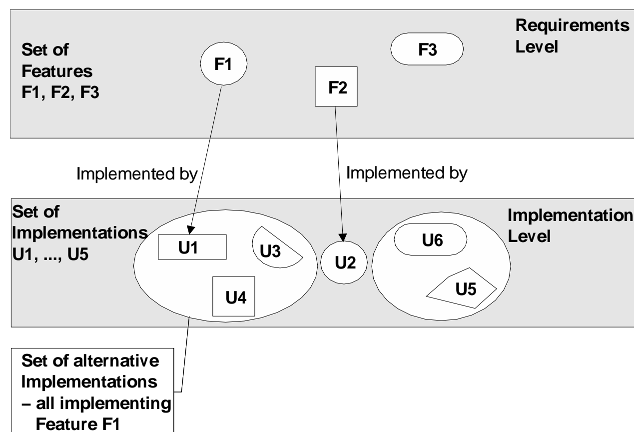


Figure 1: Relationship between Requirements and Implementation

2.2 Conditions and Features

The version model introduced in the previous section is based on conditions which determine whether a version is valid or not. Conditions are the mechanisms to control versions. In contrast to these, features describe the requirements and the properties of systems. E.g. in the requirements engineering phase of e-business systems specific business processes may be considered as features which have to be part of the system. During the development these features must then be realized by software components.

Conditions may be used to formalise features. A specific version containing a specific condition realizes therefore the feature(s) corresponding to this condition.

There are two different types of features and conditions describing the features. They depend on the phase when they occur or when they are considered. The ones are referring to the high-level requirements the others to individual implementations.

- **Requirements Engineering**:
  During the requirements engineering phase features are identified. A powerful means to do that is domain engineering [9] which in particular helps to detect and capture re-occurring and thus reusable conditions on the requirements level for a certain domain. When modelling the commonalities and differences of a domain, e.g. in a feature model [11, 9] it is possible to extend this model by additional semantic information or even derive logical formulae directly from the model (the model already captures semantic relationships as feature interdependencies respective composition rules).
- **Implementation**:
  Systems are implemented by composing components. Currently there are different system generators or generator architectures [9]. The composition is performed according to the condition rules defined in the requirements engineering phase.

There is a clear connection between these two levels as exposed in figure 1. Formally this relationship can be expressed as follows: $U1$, $U2 \in Set1$; $F1$, $F2 \in Set2$ where $Set1$ is the set of implementation units and $Set2$ is the set of features on the requirements level. Assuming that $U1$ implements (besides others) feature $F1$ and $U2$ implements $F2$ then:

$$valid(U1, U2) \Rightarrow valid(F1, F2)$$
$$valid(F1, F2) \not\Rightarrow valid(U1, U2)$$

with function

$$valid(X, Y) = \begin{cases} true & \text{: X and Y form a valid combination} \\ false & \text{: X and Y form an invalid combination} \end{cases}$$

Function $valid(X, Y)$ may be calculated by evaluating the binary condition expressions.

The distinction between requirements and implementation level is not only limited to the development phase of

a system or concerns but also exists in the maintenance phase where additional conditions may appear. This is due to the fact that it is impossible to capture all relevant dependencies and conditions from beginning. Additional conditions are added as needed or detected in a piecemeal growth manner [8].

3 Application of Versions in Software Models

Versions could be used when a system is developed from scratch. In such a case versions help to deal with the varieties that occur due to alternative design decisions which allow different versions of a specific design. Additionally versions help to define proper subsystems which may then be included in the system.

Nevertheless, versions may also be applied in systems that are already established and exist as different releases. In such cases versions may be used to define specific design alternatives within a given system family.

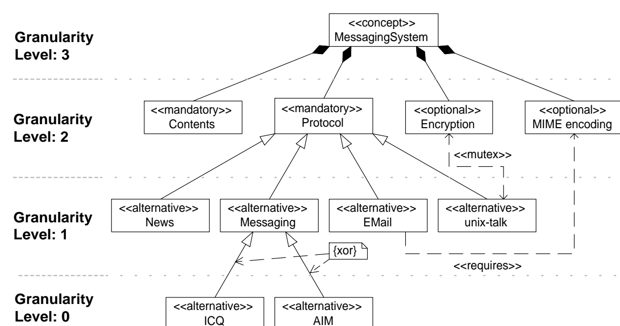


Figure 2: Feature Diagram of a Messaging System

3.1 Reuse Problem

An example for systems which are usually derived from a set of generic functionality are customisable components. Unfortunately almost none of the target systems are really the same since each customer has very individual wishes.

We demonstrate this problem at a comparative simple example: a generic messaging system that can be part of a more complex system. Figure 2 depicts the feature diagram of this system according to the notation introduced in [6]. The feature diagram shows the variability according to the experience of different already existing subsystems.

In the example the highest granularity level 3 comprises the whole system denoted by *Messaging System*. It contains the mandatory feature *Contents* that is not subject of our example. The focus of this example is on the feature *Protocol* abstracting a technical requirements of the system. The messaging system can be enhanced by adding support for encryption or *MIME-encoding* of messages. The coarse-grained feature *Protocol* can be further decomposed into different protocols for message transport. The coarse-grained feature *Protocol* can be further decomposed into different protocols for message transport.

The *Messaging System* provides of mandatory features like *Contents* and *Protocol*. Moreover it has optional features (e.g.*Encryption* or *MIME encoding*). The *Protocol* consists of one or more of the alternative protocol types (e.g.*News*, *Messaging*, *EMail* or *unix-talk*). *Messaging* has either *ICQ* or *AIM*, both are impossible. The feature tree is cross-cut by cross-tree constraints. These relationships can not be represented by a strict feature hierarchy. Cross-tree constraints describe the feature interactions not recognised by the feature hierarchy. For this messaging component the optional encryption of messages con not be used with the *unix-talk* protocol. In the example we have both types of cross-tree constraints: the mutual exclusion `mutex` which prohibits the combination of *unix-talk* protocol and *Encryption* and the requires cross-tree dependency between *EMail* and *MIME encoding*[2].

All the relationships in the feature diagram in UML-notation may be mapped to the logical version operators presented in section 2. Mandatory features may be expressed by a logical and, options by or, strict alternatives as well as mutexes by xor and requirement cross-tree constraints again with and.

If such systems are getting bigger they can consist of a very large number of small subsystems. All the dependencies and relations of such a large system can not be controlled by human beings since it is just too complex.

A solution how to deal with the complexity of large(r) systems is to construct them from smaller subsystems

[2]Multi-purpose Internet Message Extension

whereas their features are synthesised from other features. The logical model of a complete system is built by inserting the logical formulae of the sub-features (of lower granularity level) into the logical formulae of the sub-features. Then the large model captures all the possible variability of the system and serves as a base for building versions of the system. This decomposition of a complex system is exemplarily shown on the example in figure 3. It shows the version of some features and captures their conditions for valid compositions. These conditions can be much more detailed than it is reasonable in feature diagrams and therefore describe detailed composition rules on requirements level.

$$
\begin{aligned}
V_i^3 \text{ 'Messaging System'} &= (V_0^2 \text{ 'Contents' } \wedge V_1^2 \text{ 'Protocol'}) \vee \\
&\quad V_2^2 \text{ 'Encryption' } \vee V_3^2 \text{ 'MIME encoding'} \\
V_1^2 \text{ 'Protocol'} &= V_0^1 \text{ 'News' } \vee V_1^1 \text{ 'Messaging' } \vee \ldots \\
V_1^1 \text{ 'Messaging'} &= V_0^0 \text{ 'AIM' } \vee V_1^0 \text{ 'ICQ'}
\end{aligned}
$$

Figure 3: Versions of the Messaging System

3.2 System Version

A concrete version of a system is built by the concrete choice of alternative versions. This design has to be done according to the given rules in a feature model defined by the system version and its sub-versions (e.g. modeled in a feature diagram like in figure 2).

In the best case the components realizing the features of the versions already exist. In such a case the appropriate component may be inserted into the place within the system. The features of the version indicate which component has to be integrated and may serve as a specification against which the components of the real system and their arrangement may be checked.

If there are no components available to implement a version the version definition specifies the requirements of the component(s) to be newly developed. In contrast to other requirements definitions like *Use Cases* or lists of non-functional requirements the versions give a comparatively precise definition of the needs since they are derived from the surrounding existing pieces of a system. Additionally they integrate non-functional and functional requirements in a natural way which means that they clearly show which functionality at which place of the system realizes or supports a non-functional requirement and interacts with which other sub-versions (realized by components).

Figure 4 depicts some components of a implementation of the *Messaging System*. The components and their arrangement are derived from the version model as given in figure 2.

A version (with its sub-versions) may be realized in two ways:

1. Inclusion:
   The *Messaging System* includes other components defined by the sub-versions (*Contents*, *Protocol*, *Encryption* and *MIME encoding*).

2. Facade:
   The super-version may also serve as a facade of the sub-versions. Usually this happens when a super-component does not only contain sub-versions but also controls the sub-versions. The component *AbstractMessaging* of figure 4 is such an example. It has sub-versions and controls and triggers their services.

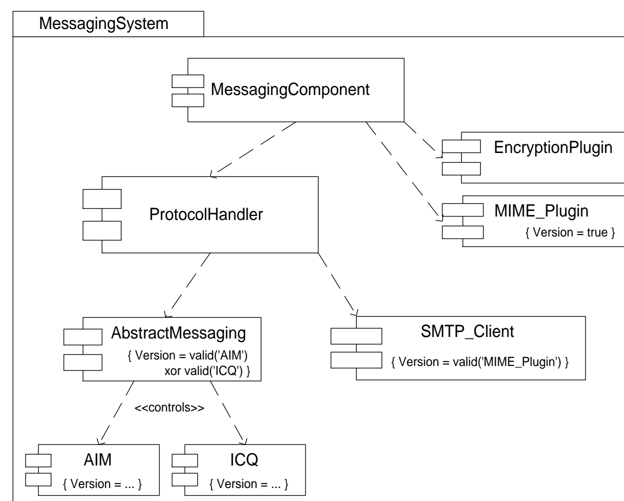


Figure 4: Components of the Messaging System

3.3 Using versions in UML models

Figure 4 also exemplarily shows the notation of versions in UML models. The version assigned to a specific modelling element (as components and classes) is denoted as the tag `Version` having a logic formulae as value. This tag is typed as a BooleanExpression (metaclass defined by UML, ([16] pages 2 – 79) and can therefore be evaluated to either true or false. If the evaluation of the versions condition results in true this version is valid and can be included in the system.

The syntax of the value must always result in a boolean value at computation. It can be described, e.g., in natural language, as mathematical formulae or as OCL[3] expression according to the UML's definition of `BooleanExpression`. In the example we prefer the latter since its syntax is well-defined and enables the textual description of mathematical expressions.

To refer subversion in a version expression we use the function $valid$ defined in section 2.2. Its definition in OCL syntax is:

```
valid (elementname : Name) : Boolean
```

It checks the correctness of the version of the model element whose name given in the argument. If the version of this element is valid, the function returns true, otherwise false. This way it is used to reference other versions in OCL constructs the same way as a version refers to a subversion.

Having this function the description of a version according to the definition in section 2.1 can be completely mapped to an OCL expression. Moreover the version of a modelling element is given in place of the element itself thus improving integrity of the model and enabling an easy tracing from a version to its according system artefact.

3.4 Meta-model problems of UML

The need to use a function `valid` shows a problem in using UML: UML is designed for modelling single software system and therefore lacks of support for describing generic systems [5]. Generic systems (sometimes called meta-modelling) describe an abstract system design including variabilities that are bound to derivate the design of a concrete system. You can also look at generic models as an meta level between the model level (named M1) and the meta-model level (M2) of the UML. This virtual meta-level is like M1 since it uses constructs defined in M2 but on the other hand it serves as abstraction of M1 and is therefore not identical with M1.

This missing meta-level for generic modelling prohibits the direct evaluation of a versions condition denoted in OCL. Despite this is not a real disadvantage since there is currently no known modelling tool that integrates the evaluation of OCL expressions into the modelling process. Instead it is possible to extract the version information from the model, e.g. using XMI, and generate input for a model-validation tool from it. Concluding currently the OCL serves as well-defined and standardised expression language that can be translated in any language supported by modelling or validation tools.

[3]Object Constraint Language

4 Related Work

The version approach of this paper is influenced by prior work about versioning at Bell Laboratories and in [20], for instance. A similar approach of describing systems in versions by applying logical formulae to describe their relationships may be found in [24]. It keeps close to feature logic while our approach concentrates on the realization and concrete application in a component world.

Methods and techniques which are covered by the area of "Separation of Concerns" such as Aspect-oriented Programming [12], Subject-oriented Programming [10], Composition Filters [2], Adaptive Programming [15] and generators like GenVoca [3] or system generators in general [9] may be used to realize tools to support the version approach we proposed in this paper. An application of such a technique may be found in [18].

The modelling notation used for the feature model is based on prior work on modelling variabilities in UML [6]. Meta-modelling techniques for software system families are described in [23]. Generic modelling in UML has been discussed in [5].

5 Conclusion

This paper proposes a version-based approach for the development of component systems. The versions may be arranged hierarchically and define a set of features which may be realized by components. The feature set of a specific version is determined by logical formulae.

The description of the static relationships and their variability (versions) in logic allows a tool-supported validation. The issues of such a validation may be the consistency of a system specification especially when parts of this specification are reused. Moreover the definition of components may be verified against the requirements of a version system specification. Also it supports to derive a specific version (with special conditions) from an existing versions system.

The descriptions of variants directly in the model enables a tight integration and provides the first step towards tool support for the versioning concept. Modelling tools can extract the version information directly from the model and provide it to any validation tool for evaluation.

Further steps in our work may be to handle not only boolean conditions but also fuzzy requirements and specification of reused components. Since the un-precise analogue expression of requirements seems to be much more natural than digital decisions such an approach may improve the version-based system engineering considerably.

Additionally our version concept does not cover the dynamic behaviour of systems. The introduction of the system's activities will provide important additional information.

References

[1] *RCE, VRCE, BDE; RCE: the Revision Control Engine.* http://wwwipd.ira.uka.de/~RCE/, 2001.

[2] M. Aksit. Composition and Separation of Concerns in the Object-Oriented Model. *ACM Computing Surveys*, 28(4), December 1996.

[3] D. Batory and B. Geraci. Composition Validation and Subjectivity in GenVoca Generators. In *IEEE Transactions on Software Engineering*, pages 67 – 82, 1997.

[4] G. Booch. *Object-Oriented Analysis and Design, Second Edition.* Benjamin/Cummings, Redwood City, CA, 1994.

[5] M. Clauss. Generic Modeling using UML extensions for variability. In *Proceedings of OOPSLA Workshop on Domain-specific Visual Languages*, pages 11 – 18, Tampa, FL, USA, 2001.

[6] M. Clauss. Modeling variability with UML. In *Proceedings of the Young Researchers Workshop GCSE'01, Third International Symposium on Generative and Component-Based Software Engineering*, Erfurt, Germany, September 2001.

[7] R. Conradi and B. Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232 – 282, 1998.

[8] J. O. Coplien. Re-evaluating the Architectural Metaphor: Towards Piecemeal Growth, Guest editor introduction to IEEE Software Special Issue on Architecture Design. *IEEE Software*, 16(5):40 – 44, 1999.

[9] K. Czarnecki and U. Eisenecker. *Generative Programming - Methods, Tools, and Applications*. Addison-Wesley, 2000.

[10] O. H. and P. Tarr. Using Subject-Oriented Programming to overcome common Problems in Object-Oriented Software Development/Evolution. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 687 – 688, May 1999.

[11] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *LNCS 1241*, ECOOP. Springer-Verlag, June 1997.

[13] H. Klaeren, E. Pulvermüller, A. Rashid, and A. Speck. Aspect Composition applying the Design by Contract Principle. In *Proceedings of the GCSE'00, Second International Symposium on Generative and Component-Based Software Engineering*, LNCS, Erfurt, Germany, September 2000. Springer.

[14] E. Lippe and G. Florijn. Implementation Techniques for Integral Version Management. In *Proceedings of ECOOP'91, European Conference on Object-Oriented Programming*, LNCS 512. Springer, 1991.

[15] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development. In *ACM SIGPLAN notices*, volume 33, October 1998.

[16] Object Management Group, http://www.rational.com/uml/resources/documentation/. *Unified Modeling Language (UML) Specification version 1.3*, June 1999.

[17] D. Parnas. On The Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053 – 1058, December 1972.

[18] E. Pulvermüller, A. Speck, and J. O. Coplien. A Version Model for Aspect Dependencies. In *Proceedings of 2nd International Symposium of Generative and Component-based Software Engineering (GCSE 2001)*, LNCS, Erfurt, Germany, September 2001. Springer.

[19] S. Robak, B. Franczyk, and K. Politowicz. Extending UML for Modeling Variability for System Families. *International Journal of Applied Mathematics and Computer Science*, 12(2), 2002.

[20] M. Rochkind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4):364 – 370, December 1975.

[21] S. Szostak, S. Robak, R. Stryjski, and B. Franczyk. UML extensions for modeling real-time and embedded systems. In *Discrete - Event System Design - DESDes '01 : Proceedings of the International Workshop*, pages 109 – 114, Zielona Góra, Poland, 2001.

[22] C. Szyperski. *Component Software.* Addison-Wesley, ACM-Press, New York, 1997.

[23] J.-P. Tolvanen and S. Kelly. Modelling languages for product families: a method engineering approach. In *Proceedings of OOPSLA Workshop on Domain-specific Visual Languages*, pages 135 – 140, Tampa, FL, USA, 2001.

[24] A. Zeller and G. Snelting. Unified Versioning through Feature Logic. *ACM Transactions on Software Engineering and Methodology*, 6(4):398 – 441, 1997.

Stable and Reusable Model-Based Architectures

Ahmed Mahdy, Mohamed E. Fayad, Haitham Hamza, and Peeyush Tugnawat
Computer Science and Engineering Dept.
University of Nebraska-Lincoln
Lincoln, NE 68588, USA
{amahdy, fayad, hhamza, peeyush}@cse.unl.edu

Abstract

The purpose of this paper is to show, by the use of examples, how software stability provides model-based reusability. The strengths of Enduring Business Themes and Business Objects are shown; they can be reused among common-core applications without change. Three case studies are modeled. These case studies show how Software Stability Models (SSMs) can be reused as a base for single or multiple applications. The key contribution of this paper is in showing that SSMs are indeed reusable, stable over time and help to avoid the hassle of reinventing the wheel in software development.

1. Introduction

"Why should we reengineer a whole system if only the exterior part has changed?" this question was the main motive behind the introduction of Software Stability [2]. SSMs are based on the concepts of Enduring Business Themes (EBTs) [1, 2, 3, 4] and Business Objects (BOs) [2, 3, 4]. Both of these concepts address the core knowledge of the system and how well this core knowledge is understood.

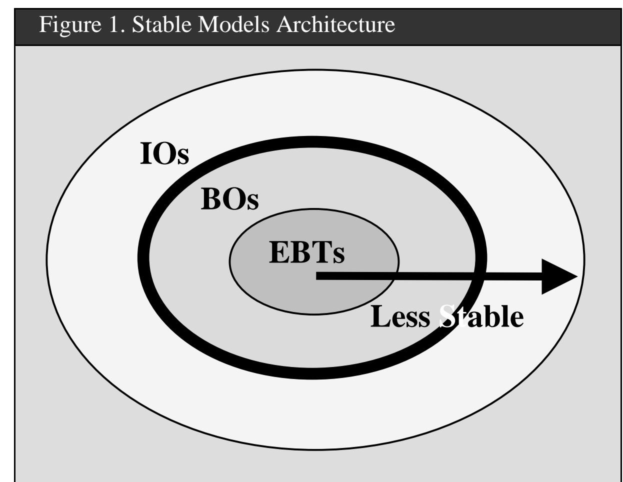
The way SSMs are built directly guarantees system reusability. They provide a stable core that can serve one or more applications that share similar goals. Accordingly, we should be able to build multiple systems that have identical cores. This is what reusability is all about.

This paper emphasizes the reusability of SSMs. In section 2, software stability is briefly introduced followed by a discussion of reusability as a software stability merit. In section 3, three different systems are modeled using stability concepts to show the reusability of SSMs. Section 4 concludes the paper.

2. Stable and Reusable Architectures

A SSM can be partitioned into three different levels: EBTs, BOs, and Industrial Objects (IOs). EBTs represent intangible objects that remain stable internally and externally. BOs are objects that are internally adaptable but externally stable, and IOs are the external interface of the system [2]. In addition to the conceptual differences between EBTs and BOs, a BO can be distinguished from an EBT by tangibility. While EBTs are completely intangible concepts, BOs are partially tangible. These artifacts develop a hierarchal order for the system objects, from totally stable at the EBTs level to unstable at the IOs level, through adaptable though stable at the BOs level. The stable objects of the system are those do not change with time. Some common-core systems may look different, but in fact, the only differences lay on the surface (i.e. IOs). If they have a common internal structure, why should we deal with them differently? It suffices to develop one system for all of these similar applications. At this point, the importance of SSMs becomes apparent. The more the systems share, the less will need to be changed. Changes will be made to the IOs, the EBTs and BOs need not be touched. Figure 1 shows the Stable Models Architecture. The EBTs represent the nucleus of the model, while the IOs represent the surface of the system. The BOs lay in between. Intuitively, the further objects are placed from the interface the more stable they will be. As a result, common-core applications share the inner layers (i.e. EBTs and BOs), and differ at the outer layer.

Figure 1. Stable Models Architecture



3. Case Studies

In this section of the paper, three applications are modeled. On the surface these applications appear to be different. However, they can all be developed on a common core using software stability. As discussed in section 2, the EBTs and BOs remain the same among applications with a common core.

Case I- Computers Trading

This case study depicts the trading of computers using a bidding process. One individual identifies the needed specifications, maximum price…etc. Another individual or entity replies back, if interested. A negotiation process takes place, until a deal is struck or an impasse is reached. Involved entities might be computer shops, dealers, manufacturers, or even individual users. If an agreement is reached, the deal is finalized A stable model for this application is shown in Figure 2.

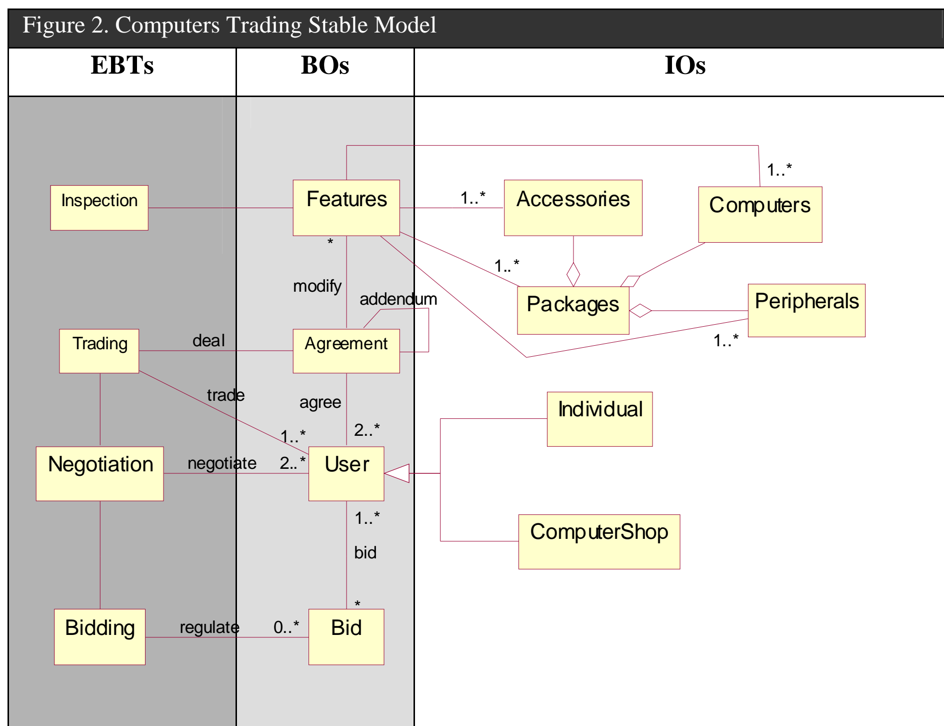


Figure 2. Computers Trading Stable Model

The model has four EBTs: Trading, Negotiation, Inspection, and Bidding. Trading is an enduring theme because it remains stable externally and internally as long as this system lasts since the main objective is to sell/buy computers. As for Negotiation, the bidding process conceptually implements the negotiations between the different entities. Again, it remains as a concept of this system as the system will forever be based on negotiations. Inspection is an enduring concept in any trading as the buyer inspects the product she is interested in. Bidding is also an enduring object since one of the main purposes is to have the ability to bid on certain items. Bid, Agreement, Features, and User represent the BOs. Each of these objects is externally stable though highly adaptable internally, for example the user is always a user to the system although he maybe an individual or a shop.

Case II- Buying a House

Trading a house is much more complicated than trading a computer. They look different, however they share the same core. When you buy a house, aren't you trading, negotiating, and bidding? The answer is yes, so buying a house is not that different from buying a computer. Thus, the EBTs and BOs can and should be the same. Figure 3 shows the stable model of this case study.

Case III- Bidding on a Football Team

Does the concept of buying a football team look different from the concept of buying a computer? Yes, it looks different, but using a similar argument as provided in Case II is applicable to this application. As in a house purchase bidding process, it starts by placing a bid. If it is reasonable, negotiations take place until an agreement is reached or an impasse reached. The deep structure is the same as for the other two case studies. Figure 4 shows a stable model.

Although the three case studies deal with different types of trading, the inner core is the same among these applications. Recognizing this fact results from how software stability approaches the problem domain. Identifying the system EBTs directly reflects the goals of the application. If different applications share the same

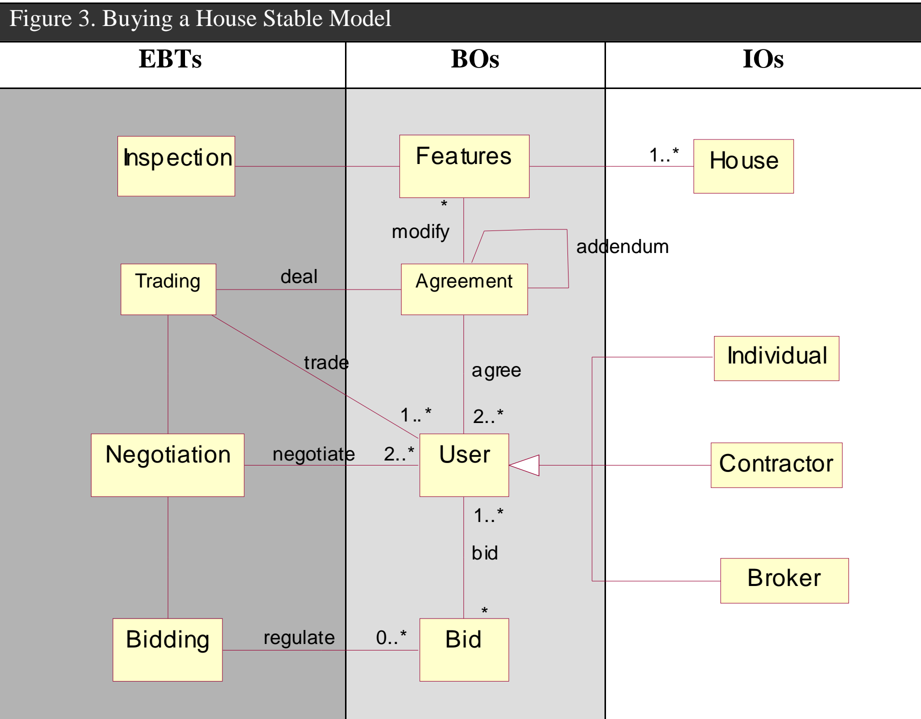


Figure 3. Buying a House Stable Model

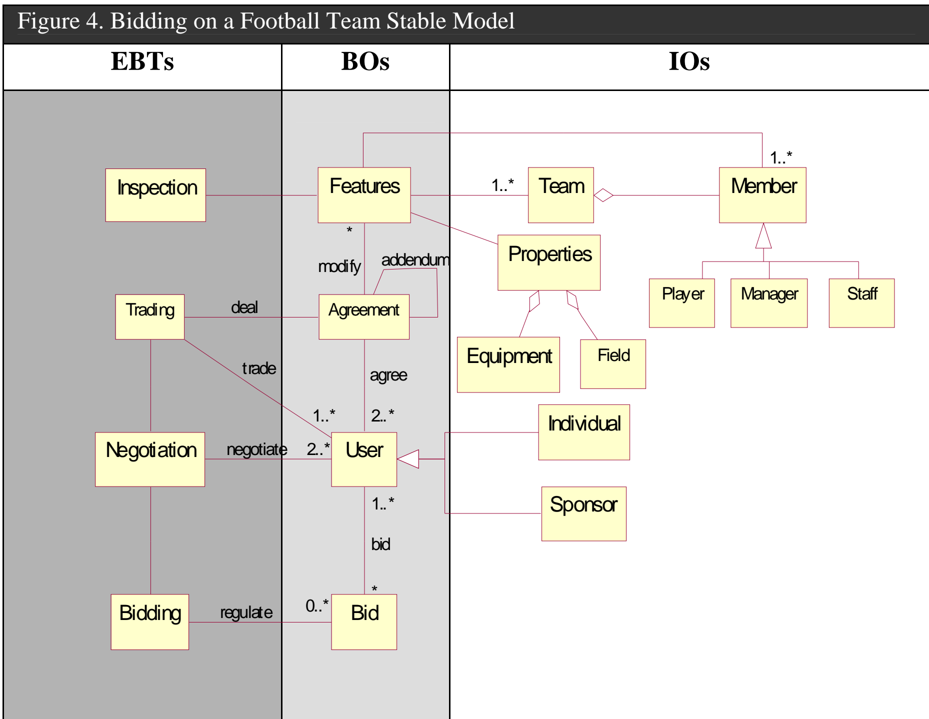


Figure 4. Bidding on a Football Team Stable Model

goals, it would be easy to determine their similarities by following the stability approach. Moreover, the BOs most probably will be exactly the same. The only differences would appear at the IOs level. In our case studies, the three applications have the same EBTs and BOs. Figure 5 plots the three stable models on top of each other to show how their similarities.

reusability of SSMs comes naturally as a result of having stable cores.

4. Conclusion

Software Stability goes deep into the structure of the system fleshing out the core knowledge of the system.

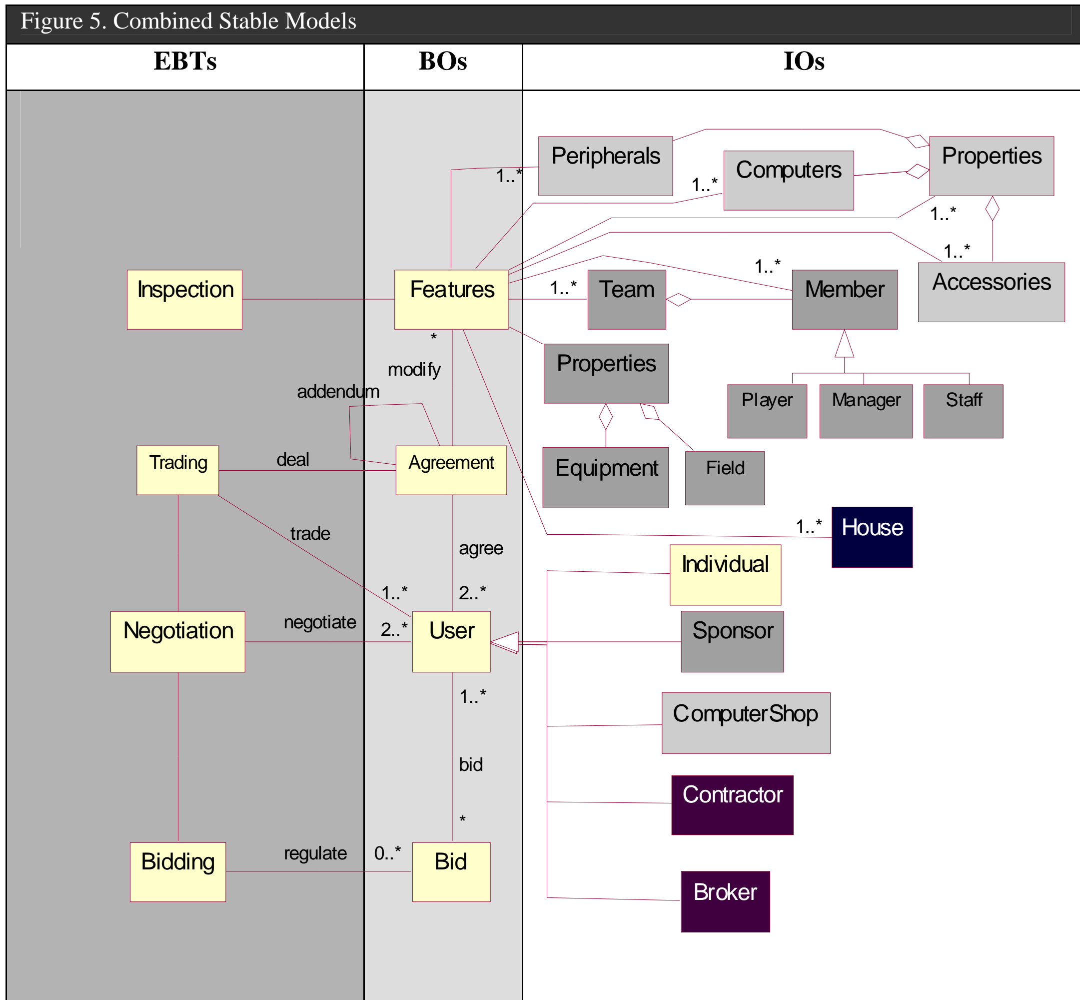


Figure 5. Combined Stable Models

As shown in the figure, the EBTs and BOs of the three applications are identical. The various IOs from each application are depicted using a different color (i.e. light gray for Case I, black for Case II, and dark gray for Case III). The message that we get from this figure is that if we develop one of these applications, it will not be necessary to develop other applications from scratch. In fact, minimal work needs to be done, mainly with the IOs, and we obtain a model for the new applications. Although the purpose of using software stability in model-based reuse is similar to that of domain engineering, software stability is not bounded to domain-oriented reuse. In fact, the

Having this knowledge about the system, a stable core can be well engineered. This stable core not only serves this application, but also can be reused to build similar applications, similar in the sense that they share only that common core. Limiting the required similarity to the core widens the domain of applicability.

In conclusion, software stability models are reusable. The examples studied in this paper show the reusability of the EBTs [1, 2, 3, 4] and BOs [2, 3, 4], as they remain common among different applications due to their stable nature.

5. References

[1] M. Cline and M. Girou, “Enduring Business Themes”, ***Communications of the ACM,*** Vol. 43, No. 5, May 2000, pp. 101-106.

[2] M.E. Fayad, “Accomplishing Software Stability”, ***Communications of the ACM,*** Vo. 45, No. 1, January 2001, pp 95-98.

[3] M.E. Fayad, and A. Altman, “Introduction to Software Stability”, ***Communications of the ACM,*** Vo. 44, No. 9, September 2001, pp 95-98.

[4] M.E. Fayad, “How to deal with Software Stability”, ***Communications of the ACM***, Vo. 45, No. 4, April 2002, pp109-112.

Stable Model-Based Software Reuse

Mohamed E. Fayad, Shasha Wu, and Majid Nabavi
Computer Science and Engineering Dept.
University of Nebraska-Lincoln
Lincoln, NE 68588, USA
{fayad,shwu}@cse.unl.edu and mnabavi@unlnotes.unl.edu

Abstract

Model-based software is expected to provide a higher-level of reusability than the software that came before. However, conventional models do not satisfy this expectation because they can become unstable when changed. The Software Stability Model (SSM) offers an innovative approach to expressing the core purpose of a problem and a method to realize that approach that typically yields extensible and stable design models. SSM especially emphasizes the model's stability over changes, a critical issue in the realm of software reuse.

1. Introduction

Computer techniques have progressed very rapidly in the past 20 years. Every year, researchers and manufacturers make computers more and more powerful. At the same time, the size and complexity of software has also increased. This situation has caused the problem of how to efficiently utilize previous works in software development to grow into a critical issue in the field of software engineering. When we compare software systems, we usually find 60% to 70% commonality from one software application to another [1]. People have generated many approaches to achieve the goal of software reusability and progress has been made every year. Design Patterns, tests cases, prototypes, plans, documentation, frameworks, and templates are all progeny of this goal of reuse. Among all of these approaches, object-oriented modeling provides the best high-level description of software. Therefore, we expect that model-based reuse would provide a higher level of reusability in software development. However, conventional modeling approaches cannot satisfy this expectation.

Unlike the products of other engineering fields, there are few extra costs or physical impediments to the implementation of software in different locations. All copies of the software are exactly the same. Software should have excellent reusability. Unfortunately, the problems for which software is designed are always different. Every specific problem has its particular requirements, even if their main purposes are very similar. Currently, when we try to reuse an existing conventional model, a re-engineering process is inevitable in most situations. It does not matter if the change is due to new technology or a change in clientele [2]. The reusability of the conventional model is very limited. On the other hand, the stable and extensive nature of the Software Stability Model (SSM) naturally makes it a better approach for reuse. To demonstrate the advantages of the Software Stability Model in reusability, we shall apply the SSM to a case study of open-pit mining[1] and compare it with corresponding conventional models.

2. The Case Study

2.1. Original problem

Underlying the requirements of open-pit mining is a common transportation problem (Figure 1) [5]. In an open-pit mine, mechanical shovels load material from depots into dump trucks. These trucks then transport material to various destinations. Ore is transported to mineral processing facilities to be processed and prepared for market. Waste is delivered to dumps.

A scheduling or dispatching system is required to assign the empty trucks to proper destinations at proper times. This system checks the status of each truck and shovel. As soon as a truck�означ's status is "free", it is assigned to a shovel that will be free when the truck arrives. If all of the shovels are busy, the dispatcher system selects the shovel with minimum wait time.

A conventional model for this problem is given in Figure 2. The model illustrates how the transport system works. There is extracted "Ore" in "Ore Extraction Openings" and "Waste" rocks in "Waste Removal Openings". This material is transported to a "Mineral Processing Facility" and "Waste Dumps" respectively. "Dump Trucks" are assigned to these openings according

[1] The detailed description of this problem can be found at http://www.cse.unl.edu/~fayad/SoftwareStability/OOPSLA01-DesignFest-Final1.doc.

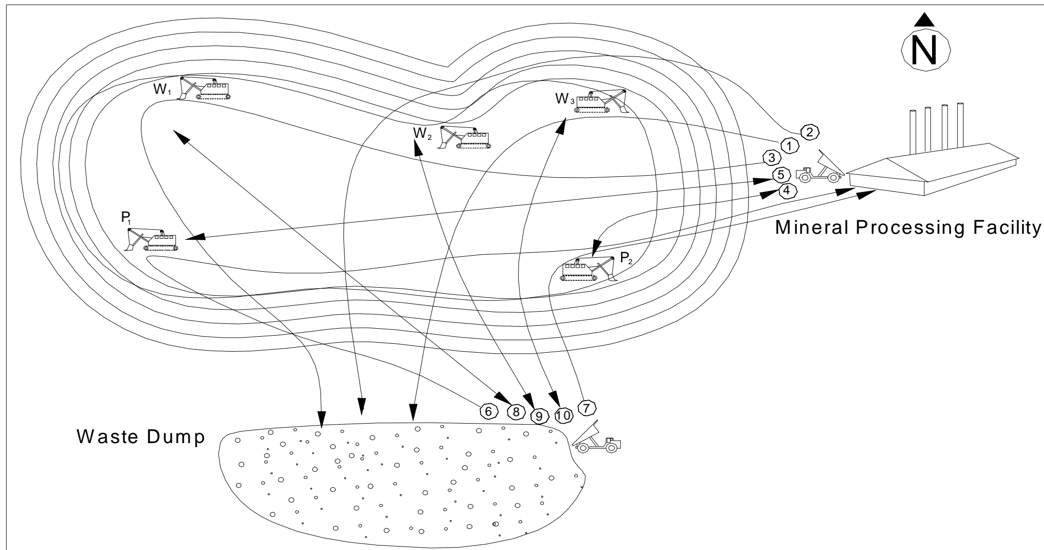


Figure 1. Open pit mining [5]

to a "Schedule". "Shovels" load "Ore" and "Waste" into these trucks.

The desired solution to this problem optimizes the operation of the trucks and the overall efficiency of the mining operation. The goal of the system is to maximize profitability of the mine and to satisfy production quotas.

With the stability approach, we first define the purpose of the system and delineate why the system is needed. Although this aspect of analysis is not unique to the SSM, SSM formalizes the analysis and focuses the analysis activities around the concept of Enduring Business Themes (EBTs). Enduring Business Themes are the core abstractions of a problem domain. These are the themes that are unlikely to change over time. For example, one important theme of the open-pit mining problem is "Efficiency„. The concept of efficiency clearly has a role as an EBT, because without efficiency there is no purpose for the scheduling system. When we derive a schedule for trucks, we are actually trying to make our system work more efficiently, hence, more profitably.

Another important theme is concurrency. Consider the interaction between different components of the transport system; they are assigned to work together when they are available. Dump trucks are assigned to shovels; trucks and shovels can only work together when shovels are available. In other words, they have concurrency. This concept is also used to understand material flow through the conveyor belt and pipeline systems. Any of the components in the system has to pass a specific amount of material to the next component within a specified time period. All of these material flows have to be equal in a system and all the components of the system must be available and working at the same time, to be concurrent. Without this concurrency we cannot design a transport system; there will be materials congestion and system failure. Therefore, it is a very important concept and one of the core purposes of our scheduling system.

After identifying the enduring business themes associated with our problem, we can identify and associate those Business Objects that provide the necessary abstractions of the processes that underlie the business operations (such as Origin, Destination and Transport). The third step of the Software Stability Process is to define those Industrial Objects (IOs) that refine (or instantiate) the various Business Objects. The result of this process is displayed in figure 3.

2.2. "New" problem

We can readily imagine that over time or across several mining pits, production techniques or mining conditions could change. Perhaps conveyor belts are used as the main transportation methods instead of trunks. In this method, loaders carry materials from the depot and dump it into feeders placed a short distance from the extraction site. The feeder feeds a crusher at a steady rate and the crusher reduces the size of the rocks to a proper size for transport by conveyor belt. The whole system must be consistent in terms of the capacity of materials flow. Loaders must carry the required volume of material in a time say, ton/hour, the feeder must feed the crusher at the same rate and the crusher must deliver the same amount of material with the proper size to the conveyor belt. Conveyor belts, in turn, must have the proper width and speed to transport the material. If any of the elements of this system fails to keep up with the required rate of materials flow, there will be a material congestion at that point and the whole system will fail to achieve its goal of transporting the designed quota of material to the destination.

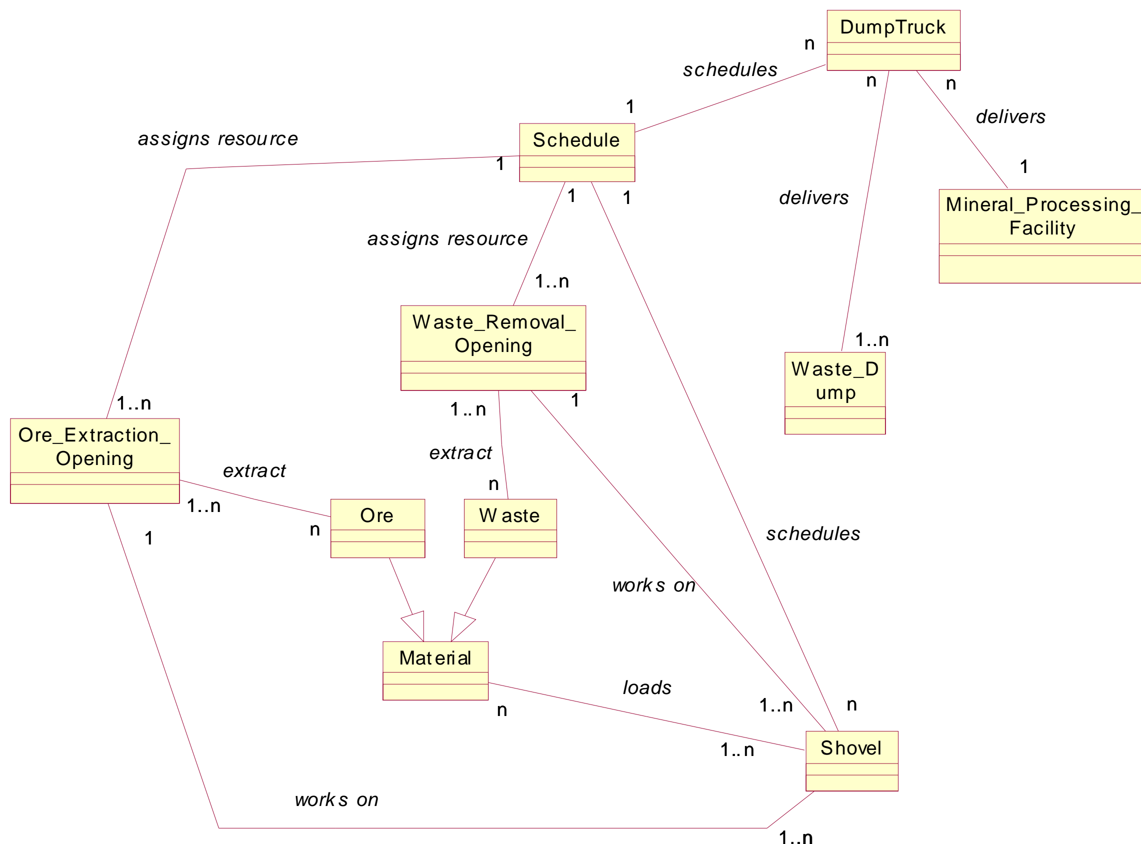


Figure 2. Typical class model for open pit mining [5]

With the introduction of the new technique, the old model fails to satisfy the business requirements. We must modify it. A new class diagram might be generated as in Figure 5, and we note that the new model bears little resemblance to our original model, nor would it satisfy the requirements of our first open pit mine.
Since the core purpose of the problem is unchanged during this alteration, we can directly reuse the previous stability model by modifying or adding some IOs. Most of the contents in figure 3 are reused in Figure 6. Some new IOs are Feeder, Loader, Crusher, and Conveyor. Acting as the core part of the model, our EBTs and BOs are unchanged in these two models.

2.3. The third change

Consider the idea that other locations may use a pipeline as the transportation mode. Figure 7 [5] characterizes the pipeline technique. Here, material is loaded into feeders by loaders. Each feeder passes the material to a crusher and then on to a ball mill. Ball mills reduce the size of the materials and prepare them for transportation through pipelines. The milled materials are poured into mixers, where water is added to them, making slurry, which is then pumped into the pipeline. The pipeline then conducts the material to the destination. With conventional models, the tendency is to remodel as in Figure 8 and again, the result is a substantial change. As in the previous modifications, the core purpose of the problem is still unchanged during this evolution. Therefore, simply by adding some IOs, we can cater to this new change. In figure 9, these new objects include pump, pipeline, mixer, and ballmill. It is obvious that our EBTs and BOs are still constant and reused in the new model.

2.4. More possible extensions

We can consider the pipeline transport method as a general method of transporting liquids like oil and water. What is the similarity of this system with transporting rocks and soil? Let‟s explain the model of transport for oil and then develop the model to see how it can fit into a transport model.
As we can see in figure 10, crude oil is extracted from the oil well, and depending on the nature of the oil reserve some pressure control process is necessary to get the flow under control. This is a complicated operation and out of the scope of transport. After controlling the extracted oil, it has to be transported to refineries or reserve tanks in

ports. The pumping system pumps oil into the pipeline; several pumping stations may work along a pipeline to keep the pressure high enough to propel the flow at the desired speed. The traditional model of this transport system (figure 11) is quite different from the material transport system we have developed for the mining industry.

What changes are needed to adapt our mining transport system to this new transport system? Very few changes are actually needed. We only need to add some new IOs such as ”Oil Well,,, ”Tank,,, and ”Pressure Control,, to the model in figure 9. Comparing figure 3, 6, 9, and 12, we can easily see that during all of these modifications, our EBTs and BOs remain the same. We duplicate them again and again to satisfy many different requirements, thanks to the Stability Model.

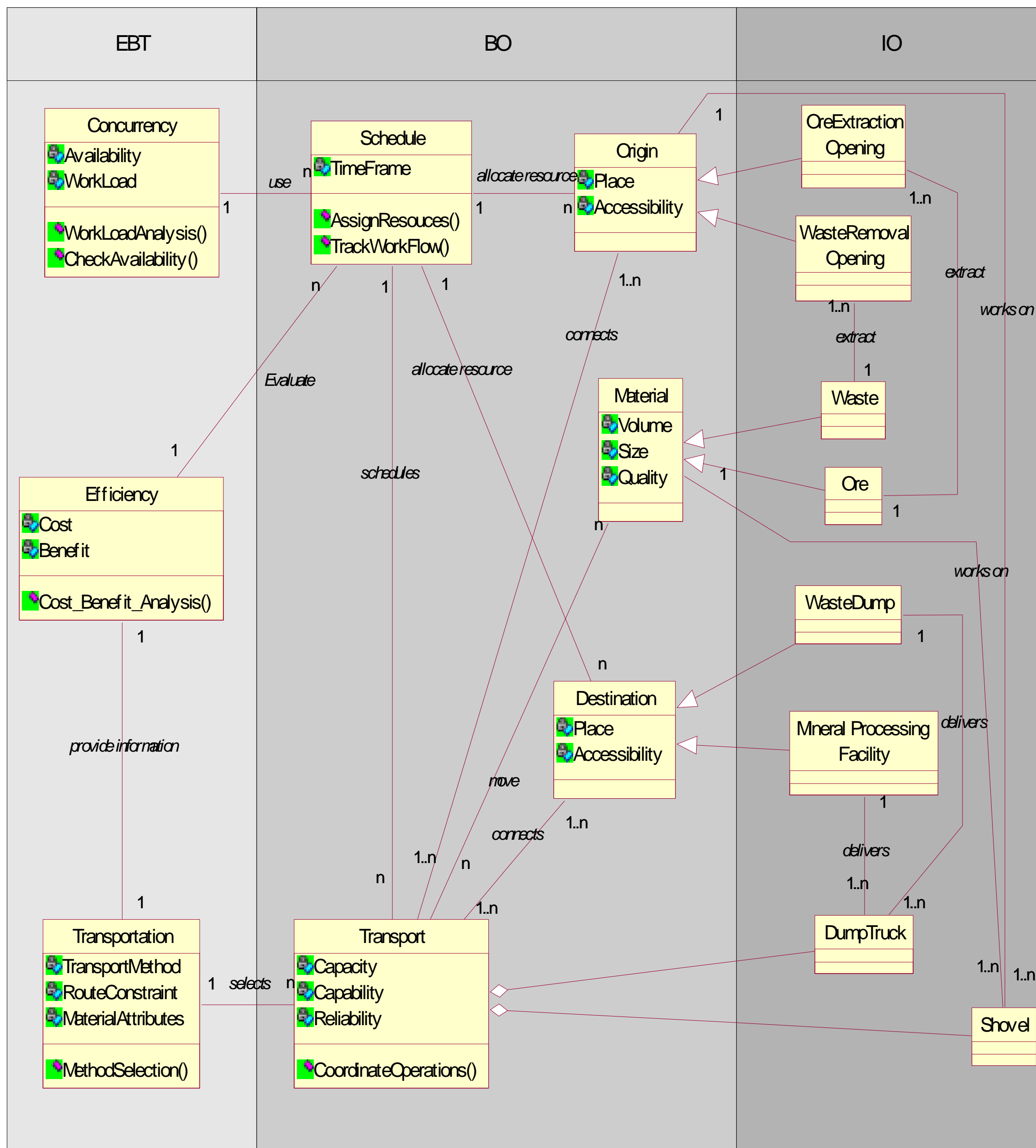


Figure 3. Class diagram of stability model for open pit mining

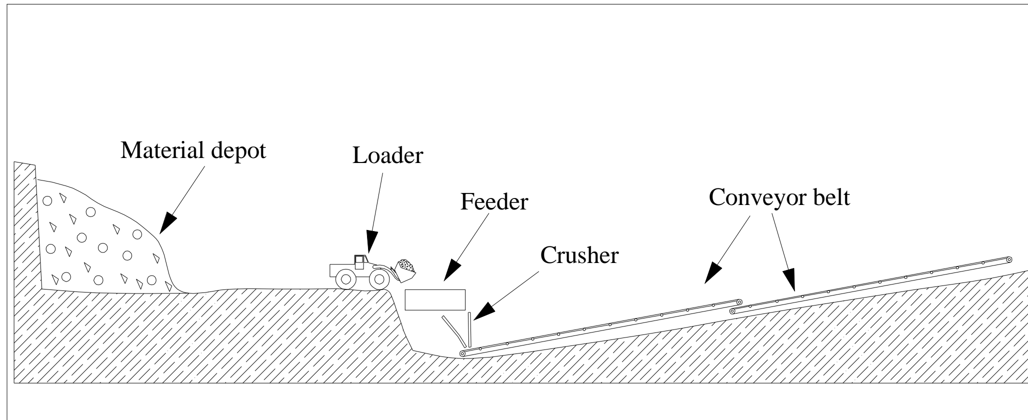


Figure 4. Transport materials by conveyor belt [5]

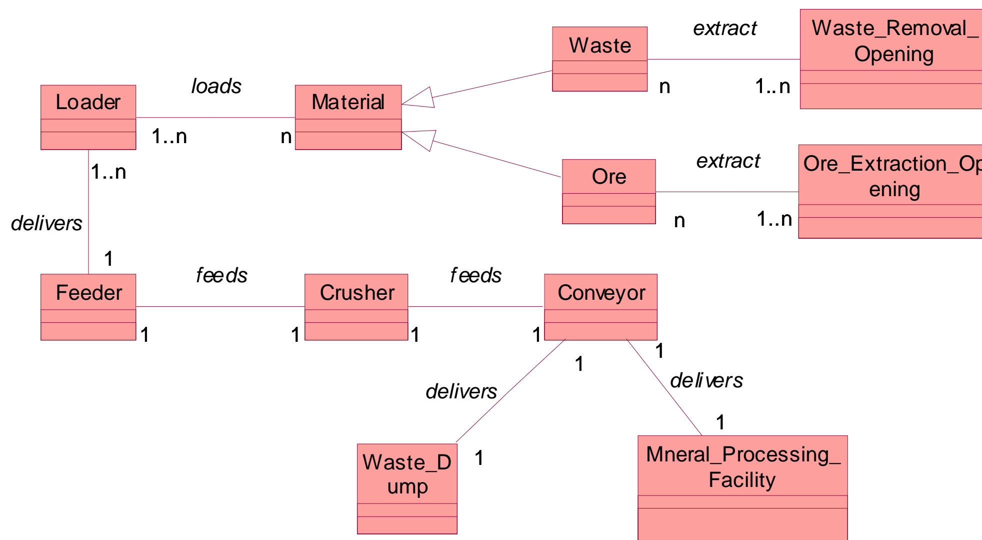


Figure 5. Class diagram of conventional model for conveyor belt

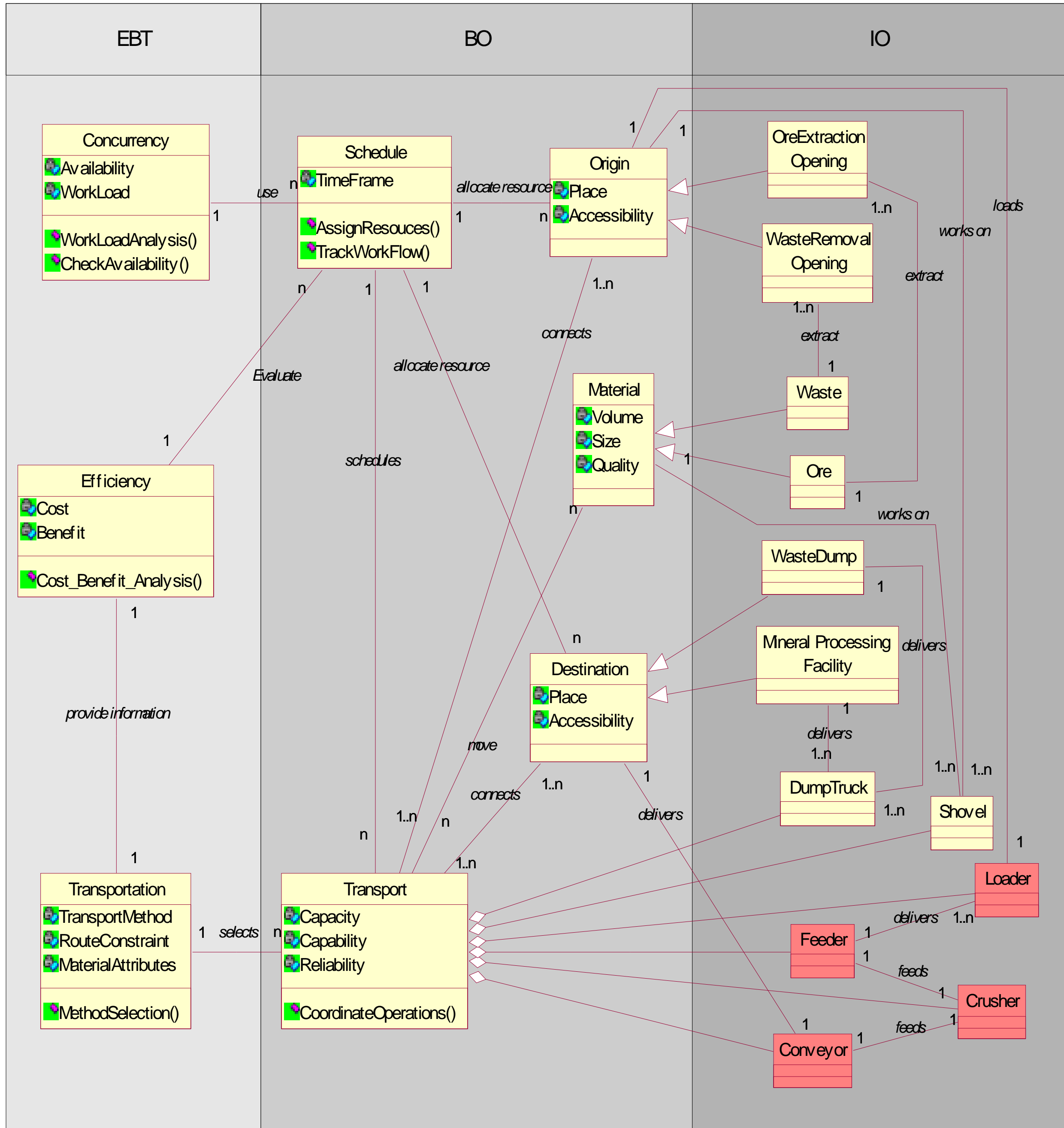


Figure 6. Class diagram of stability model for conveyor belt

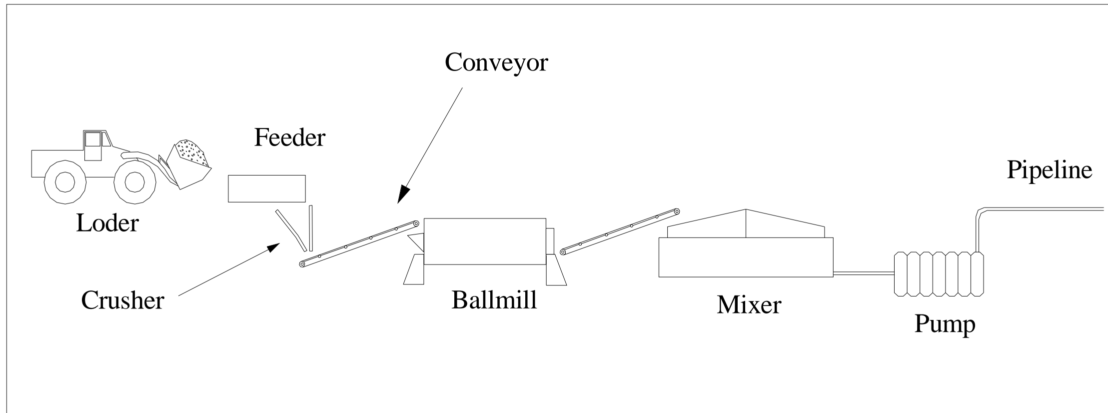


Figure 7. Transport materials through pipeline [5]

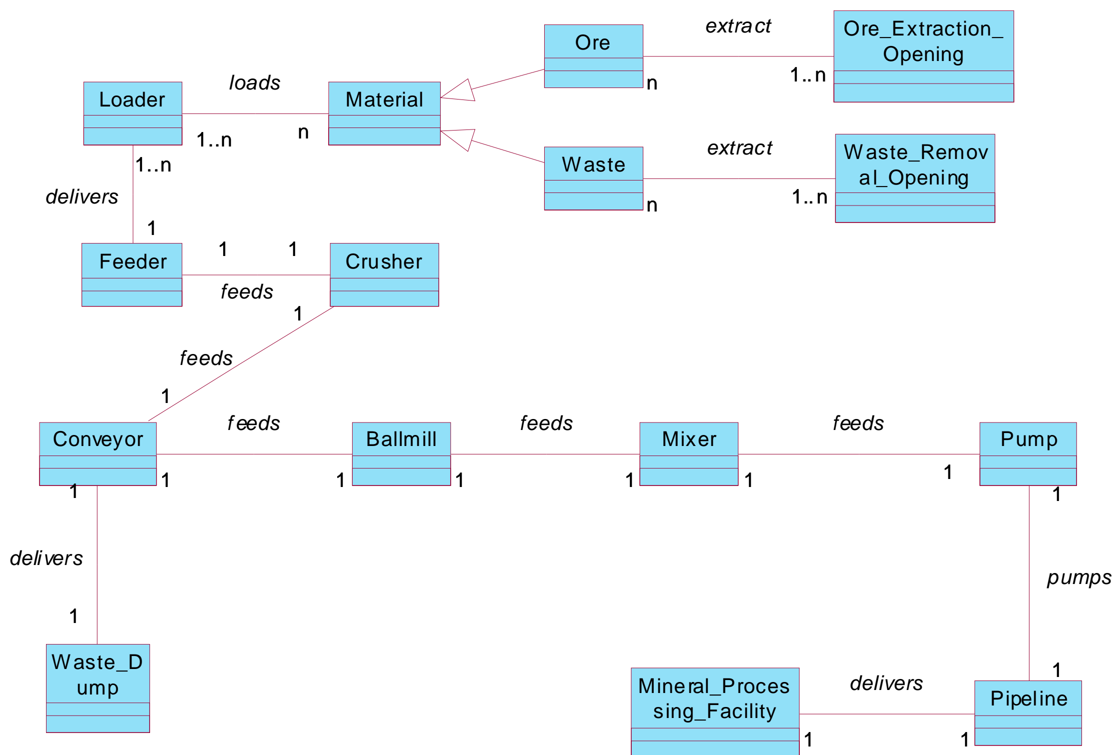


Figure 8. Class diagram of traditional model for pipeline transportation

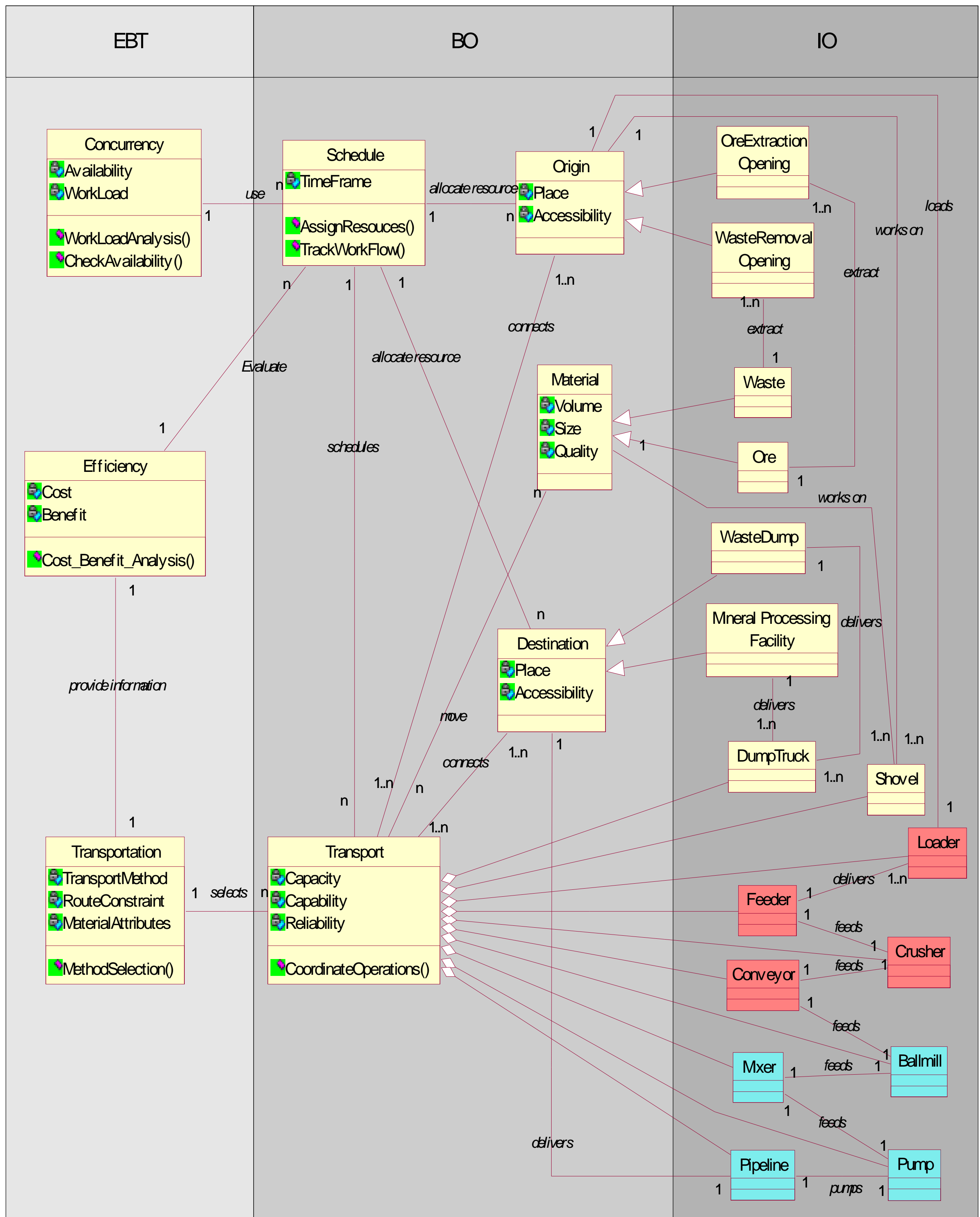


Figure 9. Class diagram of stability model for pipeline requirement

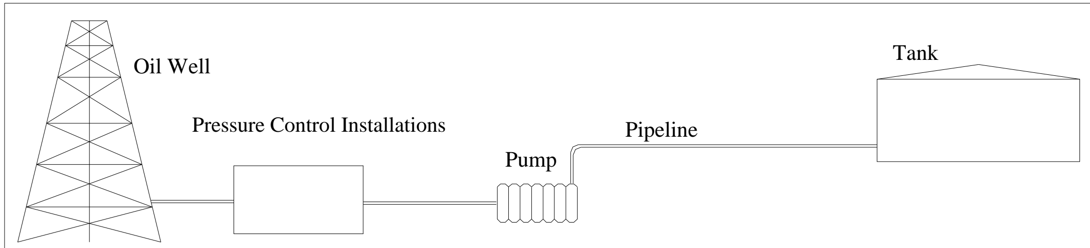


Figure 10. Oil transport system

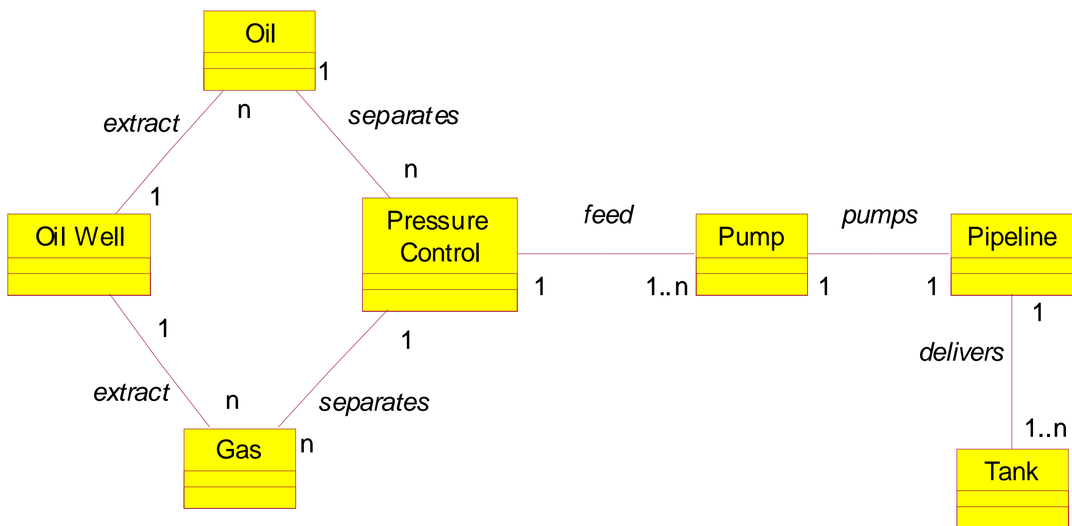


Figure 11. Traditional model for oil transport system

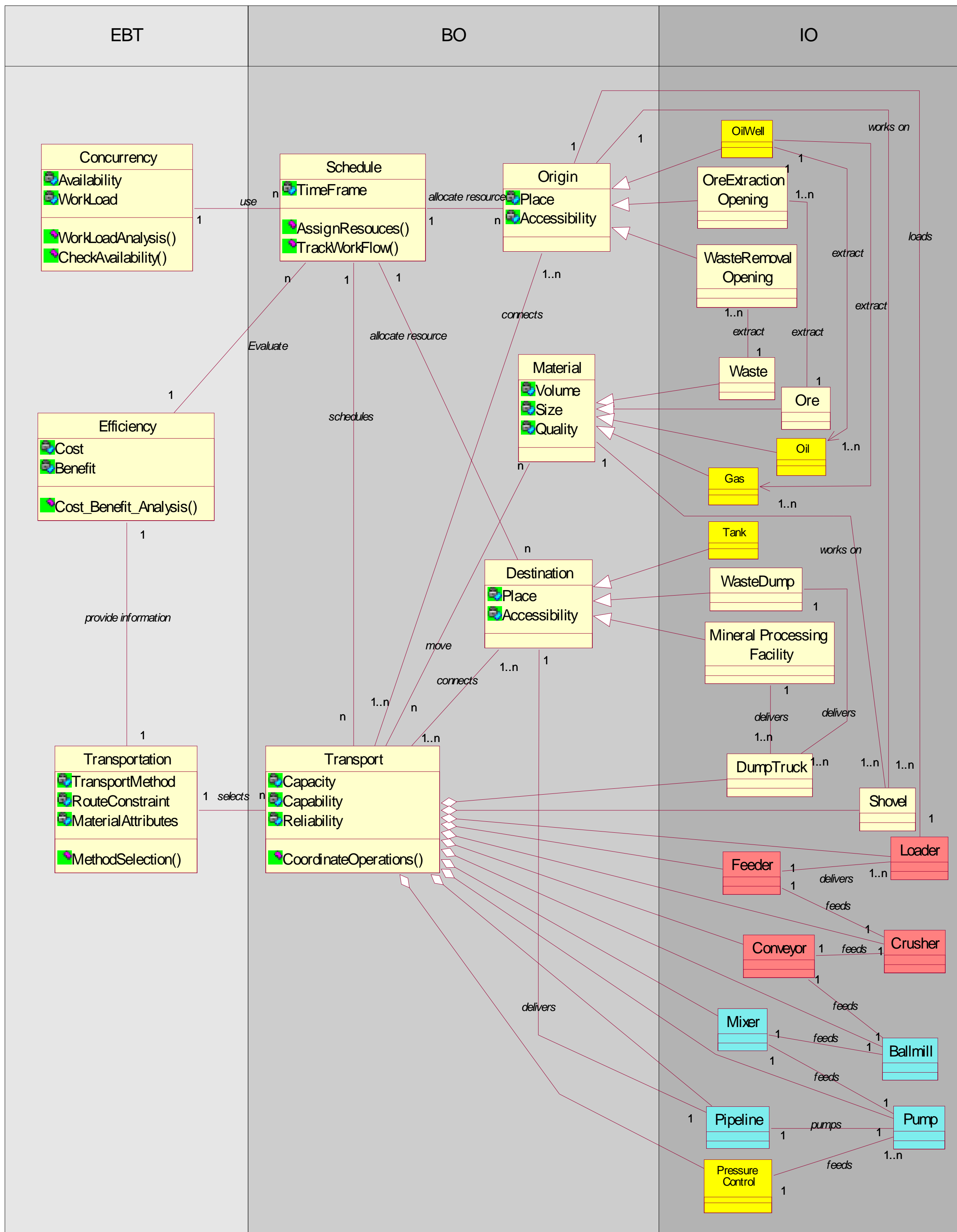


Figure 12. Class diagram of stability model for transport system

3. Conclusion

From this case study, we can see that conventional object-oriented modeling is subject to instability. Each change can prompt a re-engineering process. In contrast, the Stability Model remains well organized over the course of many design iterations. Ultimately, we believe this approach has tremendous cost benefits for software teams. The SSM is shown to be elegant and extensible and exhibits great stability through changes. It is through changes to the peripheral, industrial objects that we achieve balance between the need for changes and our goal of stability. Despite the various modifications to support various instances of the problem, our EBTs and BOs remain stable.

Conventional models are all built strictly of Industrial Objects. All of those models try to describe the problems and solutions by decomposing them into concrete objects and providing a mechanism to guide the subsequent programming process. Abstract classes are used only for reducing duplications. However, the abstract classes in the stability model, EBTs and BOs, are designed to describe the core purpose of the software product. The key to the stability modeling process is to "identify aspects of the environment in which the software will operate that will not change, and to cater the software to these areas„[3]. This concept exemplifies the essence of reuse. In conventional models, engineers often conclude their analysis when they have identified a solution to the problem. The stability model always tries to explicitly expose the core of the problem domain through EBTs and BOs. Consequently, resulting models will undoubtedly be more stable and reusable than conventional object-oriented models.

Conventional models are difficult to extend, compared to SSM. The re-engineering process involved in reusing a conventional model frequently involves more work than creating an entirely new model. We believe that the stability approach holds a promise to reduce or eliminate the costliness of the re-engineering cycles that have become commonplace in software engineering circles. "Object technology is all about managing complexity and being receptive to change„[4]. The software stability approach can be viewed as an essential refinement to existing object-oriented analysis and design processes. It provides a suitable approach to achieve model-based reuse. Our case study suggests that Software Stability Models are inherently adaptable and reusable.

4. References

[1] Tracz, W.ed., Software Reuse: Emerging Technology, *IEEE Press,* New York, 1988.

[2] Fayad, Mohamed and Altman, A. *An Introduction to Software Stability. Communications of the ACM,* Vol. 44, No. 9, September 2001.

[3] Fayad, Mohamed. *Accomplishing Software Stability. Communications of the ACM,* Vol. 45. No. 1, January 2002.

[4] Cary, James, Brent Carlson and Tim Graser. *SanFranciso design patterns: blueprints for business software*. Addison Wesley Longman, Inc. March 2000.

[5] Fayad, Mohamed, Shasha Wu, and Majid Nabavi. *Merging Multiple Traditional Models in one Stable Model. Communications of the ACM*, Vol. 45, No. 6, June 2002.

Model-based Software Reuse Using Stable Analysis Patterns

Haitham Hamza, Mohamed E. Fayad
Computer Science and Engineering Dept.
University of Nebraska-Lincoln
Lincoln, NE 68588, USA
{hhamza,fayad}@cse.unl.edu

Abstract

The challenge of building efficient reusable software artifacts is the focus of several schools of thought in software engineering. Software analysis patterns are recurring and reusable models. However, there are several deficiencies with analysis patterns. These deficiencies make it difficult to use analysis patterns as efficient reusable artifacts. This paper proposes eight essential properties to evaluate pattern reusability. In addition, the concept of Stability Analysis Patterns is introduced. This paper contrasts stable analysis patterns with some analysis patterns using the proposed properties.

1. Introduction

Since the inception of object-oriented concepts researchers and practitioners alike have held fast to the belief that reuse vastly improves the quality of software products, while simultaneously reducing cost and condensing lifecycles. Many reuse software communities have evolved in recent years, including Aspect-Oriented Programming (AOP), Component-Based Software Engineering community, and many others.

Analysis patterns are conceptual models that model the knowledge domain of the problem. Analysis patterns, as reusable artifacts have been widely heralded by the software engineering community as a major advance over conventional reuse techniques, and rightly so. However, analysis patterns have not realized their full potential. Analysis patterns are insufficiently mature to be considered as a base for building reusable software models. Understanding the cause of this immaturity is the first step in achieving real reuse of analysis patterns.

As models, analysis patterns must satisfy the six basic model properties introduced in [5]. That is, to be simple, complete and most probably accurate, testable, stable, to have visual representation, and to be easily understood. In addition to these six properties, reusable artifacts must satisfy two additional metrics: first to be general, and second to be easily and actually reused. Thus, a pattern that models a specific problem should be constructed so that it is easily reused whenever the problem occurs, and independent of the context in which the problem appears.

Software stability concepts introduced in [1] have demonstrated great promise in the area of software reuse and lifecycle improvement. Software stability models apply the concepts of "Enduring Business Themes" (EBTs) and "Business Objects" (BOs). These concepts have been shown to produce models that are both stable over time, and stable across various paradigm shifts within a domain or application context. By applying stability model concepts to the notion of analysis patterns we propose the concept of *Stable Analysis Patterns*. The idea behind the stable analysis patterns is to analyze the problem under consideration in terms of its EBTs and BOs with the goal of increased stability and broader reuse.

In the remainder of this paper we will introduce the eight essential properties of analysis patterns (Section 2), and discuss the different groups approaches for building analysis patterns (Section 3). We will study some example patterns reflecting each of the aforementioned groups (Section 4), and compare these approaches (Section 5). Conclusions are presented in Section 6.

2. Essential Properties of Analysis Patterns

In this section we examine the eight properties of efficient reusable models. Satisfying these eight properties does not guarantee an efficient reusable model; however, in practice, lacking any of these properties will affect the reusability of the model. These eight essential properties are:

1. Simple: a pattern is not intended to represent a model for a complete system; rather it models a specific problem that commonly appears within larger systems. Systems, by their nature, combine many problems. Thus, they are modeled using a collection of analysis patterns. In fact,

each analysis pattern should focus on one specific problem; otherwise, many problems arise. Without decomposing a system into components, models become unreasonably complex, the generality of the patterns are adversely affected, and the model becomes highly non-intuitive. If a pattern is used to model an overly broad portion of a system, the generality of resulting patterns is sacrificed – the maxim holds: the probability of the occurrence of all the problems together is less than the probability of the occurrence of each problem individually. For example, modeling the "payment" problem with "buying a car" is not effective since the "payment" problem may appear in unlimited number of problems. Pattern completeness is also sacrificed when we model a system at an improper level of resolution, because the analyst's focus is not on a specific problem, and it is likely that important feature of the system and its subcomponents will be overlooked.

2. Complete and most likely accurate: closely related to the concept of simplicity, this property guarantees that all the required information is present. In order to be considered complete the model should not omit any component. The model must be able to express the essential concepts of its properties. For example, trying to model the whole rental system of any property will force us to miss some of the parts of this system. Renting a car will involve something related to its insurance; however, renting a book from a library has nothing to do with the insurance problem. As a result, pattern that models the rental system, besides lacking the simplicity property, it will not be complete or accurate.

3. Testable: for the model to be testable, it must be specific and unambiguous. Thus, all the classes and the relationships of the model could be qualified and validated.

4. Stable: stability influences the reusability of a model. Stable models are easily adapted and modified without the risk of obsolescence.

5. Graphical or visual: conceptual models are difficult to visualize. Therefore, having a graphical representation for the model aids understanding it.

6. Easy to understand: Conceptual models are complex as they represent a high level of abstraction. Therefore, it is required for analysis patterns to be well described such that they aid in communicating an understanding of the system. Otherwise use of the pattern is neither attractive nor effective.

7. General: This property is essential to ensure model reusability. Pattern models lacking generality become useless, since analysts will tend to build new models rather than spending time and effort to adapt an unruly pattern to fit into an application. Generality means that a pattern that models a specific problem is easily used to model the same problem independent of context. Pattern generality may be divided into two categories: Patterns that solve problems that frequently appear in different contexts (domain-less patterns), and patterns that solve problems that frequently appear within specific contexts (domain-specific patterns). In the latter sense, the pattern is still considered to be general even if it is only applicable in a certain domain, but in this case, we should make sure that the problem that this pattern models dose not occur in other contexts.

8. Easy to use and reuse: analysis patterns should be presented in a clear way that makes them easily reused. It is important to remember that patterns are consumed in larger models. Patterns that are easy to use and designed for reuse stand a greater chance of actually being reused.

3. Classification of Analysis Pattern

One possible classification for analysis patterns is based on the construction approach. Generally, different building approaches categorize analysis patterns into three groups:

<u>Group I:</u> People in this group use their experience to build analysis patterns. Simply, patterns are produced during the course of specific projects. Since no one can be an expert in all fields, domain experts often produce domain specific patterns, even if the problem modeled occurs in many other contexts. People in this category believe that it is unsafe to further abstract patterns generated within certain projects in order to make them reusable in other contexts. They argue that the patterns resulted from extended debate and that the patterns have been tested and validated in the project. Therefore, there is no guarantee that these patterns will be successfully reused in other contexts.

<u>Group II:</u> People in this group use analogy to build their analysis patterns. According to this group, patterns that model complete systems in one context are reused by making an analogy between the pattern and the new application. Thus, by analogy, they change the names of the pattern's classes to be relevant to the new application. It is also possible to remove or add few classes to the pattern's model. Even though this group believes that analysis patterns should be built in a way the makes them reused to model the same problem regardless of its context. However, the way they choose to approach this goal makes them end up building templates rather than building patterns.

<u>Group III</u>, which is our group, our approach is based on the software stability concepts [2]. By analyzing the problem in terms of its EBTs and the BOs, the resultant pattern models the core knowledge of the problem. The goal of this approach is stability. As a result, these stable patterns could be used to model the same problem regardless its context.

4. Evaluation of Analysis Patterns Groups

In order to fairly compare the effectiveness of the patterns generated by the three different groups described in the previous section, a pattern that reflects the approach of each group will be examined against the essential properties mentioned earlier in section 2.

4.1 Group I

The *Account Pattern* provided by Fowler [5] is representative of this group. Figure 1 shows the class diagram of the *Account Pattern*. The purpose of this pattern is to provide a model for the "account" problem; thus, we can use this pattern to model banking account for instance. In fact, it was not long time ago when word account has been merely used to indicate banking and financial accounts. Today, the word account alone becomes a vague concept if it is not allied with a word related to a certain context. For instance, besides all the traditional well-known business and banking accounts, today we have e-mail accounts, on-line shopping accounts, on-line learning accounts, subscription account, and many others. As a result, using words such as balance and withdrawal while modeling the account, makes the use of the pattern to model accounts in different contexts, time and effort-consuming, if not, impossible. For instance, suppose we want to model an e-mail account using Martin's pattern, perhaps the most obvious changes are all the classes' behaviors, which are completely irrelevant to the email application.

From the simplicity point of view, Fowler's *Account Pattern* is not considered simple in the sense that it models two different problems at the same time. The first problem is the "account" problem and the second problem is the "entry" problem. In fact, these are two independent problems. Even though they used to appear together in many contexts; however, there is a possibility of having entries without an account, or having an account without entries. As a result, the generality of the pattern is limited. These factors contribute negatively to the reusability of the pattern.

Fowler's pattern is not complete in the sense that it lacks some of the "basic" concepts that appear frequently in banking accounts. For instance, suppose that we need to use this pattern to model a banking account. In banking accounts it is possible that two or more persons may be holders of the same account. Perhaps, there is a primary holder that has the full authorization to manage and control the account, while each of the other holders will have specific privileges for using the same account. Such situation cannot be handled while using Martin's account. Thus, Martin's pattern is not applicable to some of the usual financial and banking account situations. Since significant effort is required to adapt this pattern to other circumstances, the stability of the system is limited and the pattern structure is not stable over time.

This pattern is graphical in the sense that it has a graphical model that describes it (the class diagram). Having such a graphical representation will make the pattern visually testable.

Figure 1. Martin's Account pattern [5]

4.2 Group II

Figure 2 provides the class diagram of the *Resource Rental Pattern* [6]. The objective of the pattern is to provide a model that could be reused to model the problem of renting any resource. Figure 3 provides an example of the *Resource Rental Pattern* applied in the context of a library service [6]. Many examples for applying this pattern to different applications are suggested in [6].

This pattern models a complete resource rental system; thus, it models a collection of problems, whereas each of these problems could be modeled individually. For example, the "payment" process is a stand-alone problem, which could appear in many other contexts. Therefore, having a pattern that models the "payment" problem alone will be more effective since such pattern will be reused in many other applications.

The resource rental pattern lacks the simplicity. In addition, it is not general, because it is not applicable to any resource rental. One of the most basic steps in the automobile rental process is the question of insurance. There is nothing in this model that can be used to address insurance programs.

Another issue that is not addressed in this pattern, yet it is essential in many renting systems, is the verification process. In many cases, renting a resource might require a membership (as in the case of the universities library) or other identification (such as the driver license in the case of renting a car). There is a link between the completeness of the model and its stability. Reuse is challenging when applying incomplete patterns because many new classes are needed to complete the model.

In our example, suppose we need to model a car rental system using this pattern. Now we have reached the conclusion that we need to add the verification process classes and the insurance process classes. Substantial analysis is needed to complete this new model, hence the

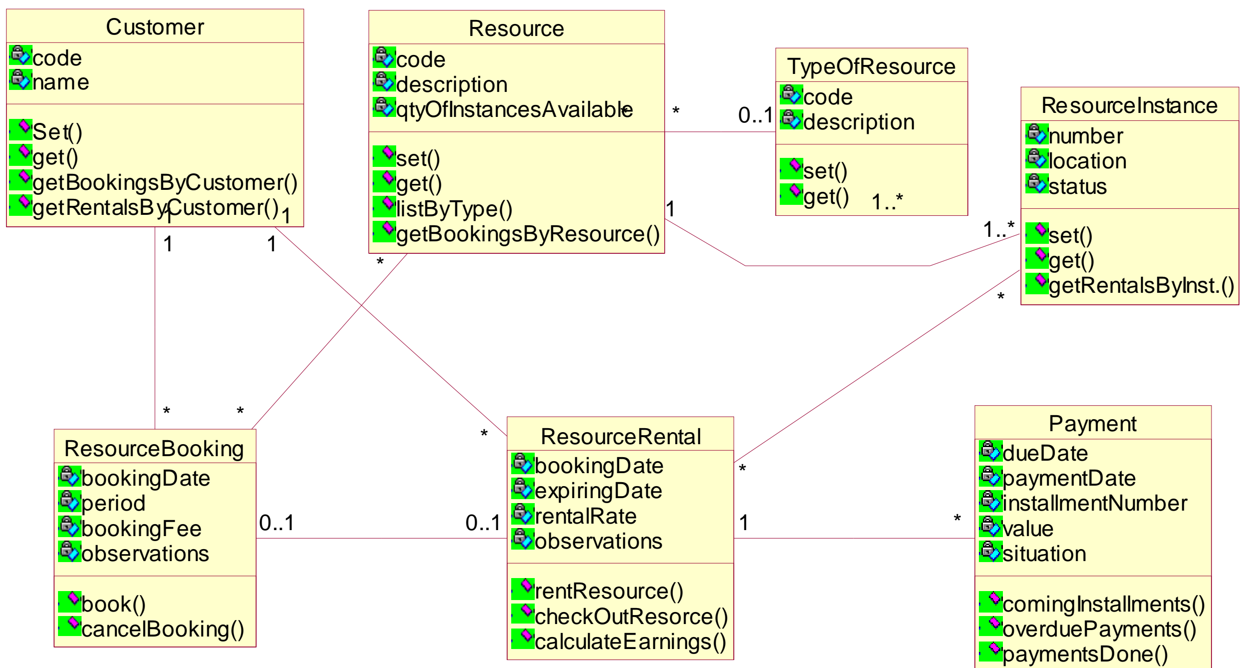


Figure 2. Resource rental pattern

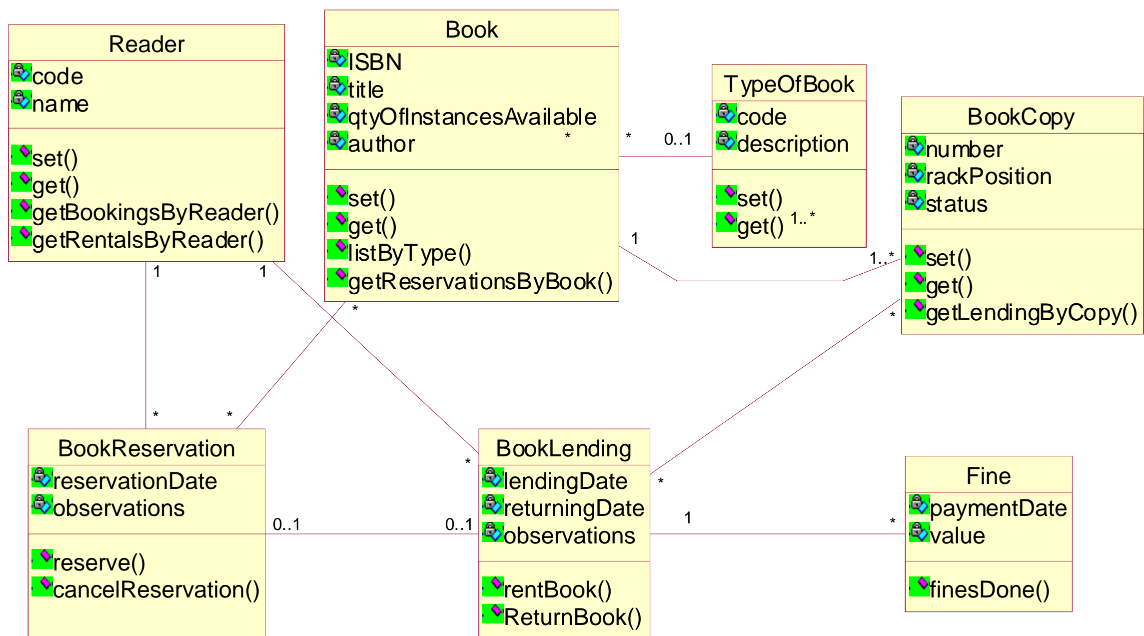


Figure 3. Instantiation of resource rental pattern for a library service [6]

analyst may be inclined to build a new model from scratch. Therefore, the pattern will not be reused.
As in the first group, the existence of a class diagram to represent the model makes the model graphical. As a result the pattern is readily tested.

4.3 Group III

This group reflects our proposed solution for building reusable analysis models and avoiding most of the common problems we found in the other groups. In order to make the evaluation of this group patterns more interesting, the pattern example that is chosen is the stability version of the *Account* pattern introduced by Fowler.

From the stability point of view, the model that focuses on the account problem has nothing to do with the entry problem. Thus, it is required to have different patterns for each individual problem. In this manner, the simplicity of our models is guaranteed since each pattern will focus on a specific problem.

Stability goes further by providing other classes that do not exist in Fowler's model. Figure 4 shows the proposed analysis pattern "*AnyAccount*" that provides the stable model for the "account" problem. The new classes that appear in the stability model help us to handle those circumstances that Fowler's model fails to cover; thus, the model becomes accurate and complete. For instance, the use of the EBT of "Ownership" and the BO of "Holder" in the modeling of the account aids in contexts where there is a difference between the account owner, and those who are authorized to use the account under certain rules, should made clear.

"Ownership" is an enduring concept, which will never change independent of context. On the other hand, the "Holder" here is externally stable and never change with time, although the holders of the account could internally change (holder may get ill, for example); however, they are still the holder of the account. As we can see in the pattern class diagram, the inherited objects from the "Holder" object model the different roles of the different levels of the usability of that account. This pattern's structure is stable over the time and general enough to handle different applications that involve accounts and the different situations within the same application as well.

On the other hand, thinking about "entry" as a stand-alone problem forces us to build a pattern that models any entry regardless of context. Using the stability concepts we were able to come up with a stable pattern the models any entry for any application. This pattern is called "*AnyEntry*" pattern and its class diagram is given in Figure 5.

By combining the pattern that models the account problem (the "*AnyAccount*" pattern) with that which models the entry problem (the "*AnyEntry*" pattern) we can demonstrate the ease of reusing stability models to construct comprehensive models. Figure 6 shows the class diagram for this third pattern. The "*AccountWithEntry*" pattern could be used to model any account that has entries associated with it, as in the case of banking accounts and email accounts for example.

5. Comparison of Analysis Patterns Groups

Based on the essential properties discussed in section 2, table 1 summarizes the results of the three analysis pattern groups.

6. Conclusion

Analysis patterns could form a foundation for building reusable software assets. However, this evaluation of some analysis patterns show that these patterns lack many essential properties. As a result, their reusability is diminished. Software stability has been proposed as a solution for the deficiencies encountered in analysis patterns. ***Stable analysis patterns*** demonstrate effectiveness by satisfying all the proposed properties. Therefore, the application of stable analysis patterns is a promising approach meriting further research among the reuse communities.

7. References

[1] M. E. Fayad, A. Altman, "Introduction to Software Stability", *Communications of the ACM*, Vol. 44, No. 9, September 2001.

[2] M. E. Fayad, "*Accomplishing Software Stability'*,, *Communications of the ACM*, Vol. 45, No. 1, January 2002.

[3] M. E. Fayad, "*How to Deal with Software Stability'*, *Communications of the ACM*, Vol. 45, No. 4, April 2002.

[4] M. E. Fayad and M. Laitinen, "Transition to Object-Oriented Software Developments", New York: Wiley & Sons, August 1998.

[5] M. Fowler, "Analysis Patterns: Reusable Object Models", Addison-Wesley, 1997.

[6] R. T. Vaccare Braga et. al., "A Confederation of Patterns for Business resource Management" *Proceedings of Pattern Language of Programs 98 (PLOP 98),* 1998.

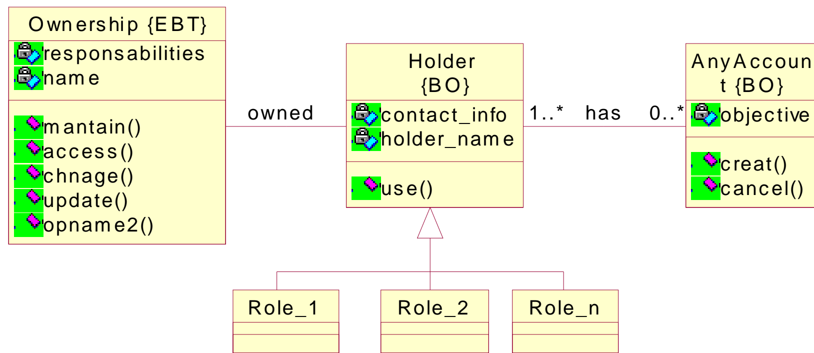


Figure 4. *AnyAccount* pattern class diagram

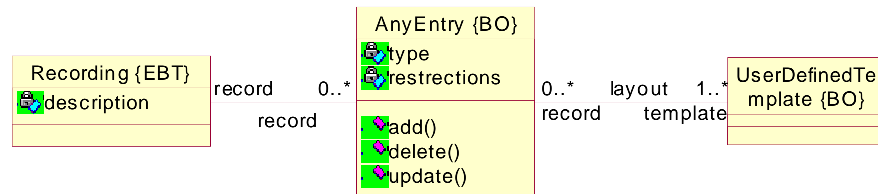


Figure 5. *AnyEntry* pattern class diagram

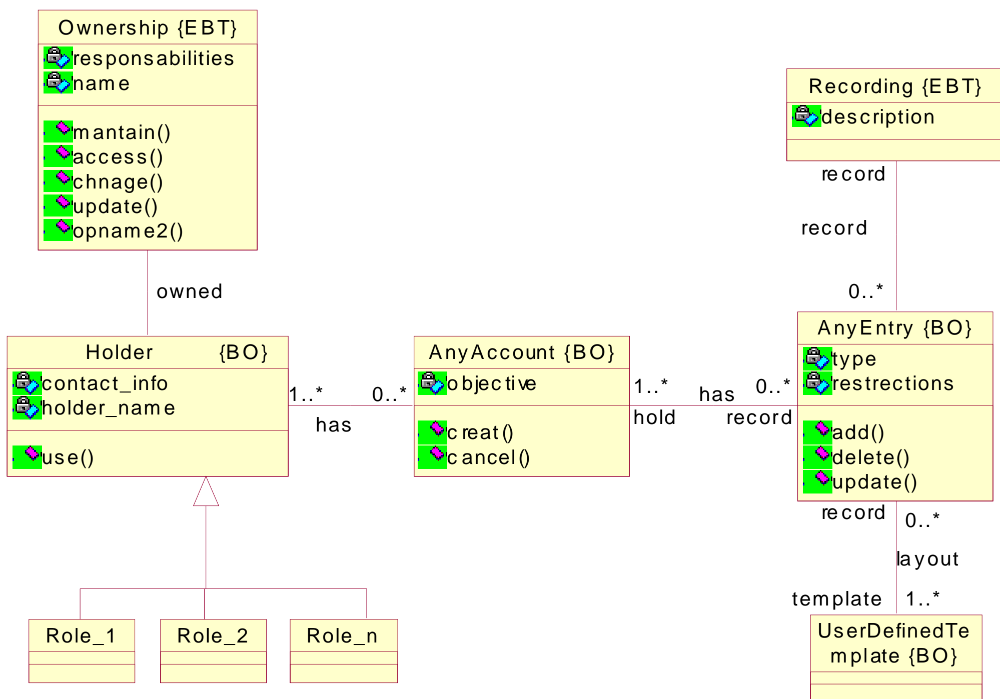


Figure 6. *AccountWithEntry* pattern

Table 1. Comparison of analysis patterns groups

	Evaluation		
Properties	**Group I**	**Group II**	**Our Group**
Simple	**No.** It models two different problems.	**No.** It models an entire system.	**Yes.** Each model focuses on one problem.
Complete and most likely accurate	**No.** It does not cover all the circumstances that might occur in the application.	**No.** It is not sufficiently general to address the requirements of different renting applications.	**Yes.** Because each model focuses on a specific problem, all situations within the problem can be easily covered.
Stable	**No.** This pattern cannot model all types of today's accounts. Thus, we will always need to do major changes to reuse this pattern for different applications.	**No.** Using this pattern to model different application will need major changes. For instance, adding the verification process to the model will need a lot of changes.	**Yes.** The patterns in this group are built with stability in mind. The use of EBTs and BOs, ensure stability in the model.
Testable	**Yes.** Since the pattern can be visualized; thus, we can, at least, visually test it.	**Yes.** Since the pattern can be visualized; thus, we can, at least, visually test it.	**Yes.** Since the pattern can be visualized; thus, we can, at least, visually test it.
Easy to understand	**Yes.** Generally speaking, despite the accuracy of the pattern, it is easy to understand its structure.	**Yes.** Despite the accuracy of the pattern, it is easy to understand its structure.	**Yes.** Since the used EBTs and the BOs reflect the concepts that we are familiars with; it is easy to understand the model structure.
Graphical or visual	**Yes.** The pattern has a graphical presentation, which is the class diagram.	**Yes.** The pattern has a graphical presentation, which is the class diagram.	**Yes.** The pattern has a graphical presentation, which is the class diagram.
General	**No.** We cannot use it to model the account in other contexts other than monetary application. Also, it dose not cover the cases of having accounts without entries and vise versa. Moreover, the pattern dose not cover some of the situations such as having more than one holder for the same account.	**No.** This pattern cannot be used to model the rental of some resources. For instance car rental, since there is nothing in the model that cover the insurance issues, which is an essential part of any cat rental process.	**Yes.** Because of the stability concept, our models focus in a specific problem trying to flush the core knowledge underneath the surface of the problem. Since the core knowledge of any problem is constant regardless the context that this problem might appear in, the model of the problem is general and can apply to the problem whenever it occur.
Easy to use and reuse	**No.** Using the patterns of this group in different application that they originally built for, if it possible is not an easy task.	**No.** As we mentioned before, we will need to do major changes to use this pattern in different application such as for car renting.	**Yes.** Using the pattern by itself or the integration of few patterns are both easy to be done. This property is demonstrated by introducing the third pattern shown in figure 6.

Modelling Component Libraries for Reuse and Evolution

Miro Casanova and Ragnhild Van Der Straeten
System and Software Engineering Lab, Vrije Universiteit Brussel
Pleinlaan 2
Brussels, Belgium
Email: {mcasanov|rvdstrae}@vub.ac.be

Abstract

If we want to compose components which were used to build a certain software application, with other components to develop a new application, we lack the necessary knowledge to reuse these components. The research on software libraries has improved reuse. Our goal is to classify software components in libraries using a multi-dimensional approach supporting reuse as well as evolution. For this purpose, ontologies will be used capturing the structure of software component libraries, Description Logic will be used to build these ontologies and the component and composition patterns approach [VW01] [WV01] will be used to support reuse. This will provide the developer with an extended support to develop an application using components and with support to manage and maintain the software component libraries.

1 Introduction

Component-based software development (CBSD) is one of the major efforts for improving reusability and maintainability of software applications. A component was defined at ECOOP 96 [SP97] as follows: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.* The great advantage of component-based software development is that new software can be built by combining bought and self-made components. To do this, it is necessary to specify in which contexts the component can be used and how it will behave. This specification should enable the user of a component to determine whether it can be used in a particular case.

In [BJ99] a component specification consists of four levels, namely the syntactic or API level, the behavioural or semantic level, the synchronization level and the quality of service level. In relation to the previous, in our lab a component composition environment PacoSuite [VW01] [WV01] has been developed which supports the development of applications through the visual composition of software components. The components are documented with *usage scenarios* and *component composition patterns*. Both kinds of documentation make use of an extension of Message Sequence Charts (MSC). These diagrams describe typical role interactions of the components in a similar way as the interaction of objects is expressed in a UML interaction diagram. Component composition patterns are high level descriptions of cooperations between several roles without any indication on how this cooperation will be implemented. The same diagrams are used to

model typical interactions of a component with its environment, i.e. usage scenario's. Based on that documentation automated compatibility checks are performed using finite automata. That research focuses on the synchronization level.

On the other hand, an important effort for improving reusability of software is the research done on software libraries. A software library is defined as a "managed collection of software assets where assets can be stored, retrieved and browsed" [AM]. An important improvement in software libraries is the use of the faceted classification approach. A facet is a "clearly defined, mutually exclusive, and collectively exhaustive aspects, properties or characteristics of a class or specific subject" [T92]. Each facet consists of several terms which describe the concrete values the facet can have.

In this paper we will bring together the aforementioned research on CBSD and the research on software libraries. We want to classify components (in the sense of CBSD) in software libraries by using the faceted approach. We will mainly focus on the semantic level by using ontologies. These ontologies will represent a multi-dimensional classification of components and will capture the behaviour of components. We envision this research as a way of improving reuse and evolution of component-based applications.

In section 2 the approach we follow will be explained. In section 3 we focus on the functionality and reuse dimension which are part of the ontology for generic components that we have developed. In section 4 we show how some components of a small application are classified using the presented approach. Finally, in section 5 we conclude and give some future work.

2 Our approach

We observe that software libraries using a faceted approach have some limitations:

- The description of the behaviour of the software artifacts, that are to be stored in the libraries, is possible if and only if they are fine-grained. However, we can have multi-behaviour components offering several services.
- Another complication appears whenever we try to classify components having state. In this case, the behaviour of a component can depend on the state the component has, which gives multiple ways of classifying the same component.
- If two software libraries of related domains are to be merged, currently, the only way of doing it is to manually add components one by one from one library to the other. This implies that the user must give the specifications for every component, which is a long and error-prone process. The best case is when in both libraries the same terminology is used, i.e. the same facets and terms are used. Otherwise, the user must also interpret each of those facets and terms and map them into concepts of the other library.

It is our intention to classify components in software libraries by using the faceted approach [PD91] [PF87]. This allows to have a multi-dimensional classification of components. We define a dimension as a set of facets that are related to the same view or the same aspect of a component. Different dimensions should be considered in the classification. For instance the functionality (what it does, inputs, outputs, etc.) of a component, the knowledge of its past uses (in which systems it has been used, with which components it has collaborated, with which composition patterns it has been used, etc.), the implementation issues (programming language, platform, etc.) and so on.

Furthermore, next to the different dimensions, also the relations among those dimensions (i.e. interdimensional relations) and the relations within one dimension (i.e. intradimensional relations) should be considered. As an example of

an interdimensional relation consider a network component sending packages to another component. Depending on the platform the performance of the component could be different. As an example of an intradimensional relation consider a component whose platform must have another value depending on the programming language (e.g. in Java, components should work on any platform). Our goal is to improve reuse and evolution in the development process of component-based systems. However, in this paper we will focus mainly on the reuse aspect.

2.1 Application-Specific Ontologies

To overcome the difficulties encountered in software libraries, we will describe the different facets of a component and their possible values together with the relations that exist between them in the same or another dimension. In other words, we want to create a layer that defines the structure of the component library. Ontologies describing the structure of the multi-dimensional classification of components will be constructed using an ontology language based on a Description Logic (DL). These ontologies will define the facets, terms, dimensions, inter- and intra-dimensional relations, etc. of the different components. The way this ontologies are defined depend largely upon the kind of application that can be constructed with these components. When using a component, the developer acquires knowledge about its use. When building new applications with that component the developer wants to be able to apply this knowledge. Component libraries are built based on such ontologies. The advantages of having such ontologies are:

- The insertion of the components in the library becomes easier because of the explicit presence of the relations. Given a term of a facet, other terms can be automatically derived due to those relationships.
- Different libraries based on the same ontology can be easily merged.
- Different libraries based on different ontologies can also be merged. In this case, first the ontologies should be merged. Some research is already done in this area [NM99], however it is still very preliminary. After merging the ontologies, the libraries can be merged based on the new ontology.
- Using the ontology, smart queries can be executed on the different software libraries.

2.2 Description Logic

The family of Description Logics originate from knowledge representation research in Artificial Intelligence. Their main strength comes from the different reasoning mechanisms they offer. The complexity of reasoning in these different languages is and has been widely investigated.

The basic elements of a Description Logic are *concepts* and *roles*. A *concept* denotes a set of individuals, a *role* denotes a binary relation between individuals. Arbitrary concepts and roles are formed starting from a set of atomic concepts and atomic roles applying concept and role constructors.

An ontology language will be developed based on the Description Logic $\mathcal{Q} - \mathcal{SHIQ}$ [CL02]. The advantages of using Description Logic to build these ontologies are:

- Concepts can be easily composed to form new concepts.
- DL allows for arbitrary binary relations which enables the expression of the different relations between the components, their terms, facets and the dimensions.
- DL offers efficient reasoning support. This support can be used to reason about and to query the constructed ontology.

We will use the OIL ontology language [FH00] for the examples shown in this paper. This is a

language for specifying and exchanging ontologies. It is based on the DL $\mathcal{SHIQ}$ [HS01], on frame-based systems and on the web standards XML and RDF.

3 Functionality & Reuse Dimension

An ontology defining the structure of a software library for generic components has been defined. For space restrictions, we focus only on the two most relevant dimensions for reuse: the functionality and the reuse dimensions. As we have said, a dimension is composed by a set of related facets. A facet in our case can have more than one term associated. The reason for this is mainly that components are intrinsically reusable entities, and in consequence, they can be used in different contexts for different purposes. Therefore they can have several functionalities, different contexts where they can be used, different provided interfaces, and so on. Thus in order to specify this multi-context behaviour using the faceted approach, it is necessary that facets have the possibility of being linked to many terms. The specification of these dimensions has been figured out by inspecting and analyzing component-based applications and their components. We have to mention that this specification is still evolving.

Two kinds of facets are present: the ones that are useful for classifying components, and the ones that are not useful for classifying but for documenting. An example of the first kind is the facets that belong to the *Functionality* dimension, and of the second kind the facet *Protocol* of the *Reuse* dimension. Although the second kind is not useful for classifying, it is indeed useful for reusing components in systems that are being created, maintained or in systems that evolve.

3.1 Functionality

Every facet that has to do with the functionality part of the component belongs to this dimension. The components in this dimension are to be classified by their behaviour. This behaviour can be multiple due to the fact that a component can perform several actions. Some facets of this dimension are:

1. Actions: the different functions the component can perform. Some terms of this facet are:
   - Add : append, prepend, insert
   - Remove : delete last, delete first, delete any
   - Link
     - Out : reference, subscribe, unsubscribe, connect, disconnect
     - In : referenced, subscribed, unsubscribed, connected, disconnected
   - Display
   - Data : send, receive, set, get, notify, stream, answer
   - Calculation : sort, search, perform_specific,...
2. Inputs: arguments needed to perform some action.
   - String
   - Number: integer, float, ...
   - None
   - ...
3. Outputs: results of the performed action.
   - String
   - Number: integer, float, ...
   - None
   - ...
4. Medium: entities that are locales where the action is performed.
   - Dictionary

- Stack
- List
- ...

5. Kind: Information about the kind or type of the component.

   - Gui: button,console,...
   - Data Structure: stack, list, tree, ...
   - Algorithm: sorting, searching,...
   - Network: client, server,...
   - ...

A hierarchy of terms has been created which structures all the possible values that a facet can have. The hierarchy presented above has been thought for classifying general purpose components. For classifying components that are specific of a given domain, other facets and terms (described in another ontology) should be used. There are also some relations between the terms of this dimension, but they will be described in 3.3.

3.2 Reuse

In this dimension the information that is useful at the moment of reusing the component in a given application is stored. The knowledge of past experiences (past uses) of the component is stored. The following are the 4 facets of this dimension:

1. Environments: Names of composition patterns in which the component has been previously used.
2. Protocol: MSC describing its protocol.
3. Related components: components that have been frequently used together with the component to be reused.
4. Related Systems: systems in which the component has been previously used.

Notice that only the third and fourth facets are useful for classifying. Indeed, the components can be grouped together following the values of those facets. On the other hand, the first and second facets are just for documenting. In other words, it is not possible to classify components by their corresponding MSCs because they are just drawings describing the manners that the components can be used.

3.3 Relations

Some relations have been found while defining the dimensions. Some of them are intradimensional and some interdimensional. For instance, there is an implicit relation between two terms of the functionality dimension, namely *Gui* and *Display*. Indeed, if the *Display* term is set, then there must be set also a term *Gui*, or vice versa. The relation can be described in $\mathcal{SHIQ}$ as follows:

$$\mathit{Functionality} \sqsubseteq$$

$$\forall \mathit{has_kind.Gui} \sqcup \neg\forall \mathit{has_actions.Display}$$

The relations can be helpful for automating part of the process of classifying a component into a software library. The selection of a particular term in a facet can trigger the automatic selection of other terms in other facets either in the same dimension or other dimensions or both. The previous process enforces the selection of some terms by inferring them from the specified relations, thus, preventing the user for making useless effort.

Both inter- and intradimensional relations can act on two levels: on the level of the ontology and on the level of the component (that must be specified by the user). The former is part of the definition of the structure of the library and indicates the relations of facets and terms that any component of the domain has. The latter is the relations that are inherent to a particular component. For instance, a typical relation in this level is when a given action in the functionality dimension is

linked to some particular inputs and outputs. The example given above belongs to the first level. An example of the second level of relations will be given in the following section.

4 Example

In this section we present how some components of a small application have been classified using the presented approach. This application is a scrabble game which has two players (Master and Slave) that communicate by means of two Network components that are connected with each other. The scrabble user interface component contains the interface of the game together with the logic of the program, the spelling checker receives a word and checks whether it corresponds to a valid English word, the Network component communicates with another Network component, and the Java button displays a button on the screen while it is waiting for a click to send a signal to another component in order to perform a given action.

The mentioned application is composed of the following components and composition patterns (i.e. the way the components communicate with each other) on the Master side:

- Scrabble user interface (ScrabbleGUI)
- Network client (Network)
- Spelling checker (Dictionary)
- Game_Master: composition pattern

On the Client side, it has in addition:

- Standard Java button (JButton)
- Game_Slave: composition pattern

As it can be seen, there are three components that are reused in both the client and server side: ScrabbleGUI, Network and Dictionary. We describe only these 3 components because we focus on reuse.

1. ScrabbleGUI
    - Actions: display, perform_specific, subscribed
    - Inputs: none
    - Outputs: none
    - Medium: none
2. Network
    - Actions: connect, disconnect, receive, send
    - Inputs: none
    - Outputs: none
    - Medium: none
3. Dictionary
    - Actions: search, answer
    - Inputs: string
    - Outputs: string
    - Medium: dictionary

By looking at this example (in particular to the component Dictionary), the necessity of having a relation among the terms of *Actions*, *Inputs*, *Outputs* and *Medium* arises. In this case, the relationship must state that the action *search* looks up a *string* (the input) in a *dictionary* and returns another *string* (the output). This is a relation at the level of the component as mentioned in 3.3. In figure 1 the expression of the relation is shown in an OIL editor. It is our aim to provide the user with a simple ontology language to enable him to write relations down in an easy way. This implies that the user does not have to know anything about DL.

As this ontology is defined only for generic components, then the action that is performed by a very specific component such as ScrabbleGUI is defined as ”perform_specific”. If a more fine-grained classification is needed, then another ontology (in this case it can be some kind of ”Game Ontology”) can be added to the software library.

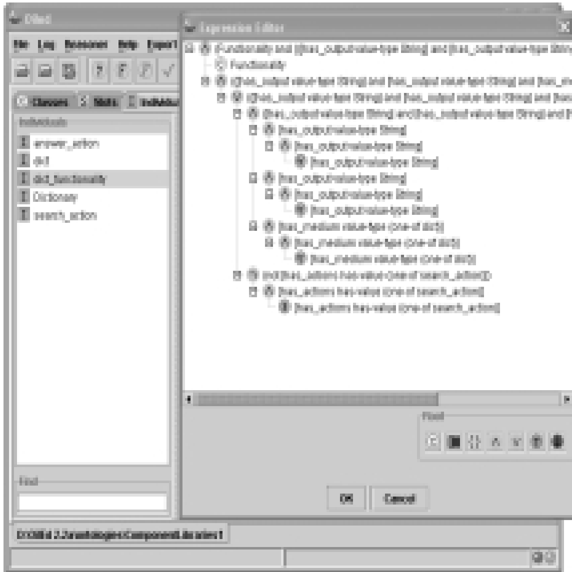


Figure 1. The relation written in OIL

The classification of a component looks as a very time-consuming process. Nevertheless we think that the previous is normal given the potential complexity of the component itself. What it is possible to do (and that is our intention) is to provide the developer with tool support for making both the classification of components and their (re)use as simple as possible, using the reasoning capabilities of DL.

5 Conclusions & Future Work

Our approach brings together several other works on software libraries, DL, ontologies and CBSD. One of our major goals is to support the software developer with a component software library for improving reuse and evolution. DL will be used for describing ontologies containing the structure of the software library (dimensions, facets, terms and relations). In this software library multi-purpose components can be classified, in contrast with other libraries that only classify atomic software artifacts (such as "input-output functions"). However, as the components to be classified can have several functionalities, its classification process becomes more difficult. By having relations, which link terms at the level of the ontology and component, the mentioned process can be semi-automated (thanks to the reasoning support offered by DL) for making it as simple as possible.

A scheme of an ontology for general-purpose components has been presented with an example of an application that follows the ideas in [WV01] and [VW01]. Also, it has been explained that if other kinds of components (some components that belong to a specific domain) are to be classified, a domain specific ontology, which specifies another dimension-facet-term-relation structure, can be used. This makes possible to have several domain-specific software libraries depending on the context in which the components they store are is used. Furthermore if there are two software libraries of domains that are very similar, they can be merged by merging their corresponding ontologies.

In the future we envision also the use of our approach for improving evolution of component-based applications. It is possible to classify and group "similar" components, i.e. components that can be interchanged in an application (when maintaining or evolving it) with a relatively low effort.

Our aim is also to provide tool support. A generic software library should be built structured only by the general purpose ontology presented above. Other more specific ontologies should be defined as well for creating the mentioned domain-specific software libraries just by plugging them in.

Another work to do is the support for components that have different functionality depending on their state. We have defined the ontology for generic components (a part has been presented in this paper) but the issue of the state of the components has not been addressed so far.

References

[SP97] Szyperski, C., & Pfister, C. (1997). Workshop on Component-Oriented Programming, Summary. In MuhlHauser M. (Ed.) Special Issues in Object-Oriented Programming - ECOOP96 Workshop Reader. Dpunkt Verlag, Heidelberg.

[BJ99] Beugnard, J. Jezequel, N. Plouzeau, D. Watkins. ”Making Components Contracts Aware”. IEEE Computer, July 1999.

[WV01] Wydaeghe, B and Vanderperren, W. ”Towards a New Component Composition Process”. Proceedings of ECBS 2001, Washington, USA, April 2001.

[VW01] Wydaeghe, B and Vandeperren, W. ”Visual Component Composition Using Composition Patterns”. Proceedings of Tools 2001, Santa Barbara, USA, July 2001

[PD91] R. Prieto-Diaz. ”Implementing Faceted classification for software reuse”. Communications of the Acm. May 1991.

[PF87] R. Prieto-Diaz and Peter Freeman. ”Classifying software for reuse”. IEEE Software, 4(1):6-16, January 1987.

[AM] S. Atkinson, A. Mili. ”Software Libraries”. http://citeseer .nj.nec.com/17413.html

[T92] Taylor A. ”Introduction to Cataloging and Classification”. 8th ed. Englewood, Colorado: Libraries Unlimited, 1992.

[NM99] N. F. Noy & M. A. Musen. An Algorithm for Merging and Aligning Ontologies: Automation and Tool Support. Sixteenth National Conference on Artificial Intelligence (AAAI-99), Workshop on Ontology Management, Orlando, FL, . 1999.

[CL02] C. Lutz. Adding Numbers to the $\mathcal{SHIQ}$ Description Logic—First Results. To appear in Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002). Morgan Kaufman. Toulouse, France. 2002.

[FH00] D. Fensel, I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann & M. Klein. OIL in a Nutshell. Proceedings of the 12th European Workshop on Knowledge Acquisition, Modeling, and Management (EKAW’00). Springer-Verlag. Juan-les-Pins, France. October 2000.

[HS01] I. Horrocks and U. Sattler. Ontology Reasoning in the SHOQ(D) Description Logic. Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence. Seattle, USA. August 2001.

Modeling With Components
– Towards a Unified Component Meta Model –

Uwe Rastofer
method park Software AG, Erlangen, Germany
Uwe.Rastofer@methodpark.de
and
Department of Computer Science 4 (Distributed Systems and Operating Systems),
University of Erlangen-Nuremberg, Germany
Uwe.Rastofer@informatik.uni-erlangen.de

Abstract

Despite their popularity, object-oriented technologies have generally failed to reuse assets across multiple projects. By contrast, components have recently become a promising reuse technology. But current component technologies address the problems of reuse and composition at the programming level only. What is needed is an integration of components into a modeling paradigm. First and foremost a model that is capable of modeling the various existing component models – a component meta model – has to be designed.

In our work we develop a meta model that unifies the basic features of the well-known component models. We shown that the meta model is complete by defining it with the help of its own features. This also makes it its own meta model and terminates the meta-level hierarchy.

To be able to describe the features of existing component models the minimal meta model has to be extended. For two well-known component models we show how this is done. We also provide mappings between the meta model and the component models to enable the reuse of modeling information. Finally we demonstrate how easily an extension for a behavior model can be integrated into our modeling framework.

1 Introduction

Object-oriented models, especially the Unified Modeling Language (UML) [1], are nowadays the most popular modeling paradigms. In spite of their success, object-oriented design and programming have a number of shortcomings that limit their applicability to large and complex systems [2]. Methods and interfaces are used to specify the services a class offers. But the required services, that a class needs to fulfil its tasks, do not have to be made explicit. They are buried in method calls and references inside the class code. Furthermore it is impossible for a class to offer the same service more than once because adding the same method or implementing the same interface again makes no difference. But probably the most severe problem is implementation inheritance, which is the sole mechanism for reuse in object-orientation. It breaks encapsulation boundaries and therefore allows uncontrolled modifications of classes. Aggregation could be used to achieve the same goals as inheritance in a more restricted way, but classes cannot be composed from other classes. Although multiple inheritance at first looks similar to class composition it is a different concept indeed. By and large, object-oriented design and programming have failed to provide the necessary mechanisms for large-scale reuse. Therefore it is reasonable to search for other more suitable meta-modeling paradigms.

In recent years component technologies have been successfully applied to ease the problem of reuse at the programming level [3]. They allow independently created components to be composed into complete applications. Similar to the evolution of object-oriented technologies, a shift is currently happening from using ad-hoc mechanisms at the programming level to modeling and designing with components. The emerging discipline of Component-Based Software Engineering aims at formalizing component-based systems.

In our work we contribute a meta model that is not only capable of describing various component models but also is a component-based model itself. We expect that component-based models exhibits the same reuse potential and ease of integration of independent model developments that have been observed in component-oriented programming.

This position paper is organized as follows. Section 2 describes a simple meta model for component-based systems and a hierarchical extension of this meta model. In Section 3 the meta model is used to model real component models. Finally an independently developed extension is integrated into a component model in Section 4.

2 A Component Meta Model

Meta-level Architecture. Modelers of any kind of system are confronted with different models at different levels of abstraction. In our case modeling component instances leads to components themselves. On the next level abstracting from components leads to a model of what the allowed features of the different components are. This meta model is often called a component model. Examples for models at this level are formal specifications of the various component technologies like JavaBeans [4], COM [5], and the CORBA Component Model [6]. The step of meta modeling could be repeated indefinitely but it is desirable to terminate the chain of meta-levels at some point. In our case we look for a meta model that can describe any component model.

Normally each meta level is an instantiation of a model at a level above it. For the sake of simplicity we assume that the resulting meta-level architecture is layered as shown in Figure 1, even though there are valid arguments for a nested architecture [7].

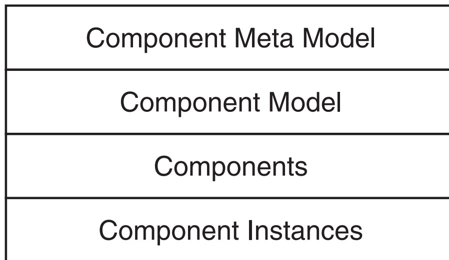


Figure 1. Layered meta-level architecture

Modeling Notation. Figure 2 illustrates the graphical notation that will be used for visualizing models. *Components* are named boxes that contain a list of *Properties*. The small filled squares at the sides of the components are *Provided Ports* - open squares are *Required Ports*. Ports also have a name and an associated type. In addition each port specifies the number of connectors that may be attached to that port[1]. A *Connector* is drawn as a solid line between two ports.

Simple Component Meta Model. While designing the component meta model we looked at the features that are

[1]Readers familiar with UML may find this notation confusing because connectors with connector multiplicities look like UML associations with their association end multiplicities interchanged.

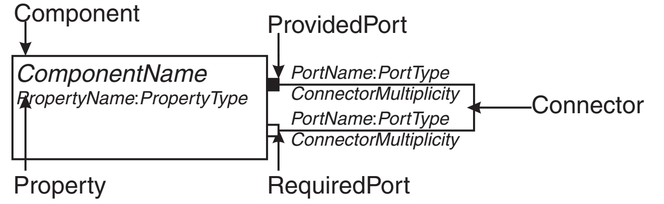


Figure 2. Component modeling notation

common to various component models and can be subsumed under the same abstraction. Every model needs an abstraction that holds together the elements that belong to that model. This is the *Model* component. It provides ports to hold all Components and all Connectors within that model. The Model like all other model elements has a name property.

Each Component has a number of properties that can be associated with it. A Property in turn requires exactly one Component to be connected to it. The same pattern applies to Provided Ports and Required Ports. A Provided Port denotes a service that the Component provides to other components whereas a Required Port shows that the Component needs this service to fulfil its tasks. In addition to its name, each Port also specifies a multiplicity for the number of Connectors that may be connected to this port. Whether a connection is possible or not is determined by the Port Type. If the Port Type of the Required Port is the same or more general than the one of the Provided Port, the two ports may be connected. Property Types follow the same rules for the substitution of properties in compatible components. Figure 3 shows the resulting simple meta model. Note that the model is both complete and minimal, i.e. it needs all model elements to describe itself.

Conformance. To be able to substitute one component for another it is necessary to define a conformance relation between components. If component A conforms to component B (in short: $A \supseteq B$) then A can be used in any place where B has been used. A conforms to B if all of the following conditions hold:

- $\forall p \in B.providedPort, \exists q \in A.providedPort :$
  $p.name = q.name \wedge$
  $p.multiplicity \leq q.multiplicity \wedge$
  $p.portType \subseteq q.portType$
- $\forall r \in A.requiredPort, \exists s \in B.requiredPort :$
  $r.name = s.name \wedge$
  $r.multiplicity \leq s.multiplicity \wedge$
  $r.portType \subseteq s.portType$
- $\forall p \in B.property, \exists q \in A.property :$
  $p.name = q.name \wedge$
  $q.propertyType \subseteq p.propertyType$

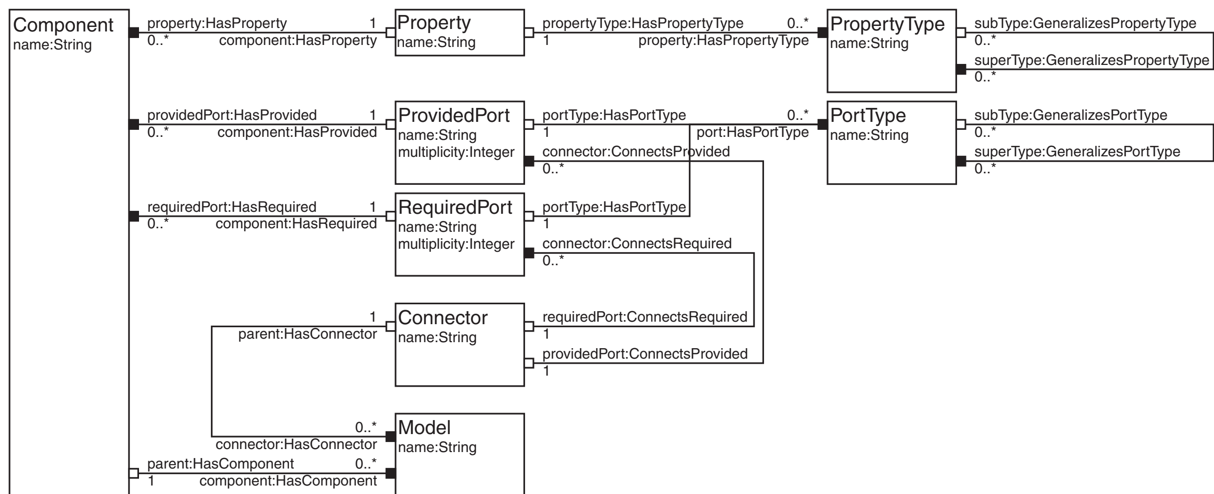


Figure 3. Simple component meta model

The relation $p.type \subseteq q.type$ means that the type of q is the same or more general than the one of p. This relation is defined via the *subType* and *superType* ports on the Property Type and Port Type components.

Obeying the conformance rules allows us to design extensions to the current meta model.

Adding a Hierarchy. For a large number of components the simple meta model becomes rather complex and confused. A well-known technique to conquer this complexity is the introduction of a hierarchy.

In our meta model a component that contains other components is called a *Composite Component*. It contains a component of Component and a component of Model. To allow the composite component to map its external ports to the ports of the internal components special relay ports are added. These conform to their non-relay counterparts. Therefore, Composite Component conforms to both Component and Model. In addition, through their internal connector ports the *Provided Relay Port* acts internally as a Required Port and the *Required Relay Port* as a Provided Port in turn. A similar mapping can be designed for Properties but has not yet been integrated. Figure 4 shows the extended meta model.

If all relay ports and the composite component are removed from Figure 4 it yields the same structure as in Figure 3. Indeed, it can be observed that each hierarchical model can be transformed into a non-hierarchical one that has the same connector structure between the contained components. Although important structuring information is lost in the transformation process, it may be desirable to remove the hierarchy for efficiency reasons in implementation models [8].

Up to now the component model has only been used to model itself. In the next section we show how to model real component models.

3 Modeling Real Component Models

Component models are at a meta level below the component meta model. Their model elements are also components, so it is reasonable to reuse the component meta model at this level too. The special features of each component model can be designed as extensions of the basic meta model.

JavaBeans. JavaBeans [4] is a non-hierarchical component model. Therefore, it is sufficient to extend only elements from the non-hierarchical meta model. The JavaBeans components are called *Beans* and their model component conforms to Component from the meta model. A Bean may possess *Bean Properties* which can have any *Java Classifier* as their property type. A peculiarity of JavaBeans is the event-based communication between components. A Bean can fire *Events* which are delivered to another Bean's *Method*. Therefore, Event components conform to Provided Ports and Method components to Required Ports in turn. The delivery of Events to Methods is achieved through *Adaptors* who's model component conforms to the Connector component. The hierarchy of *Event Types* (which are equivalent to Port Types) is determined by the inheritance relationship between the classes that are derived from java.util.EventObject with the exception of void which is the most general port type. Figure 5 shows the JavaBeans component model.

Because in the JavaBeans component model each model element has a one-to-one relationship to a component meta model element the mapping between the two models is straightforward in both directions. The most problematic part of the mapping is preserving the type information in

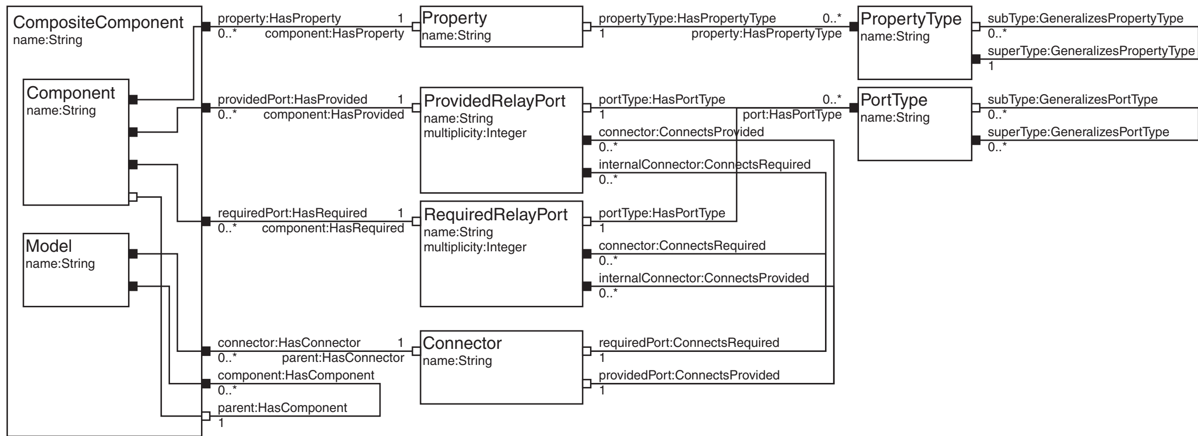


Figure 4. Extensions for the hierarchical component meta model

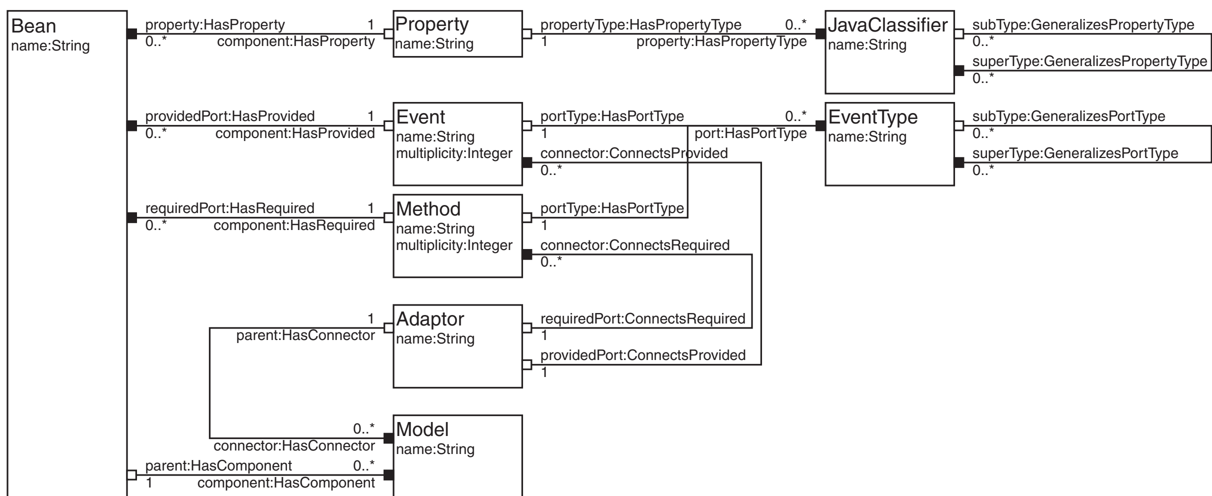


Figure 5. JavaBeans component model

property and port types.

CORBA Component Model. Although it is considerably more complex than JavaBeans, the CORBA Component Model (CCM) [6] still is a non-hierarchical component model. CCM supports both event-based and interface-based communication between components. The event-based communication part is similarly structured as in JavaBeans: *Event Sources* are Provided Ports, *Event Sinks* are Required Ports, and there are *Event Connectors* and *Event Types*.

The interface-based Provided Ports are called *Facets*, the Required Ports are *Receptacles*. We named the Connectors for Facets *Interface Connectors* and the Port Types are simply *CORBA Interfaces*. Unfortunately, the component model would now allow ports to be connected to the wrong kind of port types, e.g. Factets to EventTypes. To prohibit these wrong port type connections two new subtypes of HasPortType were introduced that are unique to event and interface ports. The same pattern is used to prevent mixing connectors and ports of the wrong kind.

The reverse mapping of the CCM model to the component meta model is easier than the forward mapping because there is a one-to-two relationship for Ports, Connectors, and Port Types. Once again it is crucial that the Port Type information is preserved in the mappings. When the Port Types are considered even Ports and Connectors can be mapped unambiguously from the component meta model to the CCM model.

Summary. As shown in this Section, the component meta model can also serve as a platform-independent model who's models can be reused and mapped to various platform-dependent component models. Similar models and mappings have been developed for other real component models that have been designed for special application fields. Describing them would go beyond the scope of this position paper.

4 Integrating Independently Developed Extensions

During our work on components for distributed, embedded real-time systems we have developed a special component model that is suitable for this special application field [8]. In this model the real-time properties of a component-based application have to be analyzed. To this end, a formalism has been developed which is capable of describing abstract behavior of components. The basic concept of the behavior model is a *Path* which has a number of entry and exit points and contains sub-paths. There are several primitive paths that can be used to build more complex ones. A *Computation Path* simply consumes computing power, *OR forks* start alternatives, *AND forks* start parallel computations, whereas the corresponding *OR joins* and *AND join*s stop these activities. The computational model of the connected paths is compatible with real-time analysis and scheduling methods. It can be used to compute optimal priorities and worst-case response times for paths.

The aim of our modeling work was to integrate the path model with an event-based component model in a way that the new model is capable of simultaneously connecting paths when the corresponding components are connected. This has been achieved by firstly developing a component-based model for paths that is completely independent of the event-based component model. Secondly a new component has been introduced that aggregates the component and the path into a new component. Thirdly all event sources combine provided ports and AND forks while event sinks are required ports with OR join behavior. Finally the connectors are at the same time component and path connectors because they are both aggregated into connector components. Figure 6 shows details of the integrated model.

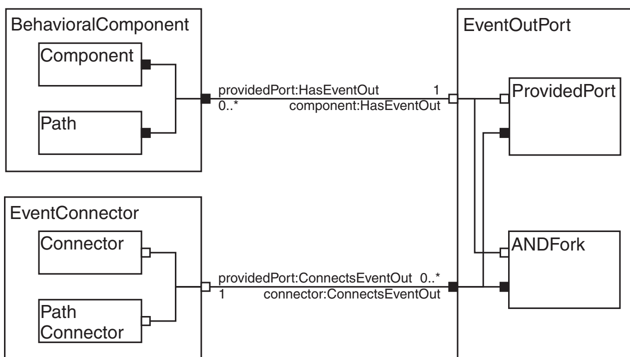


Figure 6. Integration of paths (detail)

The successful integration of the two models demonstrates how easily new models can be incorporated.

5 Related Work

The field of modeling and meta modeling as been intensively explored in the context of object-oriented design [7, 9]. The Unified Modeling Language is now based on a formal meta model, the Meta Object Facility (MOF). MOF is an abstract object model and a set of mappings to various object-oriented platforms like CORBA and Java. This is similar to our meta modeling architecture. The main difference is that MOF (the platform-independent model) is directly mapped to object-oriented platforms, whereas we have introduced platform-dependent models and mappings between models. Current efforts of the Object Management Group to establish a Model-Driven Architecture point into the same direction.

By contrast, meta models have not yet been used extensively in component-oriented programming. The closest approximation of a meta model is the description of the abstract component model for CORBA Components [6]. But there is no formal specification of that meta model. The same applies to ROOM [10], an object-oriented model that bears many similarities to component-based models.

The Software Architecture community has been dealing with component-based models for years [11]. Several groups have developed various Architecture Description Languages (ADLs). These ADLs belong into the same meta level as component models and are similar to our platform-dependent models. As observed in [12] software architecture and component models are two sides of the same coin. In an effort to improve the interoperability of ADLs the common language ACME [13] has been developed. In contrast to our meta model, the focus of ACME has been on the interchange of model information. ACME was not designed to model other ADLs.

6 Conclusion

In our position paper we describe a meta model that is not only capable of modeling various component models but is also a component-based model itself. This makes it an ideal basis for modeling various component models and extending existing models. By developing extensions for modeling real component models and integrating a behavior model we show that component-based models exhibit the same reuse potential and ease of integration of independent model developments that have been observed in component-based programming.

References

[1] Object Management Group: *Unified Modeling Language, Version 1.4*, OMG document formal/01-09-67, 2001

[2] M. Broy, J. Siedersleben: “Objektorientierte Programmierung und Softwareentwicklung – Eine kritische Einschätzung”, *Informatik Spektrum*, **25**(1), 2002, pp. 3–11

[3] C. Szyperski: *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, Reading, MA, 1998

[4] G. Hamilton: *JavaBeans API Specification, Version 1.01*, Sun Microsystems, 1997

[5] G. Eddon, H. Eddon: *Inside Distributed COM*, Microsoft Press, 1998

[6] Object Management Group: *CORBA Components – Volume 1*, OMG document orbos/99-07-01, 1999

[7] J. Alvarez, A. Evans, P. Sammut: *MML and the Metamodel Architecture*, available at http://www.puml.org/

[8] U. Rastofer, F. Bellosa: “Component-Based Software Engineering for Distributed Embedded Real-Time Systems”, *IEE Proc. Software*, **148**(3), 2001, pp. 99-103

[9] Object Management Group: *Meta Object Facility, Version 1.3.1*, OMG document formal/01-11-02, 2001

[10] B. Selic, G. Gullekson, P.T. Ward: *Real-Time Object-Oriented Modeling*, Wiley, 1994

[11] D. Garlan, M. Shaw: “An Introduction to Software Architecture”, *Advances in Software Engineering and Knowledge Engineering*, World Scientific, 1993

[12] K. Wallnau, J. Stafford, S. Hissam, M. Klein: “On the Relationship of Software Architecture to Software Component Technology”, *Sixth International Workshop on Component-Oriented Programming (WCOP 2001)*, 2001

[13] D. Garlan, R. Monroe, D. Wile: “Acme: An Architecture Description Interchange Language”, *Proceedings of CASCON ’97*, 1997

Describing and Reusing Software Design Assets for System Family Engineering*

Alexander Fried, Herbert Prähofer
Institute of System Science
Systems Theory and Information Technology
{af,hp}@cast.uni-linz.ac.at

Abstract

System-family engineering is a new paradigm in software engineering. A system-family is defined as a group of systems sharing a common, managed set of features that satisfy core needs of a scoped domain. The idea behind a system-family approach is to build a new system or application from a common set of assets (domain model, reference architecture, components) defined from earlier developed systems belonging to the same family. The structuring of systems into system-families allows sharing of development effort within the system-family and as such counters the impact of ever growing system complexity.

A system-family engineering methodology demands tool support for management and reuse of design assets. In this paper we will present ideas and concepts to support a designer in working with a design asset base. Different categories of design assets that will form the asset base and design steps which define the usage of the assets will be introduced. To automate these steps, a tool is necessary that manages the asset base and provides functionality for browsing and searching. We will outline the basic principles of such a tool and describe a realization within a commercial tool chain.

1. Introduction

The development of large software systems is getting more and more complex. On the other side, for many problems the same or similar solutions have been reinvented several times. Therefore, endeavor to establish a reuse-based software development process reaches back to the 1960s [6]. Different types of reuse strategies have been pursued since then, ranging from method libraries to component frameworks [3, 2]. *Software system families*, or *product lines* as they are sometimes called, represent a new approach in software engineering and introduce a way to deal with the complexity of software development by incorporating a planned reuse strategy [2]. In contrast to previous reuse strategies, system families are not based on simple code reuse of small pieces, but demand a high order reuse process [3].

A system-family is defined as a group of systems sharing a common, managed set of features that satisfy core needs of a scoped domain. Systems in the family are developed from a set of assets. The assets have been specifically created and prepared for each family [3]. The asset base set is not limited to ready to use software components, but usually contains domain models, architecture specifications, protocols, a domain glossary, etc.

*Research supported by the Federal Ministry of Education, Science and Culture of Austria. Work done in cooperation with Siemens, Corporate Research, SE1

As the methodology of system family is just emerging, development support is very weak. Todays design tools have not been designed with system-family engineering in mind. Therefore, research in the development and enhancement of a design methodology on the one hand and in the development of tools supporting this methodology on the other hand is a prevailing challenge. In this paper we will outline an approach to support the development process in a system-family engineering environment which allows management of assets in an asset base and supports the designer in the usage of the assets in system development. The approach relies on AI techniques - most notable case based reasoning [7, 1] - and is inspired by the previous project CASA (computer aided systems architecting) [14, 11] which has successfully used those techniques in the system design domain.

2. Design Asset Categories

In the following, different categories of software design assets which can be found in software system families are introduced. Those categories represent design assets which are used and treated differently in the design process. In the next section, different design steps are introduced which show how the different design categories of design assets can be used.

The design asset categories are as follows:

- *Component*: A component (Figure 1a) is a *full* implementation of a design asset. It only has limited customization capabilities and variability. For example, a COM component will fall into this category. Essential annotations for this component are:
  - logical view: interface specifications in the form of requirements and guarantees for communication with its environment
  - physical view: requirements on its runtime environment
  - implementation view: the object code which contains the implementation of the component
  - QoS annotations which describe the performance of the component
- *Configurable components* (Figure 1b): This is a design asset which owns more variability than the first category. In particular, its inner structure is not fixed but may be further configured by the designer by plugging in subcomponents. The plug-ins are defined by requirements, e.g. the interfaces that they have to implement. Additionally to the annotations of *components*, as listed above, essential annotations for this category are:
  - the static component structure with the variability points
  - the specification of the interfaces and the requirements for the plug-ins
  - the collaboration protocols between its subcomponents
- *Component templates and collaboration protocols* (Figure 1c): Component templates and collaboration protocols are design assets which not really represent design components in the closer sense but only give templates to show how components can be put together to form a reasonable whole or define a way how components can collaborate. In [12] *role modeling* has been introduced as an object oriented approach to system family design. Collaboration protocols are in the sense of role models as they are intended to model general reusable forms of component collaborations in fulfillment of particular tasks.
  Essential annotations for assets of this category are:
  - the specifications of the roles of the components in the form of interface specifications

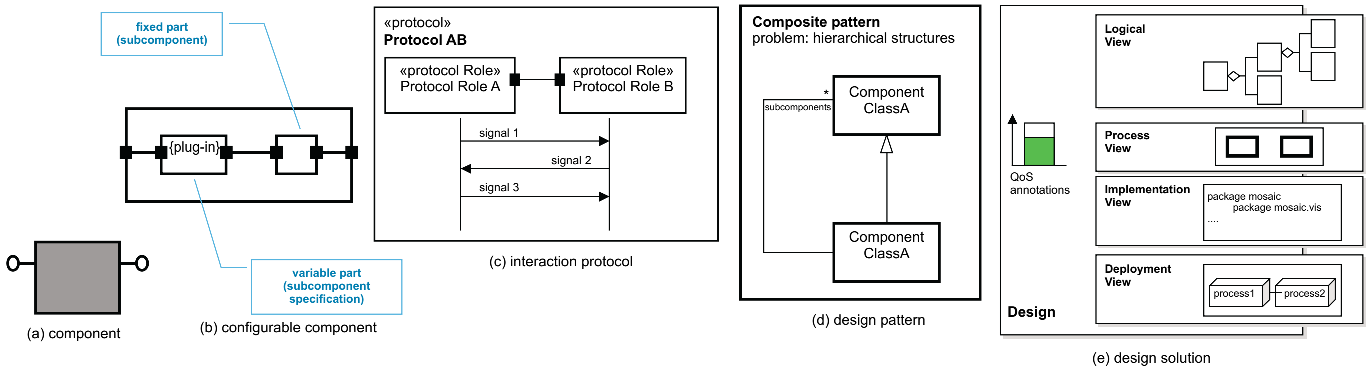


Figure 1. Categories of design assets

  - the collaboration protocol between the components in the form of interaction diagrams
  - QoS annotations which describe the achieved performances by using the asset.

- *Design Patterns* (Figure 1d): Classical software design patterns like the GoF patterns [4] represent a further category of design assets. In distinction to the former collaboration diagrams they are more general as they abstract from the application domain but represent general object oriented design solutions for typical software design problems. Design patterns are typically described by:
  - problem specification in the form of a textual specification of the problems they are intended to solve
  - a generic class or object structure
  - a generic protocol which shows how the objects in the structure collaborate for fulfillment of the task
  - QoS annotations which describe the consequences of the usage of the pattern
- *Design solutions* (Figure 1e): Whole design solutions from earlier design undertakings can be seen as design assets. The design solutions will be described at different levels of abstraction and detail and in the different views.

3. Categories of Design Steps

The introduction of the categories of reusable design assets above was motivated by the fact that those are used differently in the design process. Design steps are introduced here because they pose different requirements for an automatic design support. In the following we describe the design steps where those categories of design assets are used.

- *Selection and customization of a component* (Figure 2a): In this design step a customizable component is searched and selected for a particular design situation. Based on a set of given requirements, the component data base is searched and a best fitting component can be selected by the designer.
- *Configuration of a component* (Figure 2b): When a configurable component is given, one of the plug-ins has to be configured. Therefore, the design situation is that a set of requirements for the plug-in are given and a component, or again a configurable component, which matches the requirements best should be found.
- *Incorporation of a interaction protocol or template*(Figure 2c): In this design step an

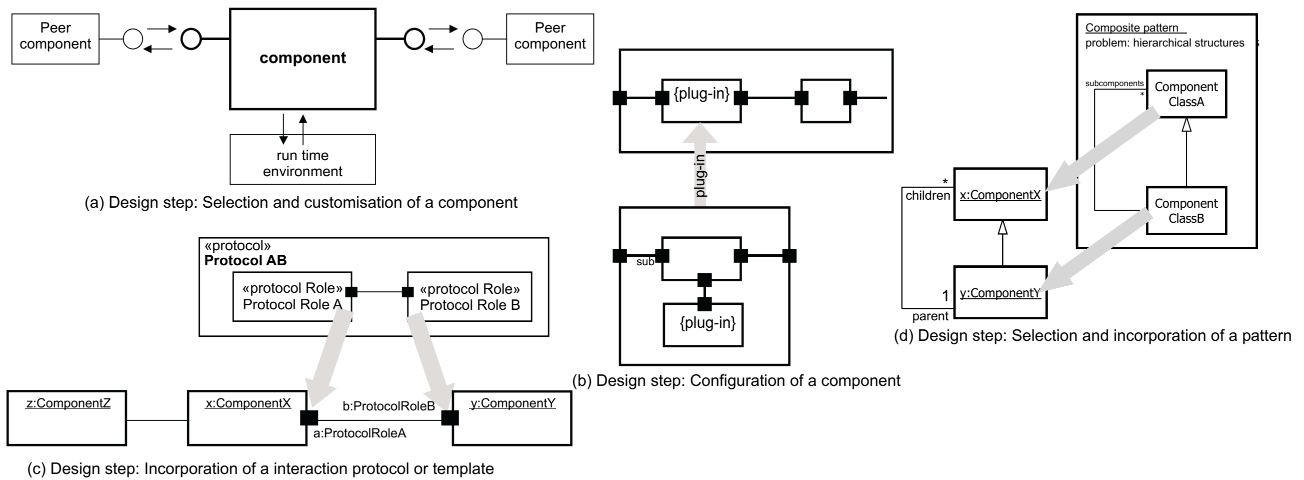


Figure 2. Design Steps

interaction pattern which is capable to fulfill a particular task should be incorporated into the current design. Given a functional requirement, possibly stated as a use case, a collaboration which fulfills this functional requirement should be found.

- *Selection and incorporation of a pattern* (Figure 2d): In this step, typically a problem situation will suggest the employment of a pattern.
- *Adaptation of a design solution*: The most complex but also the most profitable design step is the reuse of a whole or partial design from a former project. This step proceeds in the following substeps:
  - search for design situations which are similar to the current design situation, which means designs with similar requirements, which are done in a similar context, and/or which show similarity to the current (partial) design architecture.
  - detailed evaluation of previous design solution for similarities and deviations to get hints for its adaptations
  - adaptation of previous design to meet current design situation.

4. Tool Support for Asset Reuse

The above design steps describe how the design assets (section 2) are used to design new applications. Although these steps can be used manually, only when backed up by a tool (figure 4) an efficient development in system family design environments is possible.

Based on the current, i.e. not finished, design and the requirements it has to fulfill, the designer has to use the above design steps to advance his design. The supporting tool environment should connect the different tools and the asset base.

Given functional and non-functional requirements, an asset has to be retrieved from the asset base. To retrieve these assets we suggest two ways:

1. *Browsing*
2. *Automated retrieval*

The problem specification used in an automated retrieval is given in the same way as the assets are specified themselves:

- *Component specifications*: Using the interface specifications of a component (list of methods), a component or configurable component, implementing the same or a similar interface, is searched for.

- *Protocol*: A protocol, e.g. as sequence diagram, is used to search for a design pattern, protocol or a component implementing this protocol.

- *Textual specification*: A list of keywords, possibly out of a glossary, can be used to find an asset.

Using these specifications, the retrieval mechanism searches the asset base and presents the designer the best matches. In the following we will present two approaches to accomplish automated asset retrieval.

4.1. CBR

Case-based reasoning (CBR) [7, 1] is a well established paradigm of artificial intelligence. In distinction to usual AI-techniques, which require a closed and complete form of knowledge representation, in case-based reasoning knowledge is coded in the form of *cases* which represent a kind of expert knowledge. Problem solving is accomplished by retrieving relevant cases from a case base and adapting the best matches to meet the problem in hand.

Figure 3 shows the problem solving cycle in case-based reasoning [1].

Design knowledge usually is very heterogeneous, unstructured, and not supported by a formal theory. Capturing design knowledge therefore has shown to be very difficult with conventional techniques. Case-based reasoning however is very well suited for design problems, as is shown by the many projects in case-based design (e.g. [13, 8]).

We want to take this general idea of CBR and use it to assist a designer in the creation of a new application for a product line.

4.2. Graph Matching

To apply the CBR cycle to system families and retrieve certain assets from the asset base, it is necessary to possess an evaluation mechanism

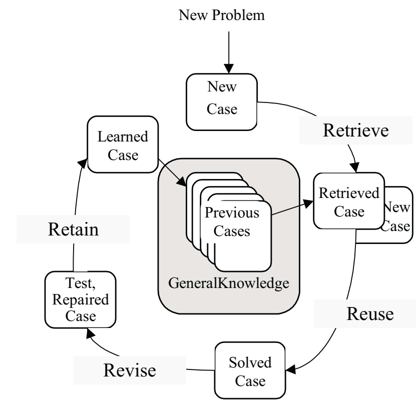


Figure 3. Problem solving cycle in CBR [1]

which measures the similarity of the assets to a given specification. Graph matching in general and the GUB (*G*raph matching toolkit *U*niversity of *B*ern) algorithm in particular [9] is one of these mechanisms being able to compare structural properties of design assets. We think that especially for collaboration protocols, which we intend to represent as UML sequence diagrams (section 5), the structural features are very important and a comparison mechanism that incorporates them will be very effective.

5. Vision of a Tool

A tool environment which supports a system family engineering approach has to fulfill the following requirements:

- Close cooperation with standard tool chains, like Rational Developer Suite and similar.

- Integrate the different tools in a chain.

- Maintain additional information required in a systems family approach (e.g. for design asset retrieval and reuse)

- Support maintaining and organizing of design assets

- Fast and accurate retrieval of design assets from the asset base based on requirement and partial design specifications.

The tool environment that we are prototyping will rely on the following concepts:

- *Unified Modeling Language (UML)* for specifications
- *UML metamodel*
- *UML metamodel integration*: Special specification forms will be introduced
- *Open APIs*: to access various tools

Figure 4 shows the overall architecture of the proposed tool environment. It consists of the following parts:

- *External Tools*: In the external design tools with their Open API the various specifications are kept and accessed.
- *UML metamodel layer*: The UML metamodel layer provides a standardized access layer to handle the various specifications.
- *Asset Base*: Design assets will be described as meaningful aggregations of other specifications, mainly UML specifications.
- *Indexing Trees & Domain Glossary*: In this part, information is maintained which is kept to support efficient browsing and retrieval of design assets.
- *Retrieval & Browsing*: There will be design modules to assist the user in retrieval of design assets.
- *Asset Retaining & Organization*: Also modules will be needed to assist the user in retaining new design assets and organizing the asset base.

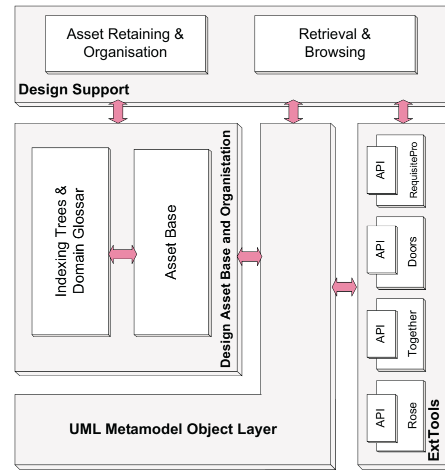


Figure 4. Architecture of the tool environment

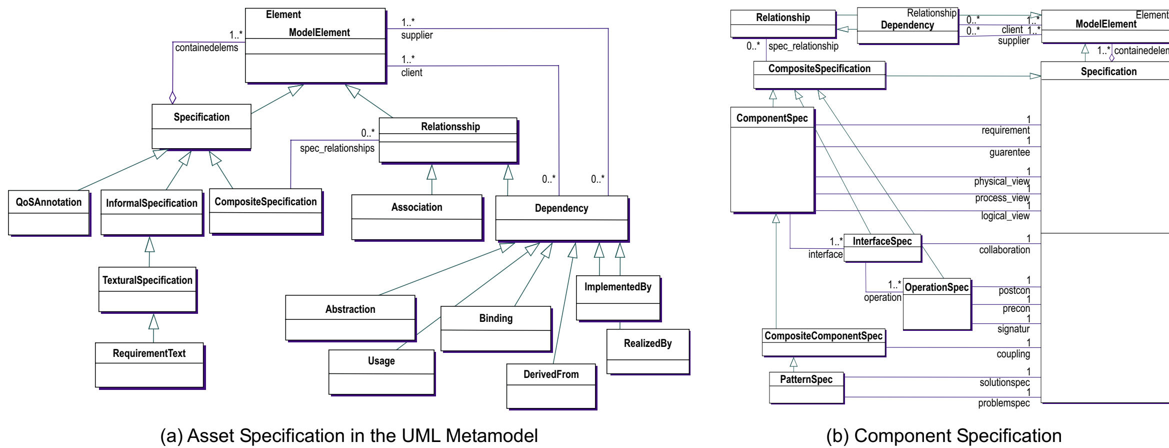
5.1. Assets Description within the UML Metamodel

After having outlined the principles of asset description techniques in the last sections, we continue to describe how design assets should be specified in the context of the UML modeling notations.

Asset specification should be closely integrated into the UML modeling and metamodeling concepts. An approach is proposed where asset specifications are basically meaningful aggregations of other specifications. Referring to the description techniques proposed in [5] and reviewed above, a design asset consists of several other specifications each with its particular meaning and purpose. Therefore, design assets just group several other specifications.

In the following a specification formalism for design asset description is introduced. This specification forms are all derived from the most general `ModelElement` type of UML metamodel [10] and in that way are integrated into the UML metamodel type hierarchy.

Figure 5a shows a type hierarchy with special model elements introduced for design asset description. The purpose of the various types are



(a) Asset Specification in the UML Metamodel

(b) Component Specification

Figure 5. Specification Overview

as follows:

- `Specification`: Base type for design asset descriptions. A `Specification` is just a grouping of other `ModelElements` (designed after the composite design pattern [4]).

- `QoSAnnotation`: A `QoSAnnotation` gives a mostly informal notation of the performance and quality of service aspect for a design asset.

- `InformalSpecification`: The `InformalSpecification` type is a base type for specifications which are not in a formal language. Derived from this base type is the type `TextualSpecification` and from this the special type `RequirementText`.

- `CompositeSpecification`: A `CompositeSpecification` not only maintains a set of sub model elements but also defines various relationships between those.

- `ComponentSpec` (Figure 5b): This type is used to describe the different categories of design assets as introduced in section 2. The specification is just an organization of various other `Specifications` in accordance to the notion of a component as introduced in [5] and shown in Figure 1.

  A `ComponentSpec` maintains a collection of `Specifications` consisting of `guarantees` (e.g. interfaces) and `requirements`.

  Additionally, the `ComponentSpec` also maintains a list of interface declarations (`interface`) which are most important in the functional view. An interface normally will contain information regarding the guarantees and requirements.

  Also the component groups all its specifications into subgroups according to its membership to the `logical_view`, `physical_view`, and `process_view`.

- `InterfaceSpec`: The `InterfaceSpec` is a subtype of `ComponentSpec` which only contains information regarding the interface description of a component.

- `OperationSpec`: An `OperationSpec` is a `ComponentSpec` which has pre- and postconditions and an operation signature specification.

- `CompositeComponentSpec`: A `CompositeComponentSpec` is a `ComponentSpec` which defines subcomponents and a coupling.

- `PatternSpec`: A `PatternSpec` is used to describe a design pattern [4].

6. Summary and Outlook

In this paper we have presented concepts and techniques for design asset management, retrieval, and reuse within a system-family engineering approach. Different categories of design assets have been introduced and we have shown how to represent them as UML metamodel extensions. Moreover, we have outlined how various design steps should be supported by a tool environment.

Currently,a prototype of a tool environment is in development. An abstract interface based on the UML metamodel is defined and a partial implementation of this interface to connect to Rational Rose is realized. The GUB toolkit is integrated into the prototype and it is possible, given a sequence diagram as query, to search for similar sequence diagrams, which are stored in Rose.

Next steps in the development will be the implementation of the asset specification techniques and the set up of an example asset base for a domain. Then we will experiment with different retrieval techniques and investigate methods to allow adaption of design assets to meet the special design requirements.

References

[1] A. Aamodt and E. Plaza. Case-based reasoning: Foundational issuesissues, methodological variations, and system approaches. *AI Communications*, 7(1):39–59, 1994.

[2] Jan Bosch. *Design and use of software architectures*. Addison-Wesley, 2000.

[3] Paul Clements and Linda Northrop. *Software Product Lines*. Addison Wesley, 2001.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison Wesley, 1994.

[5] Peter Graubmann and Axel Klein. Requirements and techniques for component interface description and component selection. Technical report, Siemens ZT, 2000.

[6] Ivar Jacobsen, Martin Griss, and Patrik Jonsson. *Software Reuse - Architecture, Process and Organization for Business Success*. Addison Wesley, 1997.

[7] Janet Kolodner. *Case-based reasoning*. Morgan Kaufmann, San Mateo, CA, 1993.

[8] M.L. Maher and G. de Silva Garza. Developing case-based reasoning for structural design. *IEEE Expert*, 11(3), 1996.

[9] B.T. Messmer. *Efficient graph matching algorithms for preprocessed model graphs*. PhD thesis, University Bern, Switzerland, 1996.

[10] OMG. Unified modeling language specification - version 1.4. www.omg.org, 9 2001.

[11] H. Praehofer and J. Kerschbaummayr. Case-based reasoning techniques to support reusability in a requirement engineering and system design tool. *Engineering Applications of Artificial Intelligence*, 12, 1999.

[12] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working With Objects*. Manning Publications, Greenwich, 1995.

[13] Y. Reich. The development of bridger: A methodological study of research on machine learning in design. *Artificial Intelligence in Engineering*, 8:217–231, 1993.

[14] C. Schaffer. *Computer Aided System Architecting (CASA): Requirement-driven design of multi-disciplinary systems*. PhD thesis, JKU Linz, Austria, 1999. (in German).

A Preliminary Analysis in Mapping UML Use Cases to State Machines

Luca Pazzi

University of Modena and Reggio Emilia
Dipartimento di Ingegneria dell'Informazione
Strada Vignolese 905, Modena, Italy
email: *pazzi@unimo.it*

Abstract. UML behavioral modelling has its roots in two well distinguished paradigms: Jacobson's use case modeling, which elicits user specification knowledge to sequence diagrams through ITU Message Sequence Charts (MSC), and Rumbaugh's OMT state-based modelling, which allows to describe the behavior of object classes and more complex design artifacts through Harel's Statecharts. Although MSCs remain the main tool for describing the separate scenario which make a UML use case, a state-based global view of a whole use case has been advocated by many authors and seems desirable, since Sequence Charts – inherently bound to a linear view of time – are less expressive than state based formalisms – allowing for modelling branching and cyclical behavior. This paper reports a preliminary study in bridging the gap between the two paradigms, by showing some initial steps towards the integration od *separate* linear descriptions into a *single* state description of the world involving different actors.

1 Introduction

A use case is the "high level description of a coherent unit of functionality to be provided by a system under design to a set of one or more actors participating in it" [2]. Use cases are realized by collaborations of objects acting in concert [1] that provide some behavior that is *bigger than the behavior represented by the sum of the parts.*

The use case behavior *is not represented as a whole*, rather by means of its instances, named *scenarios*, which are linear sequences of events drawn from the whole use case. Scenarios can be thus thought of as being *slices* of the whole use case behavior and as such analysed and described separately one from the another. Scenarios are described within the UML by the messages exchanged the participating actors through *sequence diagrams.* State based formalisms provide obvious representational advantages over sequence

diagrams. In first place they are not bound to a strictly linear representation of time. In second place states can be seen as *snapshots* of complex situations occurring within a use-case scenario; state transitions, conversely, represent the feasible connections among such snapshots, that is the events occurring in the scenario. The real advantage is that, by the concept of state, it can be easily established that a situation within a scenario *is the same* situation within another scenario. This allows an easy merging of different scenarios into a single representation, that is the use case seen as a whole.

The paper shows how, by a simple textual decomposition criterion, global states belonging to a whole use case can be easily identified from the use case component scenarios. Once such global states are found it is shown how scenario subparts (subscenarios) may be easily integrated into such a state based vision as state transitions. Finally, the translation of a sequence diagram into a state transition is fully detailed.

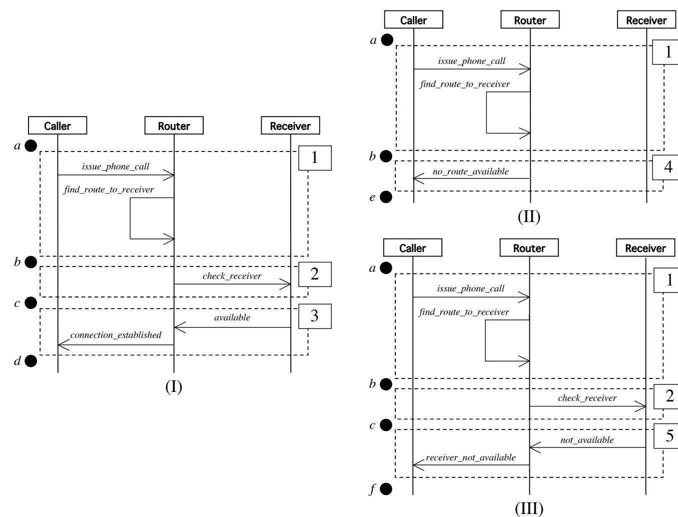


Fig. 1. The Message Sequence Charts depicting three scenarios in a phone call use case. The diagrams show scenario fragmentation obtained through the *bifurction points* identified in the text.

2 Adding modularity to textual scenarios

Sequence diagrams (Message Sequence Charts) allow to represent the *sequence information* pertaining the interactions among different entities involved in a complex behavior, namely the whole use case. In the example carried out in this Section, the use case "telephone communication" is depicted by different sequence diagrams (Figure 1). We start by reporting the full text of the corresponding scenarios:

Scenario 1 • *the caller issues a call to the receiver through the router; the router seeks a free route to the receiver* • *once the free route to the receiver is found, the router checks whether the receiver accepts the call from the caller* • *the receiver accepts the call and the connection is established* •

Scenario 2 • *the caller issues a call to the receiver through the router; the router seeks a free route to the receiver* • ***no route is available to the receiver; the caller is reported a no-route error*** •

Scenario 3 • *the caller issues a call to the receiver through the router; the router seeks a free route to the receiver* • *once the free route to the receiver is found, the router checks whether the receiver accepts the call from the caller* • ***the receiver does not accept the call from the caller; the caller is reported a specific error*** •

The first step towards scenario modularization is achieved by identifying subparts of the original scenarios; these on their turn correspond to subsequences in the MSCs (shown as dashed regions identified by boxed labels, such as [1], [2], [3], etc.). We segment scenarios by first identifying *bifurcation points* in the text of the scenarios, that is points in which scenarios *become different* one from the anothers. We denote such point by bullet dots (•), as in the text of Scenario 1. Subsequently segmentation points have been transferred to the corresponding MSC diagram (Figure 1, Diagram (I), yelding the three *subsequences* denoted respectively by [1], [2], and [3]. Other *subsequences* (namely [4], and [5]) are identified in Scenarios 2 and 3

and matched to the corresponding MSCs (Figure 1, Diagrams (II) and (III). The original scenarios can now be rewritten in a compact form (we name bullet points as in Figure 1):

Scenario 1 $\overset{a}{\bullet}\ \boxed{1}\ \overset{b}{\bullet}\ \boxed{2}\ \overset{c}{\bullet}\ \boxed{3}\ \overset{d}{\bullet}$

Scenario 2 $\overset{a}{\bullet}\ \boxed{1}\ \overset{b}{\bullet}\ \boxed{4}\ \overset{e}{\bullet}$

Scenario 3 $\overset{a}{\bullet}\ \boxed{1}\ \overset{b}{\bullet}\ \boxed{2}\ \overset{c}{\bullet}\ \boxed{5}\ \overset{f}{\bullet}$

Unfortunately, compactness does not correspond here to clarity. For example it would be desirable to view at a glance that the use case behavior presents bifurcations (*branches*) in points b and c. Moreover it can be observed can be better observed that the use case presents different end points correspond to three different feasible evolutions of the state of the world, respectively points d, e and f. The same information may be better presented by a graph-like diagram like the one in Figure 2.

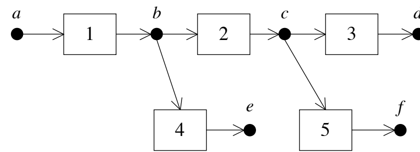


Fig. 2. The overall topology of the use case of Figure 1 shown by a directed graph.

3 From sequence diagrams to state machines

As shown by the example above it is clear that a linear view of time is not sufficient for most of the complex situations, which often present branches (and loops) in their state evolution. Although there have been various attempt in providing MSC with conditional (and looping) constructs, (we note that) the *subtle* point is that no construct is provided for the *lifeline* which denotes the evolution of the system as a whole.

Since we already put bullet points in correspondence of the states of the world, it seems appropriate to consider the *whole behavior* of the use case as a state diagram. Such intuition is also enforced by the fact that a state diagram is a directed graph, that is it allows to represent branches, cycles and loops in the evolution of the system, thus allowing full expressivity to the use case.

For example, in Figure 2, the line joining point a to b is, essentially, a *state transition* from state a to state b labeled by the sequence diagram of MSC $\boxed{1}$. The intended meaning is that, starting from the world in state a and by applying the trasformations described in the MSC $\boxed{1}$, we reach a state of the world which is named b.

Observe Figure 3: suppose the world (i.e., the system) is found in a state called a before the event e happens, and in a state b afterward. Since an event is meant to denote a change of state, entity X moves from state x to state x' as event e *is sent* to Y, as well as entity Y moves from y to state y' as event e *is caught* and processed.

4 Conclusions and open issues

Representing use cases by state diagrams allows to express the behavior of complex system in a way more clear and effective than single sequence diagrams. We have shown how to achieve the translation of sequence diagrams into more general graph representations, which can be read as the *state diagram representation* of the use case taken as a *whole*. Moving from sequence diagrams to state machines requires, however, to address the following open issues:

1. the *system* line may not be present in the scenarios; in other words, most of the designs may consider the state evolution of

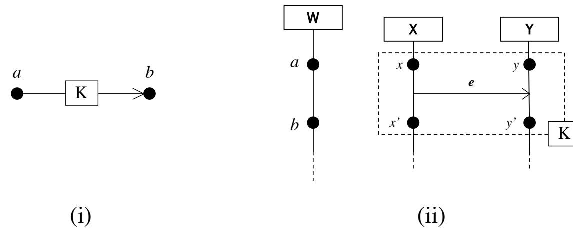


Fig. 3. Comparing a state transition (i) with a MSC (ii). We added a lifeline W to the MSC which gathers fragmentation points (system states). Corresponding states x, x', y, y' have been marked on the original components' lifelines.

the system implicitly, as in the example, where we added a further lifeline, named w in order to suggest "behavior of the *whole*";

2. modelling a complex behavior by a state machine requires to address consistency issues in the following cases:
   (a) different paths branching out of a node;
   (b) different paths joining into a node; in this case it is required to state that the system is left in the *same state* by any of the converging paths, since they converge to the *same point*. This in turn requires to fully exploit the notion of system state semantics.

References

1. Bruce Powel Douglass. *Real-time UML: Developing Efficient Objects for Embedded Systems*. Addison-Wesley, 1998.
2. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

Coupling MDA and Parlay to increase reuse in telecommunication application development

Babak A. Farshchian Sune Jakobsson
Erik Berg
Telenor Research and Development
Otto Nielsensvei 12
NO-7004 Trondheim, Norway
{Babak.Farshchian, Sune.Jakobsson, Erik.Berg}@telenor.com

Abstract

MDA (Model-Driven Architecture) has been coined by OMG (Object Management Group) as the next step in application integration. Being based on standards already embraced by a large segment of the software engineering industry, MDA promises fully automatic model transformation. MDA enables application developers to use formalisms such as UML to specify their applications in a totally platform-independent way. Later transformations to platform-dependent software are automated. Parlay, a middleware specification developed for the telecommunication domain, is on the other hand promising network independent development and deployment of telecommunication services and applications. In this position paper we report on our experience from a Eurescom project where we try to couple MDA and Parlay in order to increase reuse in the telecommunication domain. Telecommunication service development is hampered by long development cycles and low level of reuse. We describe how MDA approach can be applied to telecommunication domain through the use of Parlay. We believe this approach has substantial potential for reducing development costs for many telecommunication operators. In addition, developed models and applications can be deployed on a wide variety of platforms without much change.

1. Introduction

Market situation in telecommunication is changing rapidly due to the convergence of Internet and telephony networks and the removal of monopoly situations in many countries [7]. This change poses great challenges and opportunities for telecommunication operators and service providers. One such challenge has been the increased competition in providing advanced value-added services to the customers [3]. The abandoning of monopolist market models has brought with it a pressure on incumbent carriers to make available their network resources to be used by third-party service and application providers [8]. This creates a specialization of the telecommunication application development business that again puts pressure on both network operators and third-party application providers to optimize their development processes.

Value-added services and applications in convergent networks increasingly contain a large software part, as opposed to traditional telecommunication services where software played a more modest role. Application developers resort to (often complex) software solutions for accessing and deploying network resources offered by network operators. Software is also used in order to create innovative applications that make use of a mixture of network resources (e.g. applications operating transparently on both Internet and telephony networks and utilizing corporate databases). Increased competition and the innovative nature of convergent networks make it necessary to deploy development processes that allow for rapid application development so that developers can experiment with different application types without binding too much resources. Issues that have been of central importance for the software engineering community, such as reuse and object-oriented software development, is becoming more and more important also in the telecommunication domain.

One of the main software engineering issues facing telecommunication application developers is reuse. Telecommunication networks have traditionally been proprietary, developed as a result of strategic alliances between equipment producers and incumbent network operators. Similar applications have had to be developed multiple times on each proprietary platform. The situation is not to be changed in the near future due to large invest-

ments in proprietary network technologies. Nevertheless, the increasing number and variation of offered services and applications make the situation critical. There is an urgent need to enable application developers to reuse their applications on different proprietary platforms.

Important initiatives are taken in order to address this issue. In this paper we will look at two of these initiatives, i.e. Parlay and MDA (Model-Driven Architecture). Parlay is a middleware specification developed specifically for the telecommunication domain. Parlay enables network operators to open up their networks and make available their network resources in a systematic and standard manner [8]. Parlay enables reuse at the component level. Although Parlay allows service providers to reuse their services on multiple networks, there is little support for reuse at the application modeling level. Another relevant initiative, started by the software engineering community, is OMG's MDA [12]. MDA addresses reuse at the business modeling level. MDA aims at increasing automation in software development by deploying automatic model transformation techniques. The goal is to specify applications in a business modeling language such as UML, and to have some tool automatically create the software.

Although both Parlay and MDA are in their early days, they have gained large industry support. Supporting tools are entering the market, and an increasing number of companies are deploying solutions based these technologies. In order to evaluate MDA we are participating in a Eurescom project (see [4]) together with a number of telecommunication operators and tool vendors. Our goal in this project has been mainly to evaluate MDA as a possible enabler for improving application development activities undertaken by many telecommunication companies. This paper describes some of our experiences so far, in particular those related to reuse at application modeling level.

The paper is organized as follows. First we give a short overview of Parlay and MDA. We then describe how MDA and Parlay can be used in combination in order to improve application development activities. A discussion of the implications on reuse is then provided before we conclude the paper.

2 Overview of Parlay

Parlay [8, 13] is a platform-independent, object-oriented middleware specification developed specifically for the telecommunication domain. Parlay is being developed by the non-profit Parlay group as an open specification (see www.parlay.org). Figure 1 shows an overview of how Parlay works.

The conventional approach to service provision in telecommunication networks allows only the network operator to develop applications (e.g. voice messaging, follow-

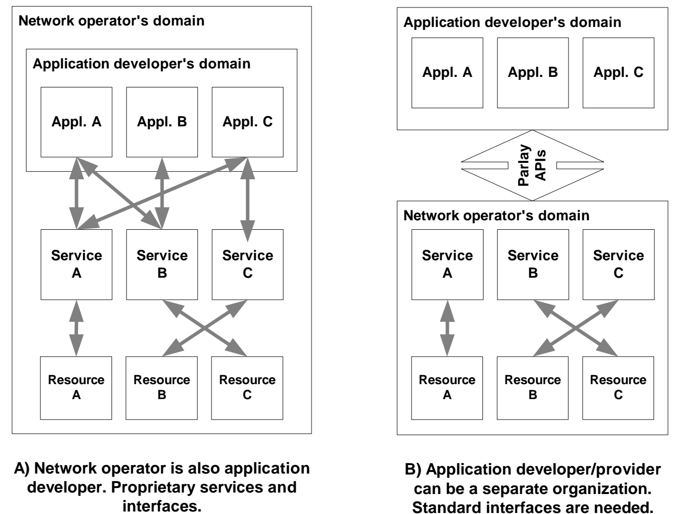


Figure 1. Parlay enables third-party application developers make use of network resources.

me functions) based on its own services. These services and applications are normally developed specifically for the network operator's proprietary network, and are not reusable in other contexts (Figure 1.A). Parlay defines a standard interface towards the network operator's environment (Figure 1.B). This environment includes resources such as fixed and mobile networks. Services that provide access to these resources are standardized through Parlay APIs. Application developers can call API methods on network operator's services and make use of the underlying resources in their own applications. This approach allows application developers to develop advanced applications utilizing a mixture of services and resources (e.g. fixed and mobile telephony, Internet, corporate databases). In addition, the approach increases reuse by making application specification independent of underlying network.

Parlay APIs' architecture is shown in Figure 2. The Framework Interface is an authentication and service discovery component that is a standard part of any Parlay implementation. This interface provides standard methods for authentication of third-party external applications that wish to make use of network services. Framework Interface also provides standard methods for publishing and discovering existing services in a network.

Service interfaces provide standard access methods to common telecommunication services. Parlay specification contains a large number of such service definitions (e.g. call control, session control, messaging, account management) but allows for new services to be added. An (external) application has to be authenticated before it can look for desired

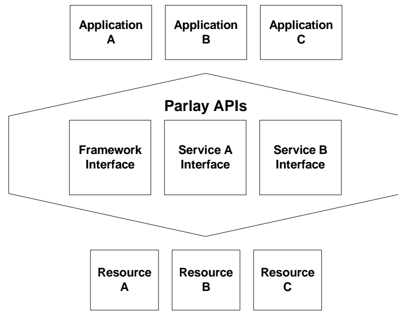


Figure 2. Parlay APIs' overall architecture.

services. Once contact with a service is established, the application can make use of the service. Parlay also specifies a callback mechanism that allows services to call methods on the applications (e.g. to notify an application if a call is made to a specific number).

Parlay enables reuse beyond what is available in current telecommunication networks. Service definitions can be reused. The same service definition can provide access to a variety of resource types. For instance, a call control service can be used to connect users using mobile, fixed, or Internet phones, without the service interface being changed for each type of phone. In addition, Parlay opens for reuse at the application level. Applications need to deal only with Parlay APIs. Application developers can specify their applications without taking into consideration what kind of network they will be deployed on.

Although Parlay enables reuse of components (i.e. services), it does not support higher-level reuse, such as reuse at the application model level or reuse of component compositions. What happens in the application level, e.g. what kind of business models are developed and reused, is outside Parlay's scope. This is where the MDA approach promises to be of benefit.

3 Overview of MDA

MDA (Model-Driven Architecture) is OMG's vision of enterprise application integration [2, 10]. MDA is based on model transformation principles, some known from earlier research and development within the CASE (Computer-Aided Software Engineering) community. MDA is a promising approach mainly because it is based on already-embraced industry standards such as UML (Unified Modeling Language) and XML (eXtensible Markup Language), which provide an organizational and technological springboard for the approach. In addition, the deployed standards already offer a level of formalism that makes it feasible to perform model transformations with a reasonable level of automation.

Figure 3 shows an overview of the MDA approach. At the heart of the approach is the Meta Object Facility (MOF) [9]. MOF is a meta metamodel that is used to define all the metamodels (i.e. modeling languages) in an MDA architecture. MOF is used to define UML, which is a business metamodel. MOF is often mapped into an OMG-defind XML standard called XMI (XML Metadata Interchange). XMI is used for exchanging models among tools, e.g. for transformation purposes.

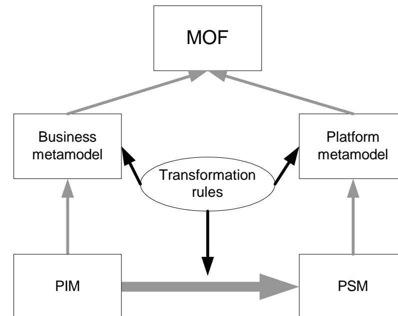


Figure 3. An overview of the MDA approach.

Any number of metamodels can be defined using MOF. MDA approach is based on developing separate metamodels for business domains (e.g. telecommunication) and technological platforms (e.g. Parlay). A set of transformation rules formally define how models defined using one metamodel can be transformed to models defined using another metamodel.

An application developer can use a business metamodel to model business applications. An application model is in this way developed to be totally independent of technological platforms the application will be deployed on. Such an application model is called a Platform-Independent Model (PIM). Platform-Specific Models (PSMs) are on the other hand defined specifically for the target platform. PSMs are ideally not developed by people but are generated automatically using proper transformation rules.

MDA is an enabler of reuse at application model and metamodel levels. Reuse at the model level is mainly due to platform-independent development of application models (i.e. PIMs), but also because of the possibility to reuse model segments and component compositions. The same PIM can be used to generate several different PSMs by simply defining new transformation rules. At the metamodel level, one metamodel needs to be developed for each business domain and each technological platform. Transformation rules themselves are of course reusable across metamodels.

Although MDA is notation-independent, OMG's recommendation and a large part of the development within the industry are based on UML. UML is particularly suited for the

MDA approach because of its extension mechanisms [1]. Business and platform metamodels can be defined as UML profiles (for an example of UML profiles see [6]). In addition, having a formal meta metamodel ensures consistency and interoperability.

4 Using MDA and Parlay to develop applications

Using the MDA approach in combination with Parlay is one of our main research goals. Such a combination has the potential to increase the level of reuse significantly. PSMs for each proprietary platform need to be developed only once, by the network operator or other software vendors. Third-party application developers will only need to develop PIMs for their applications and reuse these PIMs on multiple platforms, possibly belonging to multiple network operators. The overall process is shown in Figure 4. To the left of this figure we see the application developer who develops the PIM for the desired application. Transformation rules are used by the development tool to automatically create a PSM, which is an application directly written for Parlay. This application will make use of any network resource that offers a Parlay service interface.

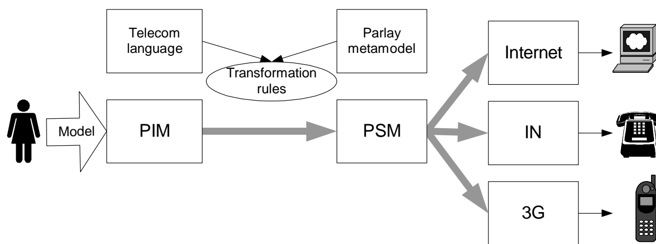


Figure 4. MDA-enabled development of telecommunication applications.

In order to enable this approach we need to start by defining a business metamodel (modeling language) for the telecommunication domain. This metamodel will be MOF-compliant and will focus on the specific needs of the telecommunication industry. Some of these needs are identified to be [5]:

- Telephony networks: UML needs to be extended with concepts that constitute the domain of telephony networks and related services. Examples of such concepts are call, conference, voice message, and telephone number. These concepts are independent of any underlying technology and should be specified in application PIMs.
- Telecommunication service access and subscription: UML needs to be extended with concepts to allow application developers to model how customers, customer profiles, services, access to services, subscription etc. will be managed for the purpose of billing and resource allocation. Access and subscription management is also mainly independent of the underlying technologies (except where there are differences in prices based on the used technology) and should be modeled in PIMs.
- Telecommunication network management: Assuring that a telecommunication network is functioning properly is crucial for the customers. A number of parameters, such as fault management policies, performance management, and security management can be modeled independently from the underlying platform.
- Quality of service: Quality of service is in many cases a part of the application domain, independently of what platform the application is running on. Some applications, e.g. emergency numbers, will require very high availability, while other applications e.g. video conferencing will require high performance. These parameters should typically be defined in a PIM.

Developing a metamodel for the telecommunication domain is a great challenge. Such a metamodel should have enough expressive power in order to satisfy both conventional telecommunication needs and the needs of the increasing number of services and applications that are based on convergent networks. Such a metamodel can be developed as a completely new modeling language, or it could be a specialization of UML. In both cases compliance to MOF is crucial in order to allow for interoperability with future MDA-enabled tools.

A platform metamodel, and corresponding transformation rules, will take as a starting point the Parlay API specifications. These specifications are already documented in UML, and it is expected that the influence of UML on Parlay specifications will increase as UML 2.0 is accepted by the industry. Compliance with UML will make it easier to develop transformation rules that can automate the transformation task to a satisfactory degree. In fact, the formal specification of UML 2.0, with the increased centrality of MOF, also makes it feasible to talk about full-scale roundtrip engineering.

In applying the scenario described above, development tools and environments will play a crucial role [11]. MDA-enabled tools are entering the market. This new generation of tools will have a meta-CASE flavor in that they will enable flexibility both in front-end and back-end in order to allow different business and platform metamodels to be used in combination. Defining architectures for such tools is an important part of our work.

5 Enabling reuse

To sum up, our approach will increase reuse in the following specific points:

- Reuse of application models: The business metamodel and the resulting application models will be totally independent of the underlying service infrastructures, and will focus solely on the business area to be supported by the telecommunication application. This is in accordance with the MDA vision. This means that when new services with better quality make their way to the market (e.g. 2.5G and 3G services) the application models can be reused without much change. In fact, we envisage UML packages that model basic telecommunication applications (e.g. number conversion, personal answering machines, voice mail) to be developed by third-party vendors.

- Reuse of application and architecture patterns: The approach will offer the possibility to reuse specific combinations/patterns of service components. Such patterns will offer a significant advantage for application developers who want to customize their applications to the needs of different customer groups.

- Reuse of service components: Services defined in form of Parlay service interfaces will be reusable in case of changes in the underlying network technologies.

- Metamodels and transformation rules: Metamodels and transformation rules developed during our research are generic and can be reused. In fact, they are planned to be incorporated into MDA-enabled tools developed by our partners. Knowledge transfer activities, in particular towards standards bodies (e.g. OMG) is also a high priority activity within our research.

6 Conclusions and future work

We have described our preliminary results from a European Eurescom project where we aim at coupling Parlay and the MDA approach to increase reuse in telecommunication application development. The approach is based on developing a MOF-compliant modeling language for the telecom domain that addresses the specific needs of this domain. Models developed using this language will be automatically transformed into Parlay applications using a set of transformation rules. The approach is ideal for third-party application developers who want to build applications on top of the network infrastructures offered by large network operators.

The work reported here will be developed further in a European IST project starting in 2002. Our future work in the context of this upcoming project is to develop the different components of the approach. Our particular focus will be on developing a business metamodel for the telecommunication domain, an architecture for MDA-enabled tools, prototypes of MDA-enabled tools, and methodologies for MDA-enabled application development within telecommunication. We also plan to run a number of experiments involving real world cases. These experiments will allow us to measure the gained amount of reuse, in particular with respect to component reuse, pattern reuse and application model reuse, which are the main expected advantages of using the combined MDA-Parlay approach.

7 Acknowledgements

The work reported here is a result of the Eurescom P1149 project. We thank all project members for cooperation and feedback on the ideas presented here.

References

[1] S. S. Alhir. Unified modeling language extension mechanisms. *Distributed Computing*, pages 29–32, Dec. 1998.
[2] J. Bézivin. From object composition to model transformation with the MDA. In *TOOLS' USA, Santa Barbara, CA USA*. IEEE, 2001.
[3] Y. De Serres and L. Hegarty. Value-added services in the converged network. *IEEE Communications*, 39(9):146–154, Sept. 2001.
[4] Eurescom P1149 project team. Impacts of changes in enterprise software construction for telecommunications, 2002. http://www.eurescom.de/mda4telecom.
[5] Eurescom P1149 project team. Model driven architecture - Adaptations and impacts for the telecom domain. Project deliverable, Eurescom, Heidelberg, Apr. 2002.
[6] M. Fontoura, W. Pree, and B. Rumpe. *The UML profile for framework architectures*. Addison–Wesley, Boston, 2002.
[7] R. M. Frieden. *Managing Internet-driven change in international telecommunications*. Artech House, 2001.
[8] A.-J. Moerdijk and L. Klostermann. Opening the networks with Parlay/OSA APIs: Standards and aspects behind the APIs. Draft of submitted article, 2002.
[9] Object Management Group. Meta Object Facility (MOF) v1.3.1. Specification, OMG, 2002.
[10] OMG Architecture Board MDA Drafting Team. Model driven architecture – A technical perspective. Technical report, OMG, 2001.
[11] J. Siegel and OMG Staff Strategy Group. Developing in OMG's model-driven architecture. Technical report, OMG, 2001.
[12] R. Soley and OMG Staff Strategy Group. Model driven architecture. White paper, OMG, 2000.
[13] The Parlay Group. Parlay APIs 2.1. Technical white paper, 2001.