Workshop Materials of the 1st Workshop on

Quality in Modeling

Co-located with

the ACM/IEEE 9th International Conference on

Model Driven Engineering Languages and Systems
(MoDELS 2006)

October 1st, 2006

Genoa, Italy

Organized by:

Ludwik Kuzniarz
Jean Louis Sourrouille
Ragnhild Van Der Straeten
Miroslaw Staron
Michel Chaudron
Alexander Förster
Gianna Reggio

Organizers

Ludwik Kuzniarz - chair
Blekinge Institute of Technology, Ronneby, Sweden
lku@bth.se

Jean Louis Sourrouille – co-chair, review process
INSA Lyon, Villeurbanne Cedex, France
Jean-Louis.Sourrouille@insa-Lyon.fr

Ragnhild Van Der Straeten – co-chair, publication process
System and Software Engineering Lab, Brussels, Belgium
rvdstrae@vub.ac.be

Miroslaw Staron - publicity
IT University, Gothenburg, Sweden
miroslaw.staron@ituniv.se

Michel Chaudron – workshop materials
Eindhoven University of Technology, The Netherlands
m.r.v.chaudron@tue.nl

Alexander Förster
University of Paderborn, Germany
alfo@uni-paderborn.de

Gianna Reggio
Università di Genova, Italy
gianna.reggio@unige.it

Program committee

Uwe Assmann, TU Dresden, Germany
Colin Atkinson, TU Braunschweig, Germany
Thomas Baar, EPFL, Switzerland
Benoit Baudry, IRISA-INRIA, Rennes, France
Lionel C. Briand, Simula Research Labs, Norway
Betty Cheng, Michigan State University,USA
Krzysztof Czarnecki, University of Waterloo , Canada
Alexander Eyged, University of Southern California, USA
Robert France, Colorado State University, USA
Gregor Engels, University of Paderborn, Germany
Martin Gogolla, University of Bremen, Germany
Reiko Heckel, Leicester University, UK
Heinrich Hussmann, TU Munchen, Germany
Anneke Kleppe, University Twente, The Netherlands
Mieczyslaw Kokar, Northeastern University, USA
Kai Koskimies, TU Tampere, Finland
Ludwik Kuzniarz, BTH, Sweden
Alexander Pretschner, ETH Zurich, Switzerland
Gianna Reggio, Universita di Genova, Italy
Bernhard Rumpe, Technische Universität Braunschweig, Germany
Pierre-Yves Schobbens, FUNDP, Belgium
Bran Selic, IBM Rational Software, Canada
Jean Louis Sourrouille, INSA Lyon, France
Miroslaw Staron, IT University, Sweden
Perdita Stevens, University of Edinburgh, UK
Hans Vangheluwe, McGill, Canada

Preface

Quality assessment and assurance constitute an important part of software engineering. The issues of software quality management are widely researched and approached from multiple perspectives and viewpoints. The introduction of a new paradigm in software development – namely Model Driven Development (MDD) and its variations (e.g., MDA [Model Driven Architecture], MDE [Model Driven Engineering], MBD [Model Based Development], MIC [Model Integrated Computing]) – raises new challenges in software quality management, and as such should be given a special attention. In particular, the issues of early quality assessment, based on models at a high abstraction level, and building (or customizing the existing) prediction models for software quality based on model metrics are of central importance for the software engineering community.

The workshop is continuation of a series of workshops on consistency that have taken place during the subsequent annual UML conferences and recently MDA-FA. The idea behind this workshop is to extend the scope of interests and address a wide spectrum of problems related to MDD. It is also in line with the overall initiative of the shift from UML to MoDELS.

The goal of this workshop is to gather researchers and practitioners interested in the emerging issues of quality in the context of MDD. The workshop is intended to provide a premier forum for discussions related to software quality and MDD. And the aims of the workshop are:

- Presenting ongoing research related to quality in modeling in the context of MDD,
- Defining and organizing issues related to quality in the MDD.

The format of the workshop consists of two parts: presentation and discussion. The presentation part is aimed at reporting research results related to quality aspects in modeling. Seven papers were selected for the presentation out of 16 submissions; the selected papers are included in these proceedings. The discussion part is intended to be a forum for exchange of ideas related to understanding of quality and approaching it in a systematic way.

Ludwik Kuzniarz
Workshop chair

Table of Contents

Efficient Decision of Consistency in UML Diagrams with Constrained Generalization Sets

Azzam Maraee[1] and Mira Balaban[2] *

[1] Information Systems Engineering Department
[2] Computer Science Department
Ben-Gurion University of the Negev, Beer-Sheva 84105, ISRAEL.
mari@bgu.ac.il, mira@cs.bgu.ac.il

Abstract. UML class diagrams are central to software development and to ontology engineering. The quality of a UML model affects the resulting software or ontology. One problem that relates directly to quality has to do with the consistency of the designed model. In presence of constraints, it is possible to specify class diagrams that cannot be realized by any finite, non-empty world. We believe that early detection of inconsistencies improves the quality of modeling, and yields a more reliable software.
In this paper we present an algorithm for determining consistency of class diagrams with constrained generalization sets. The algorithm is simple and efficient, and can be easily added to case tools. It improves over existing methods that require exponential resources. We study the limitations of this approach in presence of unrestricted class hierarchy structures.

Keywords:
UML class diagram, finite satisfiability, cardinality constraints, generalization set constraints, class hierarchy structure.

1 Introduction

The Unified Modeling Language (UML) is nowadays the industry standard modeling framework, including multiple visual modeling languages, referred to as UML models. Traditionally, UML models are used for analysis and design of complex systems. Their relevance has increased with the advent of the Model-Driven Development (MDD) approach, in which analysis and design models play an essential role in the process of software development. Recently, with the emergence of web-enabled agent technology, UML models are used also for ontology representation, construction and extraction [1, 7, 8].

Consequently, it is highly important that UML models provide a reliable support for the designed systems, and are subject to stringent quality assurance and quality control criteria [22]. Indeed, an extensive amount of research efforts is devoted to formalization of UML models, specification of their semantics,

* Supported by the Lynn and William Frankel Center for Computer Sciences.

and development of reasoning and correctness checking methods [3, 17]. Current UML case tools still do not test diagram correctness, and implementation languages still do not enforce design level constraints. Yet, with the prevalence of the Model Driven Engineering approach, it is expected that all information in a design model will be effective in its successive models.

Class Diagrams are probably the most important and best understood among all UML models. A Class Diagram provides a static description of system components. It describes systems structure in terms of classes, associations, and constraints imposed on classes and their inter-relationships. Constraints provide an essential means of knowledge engineering, since they extend the expressivity of diagrams. UML supports class diagram constraints such as cardinality constraints, class hierarchy constraints, and inter-association constraints. Example 1 below, presents a class diagram that includes cardinality and hierarchy constraint.

Example 1. Figure 1 presents a class diagram with three classes named *Advisor*, *MasterStudent* and *PhDStudent*, one association named *Advise* between instances of the *Advisor* and the *MasterStudent* classes, with roles named *advisor* and *ma*, respectively, a cardinality constraint that is imposed on this association, and a genralization set with a super class *Advisor* and sub-classes *MasterStudent* and *PhDStudent*. The cardinality constraint states that every *Master* student must be advised by exactly one *Advisor*, while every *Advisor* must advise exactly two *Master* students. The generalization set states that *Master* students and *PhD* students are *Advisors* as well, implying that the *Advisor* of a *Master* student can be a *Master* or a *PhD* student.

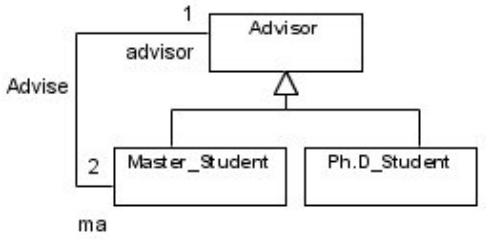


Fig. 1. A Class Diagram that is Strongly Unsatisfiable

In the presence of constraints, a model diagram may turn vacuous as it might impose constraints cannot be satisfied. For class diagrams, it means that no intended world can satisfy all constraints simultaneously. In Figure 1, the interaction between the cardinality constraint of the *Advise* association and the class hierarchy causes an inconsistency problem: Every *Master* student must have a single *Advisor*, while each *Advisor* should advise two *Master* students.

However, since the *MasterStudent* class specializes the *Advisor* class, every *Master* student must advise two *Master* students which have each a single *Advisor*. In order to satisfy this constraint, the *Master* class must be either empty or infinite. Since the intention behind class diagrams is to describe non-empty and finite classes, this diagram does not admit any intended world. Such diagrams are termed *strongly unsatisfiable*.

Inconsistency problems decrease the quality of a UML model, since they reflect a symptom of bug in the analysis phase [5]. Inconsistency might delay system development and increase its cost [14]. [15] shows that defects often remain undetected, even if the model is read attentively by practitioners. Therefore, it is necessary to enrich case tools with automatic inconsistency checking.

The problem of satisfiability has been studied in the context of various kinds of conceptual schemata [2, 4, 6, 9, 10, 16, 21]. There are methods for testing strong satisfiability, for detecting causes for unsatisfiability, and for heuristic suggestions for diagram correction. Yet, no method provides a feasible solution for detecting strong unsatisfiability for the combination of cardinality constraints and UML2.0 class hierarchy constraints.

In this paper, we present an algorithm for testing the satisfiability of UML class diagrams that include binary associations and class hierarchies with "disjoint/overlapping" and "complete/incomplete" constraints. The algorithm is based on a reduction to the algorithm of Lenzerini and Nobili [16] that was applied only to ER-diagrams without class hierarchies. It is simple and feasible since it adds in the worst case only a linear amount of entities to the original diagram. It improves over previous extensions of the Lenzerini and Nobili method that require the addition of an exponential number of new entities to the original diagram [6]. An implementation of our method within a UML case tool is currently under development. The paper is organized as follows: Section 2 summarizes relevant methods for detecting strong unsatisfiability in class diagrams, introduces the *Generalization Set* notion of UML2.0, and classifies different class hierarchy structures. In Section 3 we present polynomial time algorithms for testing satisfiability of UML class diagrams with unconstrained generalization sets. Section 4 extends the algorithms to operate on constrained generalization sets, and investigates the limits of these methods. Section 5 is the conclusion and discussion of future work.

2 Background

The standard set theoretic semantics of class diagrams associates a class diagram with *class diagram instances* in which classes have extensions that are sets of objects that share structure and operations, and associations have extensions that are relationships among class extensions. We denote class symbols as C, association symbols as A, and role symbols as rn. Henceforth, we shorten expressions like "instance of an extension of C" by "instance of C" and "instance of an extension of A" by "instance of A".

A cardinality constraint (also termed multiplicity constraint) imposed on a binary association A between classes C_1 and C_2 with roles rn_1, rn_2, respectively, is symbolically denoted:

$$A(rn_1 : C_1[\ min_1,\ max_1], rn_2 : C_2[\ min_2,\ max_2]) \tag{1}$$

The multiplicity constraint $[min_1, max_1]$ that is visually written on the rn_1 end of the association line is actually a participation constraint on instances of C_2. It states that an instance of C_2 can be related via A to n instances of C_1, where n lies in the interval $[min_1, max_1]$. A class hierarchy constraint between a super class C_1 and a subclass C_2 is written $ISA(C_2, C_1)$ and called also *ISA constraint*. It states a subset relation between extensions of C_2 and C_1.

A *legal instance* of a class diagram is an instance where the class and association extensions satisfy all constraints in the diagram. A class diagram is *satisfiable* if it has a legal instance. However, following [16], it appears that a more meaningful satisfiability property involves intended (desirable) instances, in which all class extensions are *finite*, and not all are *empty*. Instances in which all class extensions are empty are termed *empty instances*. A class diagram is *strongly satisfiable* if for every class symbol in the diagram there is an instance world with a non-empty and finite class extension[3]. Lenzerini and Nobili also show, for the restricted version of class diagrams that they deal with, that a strongly satisfiable class diagram has an instance in which all class extensions are non-empty and finite. The problem of testing whether a class diagram is strongly satisfiable is called the *consistency problem* of class diagrams.

2.1 Methods for Testing Consistency of Class Diagrams

The method of Lenzerini and Nobily is defined for *Entity-Relationship* (*ER*) diagrams that include *Entity Types* (*Classes*), *Binary Relationships* (*Binary Associations*), and *Cardinality Constraints*. The method consists of a transformation of the cardinality constraints into a set of linear inequalities whose size is polynomial in the size of the diagram. Strong satisfiability of the ER diagram reduces to solution existence of the associated linear inequalities system. The linear inequalities system is defined as follow:

1. For each association $R(rn_1 : C_1[min_1, max_1], rn_2 : C_2[min_2, max_2])$ insert four inequalities:

$$r \geq min_2 \cdot c_1,\ \ r \leq max_2 \cdot c_1,\ \ r \geq min_1 \cdot c_2,\ \ r \leq max_1 \cdot c_2$$

2. For every entity or association symbol T insert the inequality: $T > 0$

Lenzerini and Nobili also present a method for identification of causes for non-satisfiability. This method is based on a transformation of the conceptual schema into a graph and identification of critical cycles. Similar approaches are

[3] The notions of satisfiability and strong satisfiability were introduced by Lenzerini and Nobili [16] with respect to Entity-Relationship diagram.

introduced in [21, 9]. Hartman, in [10] further develops methods for handling satisfiability problems in the context of database key and functional dependency constraints. Heuristic methods for constraint corrections are presented in [11, 12].

Calvanese and Lenzerini, in [6], extend the inequalities based method of [16] to apply to schemata with class hierarchy constraints. The expansion is based on the assumption that class extensions may overlap. They provide a two stage algorithm in which the consistency problem of a class diagram with *ISA* constraints is reduced into the consistency problem of a class diagram without *ISA* constraints. Then, similarly to [16], they check satisfiability of the new class diagram by deriving a special system of linear inequalities (different from that of [16]).

The class diagram transformation process of [6] is fairly complex, and might introduce, in the worst case, an exponential number, in terms of the input diagram size, of new classes and associations. The method was further simplified in [5], were class overlapping is restricted to class hierarchy alone. The simplification of [5] reduces the overall number of new classes and associations, but the worst case is still exponential. Example 2 presents the application of [5] to Figure 1.

Example 2. The application of the [5] method yields four classes and eight associations. Each class and association is represented by a variable in the resulting inequalities system. The variables are:

1. **Class variables:** a_1 for an *Advisor* that is neither a *Master* nor a *PhD*; a_2 for an *Advisor* that is a *Master* but not a *PhD*; a_3 for an *Advisor* that is a *PhD* but not a *Master*; a_4 for an *Advisor* that is simultaneously a *Master* and a *PhD*;
2. **Association variables:** $\{ad_{ij} | 1 \leq i \leq 4 \wedge j \in \{2,4\}\}$. Every specialized association relates two new classes, one for the advisor role and the other for the *ma* role. The indexes represent the index of the class variable. For example the variable ad_{12} represents the specialization of the *Advise* association to an association between *Advisors* who are neither *Masters* nor *PhD* students (the a_1 variable) and *Advisors* specialized to *Masters* but not to *Ph.Ds* (the a_2 variable).

The inequalities system below results from application of the method of [5] to Figure 1. Equations 1-8 translate the 2..2 multiplicity, equations 9-12 translate the 1..1 multiplicity, and the inequalities in 13-26 represent the satisfiability conditions. The inequalities system is unsolvable, implying that the class diagram in Figure 1 is inconsistent.

- $1, 2.$ $\quad 2a_1 \leq ad_{12} + ad_{14}, \;\; 2a_1 \geq ad_{12} + ad_{14}$
- $3, 4.$ $\quad 2a_2 \leq ad_{22} + ad_{24}, \;\; 2a_2 \geq ad_{22} + ad_{24}$
- $5, 6.$ $\quad 2a_3 \leq ad_{32} + ad_{34}, \;\; 2a_3 \geq ad_{32} + ad_{34}$
- $7, 8.$ $\quad 2a_4 \leq ad_{42} + ad_{44}, \;\; 2a_4 \geq ad_{42} + ad_{44}$
- $9, 10.$ $\quad a_2 \leq ad_{12} + ad_{22} + ad_{32} + ad_{42}, \;\; a_2 \geq ad_{12} + ad_{22} + ad_{32} + ad_{42}$
- $11, 12.$ $\quad a_4 \leq ad_{14} + ad_{24} + ad_{34} + ad_{44}, \;\; a_4 \geq ad_{14} + ad_{24} + ad_{34} + ad_{44}$

– $13 - 24.$ $a_1, a_2, a_3, a_4, ad_{12}, ad_{14}, ad_{22}, ad_{24}, ad_{32}, ad_{34}, ad_{42}, ad_{44} \geq 0$
– $25.$ $a_1 + a_2 + a_3 + a_4 > 0,$
– $26.$ $ad_{12} + ad_{14} + ad_{22} + ad_{24} + ad_{32} + ad_{34} + ad_{42} + ad_{44} > 0$

2.2 UML2.0 Class Hierarchy Concepts: Generalization Sets

In UML2.0 class hierarchy constraints are expressed using the *Generalization Set* (*GS*) concept, which is similar to the former class hierarchy grouping construct [19]. *GS*s may be constrained as follows [19, 20]:

1. *complete* - Every instance of the super class must be an instance of at least one of the subclasses in the set.
2. *incomplete*- There are one or more instances of the particular general superclass of the that are not instances of any subclass in the set.
3. *disjoint*- The subclasses in the set are mutually exclusive.
4. *overlapping* - The subclasses in the set are not mutually exclusive. The specific subclasses in the set have one or more members in common.

The *GS* constraints can be combined to form one of the following valid combination: {complete, disjoint}, {incomplete, disjoint}, {complete, overlapping}, {incomplete, overlapping}. For example, Figure 2 includes a constrained *GS* that means: (1) there is at least one *Advisor* who is neither a *Master* nor a *PhD* (2) there is no *Advisor* who is both a *Master* and a *PhD*.

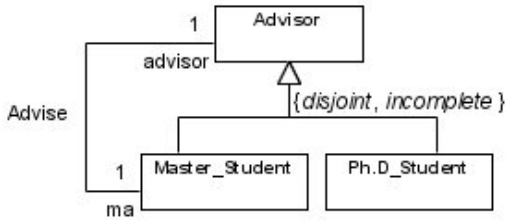


Fig. 2. Constrained Generalization Set

2.3 Classification of Class Hierarchy

Class hierarchy can arise in various structures that affect the satisfiability decision algorithm. We distinguish three parameters that determine the class hierarchy structures and content:

1. **Hierarchy Structure**: *ISA* constraints can form three graph structures:
   (a) **Tree Structure**: A subclass has only one super class, For example: Figure 1.

(b) **Acyclic Structure**: Multiple inheritance is allowed, but the undirected graph formed by the *ISA* constraints is acyclic. For example, in Figure 3, the hierarchy structure is not a tree, as F is a sub class of both C and D, but the undirected class hierarchy graph as acyclic. The acyclic structure prevents multiple inheritance with a common ancestor-class.

(c) **Graph Structure**: unrestricted multiple inheritance. For example, Figure 4.

2. **Presence of *GS* Constraints**.
3. **Number of *GS*s per Super-Class**: The case of multiple *GS*s per super-class is distinguished from the simpler case of a single *GS* per super-class. For example in Figure 4, class A has 2 *GS*s.

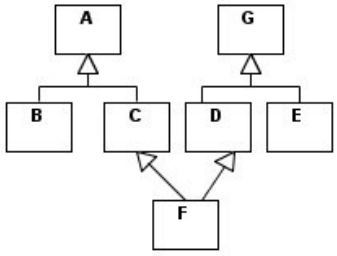


Fig. 3. Unconstrained Acyclic Hierarchy Structure

We use an abbreviated notation that specifies the value of each of the three parameters in a hierarchy under consideration. The *hierarchy structure* is denoted by one of {T,A,G}, standing for *Tree structure, Acyclic graph*, and *Graph*, respectively. The presence of *GS* constraints is denoted by *C*, and the presence of multiple *GS*s per super-class is denoted by *M*. The resulting hierarchy variants are: [T]-GS for tree structured unconstrained hierarchy with a single *GS* per super-class; [T-C]-GS for tree structured constrained *GS*s with a single *GS* per super-class; [T-C-M]-GS for a constrained tree hierarchy, with multiple *GS* per super class, [A]-GS for an unconstrained acyclic hierarchy; [A-C]-GS for a constrained acyclic hierarchy; [G]-GS for unconstrained graph hierarchy; [G-C]-GS for a constrained graph hierarchy. The multiple *GS*s per super-class is irrelevant outside the tree structure hierarchy.

3 Testing Consistency of UML2.0 class diagram with Unconstrained Generalization Sets

In this section, we present a method for testing the strong satisfiability of class diagrams with unconstrained *GS*s. We start with a tree structured hierarchy [T]-GS, and extend it to the rest of the hierarchical structures: {[T-M],[A],[G]}-GS.

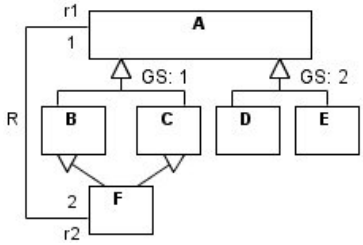


Fig. 4. Unconstrained Graph Hierarchy Structure

The method builds on top of the Lenzerini and Nobili [16] algorithm described in Section 2. We reduce the consistency problem of a class diagram with *ISA* constraints, into the consistency problem of a class diagram that is handled by [16]. First, we state that strong satisfiability implies a fully populated legal instance also in the presence of generalization sets. The proof is similar to the proof for the case of the restricted ER diagrams, as presented in [16]. The result applies only to acyclic structures.

Theorem 1. *If a class diagram with acyclic hierarchy structure and possibly constrained generalization sets is strongly satisfiable, then it has an instance in which all class extensions are non-empty and finite.*

Testing the Consistency of Class Diagrams with [T]-GS

Algorithm 1:

- **Input**: A class diagram CD that includes binary associations and [T]-GS.
- **Output**: True, if CD is strongly satisfiable; false otherwise.
- **Method**:
  1. Class diagram reduction - Create a new class diagram CD' as follows:
     (a) Initialize CD' by the input class diagram CD.
     (b) Remove from CD' all generalization set constructs.
     (c) For every removed generalization set construct create new binary associations between the superclass to the subclasses, with 1..1 participation constraint for the subclass (written on the super class edge in the diagram) and 0..1 participation constraint for the super class.
  2. Apply the Lenzerini and Nobili algorithm to CD'.

Example 3. Figure 5 is the reduced class diagram of Figure 1, following step 1 in the algorithm. Applying the inequalities method of [16] (step 2) yields the inequalities system presented below. We describe the inequalities system for Figure 5 using the symbols a for *Advisor*, m for *Master*, p for PhD, ad for *advise* and isa_1, isa_2 for the new associations $ISA1$, and $ISA2$ respectively.

$$ad \geq 2a,\ ad \leq 2a,\ ad \leq m,\ ad \geq m,\ isa_1 \geq m,\ isa_1 \leq m,\ isa_1 \leq a,\ isa_2 \geq p,$$
$$isa_2 \leq p,\ isa_2 \leq a$$

This system has no solution and therefore the [16] algorithm returns False. The same result was obtained in Section 2 by applying the [6],[5] algorithm to Example 1.

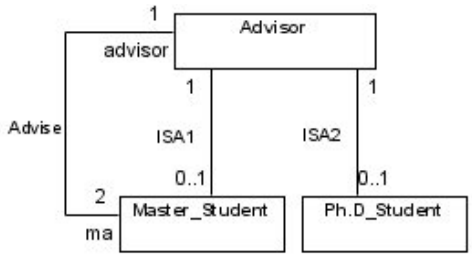


Fig. 5. The Reduced Class Diagram of Figure 1

Claim 1: [Correctness of Algorithm 1] Algorithm 1 tests for satisfiability of class diagrams with [T]-GS.

Proof. (sketched) the claim builds on showing that the translated class diagram CD' preserves the satisfiability of the input class diagram CD. Full proof appears in [18]

Claim 2: [Complexity of Algorithm 1] Algorithm 1 adds to the [16] method an $O(n)$ time complexity, where n is the size of the class diagram (including associations, classes and ISA constraints).

Proof. The additional work involves the class diagram reduction, which creates a class diagram with the same set of classes and one additional association that replaces every class hierarchy constraint. Since there is a linear additional work per generalization set, the overall additional work is a linear to the size of the class diagram.

Extensions for {[M], [A], [G]}-GS

Algorithm 1 applies properly also to the rest of the unconstrained structured: {[M], [A], [G]}-GS. The extensions preserve the correctness of the algorithm since the reduction of strong satisfiability of CD to that of CD' is still correct. The more complex structure does not break the reduction because in the lack of constraints on the GSs, ISA constraints can be simulated by regular links between the involved classes. Different instances of a super-class C in CD' can be unified into a single instance of C in CD, without breaking any constraints.

4 Testing the Consistency of Class Diagrams with Constrained Generalization Sets

In order to test satisfiability under constraints, the consistency problem of a class diagram CD with *ISA* constraints, is reduced into the consistency problem of a "constrained" class diagram CD' without class hierarchy. The additional constraints on CD' preserve the constraints on the *GS*s of CD. The inequalities system obtained by applying the method of [16] to CD' is expanded with new inequalities that reflect the *GS* constraints. We begin with an algorithm for [T-C]-GS. Then we show that it can be extended for [T-C-M]-GS. Finally we explore the limits of the algorithm for the {[G-C], [A-C]}-GS structures, where we show that for certain *GS* constraints it applies, while for other constraints it falls short for deciding consistency.

4.1 Testing the Consistency of Class Diagram with [T-C]-GS

Algorithm 2

- **Input**: A class diagram CD that includes binary associations and [T-C]-GS.
- **Output**: True, if CD is strongly satisfiable; false otherwise.
- **Method**:
  1. Class diagram reduction:
     (a) Steps 1.a, 1.b, 1.c from Algorithm 1.
     (b) For every generalization set $C, C_1, ..., C_n$ in CD, add constraint *Const* on its classes as follows:
     for *disjoint/overlapping* constraint, *Const* is: "there is no/(at least one) instance of class C who is associated with more than one instance from C_1,... C_n via the ISA links";
     for *complete/incomplete* constraint, *Const* is: "all/part of the instances of class C are associated with the instances of the classes $C_1, ..., C_n$ via the ISA links".
  2. Inequalities system construction:
     (a) Create the inequalities system for CD' according to the Lenzerini and Nobili algorithm.
     (b) For every constraint *Const* added in step 1b, extend the inequalities system, as follows:
        i. *Const* = *disjoint,incomplete*: $C > \sum_{j=1}^{n} C_j$
        ii. *Const* = *disjoint,complete*: $C = \sum_{j=1}^{n} C_j$
        iii. *Const* = *overlapping, complete*: $C < \sum_{j=1}^{n} C_j$
        iv. *Const* = *overlapping, incomplete*: $\forall j \in [1, n].C > C_j$

Example 4. Consider Figure 2. The interaction between the cardinality constraint, the hierarchy, and the *GS* constraints causes an inconsistency problem. Applying the method of [16] with the extension in Algorithm 2, step 2.b.i, to the reduced class diagram of Figure 2 yields the unsolvable inequalities system (same variables from Example 3) presented below, implying that the class diagram is inconsistent.

1. $ad \geq a$, $ad \leq a$, $ad \leq m$, $ad \geq m$, $isa_1 \geq m$, $isa_1 \leq m$, $isa_1 \leq a$, $isa_2 \geq p$, $isa_2 \leq p$, $isa_2 \leq a$
2. $a > m + p$.

Claim 3: [Correctness of Algorithm 2] Algorithm 2 tests for strong satisfiability of class diagrams with [T-C]-GS hierarchy structure.

Proof. (Sketched) The claim builds on showing that the translated class diagram CD' together with its associated constraints, preserves the strong satisfiability of the input class diagram CD. As for the second step of the algorithm, we show that for each constraint the additional inequality (or equality) provides a necessary and sufficient condition for the existence of a CD' instance that satisfies the generalization set constraint. For example, inequality [i] in step 2.b of Algorithm 2 characterizes the existence of a CD' instance that satisfies the referenced *disjoint, incomplete* constraint. Note that the additional inequalities are not exclusive. For example, inequality [i] implies inequality [iv], which implies the existence of a CD' instance that satisfies an *disjoint, incomplete* constraint. For full proof consult [18].

Claim 4: [Complexity of Algorithm 2] Algorithm 2 adds an $O(n)$ time complexity to the [16] method, where n is the size of the class diagram (including associations, classes and ISA constraints).

Proof. The additional work involves the class diagram reduction, which creates a class diagram with the same set of classes and one additional association that replaces every class hierarchy constraint. In addition, every *GS* constraint adds a single inequality. Since the work per generalization set is linear in its size, the overall additional work is linear in the size of the class diagram.

4.2 Extension for Class Diagrams with [T-C-M]-GS Hierarchy Structure

Algorithm 2 can determine consistency in class diagrams with [T-C-M]-GS hierarchy structure. We notice that the presence of single *GS* constraints, as in *disjoint*, requires refinements of the generalization set constraints.

1. *disjoint*: $C \geq \sum_{j=1}^{n} C_j$.
2. *complete*: $C \leq \sum_{j=1}^{n} C_j$.
3. *incomplete*: $\forall j \in [1, n].C > C_j$.
4. *overlapping*: No additional inequality is needed here.

4.3 Extension For Class Diagrams with {[G-C], [A-C]}-GS Hierarchy Structure - Exploring the Limits of the Suggested Method

The combination of *GS* constraints with a non-tree hierarchy structure provides additional expressivity, but makes consistency decision harder. Our method does not extend to the full case [G-C]-GS. Therefore, we carefully investigate its limits. We do it in a stepwise manner. First, we investigate [G-C]-GS hierarchies with a single *GS* constraint. Then, we extend our investigation to pairs of *GS* constraints.

[G-C]-GS Hierarchy Structure with a Single *GS* Constraint The Refinements for single constraint inequalities is like the ones in Section 4.2:

1. **Only *overlapping or complete***: Our method extends properly. Inconsistency problems caused by the *complete* constraint are directly reflected in the inequalities system and force it to be unsolvable. The *overlapping* constraint can alway be satisfied.
2. **Only *disjoint***: Our methods cannot determine consistency of class diagrams that include a disjointness *GS* constraint. The disjointness constraint can cause problems of enforced *class emptiness* or enforced *class infinity*. Both cannot be detected by our method.
   (a) **Emptiness**: The emptiness problem arises in [G-C]-GS structures that cyclic (as undirected graphs) and are assigned a disjointness constraint. For example, in Figure 4, the addition of a *disjoint* constraint enforces and empty extension to F.
   (b) **Infinity**: Infinity is caused by the interaction between cardinality constraints and a graph hierarchy structure. In Figure 6, for example, every legal instance is has either an empty or infinite extension for C.
3. **Only *incomplete*** : Our methods cannot determine consistency of class diagrams that include incomplete *GS* constraint. An incomplete constraint can cause an enforced infinity problem that cannot be detected by our method. For example, in Figure 6, if we replace the *disjoint* constraint by an *incomplete* constraint, and let class A be a subclass of E, then legal instance is has either an empty or infinite extension for D

[G-C]-GS Hierarchy Structure with Pairs of *GS* Constraints Our method can handle the pair constraint *complete, overlapping*, and falls short of handling either *complete, disjoint* or *incomplete, disjoint*.

Extension for [A-C]-GS. The problems which arise in unrestricted graph hierarchy with *GS* constraints do not arise when the hierarchy structure is acyclic (as undirected graph). Every instance of a class with multiple super-classes does not violate the disjointness constraint since all the parents of this super-classes are pairwise disjoint. Therefore, our method recognizes all the inconsistency problems occurring in this structure, both in the single constraint case and in the pair constraint case.

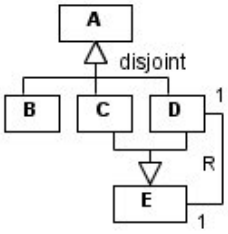


Fig. 6. Graph Hierarchy: Finitely Inconsistent Problem

Summary of the Extensions

Our method succeeds in determining inconsistency for the following cases: [G]-GS and {[T-C], [T-C-M], [A-C]}-GS with all the *GS* constraints. For the [G-C]-GS case, the *complete* and *overlapping* constraints can be handled, while the *disjoint* and *incomplete* cannot. The emptiness problem is instigated in [1, 3, 13].

5 Conclusions and Future Work

In this paper, we introduced a simple and effective algorithm for checking strong satisfiability of class diagrams with constrained *GS*. The advantage of our method lies in its simplicity and efficiency. In this paper we have also studied the limits of this method with respect to the interaction between class hierarchy structure to the kind of *GS* constraints.

In the future, we plan to explore the possible extension of the presented method for testing inconsistency in the presence n-ary association with complex cardinality constraints, qualifier constraints, and association classes.

Another direction involves the possibility of expanding our method with heuristics for detecting and repairing inconsistency constraints, following the ideas of [11, 12]. The intension is to apply similar strategies for repairing inconsistency in UML2 class diagrams with class hierarchy constraints.

References

[1] Baclawski, K., Kokar, M., Smith, J., Letkowski, J.: Consistency Checking of Ontologies Expressed in UML. International Conference on Formal Ontologies in Information Systems, (2001)

[2] Balaban, M., Shoval, P.: MEER-An EER Model Enhanced with Structure Methods. Information Systems, Volume 27, Issue 4 (2002)

[3] Berardi, D., Calvanese, D., Giacomo, De.: Reasoning on UML class diagrams. Artificial Intelligence (2005)

[4] Boufares, F., Bennaceur, H.: Consistency Problems in ER-schemas for Database Systems. Information Sciences, Issue 4 (2004)

[5] Cadoli, M., Calvanese, D, De Giacomo, G, Mancini, T.: Finite Satisfiability of UML Class Diagrams by Constraint Programming. In Proc. of the CP 2004 Workshop on CSP Techniques with Immediate Application, (2004)
[6] Calvanese, D., Lenzerini, M.: On the Interaction between ISA and Cardinality Constraints. Proc. of the 10th IEEE Int. Conf. on Data Engineering (1994)
[7] Cranefield, S., Hausteiny, S.: Purvis, M.: UML-Based Ontology Modelling for Software Agents. Proc. of Ontologies in Agent Systems Workshop, Montreal, (2001)
[8] Guizzardi, G., Wagner G., Guarino, N., van Sinderen, M.: An Ontologically well-Founded Profile for UML Conceptual Models. 16th International Conference on Advanced Information Systems Engineering (CAiSE), Latvia, (2004)
[9] Hartman, S.: Graph Theoretic Methods to Construct Entity-Relationship Databases. LNCS, Vol. 1017, (1995)
[10] Hartman, S.: On the Implication Problem for Cardinality Constraints and Functional Dependencies. Ann.Math.Artificial Intelligence, (2001)
[11] Hartman, S.: Coping with Inconsistent Constraint Specifications. LNCS, Vol. 2224, (2001)
[12] Hartman, S.: Soft Constraints and Heuristic Constraint Correction in Entity- Relationship Modeling. LNCS, Vol. 2582, (2002)
[13] Kaneiwa, K., Satoh, S.: Consistency Checking Algorithms for Restricted UML Class Diagrams. In Proceedings of the Fourth International Symposium on Foundations of Information and Knowledge Systems (2006)
[14] Kozlenkov, A., Zisman, A.: Discovering Recording, and Handling Inconsistencies in Software Specifications. Int. J. of Computer and Information Science 5(2), (2004)
[15] Lange, C., Chaudron, M., Muskens. J.: In Practice: UML Software Architecture and Design Description. IEEE Software, vol. 23, no. 2, (2006)
[16] Lenzerini, M. Nobili, P.: on the Satisfiability of Dependency Constraints in Entity-Relationship Schemata. Information Systems, Vol. 15, 4, (1990)
[17] Liang. P.: Formalization of Static and Dynamic UML Using Algebraic. Master's thesis, University of Brussel (2001)
[18] Maraee, A.: Consistency Problems in UML Class Diagram. Master' thesis, Ben-Gurion University of the Negev (2006)
[19] OMG.: UML 2.0 Superstructure Specification, (2005)
[20] Rumbaugh., J., Jacobson, G., Booch, G.: The Unified Modeling Language Reference Manual Second Edition. Adison Wesley (2004)
[21] Thalheim, B.: Entity Relationship Modeling, Foundation of Database Technology. Springer-Verlag, (2000)
[22] Unhelkar, B.: Verification and Validation for Quality of UML 2.0 Models. Addison-Wesley,(2005)

Consistency of Business Process Models and Object Life Cycles

Ksenia Ryndina[1,2], Jochen M. Küster[1], and Harald Gall[2]

[1] IBM Zurich Research Laboratory, Säumerstr. 4
8803 Rüschlikon, Switzerland {ryn,jku}@zurich.ibm.com
[2] Department of Informatics, University of Zurich, Binzmühlestr. 14
8050 Zurich, Switzerland gall@ifi.unizh.ch

Abstract. Business process models and object life cycles can provide two different views on behavior of the same system, requiring that these models are consistent with each other. Consistency is an important quality attribute for models, but in this case it is difficult to reason about consistency since the relation between business process models and object life cycles is not well-understood. We clarify the relation between these two model types and propose an approach to establishing their consistency. Object state changes are first made explicit in a business process model and then the process model is used to generate life cycles for each object type used in the process. We define two consistency notions for a process model and an object life cycle and express these in terms of conditions that must hold between a given life cycle and a life cycle generated from the process model.

1 Introduction

Business process models [9] are nowadays a well-established means for representing business processes in terms of tasks that need to be performed to achieve a certain business goal. In addition to tasks, business process models also show how *business objects* are passed between tasks in a process. Complete behavior of business objects is usually modeled using a variant of a state machine or a statechart [19] called an *object life cycle* (see e.g. [5]). Object life cycle modeling is valuable at the business level to explicitly represent how business objects go through different states during their existence.

There are situations where it is beneficial or even required to use both business process models and object life cycles. Consider an insurance company that uses business process models for execution and also maintains object life cycles for business objects. In this case, life cycles serve as a reference to employees for tracking progress of business objects. For instance, when an employee receives an enquiry about the state of a submitted claim, he/she can explain the current state of the claim to the customer in the context of the entire claim life cycle that shows all the possible states and transitions for claims. Another example is encountered in compliance checking, where existing business process models are benchmarked against best practice models (e.g. ACORD [2] and IFW [4])

given as object life cycles. Given a best practice object life cycle, it is required to ensure that an existing business process model is compliant with it.

In situations such as the ones described above, where both business process models and object life cycles are used, it is required that these models are *consistent* with each other. Inconsistencies can give rise to unsatisfied customers or to compliance violations. For example, customer discontent may emerge if the insurance company employee incorrectly informs the customer about the processing that still needs to be done before a claim is settled. On the other hand, inconsistencies between an existing process model and a best practice life cycle lead to compliance violations that can cause legal problems for a company.

Consistency of object-oriented behavioral models, such as scenarios and state machines, has already been extensively studied [11, 12, 18, 21]. However, the relation between business process models and object life cycles is not yet well-understood, which makes it impossible to reason about their consistency.

In this paper, we present our approach to establishing consistency of a business process model and an object life cycle. Prototype tool support for this approach has been implemented as an extension to the IBM WebSphere Business Modeler [1]. The remainder of the paper is structured as follows: In Section 2, we introduce subsets of *UML2.0 Activity Diagrams (UML AD)* and *State Machines (UML SM)* [3] chosen to represent business process models and object life cycles, respectively. In Section 3, we give an overview of the proposed solution, followed by a detailed description of the solution in the next three sections. Section 4 describes how we augment business process models to explicitly capture object state changes. Then in Section 5, we present a technique for generating life cycles for each object type used in a given process. In Section 6, we define two consistency notions for a business process model and an object life cycle and express these in terms of conditions that must hold between the given life cycle and the life cycle generated from the process model. Finally, we discuss related work in Section 7, and conclusions and future work in Section 8.

2 Business process models and object life cycles

UML AD constitute one of the most widely used languages for business process modeling. In this paper, we consider process models created using a subset of UML AD that includes the following elements: *action nodes* that represent tasks in a process and *control nodes* that show splits and merges of process execution paths, as well as beginning and end of process execution. Control nodes comprise *decision* and *merge nodes* that represent points in a process where one of several possible execution paths is taken, *fork* and *join nodes* that model parallel execution paths, *start nodes*[3] that model beginning of a process, and *flow final nodes* and *activity final nodes* that represent end of a process execution path and end of all process execution paths, respectively. All process nodes are connected with directed edges, which are used to represent *control* and *object*

[3] These are called *initial nodes* in UML AD, but renamed here to avoid confusion with initial states of object life cycles introduced later.

flows. Furthermore, *input* and *output pins* are used to model connection points that allow object flows to be attached to nodes, with the exception of start nodes that may not have outgoing object flows. Data inputs and outputs of processes are modeled using *input* and *output parameters*.

More advanced elements such as structured activity nodes, loop nodes, parameter sets and call behavior actions are not considered in this paper. We make a further simplification by associating data types with object flows instead of pins as done in UML AD, thus ensuring that data type is the same for two pins connected by an object flow. For the elements in the selected language subset, the semantics prescribed in the UML AD specification [3] are assumed.

Figure 1 shows an example business process model for a *Claims handling* process from the insurance industry that is represented in the chosen subset of UML AD. Elements highlighted in gray represent an extension to the notation introduced at a later stage and should be ignored at this point.

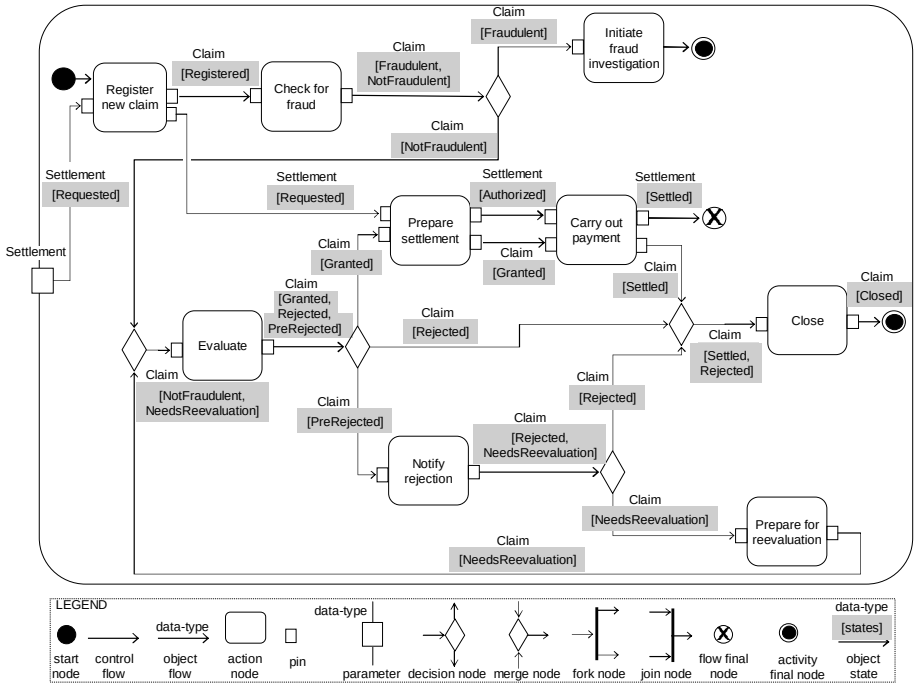


Fig. 1. *Claims handling* business process model

As can be seen from the diagram in Figure 1, the *Claims handling* process starts when a requested *Settlement* is received by the process, after which a new *Claim* is registered. The *Claim* further goes through a number of processing steps and at the end of the process it is either found to be fraudulent, or it is rejected or settled and subsequently closed.

For modeling object life cycles, we use a basic subset of the UML SM language. This subset comprises *states*, including one *initial state* and one or more

final states, and *transitions* connecting the states. Each state in an object life cycle that is not an initial or a final state has a unique name or *state label*. Transitions initiated by a particular triggering event are labeled with a *trigger label*, while *completion transitions* without an explicit trigger are unlabeled. As the main application context of this work is a business environment, a simple notation for object life cycles is chosen. Advanced features of UML SM, such as composite and concurrent states, are thus not considered here.

Figure 2 shows two example life cycles for *Claim* and *Settlement* object types. Such life cycles and the *Claims handling* process model introduced earlier could be developed in the same insurance company by the same or different modelers.

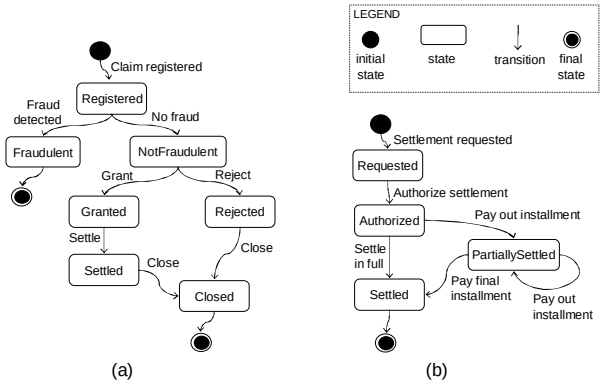


Fig. 2. Object life cycles: (a) *Claim* (b) *Settlement*

In Figure 2(a) it can be seen that all objects of type *Claim* go through state *Registered* directly after the initial state and pass through either *Fraudulent* or *Closed* states before they reach a final state. In the *Settlement* life cycle in Figure 2(b), it is shown that after a *Settlement* is *Authorized*, the payment for the *Settlement* can either be made in full or in a number of installments. Further, every object of type *Settlement* always reaches a final state through state *Settled*.

In this paper we use the following definition for an object life cycle, adapted from the definition of a UML State Machine in [16]:

Definition 1 (Object life cycle). *An* object life cycle *OLC for object type O is a tuple $(S, T, L, stL, trL, s_\alpha, S_\Omega)$, where:*

- *S is a finite set of* states*;*
- *$T \subseteq S \times S$ is a set of* transitions*, where each transition is an ordered pair (s_1, s_2) such that $s_1 \in S$ is the* source state *and $s_2 \in S$ is the* target state*;*
- *L is a finite set of* labels *that is further partitioned into a set of* state labels *L_s and a set of* trigger labels *L_t;*
- *$stL : S \setminus (\{s_\alpha\} \cup S_\Omega) \longrightarrow L_s$ is an injective function that associates each state that is not an initial or a final state with a unique state label;*
- *$trL : T \longrightarrow L_t$ is a partial function that associates a transition with a trigger label;*
- *$s_\alpha \in S$ is the* initial state*;*
- *$S_\Omega \subseteq S$ is a set of* final states.

Additionally to the above definitions, the following well-formedness constraints are defined for an object life cycle: the initial state has no incoming transitions; a final state has no outgoing transitions; all states that are not initial or final have at least one incoming and at least one outgoing transition.

3 Overview of proposed solution

For establishing consistency of business process models and object life cycles, we use an existing methodology for managing consistency of behavioral models [6, 10]. According to this methodology, the consistency problem must first be identified by determining the *overlap* between two given models. Then, aspects of the models that contribute to the consistency problem must be mapped into a suitable *semantic domain*, where *consistency conditions* can be defined and checked.

For business process models and object life cycles, the overlap between the models first needs to be made explicit. For achieving this, we augment the process model definition, such that states of objects passed along object flows in the process can be captured. Given this augmentation, there are two main alternatives for defining consistency conditions, shown in Figure 3.

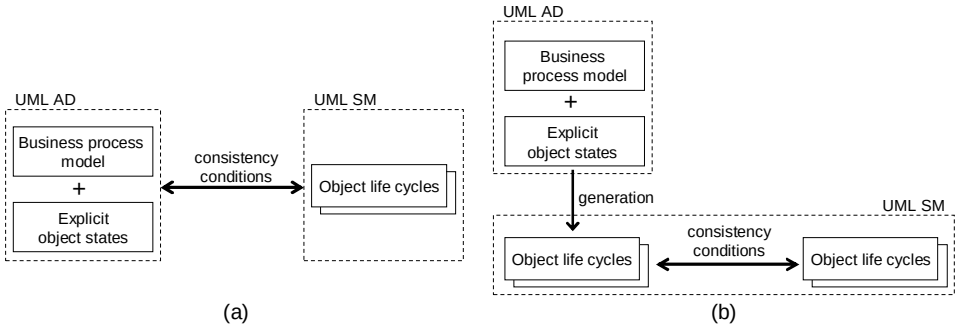


Fig. 3. Solution alternatives

Consistency conditions can be defined directly between a business process model with explicit object states and an object life cycle, as shown in Figure 3(a). In this case, no mapping of the given models is necessary, since the semantic domain contains both UML AD and SM. However, this approach requires defining consistency conditions across language boundaries, as UML AD and SM can be considered as two different modeling languages. This complicates the definition of consistency conditions.

The other alternative shown in Figure 3(b), uses UML SM as the semantic domain. Once object states are made explicit in a business process model, the model is mapped into the UML SM domain through the generation of object life cycles for each object type used in the modeled process. With this approach, consistency conditions are defined between models expressed in the same language, namely UML SM.

The latter approach is beneficial, as defining consistency conditions and interpreting inconsistencies between two models in the same language is easier than across language boundaries. Further, an object life cycle produced by the generation step provides an additional view on the original process model where one specific object type is in focus. This is valuable, as in complex business process models with many different objects being passed between actions, tracing one specific object is not easy.

As a consequence, we follow the second alternative in this paper. In the following three sections we describe the augmentation of a business process model with object states, generation of object life cycles from a process model, and consistency checking between the generated and the given object life cycles.

4 Business process models with object states

As objects are passed along object flows from one node to another in a business process model, it is only action nodes that perform work on these objects and hence make changes to their state. According to the semantics of UML AD, control nodes are executed without side-effects. Additionally, we assume that an object cannot change its state while it is being passed along an object flow.

Taking the mentioned assumptions into account, we augment business process models by providing a means of associating a set of states with an object flow to represent that objects passed along this flow are in one of the associated states. Given that all object flows in a business process model are associated with such state information, we can determine input and output states of all objects for each node in the process model. Figure 4(a) shows an example, where objects of type O are received in state s_1 by action A, passed to action B in either state s_2 or s_3 and further change their state to s_4 after action B executes.

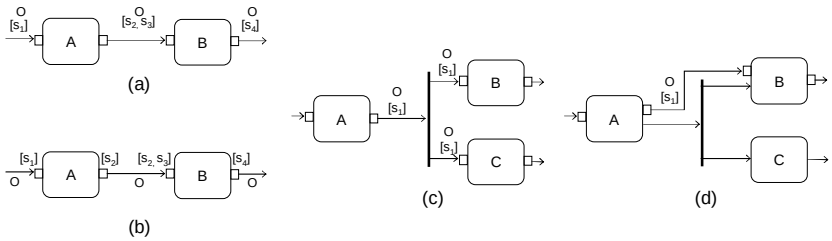


Fig. 4. Examples of explicit object state modeling

Referring back to Figure 1, we can see how state information (highlighted in gray) can be indicated for each object flow in the *Claims handling* process. For example, in this process *Claim* objects are created in state *Registered* by the *Register new claim* action. Further, objects of type *Claim* change state to either *Fraudulent* or *NotFraudulent* after *Check for fraud* action. *Claims* in state *Fraudulent* are routed along the upper object flow to *Initiate fraud investigation*, while those in state *NotFraudulent* are passed through the merge node to the

Evaluate action. It can also be seen that objects of type *Settlement* are passed to the process via an input parameter in state *Requested*.

In the current specification of UML AD [3], limited support for representing object state is provided. Input and output pins have an attribute called *inState* that can be used to associate a pin with a set of possible states of type *State* from UML SM. In the specification, this attribute defines "the required states of the object available at this point in the activity". No further semantic explanations, references or well-formedness constraints about the *inState* attribute are mentioned in the specification.

Associating state information with object flows is more appropriate than using input and output pins for process models considered here. Figure 4(b) shows an example where associating object states with pins allows for modeling of unreachable states. In the diagram, action B receives objects of type O from action A in state s_2, while state s_3 associated with the input pin of B is never reached. As our final goal is to generate object life cycles from a process model that capture states and transitions that occur in the process for a given object type, we want to exclude states such as s_3 that are not reachable in the process. Therefore, placing state information on object flows is more suitable to our needs, as it does not allow for modeling of unreachable states. Furthermore, when states are defined on pins, additional well-formedness constraints are required to forbid cases where mutually exclusive states are defined for two pins connected by an object flow.

For well-formedness of a business process model with object state modeled on object flows, it is required that an object state set associated with the incoming object flow of a decision node must be a union of the state sets associated with all the object flows going out of that decision node. All decision nodes in Figure 1 are well-formed: consider the decision node following the *Check for fraud* action and states associated with its incoming and outgoing object flows ($\{$*Fraudulent*, *NotFraudulent*$\} = \{$*Fraudulent*$\} \cup \{$*NotFraudulent*$\}$). Another well-formedness constraint applies to merge nodes and is the reciprocal of the constraint for decision nodes.

Well-formedness constraints for concurrent execution paths modeled with fork and join nodes are more challenging to define. This is partially due to the incomplete semantics of object flow on concurrent regions in the UML AD specification [3], where it is largely left up to the modeler to decide whether data is passed by reference or by value and what locking mechanisms are assumed for concurrent object access. Currently, we make a simplifying assumption that process models do not contain concurrent splits of object flows by fork nodes and merging of object flows by join nodes. Hence, we do not consider process models such as the example shown in Figure 4(c), because the fork node splits object flow of type O. However, we allow object flow to be routed to one of the execution paths in a concurrent region. This is illustrated in Figure 4(d), where the fork node splits control flow and object flow of type O is connected directly to an input pin of action node B.

In this section we have shown how object states are made explicit in business process models and what additional well-formedness constraints are necessary due to this augmentation. In the next section we explain how augmented process models are used to generate object life cycles.

5 Generation of object life cycles

An object life cycle generated from a given business process model for a particular object type should capture all possible state changes that can occur for objects of this type in the given process. Additionally, initial and final states need to be identified in the generated life cycles. We next describe how this is achieved with our generation technique.

Given a business process model P where each object flow is associated with a non-empty set of states, we generate an object life cycle for each object type used in P. For a particular object type O used in P, we first create an object life cycle OLC_P that contains only the initial state. Then, for each unique state associated with object flows of type O in process P, a state is added to OLC_P. Transitions and final states are added to OLC_P according to the generation rules shown in Figure 5.

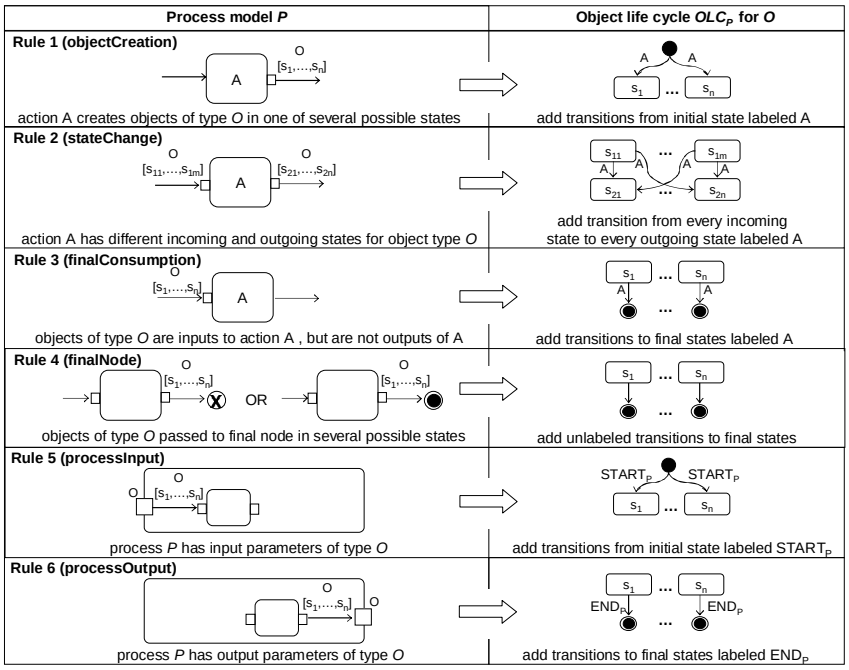


Fig. 5. Rules for object life cycle generation

Each row in Figure 5 represents a high-level generation rule, where the left-hand side shows patterns that are matched in process model P and the right-

hand side shows what is created in the generated object life cycle OLC_P. Each of the depicted rules is explained next.

Rule 1 (objectCreation) applies when some action A has an outgoing object flow of type O, but no incoming object flows of this type. The interpretation is that A creates objects of type O in one of several possible states. As shown on the right-hand side of the rule, transitions from the initial state to states associated with the outgoing object flow are added to OLC_P. These transitions are labeled A to indicate that they are triggered during the execution of this action.

Rule 2 (stateChange) is applicable when some action A has incoming and outgoing object flows of type O. When states of the outgoing object flow are not the same as those of the incoming object flow, we deduce that action A changes the state of objects of type O. In OLC_P, a transition labeled A from each incoming state to each possible outgoing state for objects of type O is added, for all cases where the outgoing state is different from the incoming state.

Rule 3 (finalConsumption) applies when an action A has an incoming object flow of type O, but no outgoing object flow of this type. The interpretation is that objects of type O are used by A, but are not passed further in the process and thus reach their final state. Transitions labeled A from each of the states of the incoming object flow to a new final state are added to OLC_P.

Rule 4 (finalNode) is applicable when there is an object flow of type O connecting some node in P to a flow final or an activity final node, as shown by the two patterns on the left-hand side of the rule. The interpretation is that on execution paths that end when the final node is reached, objects of type O are not further processed and thus reach their final state. Hence, transitions from each of the states of the object flow to a new final state are added to OLC_P. These transitions are unlabeled, as they can be considered as completion transitions.

Rule 5 (processInput) applies when the process model has an input parameter of type O and an object flow connects this parameter to some node in the process. The interpretation is that objects of type O are received by the process in one of the states associated with the object flow connected to the input parameter. In OLC_P, transitions from the initial state to each of the states of the object flow are added. These transitions are labeled START_P to indicate that they are triggered at the time when process P starts execution.

Finally, **Rule 6 (processOutput)** applies when the process model has an output parameter of type O that is connected by an object flow to some node in the process. The interpretation is that objects of type O leave the process in one of the states associated with the object flow connected to the output parameter. In OLC_P, transitions from each of the states of the object flow to a new final state are added. These transitions are labeled END_P to indicate that they are triggered at the time when process P ends execution.

As part of our prototype, we have implemented a generation algorithm that iterates over all the elements in a process model, searching for matches of the patterns defined for the generation rules and applying the rules when such matches are found. Provided that all object flows in the process model are associated with non-empty state sets, well-formedness of the generated life cycles is ensured by

the generation rules. A detailed definition of the generation algorithm is not included in this paper due to space limitations and will be the subject of a future publication.

Figure 6 shows life cycles for *Claim* and *Settlement* object types generated from the *Claims handling* process model with explicit object states (Figure 1).

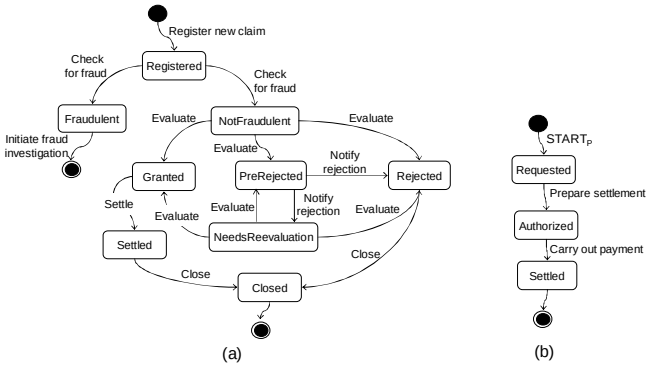


Fig. 6. Generated object life cycles: (a) *Claim* (b) *Settlement*

In the next section we show how generated object life cycles are used for defining consistency conditions to establish whether a given process model is consistent with a given life cycle for a particular object type.

6 Consistency of object life cycles

We identify two consistency notions for a given business process model and an object life cycle: *life cycle compliance* and *coverage*. A given process model is *compliant* with a given life cycle for a particular object type, if the process initiates only those state transitions for objects of this type that are defined in the given life cycle. Compliance allows objects of the given type to traverse only a part of their given life cycle in the given process. On the other hand, coverage requires that objects traverse the entire given life cycle in the given process, but additional transitions not defined in the given life cycle may be incurred in the given process model.

Depending on the circumstances, one or both of the introduced consistency types may be required to hold. For example, if the *Claims handling* process (Figure 1) is used for execution and the *Claim* life cycle (Figure 2(a)) is referenced by employees for interpreting the state of *Claim* objects, both compliance and coverage must hold. If the process is not compliant with the life cycle and takes *Claim* objects into states not shown in the life cycle or performs different transitions, this will disconcert the employees. On the other hand, customers will be incorrectly informed and thus unsatisfied if the process does not provide a coverage of the life cycle. An example of this occurs if a customer expecting a

Claim in state *Granted* to eventually reach state *Settled* according to the given life cycle, but this never happens in the *Claims handling* process.

We next give more precise definitions of compliance and coverage, providing consistency conditions that must hold between a life cycle generated from a process model for a particular object type and a given life cycle for that type. We first give two definitions that simplify the expression of consistency conditions that follow. Definitions 2 and 3 can be applied to any two object life cycles: $OLC = (S, T, L, sL, tL, s_\alpha, S_\Omega)$ and $OLC' = (S', T', L', sL', tL', s'_\alpha, S'_\Omega)$.

Definition 2 (State correspondence). *A* state correspondence *exists between a state $s \in S$ and a state $s' \in S'$, if and only if one of the following holds:*

- *$s = s_\alpha$ and $s' = s'_\alpha$;*
- *$s \in S_\Omega$ and $s' \in S'_\Omega$.*
- *$stL(s) = stL'(s')$;*

Definition 3 (Transition correspondence). *A* transition correspondence *exists between a transition $t = (s_1, s_2) \in T$ and a transition $t' = (s_3, s_4) \in T'$ if and only if there are state correspondences between s_1 and s_3, and between s_2 and s_4.*

In Definition 2, we define a *state correspondence* between two states if they are both initial states, they are both final states or their state labels are the same. In Definition 3, we define a *transition correspondence* between two transitions if there are state correspondences between their sources states and between their target states.

In Definitions 4 and 5, P is a given process model, $OLC = (S, T, L, sL, tL, s_\alpha, S_\Omega)$ is a given life cycle for object type O and $OLC_P = (S_P, T_P, L_P, sL_P, tL_P, s_{\alpha_P}, S_{\Omega_P})$ is the life cycle generated from P for O.

Definition 4 (Life cycle compliance). *A business process model P is* compliant *with an object life cycle OLC if and only if for each transition $t_P \in T_P$ that is not labeled* START_P *or* END_P*, there exists a transition $t \in T$ such that there is a correspondence between t_P and t.*

According to Definition 4, life cycle compliance requires that each transition in the generated object life cycle has a transition correspondence to some transition in the given life cycle. However, there are two exceptions to this consistency condition: transitions labeled START_P and END_P in the generated object life cycle. These transitions are generated when the given process model P has input or output parameters of object type O. We do not place restrictions on these transitions, thus allowing objects of type O to be received by and passed from the given process in any state and not necessarily a state following the initial state or preceding a final state.

Definition 5 (Life cycle coverage). *A business process model P provides a* coverage *of an object life cycle OLC if and only if all of the following conditions hold between OLC and OLC_P:*

(a) For each transition $t \in T$ there exists a transition $t_P \in T_P$ such that there is a correspondence between t and t_P.
(b) There are no transitions labeled START_P *or* END_P *in T_P.*

Condition(a) in Definition 5 requires every transition in the given object life cycle to have a transition correspondence to some transition in the generated life cycle. Furthermore, condition(b) requires that the given process does not have input or output parameters of the given type, hence objects of this type must be created and reach their final states within the process boundaries.

We next illustrate the notions of life cycle compliance and coverage using examples. Figure 7 shows the given object life cycles for the *Claim* and *Settlement* object types on the left and the object life cycles generated from the *Claims handling* process on the right. Transitions that have a correspondence between them are marked with the same number, while transitions without a correspondence are marked with a cross.

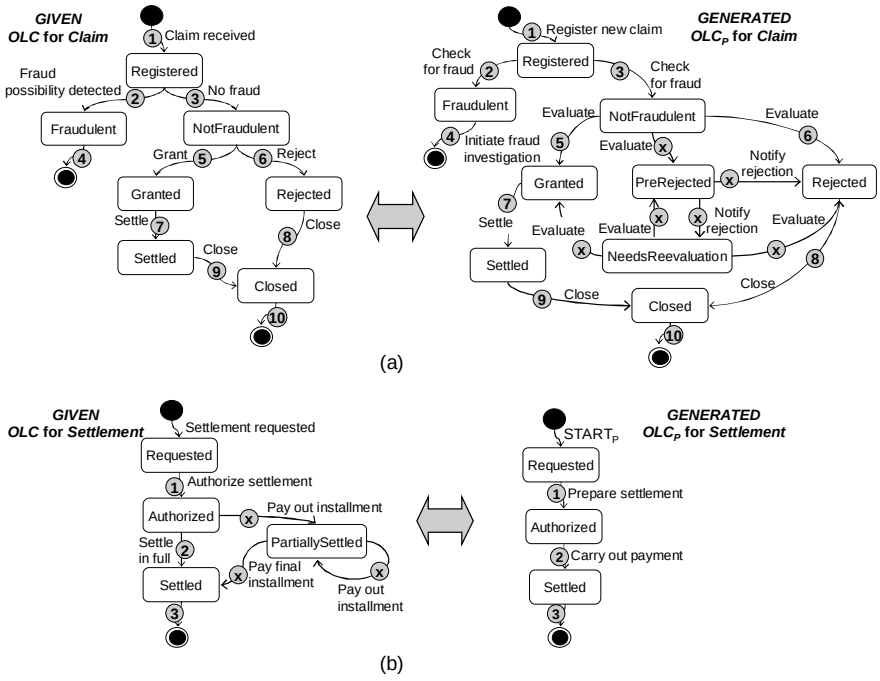


Fig. 7. Consistency of *Claim* and *Settlement* object life cycles

In Figure 7(a), it can be seen that the shown object life cycles satisfy all the consistency conditions for life cycle coverage. As all the transitions in the given life cycle for *Claim* have a correspondence to transitions in the generated life cycle, condition(a) from Definition 5 is satisfied. Further, the generated life cycle does not contain transitions labeled START_P or END_P, as required by condition(b) for life cycle coverage. Therefore, the *Claims handling* process provides a coverage of the given *Claim* life cycle. However, the *Claims handling* process is not compliant with this life cycle, due to transitions in the generated life cycle without transition correspondences to transitions in the given life cycle.

Figure 7(b) shows that the *Claims handling* process is compliant with the given *Settlement* life cycle, but does not provide a coverage for it.

The prototype that we have implemented determines life cycle compliance and coverage for a given business process model and an object life cycle by checking the consistency conditions defined in this section.

7 Related work

A research area closely related to our work is object life cycle inheritance, where consistent specialization of object behavior is required (see e.g. [5, 8, 13, 15, 16]). Ebert and Engels [5] distinguish between life cycles representing *observable* and *invocable* behavior, and define consistency notions for specialization of each life cycle type. With respect to this classification, object life cycles considered in this paper represent observable behavior of business objects. In our work, state transitions are not linked to object methods that can be directly invoked and thus invocable behavior is not in our focus.

In their comprehensive work, Schrefl and Stumptner [13, 16] define consistency conditions for life cycle specialization by *extension* and *refinement*. Different types of consistent specialization, such as *observation consistent extension*, are defined based on traces of states traversed by objects called *life cycle occurrences* and traces of executed transitions called *activation sequences*.

Currently, the main goal of our work is to establish a link between business process models and object life cycles, and thus life cycle inheritance is not in our main focus. However, there may be situations where it is required that the relation between a given business process model and an object life cycle is a certain type of specialization. Thus, it would be beneficial for our approach to make use of the consistency notions already defined for life cycle inheritance.

Another related area is synthesis of state machines from scenarios [21, 18], where the goal is to use given scenario specifications to generate state machines for different objects that participate in these scenarios. Techniques used for state machines synthesis from scenarios are different from our life cycle generation, due to several fundamental differences between business process models and scenarios. While scenarios represent interaction between objects via messages, process models show the flow of objects between tasks. Furthermore, process models generally capture all the possible task executions relevant to the modeled process and do not describe alternative scenarios. In state machine synthesis, it is possible that a synthesized state machine contains so-called *implied scenarios* [17, 14], which are behaviors that are not valid with respect to the original scenario specifications. Under certain conditions, a similar phenomenon can occur in our life cycle generation step. We leave a more detailed investigation of this issue for future work.

Consistency of behavioral models as discussed in this paper is conceptually related to notions of *equivalence* and *refinement*, which have been thoroughly studied in the context of formal process specifications [7]. However, as discussed in [20], it is challenging to directly apply the existing equivalence and refinement definitions to modeling languages such as UML AD and SM, as they do

not have an agreed formal semantics. Additionally, equivalence checking is not always well-suited for practical application due to its high computational complexity. Regardless, as future work we intend to establish a clear relation of our consistency notions to the existing equivalence and refinement definitions and investigate which are most appropriate in practice.

8 Conclusion and future work

Consistency of business process models and object life cycles needs to be established in situations where process models manipulate business objects with an explicitly modeled life cycle.

In this paper we have presented our approach to establishing consistency between business process models and object life cycles. There are three main contributions of our approach: We bridge the conceptual gap between business process models and object life cycles by making object states explicit in process models and describing the required well-formedness constraints. The second contribution is a technique for generating life cycles from process models. The third contribution is a precise definition of consistency conditions that can be used to check two consistency notions for a given business process model and an object life cycle, namely compliance and coverage. With regards to tool support, we have developed a prototype as an extension to the IBM WebSphere Business Modeler [1] that allows us to capture object states in business process models, generate life cycles from process models and check the consistency conditions.

There are several directions for future work. Firstly, we need to extend our approach to handle hierarchical nesting and concurrent object access in process models. Secondly, we intend to adapt our consistency conditions to check compliance and coverage between several process models that use objects of the same type and a given life cycle for this object type. Further future work includes an investigation of implied scenarios in the context of our life cycle generation and relation between our consistency notions and the existing equivalence and refinement definitions.

Acknowledgement: The authors would like to thank Michael Wahler for his valuable comments on an earlier version of this paper.

References

1. IBM WebSphere Business Modeler. http://www-306.ibm.com/software/integration/wbimodeler/.
2. ACORD Life & Annuity Standard. ACORD Global Insurance Standards, Final Version 2.13.00, September 2005.
3. UML2.0 Superstructure, formal/05-07-04. OMG Document, 2005.
4. IBM Industry Models for Financial Services, The Information Framework (IFW) Process Models. IBM General Information Manual, 2006.
5. J. Ebert and G. Engels. Specialization of Object Life Cycle Definitions. Fachberichte Informatik 19/95, University of Koblenz-Landau, 1997.

6. G. Engels, J. M. Küster, L. Groenewegen, and R. Heckel. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In *Proceedings of the 8th European Software Engineering Conference - ESEC'01*, pages 186–195. ACM Press, 2001.
7. A.-W. Fayez. Comparative Analysis of the Notions of Equivalence for Process Specifications. In *Proceedings of the 3rd IEEE Symposium on Computers & Communications - ISCC'98*, page 711, Washington, DC, USA, 1998. IEEE Computer Society.
8. D. Harel and O. Kupferman. On the Inheritance of State-Based Object Behavior. Technical Report MCS99-12, Weizmann Institute of Science, Faculty of Mathematics and Computer Science, 1999.
9. M. Havey. *Essential Business Process Modeling.* O'Reilly, 2005.
10. J. M. Küster. *Consistency Management of Object-Oriented Behavioral Models.* PhD thesis, University of Paderborn, March 2004.
11. J. M. Küster and J. Stehr. Towards Explicit Behavioral Consistency Concepts in the UML. In *Proceedings of the 2nd International Workshop on Scenarios and State Machines: Models, Algorithms and Tools*, 2003.
12. B. Litvak, S. Tyszberowicz, and A. Yehudai. Behavioral Consistency Validation of UML Diagrams. *1st International Conference on Software Engineering and Formal Methods -SEFM'03*, page 118, 2003.
13. M. Schrefl and M. Stumptner. Behavior-Consistent Specialization of Object Life Cycles. *ACM Transactions on Software Engineering and Methodology*, 11(1):92–148, 2002.
14. H. Muccini. An Approach for Detecting Implied Scenarios. In *Proceedings of the Workshop on Scenarios and State Machines: Models, Algorithms, and Tools - ICSE'02*, 2002.
15. J. L. Sourrouille. UML Behavior: Inheritance and Implementation in Current Object-Oriented Languages. In *Proceedings of UML'99*, volume 1723 of *LNCS*, pages 457–472. Springer-Verlag, 1999.
16. M. Stumptner and M. Schrefl. Behavior Consistent Inheritance in UML. In *Proceedings of Conceptual Modeling - ER 2000*, volume 1920 of *LNCS*, pages 527–542. Springer-Verlag, 2000.
17. S. Uchitel, J. Kramer, and J. Magee. Detecting Implied Scenarios in Message Sequence Chart Specifications. In *Proceedings of European Software Engineering Conference (ESEC/FSE'01)*, 2001.
18. S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *IEEE Transactions on Software Engineering*, 29(2):99–115, 2003.
19. M. von der Beeck. A Comparison of Statecharts Variants. In *Proceedings of Formal Techniques in Real Time and Fault Tolerant Systems*, pages 128–148. Springer-Verlag, 1994.
20. M. von der Beeck. Behaviour Specifications: Equivalence and Refinement Notions. In *Visuelle Verhaltensmodellierung verteilter und nebenläufiger Software-Systeme, 8. Workshop des Arbeitskreises GROOM der GI Fachgruppe 2.1.9 Objektorientierte Software-Entwicklung, Universität Münster*, 2000. Techreport 24/00-I.
21. J. Whittle and J. Schumann. Generating Statechart Designs from Scenarios. In *Proceedings of the 22nd International Conference on Software Engineering - ICSE'00*, pages 314–323, New York, NY, USA, 2000. ACM Press.

Model Metrics and Metrics of Model Transformation

Motoshi Saeki[1] and Haruhiko Kaiya[2]

[1] Dept. of Computer Science, Tokyo Institute of Technology
Ookayama 2-12-1, Meguro-ku, Tokyo 152, Japan
[2] Dept. of Computer Science, Shinshu University
Wakasato 4-17-1, Nagano 380-8553, Japan
saeki@se.cs.titech.ac.jp,kaiya@cs.shinshu-u.ac.jp

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

1 Introduction

To develop information systems of high quality, measuring quality in the earlier phases of the development process such as a modeling phase is one of the key issues. There are wide varieties of modeling methods such as object-oriented modeling, data flow modeling, activity modeling etc. and many kinds of models are produced following these modeling methods. For example, an object-oriented modeling method mainly adopt class diagrams consisting of classes and their associations, while in data flow modeling data flow diagrams having processes (data transformation), data flows and data stores, etc. are used. In this situation, according to models, we should have different metrics to measure their quality, and it is necessary to define the metrics according to the modeling methods. For example, in the object-oriented method, we can use the CK metrics [6] to measure the structural complexity of a produced class diagram. The technique of Cyclomatic number [12] can be applied to an activity diagram of UML (Unified Modeling Language) in order to measure its complexity, so that we can avoid constructing a structurally complicated activity diagram. These examples show that effective metrics vary on a modeling method.

The existing metrics such as CK metrics and Halstead's Software Science are for expressing the structural, i.e. syntactical characteristics of a product only, but do not reflect its semantic aspects. Suppose that we have two class diagrams of Lift Control System, which are the same except for the existence of class "Emergency Button"; one includes it, while the other one does not. It can be considered that the diagram having "Emergency Button" is structurally more complex rather than the other, because the number of the classes included in it is larger. However, it has higher quality in the semantics of Lift Control System because Emergency Button is mandatory for the safety of passengers in a practical Lift Control Systems. This example shows that the metrics

expressing semantic aspects is necessary to measure the quality of products more correctly and precisely. In particular, these semantic aspects are from the properties specific to problem and application domains.

In Model Driven Development (MDD), model transformation is one of the key technologies [2, 13] and the transformations that can improve the quality of models are significant. One of the problems in MDD is how to identify what transformations can improve the quality of models. If a metrics value can express the quality of a model, the changes of the metrics values before and after a model transformation can be the improvement of the model quality. It means that the formal definition of a transformation should include the specification of metrics so that the metrics can be calculated during the transformation. Furthermore the quality of model transformation as well as the quality of models should be considered and formally defined based on the improvement degree of the quality of the model.

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

1. Meta modeling technique
   Since a meta model defines the logical structure of models, we specify the definition of metrics, including its calculation technique, as a part of the meta model. Thus we can define model-specific metrics formally. We use Class Diagram plus Object Constraint Language (OCL) to represent meta models with metrics definitions.
2. Domain ontology
   We use a domain ontology to provide for the model the semantics specific to a problem and application domain. As mentioned in [11], we consider an ontology as a thesaurus of words and inference rules on it, where the words in the thesaurus represent concepts and the inference rules operate on the relationships between the words. Each concept of an ontology can be considered as a semantically atomic element that anyone can have the unique meaning in the domain. The thesaurus part of the ontology plays a role of a semantic domain in denotational semantics, and the inference rules can automate the detection of inconsistent parts and of the lacks of mandatory model elements [9]. Thus we can calculate semantic metrics such as the degree on how many inconsistent parts are included in the model, using information on mapping from a model to a domain ontology. Intuitively speaking, the semantic metrics value is based on the degree of how faithfully the model reflects the structure of the domain model.
3. Graph rewriting system
   Since we adopt Class Diagram to represent a meta model, a model following the meta model is mathematically considered as a graph. Model transformation rules can be defined as graph rewriting rules and the rewriting system can execute the transformation automatically. Metrics values can be evaluated and propagated between the models during the transformation. The evaluation and propagation methods can be defined within the graph rewriting rules. Furthermore we can define the quality of a transformation with an increase in the quality metrics values before and after the transformation.

The usages of the meta modeling technique for defining model-specific metrics [15] and of graph rewriting for model transformation [7, 10, 14] are not new, but the contribution of this paper is the integrated application technique of meta modeling, domain ontologies and graph rewriting to solve simultaneously the above three problems with unified framework.

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

2 Meta Modeling

Roughly speaking, the description of a modeling method consists of product and process parts. The product parts, so called meta model, specify the structure or data type of the products. On the other hand, the process part specifies the activities for constructing the products, such as "at first, identify classes and objects" to construct a class diagram. In this paper, since we discuss the quality of the models, not the activities, we focus on the product part only. In addition, we should consider constraints on the models. Suppose that we define the meta model of the models which are described with class diagrams. In any class diagram, we cannot have different classes having the same name, and we should specify this constraint to keep consistency of the models on their meta model.

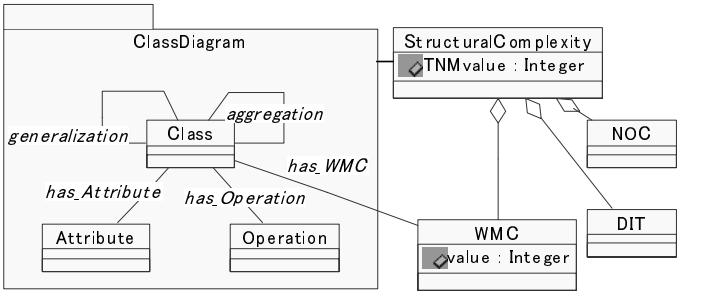
In our technique, we adopt a class diagram of UML for specifying meta models and OCL [20] for constraints on models. The example of the meta model of the simplified version of class diagrams is shown in Figure 1 (a). As shown in the figure, it has the concepts "Class", "Operation" and "Attribute" and all of them are defined as classes and these concepts have associations representing logical relationships among them. For instance, the concept "Class" has "Attribute", so the association "has_Attribute" between "Class" and "Attribute" denotes this relationship.

Metrics is defined as a class having the attribute "value" in the meta model as shown in the Figure 1 (b). The "value" has the metrics value and its calculation is defined as a constraint written with OCL. For example, WMC (Weighted Method per Class) of CK metrics is associated with each class of a class diagram and it can be defined as follows[3].

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

where predicates $has\_WMC(x, y)$ and $has\_Operation(u, v)$ express that the class x has y of the WMC class and that the class u has the operation v. These predicates are from the associations on the meta model of Class Diagram as shown in Figure 1 (a). #

[3] For the readers unfamiliar to OCL notation and its built-in identifiers, we do not use the OCL notation but usual mathematical notation including formal logic.



(a) Meta Model of Class Diagram

(b) Meta Model of Structural Complexity Metrics

Fig. 1. Meta Model with Syntactical Metrics Definitions

P denotes the cardinality of the set P. The technique of using OCL on a meta model to specify metrics was also discussed in [5, 15]. WMC and the other CK metrics are for a class not for a class diagram. Thus we use the maximum number of WMCs in the class diagram or the average value to represent the WMC for the class diagram. In this example, which is used throughout the paper, we take the sum total of WMCs for the class diagram, and the attribute TNMvalue of StructuralComplexity holds it as shown in Figure 1 (b).

We have a meta CASE tool called CAME (Computer Aided Method Engineering) [16], which takes meta models as inputs and generates diagram editors having the functions of calculating defined metrics. Figure 2 illustrates a screen snap shot of this meta CASE. The process aspect of a modeling method is represented with an activity diagram as shown in the right part of this figure. The left part is the meta model of Class Diagram, a complicated version to Figure 1, and the window CAMEPackage displays the definition of WMC metrics as a constraint. When a model engineer (an engineer to develop a model), who is the user of the generated diagram editor, reaches the second activity "CalculateStructuralComplexity", the value of WMC is automatically calculated according to the OCL description, because it is specified in the meta model that the activity produces WMC instances. The details of this tool is out of scope of the paper, and is shown in [16] and [15].

3 Using Domain Ontologies

Ontology technologies are frequently applied to many problem domains nowadays [8, 19]. As mentioned in section 1, an ontology plays a role of a semantic domain in denotational semantics. Basically, our ontology is represented in a directed typed graph where a node and an arc represent a concept and a relationship (precisely, an instance of a relationship) between two concepts, respectively.

Let's consider how a model engineer uses a domain ontology to measure the quality of his or her models. During developing the model, the engineer should map its model

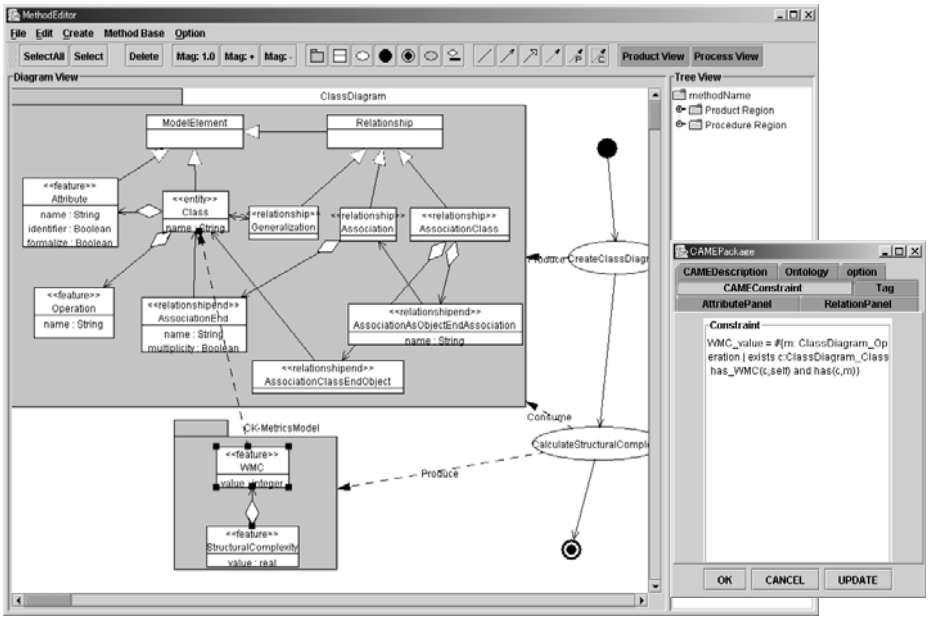


Fig. 2. Meta CASE tool

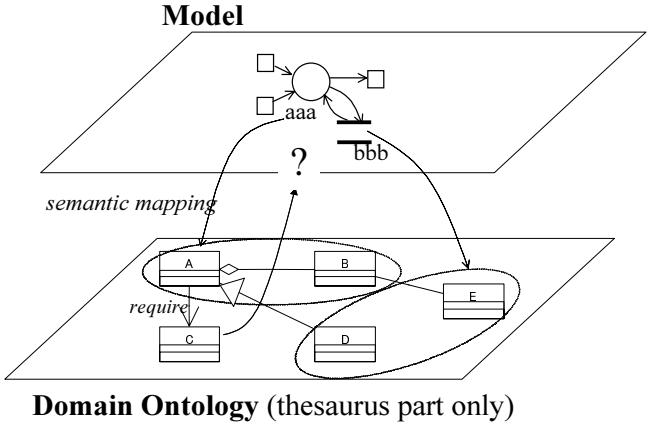


Fig. 3. Mapping from a Model to an Ontology

element into atomic concepts of the ontology as shown in Figure 3. In the figure, the engineer develops a data flow diagram, while the domain ontology is written in the form of class diagrams. For example, the element "aaa" in the data flow diagram is mapped into the concepts A, B and the relationship between them. Formally, the engineer specifies a semantic mapping $\mathcal{F}$ where $\mathcal{F}$(aaa) = {A, B, a relationship between A and B}. In the figure, although the model includes the concept A, it does not have the concept C, which is required by A. Thus we can conclude that this model is incomplete because a necessary element, i.e. the concept C is lacking, and we can have the metrics of completeness (COMPL) by calculating the ratio of the lacking elements to the model elements, i.e.

$$COMPL\_value = 1 - \#\{c1 \mid \exists e, u, c2 \cdot (require(e,u) \wedge semantic\_mapping(c1,e) \wedge semantic\_mapping(c2,u) \wedge \neg(c1 \in ModelElement) \wedge c2 \in ModelElement)\} / \#ModelElement \quad (2)$$

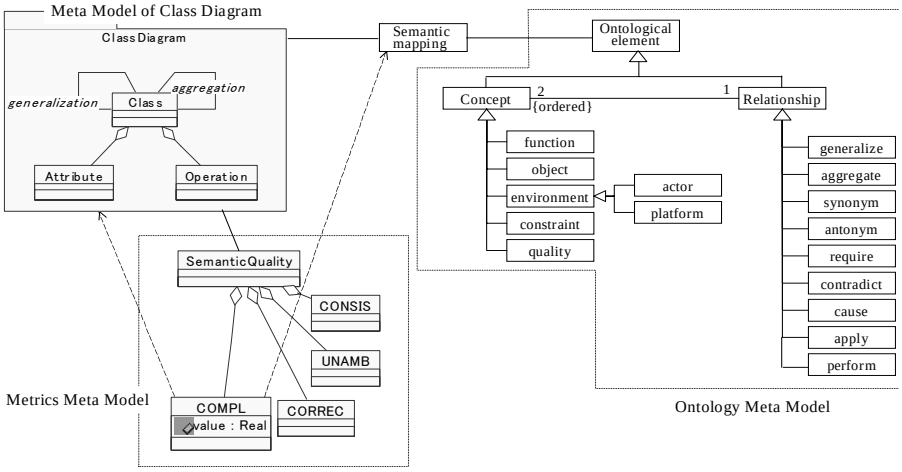


Fig. 4. Combining an Ontology Meta Model to a Meta Model

The meta model combined with this semantic metrics definition can be illustrated in Figure 4 in the same way as the syntactical metrics of Figure 1. The right part of the figure is the meta model of the thesaurus part of domain ontologies. Thesauruses consist of concepts and relationships among the concepts, and they have a variety of subclasses of the "concept" class and "relationship". In the figure, "object" is a subclass of a concept class and a relationship "apply" can connect two concepts. Concepts and relationships in Figure 4 are introduced so as to easily represent the semantics of the models of information systems. A semantic mapping plays a role of the bridges between the model written in class diagram and a domain thesaurus, and the model engineer provides the semantic mapping during his or her development of the model. In the figure, as the

examples of semantic metrics, there are four metrics completeness (COMPL), consistency (CONSIS), correctness (CORREC) and unambiguity (UNAMB), which resulted from [4]. Their values are calculated from the model, the thesaurus and the semantic mapping, and the attribute "value" of the metrics holds the calculation result. Similar to Figure 1 and the formula (1), the calculation formulas are defined as constraints and the example of $COMPL\_value$ (the attribute "value" in COMPL) was shown in the formula (2).

Figure 5 shows a part of an ontology of Lift Control Systems, and we use class diagram notation to represent an ontology. Stereo types attached to class boxes and associations show their types. For example, "Open" belongs to a "function" concept of Figure 4. An association between "Open" and "Door" is an "apply" relationship, which presents that an object "Door" participates in the function "Open". In this example, we have 12 model elements in the class diagram of Lift Control System and 2 elements (Close and Stop) to be required are lacking there. Because, in the thesaurus, the functions Open and Move requires Close and Stop respectively. As a result, we can get the completeness metrics (COMPL) $1 - (2/12) = 0.83$. As for the other semantic metrics such as consistency, correctness and unambiguity, their calculation methods were discussed in [9].

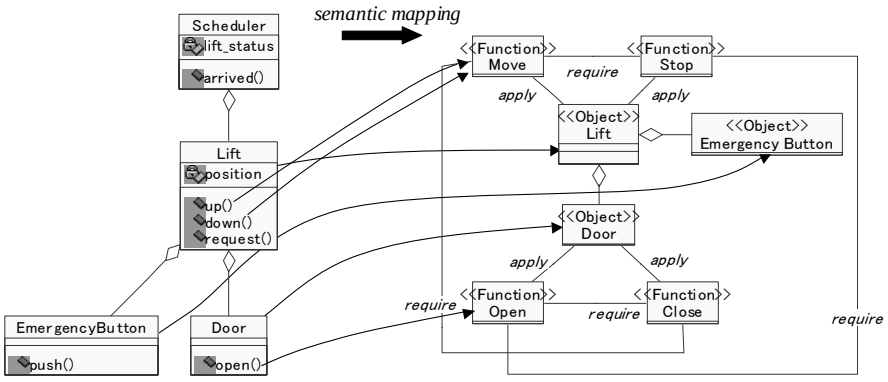


Fig. 5. An Example of Lift Control System

The calculation formula of this example is general because we calculate the ratio on how many generally required concepts are really included in the model. On the other hand, we sometimes need to define the metrics whose calculation formulas have the domain-specific properties, and these metrics can be defined as sub classes of the general metrics class. In the example of Lift Control System domain, we can consider that

the quality of the model having no emergency buttons is low from the viewpoint of completeness. As shown in Figure 6, we set the sub class DS-COMPL of COMPL and specify a new calculation formula for the domain-specific completeness value as follows.

$$DSCOMPL\_value = super.value \times (1 + \exists c \cdot (semantic\_mapping(c, EmergencyButton) \wedge (c \in ModelElement)))/2 \quad (3)$$

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

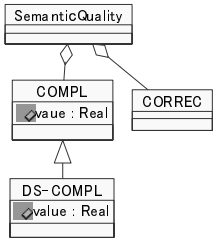


Fig. 6. Domain Specific Metrics

Three dimensional space model of Figure 7 summarizes our approach of using meta modeling for both syntactic metrics and semantics one. In the figure, we have three axes for "syntax vs. semantics", "model vs. meta model" and "product vs. metrics". For example, the left and right parts of the horizontal axis "model vs. meta model" in Figure 7 stand for model level (M1 layer in MOF) and meta model level (M2) respectively. On the other hand, syntactic entities are put on the upper area of the figure and the ontological entities on the lower area, following the vertical axis "syntax vs. semantics". A meta model and ontology meta model are used to specify how to calculate a model metrics with a metrics meta model. Following the metrics meta model, the model metrics is calculated from a triple of a model, semantic mapping and a domain ontology.

Note that we can implement semantic mapping by using UML stereo types of UML profile [3], considering a stereo type as an ontological concept. However, powerful inferences based on the relationships on ontological concepts to calculate semantics metrics are not so easy to define and implement on UML profile.

4 Model Transformation as Graph Rewriting

In Model Driven Development, one of the technically essential points is model transformation. Since we use a class diagram to represent a meta model, a model, i.e. an instance of the meta model can be considered as a graph, whose nodes have types and attributes, and whose edges have types, so called attributed typed graph. Thus in this paper, model transformation is defined as a graph rewriting system, and graph rewriting

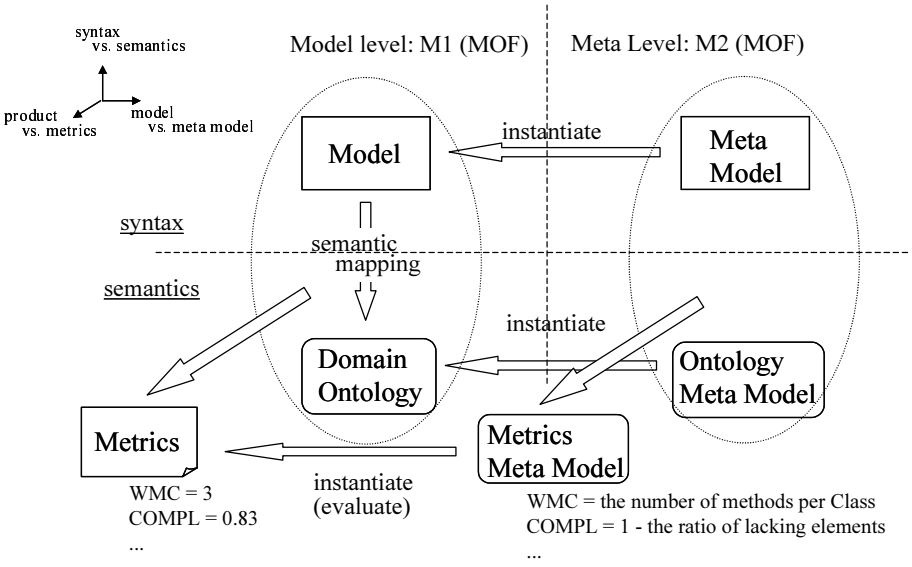


Fig. 7. Relationships among Models, Ontologies, Metrics and Meta Models

rules dominate allowable transformations. In this section, we introduce a graph rewriting system and show that metrics values can be evaluated and propagated between the models associated with graph rewriting rules. And the quality of transformation can be defined using the metrics values before and after the model transformation.

4.1 Graph Rewriting System

A graph rewriting system converts a graph into another graph or a set of graphs following pre-defined rewriting rules. There are several graph rewriting systems such as PROGRESS [17] and AGG [18]. We use the definition of the AGG system in this paper. A graph consists of nodes and edges, and type names can be associated with them. Nodes can have attribute values depending on their type. The upper part of Figure 8 is a simple example of rewriting rules. A rule consists of a left-hand and a right-hand side which are separated with "::=". A rectangle box stands for a node of a graph and it is separated into two parts with a horizontal bar. Type name of a node appears in the upper part of the horizontal bar, while the lower part contains its attribute values. In the figure, the node of "TypeA" in the left-hand graph has the attribute "val" and its value is represented with the variable "x". Numerals are used for identifying a node between the left-hand graph and the right-hand graph. For example, the numeral "1" in the node of "TypeA" in the left-hand means that the node is identical to the node of "TypeA" having "1" in the right-hand. A graph labeled with NAC (Negative Application Condition) appearing in the left-hand controls the application of the rule. If a graph includes the NAC graph, the rule cannot be applied to it. In addition, we add the conditions that are to be

satisfied when the rule is applied. In this example, “val” of the node “1:TypeA” have to be greater than 4 to apply this rule. The procedure of graph rewriting is as follows;

1. Extracting the part of the graph that structurally matches to the left-hand side of the rule. If the type names are attached with nodes and/or edges, these names should also match during this process. Suitable values are assigned into the variables appearing in attributes of the nodes.
2. Checking if the condition holds and if NAC does not appear. If successful and if none of the parts that structurally match a graph of NAC appears, the rewriting process continues, but if not so, the application of this rule is dismissed.
3. Replacing the extracted part with the right-hand of the rule and embedding the result to the original graph. New attribute values are calculated and assigned to the attributes.

The lower part of Figure 8 illustrates graph rewriting. The part encircled with a dotted rectangular box in the left-hand is replaced with the sub graph that is derived from the right-hand of the rule. The attribute values 5 and 2 are assigned to x and y respectively, and those of the two instance nodes of “TypeD” result in 7 (x+y) and 3 (x−y). Note that the value of “TypeA” is 5, greater than 4, and none of nodes typed with “TypeD” appear, so the rule is applicable. The other parts of the left-hand side graph are not changed in this rewriting process.

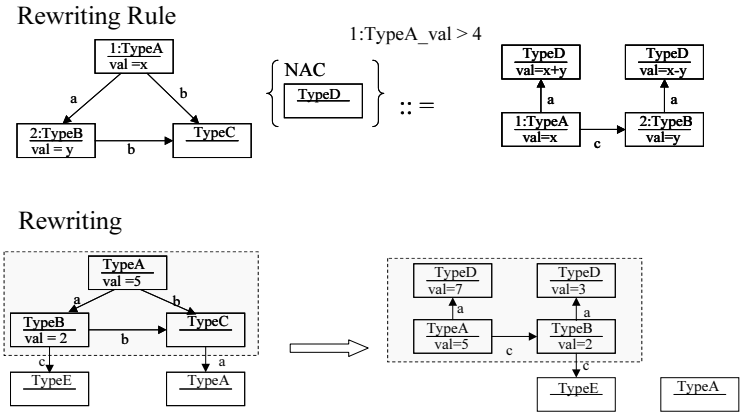
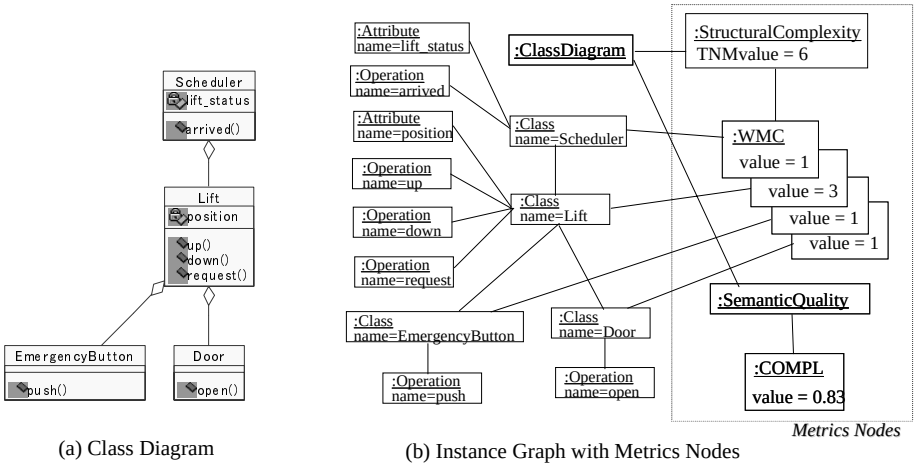


Fig. 8. Graph Rewriting Rules and Rewriting Process

4.2 Metrics on Model Transformation

The model following its meta model is represented with an attributed typed graph and it can be transformed by applying the rewriting rules. We call this graph instance graph in the sense that the graph is an instance of the meta model. Figure 9 shows the example of

a class diagram of Lift Control System and its instance graph following the meta model of Figures 1 and 4. The types of nodes are from the elements of the meta model such as Class, Attribute and Operation, while the names of classes, attributes and operations are specified as the values of the attribute "name". In the figure, the class Lift in the class diagram corresponds to the node typed with Class and whose attribute "name" is Lift. Some nodes in the instance graph have metrics values as their attribute values. For example, a node typed with WMC has the attribute "value" and its value is the number of the operations of the class, which is automatically calculated using the formula (1). The WMC value of class Lift is 3 as shown in the figure.



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

Fig. 9. Class Diagram and Its Instance Graph

We can design graph rewriting rules considering the nodes of the metrics and their values. See an example of a transformation rule shown in Figure 10. Two conditions $x2 > a$ and $x3 < y3$ are attached to the rule for rewriting the graph G1 with G2 and these conditions should be satisfied before the rule is applied. This rule includes two nodes for metrics; one is the metrics for G1 and the other is for G2. The first condition $x2 > a$ expresses that the rule cannot be applied until the value of the metrics m2 before the rewriting is greater than a certain value, i.e. "a". It means that this model transformation is possible when the model has a quality higher than a certain standard. The second condition $x3 < y3$ specifies monotonic increasing of the metrics m3 in this transformation. This formula has both values of metrics before and after the transformation as parameters and it can specify the characteristics of the transformation, e.g. a specific metrics value is increasing by the transformation. As shown in the figure, the calculation of the metrics n2 uses the metrics m1 of the model before the transformation, and this calculation formula of n2 shows that the metrics value of G1 is propagated to G2. The quality of a transformation can be formally specified by using this approach. In Figure 10, we can calculate how much the quality could be improved with the trans-

formation by using the metrics values of the model before the transformation and those after the transformation. The function g in the figure calculates the improvement degree of the quality.

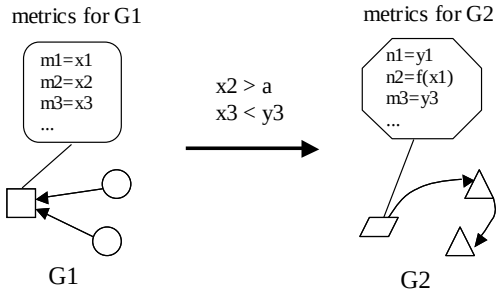


Fig. 10. Metrics and Model Transformation

Let's consider the example of a model transformation using graph rewriting rules. The model of Lift Control System in Figure 9 (a) can be considered as a platform independent model (PIM) because of no consideration of implementation situation, and we illustrate its transformation into a platform dependent model (PSM). We have a scheduler to decide which lift should be made to come to the passengers by the information of the current status of the lifts (the position and the moving direction of the lift), and we do not explicitly specify an alternative technique to implement the function of getting the status information from the lifts. If the platform that we will use has an interrupt-handling mechanism to detect the arrival of a lift at a floor, we put a new operation "notify" to catch the interruption signal in the Lift module. The notify operation calls the operation "arrived" of Scheduler and the "arrived" updates the lift_status attribute according to the data carried by the interrupt signal. As a result, we can get a PSM that can operate under the platform having interrupt-handling functions. In Figure 11, Rule #1 is for this transformation and PSM#1 is the result of applying this rule to the PIM of Lift Control System.

Another alternative is for the platform without any interrupt-handling mechanism, and in this platform, we use multiple instances of a polling routine to get the current lift status from each lift. The class Thread is an implementation of the polling routine and its instance is concurrently executed so as to monitor the status of the assigned lift. To execute the thread, we add the operations "start" for starting the execution of the thread and "run" for defining the body of the thread. The operation "attach" in Scheduler is for combining a scheduler object to the thread objects. Rule #2 and PSM#2 in the figure specifies this transformation and its result respectively. The TNMvalue, the total sum of the operations, can be calculated following the definition of Figure 1 for PIM, PSM#1 and PSM#2. It can be considered that the TNM value expresses the efforts to implement

the PSM because it reflects the volume of the source codes to be implemented. Thus the difference of the TNMvalues ($\Delta TNMvalue$) between the PIM to the PSM represents the increase of implementation efforts. In this example, PSM#1 is easier to implement because $\Delta TNMvalue$ of PSM#1 is smaller than that of PSM#2, as shown in Figure 11. So we can conclude that the transformation Rule #1 is of higher quality rather than Rule #2, only from the viewpoint of lower implementation efforts. This example suggests that our framework can specify formally the quality of model transformations by using the metrics values before and after the transformations.

Since in the above example we calculate just metrics values before and after the transformation, associating the calculating rule with the transformation rule seems not to be necessary. However, considering a whole of a transformation process consisting of a run of applying transformation rules, a calculation rule for quality should be associated with each of transformation rules and a metrics value of the model transformation can be composionally calculated from the metrics values obtained from the applications of the transformation rules. Complexity metrics of graph rewriting, e.g. the length of a transformation path and the number of the potentials of alternative applications, can be significant factors of the metrics of a model transformation.

5 Conclusion and Future Work

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

The future research agenda can be listed up as follows.

1. Development of supporting tools. We consider the extension of the existing AGG system, but to support the calculation of the metrics of transformations and the selection of suitable transformations, we need more powerful evaluation mechanisms of attribute values. This graph rewriting engine should be embedded into our meta CASE shown in Figure 2. The mechanisms for version control of models and redoing transformations are also necessary to make the tool practical.
2. Collecting useful definitions of metrics. In the paper, we illustrated very simple metrics for explanation of our approach. Although the aim of this research project is not to explore useful and effective quality metrics, making a kind of catalogue of metrics definitions and specifications is important in the next step of the supporting tool. The assessment of the collected metrics is also a research agenda.
3. Constructing domain ontologies. The quality of a domain ontology has a great effect on the preciseness of the semantic metrics, and we should get a domain ontology for each problem and application domain. In fact, developing various kind of domain ontologies of high quality by hand is a time-consuming and difficult task. Adopting text mining approaches are one of the promising ones to support the development of domain ontologies [1].

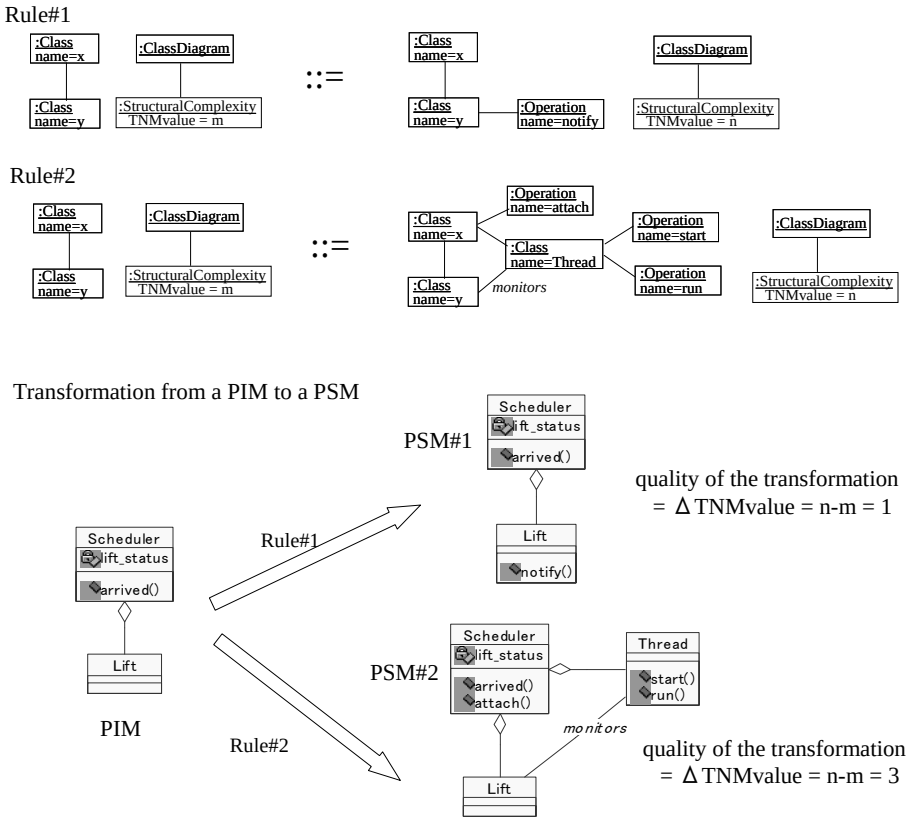


Fig. 11. Model Transformation Example

References

1. KAON Tool Suite. http://kaon.semanticweb.org/.
2. OMG Model Driven Architecture. http://www.omg.org/mda/.
3. Visual Modeling Forum – Domain-Specific Modeling. http://www.visualmodeling.com/DSM.htm.
4. IEEE Recommended Practice for Software Requirements Specifications. Technical report, IEEE Std. 830-1998, 1998.
5. F. B. Abreu. Using OCL to Formalize Object Oriented Metrics Definitions. In *Tutorial in 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2001)*, 2001.
6. S. Chidamber and C. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Trans. on Software Engineering*, 20(6):476–492, 1994.
7. K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *OOPSLA2003 Workshop on Generative Techniques in the context of Model Driven Architecture*, 2003.
8. M. Gruninger and J. Lee. Ontology: Applications and Design. *Commun. ACM*, 45(2), 2002.
9. H. Kaiya and M. Saeki. Ontology Based Requirements Analysis: Lightweight Semantic Processing Approach. In *QSIC*, pages 223–230, 2005.
10. G. Karsai and A. Agrawal. Graph Transformations in OMG's Model-Driven Architecture: (Invited Talk). In *AGTIVE*, pages 243–259, 2003.
11. A. Maedche. *Ontology Learning for the Semantic Web*. Kluwer Academic Publishers, 2002.
12. T. McCabe and C. Butler. Design Complexity Measurement and Testing. *CACM*, 32(12):1415–1425, 1989.
13. S. Mellor and M. Balcer. *Executable UML*. Addison-Wesley, 2003.
14. M. Saeki. Role of Model Transformation in Method Engineering. In *Lecture Notes in Computer Science (Proc. of CAiSE'2002)*, volume 2348, pages 626–642, 2002.
15. M. Saeki. Embedding Metrics into Information Systems Development Methods: An Application of Method Engineering Technique. In *Lecture Notes in Computer Science (Proc. of CAiSE 2003)*, volume 2681, pages 374–389, 2003.
16. M. Saeki. Toward Automated Method Engineering: Supporting Method Assembly in CAME. In *Engineering Methods to Support Information Systems Evolution (EMSISE'03 in OOIS'03)*. http://cui.unige.ch/db-research/EMSISE03/, 2003.
17. A. Schurr. Developing Graphical (Software Engineering) Tools with PROGRES. In *Proc. of 19th International Conference on Software Engineering (ICSE' 97)*, pages 618–619, 1997.
18. G. Taentzer, O. Runge, B. Melamed, M. Rudorf, T. Schultzke, and S. Gruner. AGG : The Attributed Graph Grammar System. *http://tfs.cs.tu-berlin.de/agg/*, 2001.
19. Y. Wand. Ontology as a Foundation for Meta-Modelling and Method Engineering. *Information and Software Technology*, 38(4):281–288, 1996.
20. J. Warmer and A. Kleppe. *The Object Constraint Language*. Addison Wesley, 1999.

Graph-Based Tool Support to Improve Model Quality

Tom Mens[1], Ragnhild Van Der Straeten[2], and Jean-François Warny

[1] Software Engineering Lab, Université de Mons-Hainaut
Av. du champ de Mars 6, 7000 Mons, Belgium
tom.mens@umh.ac.be
[2] Systems and Software Engineering Lab, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
rvdstrae@vub.ac.be

Abstract. During model-driven software development, we are inevitably confronted with design models that contain a wide variety of design defects. Interactive tool support for improving the model quality by resolving these defects in an automated way is therefore indispensable. In this paper, we report on the development of such a tool, based on the underlying formalism of graph transformation. Due to the fact that the tool is developed as a front-end of the *AGG Engine*, a general purpose graph transformation engine, it can exploit some of its interesting built-in mechanisms such as critical pair analyis and the ability to reason about sequential dependencies. We explore how this can help to improve the process of quality improvement, and we compare our work with related research.

1 Introduction

During development and evolution of design models it is often desirable to tolerate inconsistencies in design models. Indeed, such inconsistencies are inevitable for many reasons: (i) in a distributed and collaborative development setting, different models may be developed in parallel by different persons; (ii) the interdependencies between models may be poorly understood; (iii) the requirements may be unclear or ambiguous at an early design stage; (iv) the models may be incomplete because some essential information may be deliberately left out, in order to avoid premature design decisions; (v) the models are continuously subject to evolution; (vi) the semantics of the modeling language itself may be poorly specified.

All of these reasons hold in the case of UML, the de-facto general-purpose modelling language [1]. Therefore, current UML modeling tools should provide better support for resolving these inconsistencies in an automated way. Other types of design defects may also affect the quality of a model. Therefore, we suggest an automated approach to detect and resolve, among others, the following types of defects:

- nonconformance to standards (both industry-wide and company-specific standards);
- breaches of conventions (e.g., naming conventions);
- incomplete models, that are only partially specified and still have missing items [2];
- syntactic inconsistencies, i.e., models that do not respect the syntax of the modeling language;

- semantic inconsistencies, i.e., models that are not well-formed with respect to the semantics of the modeling language[3];
- design smells (in analogy with "bad smells") that indicate opportunities for performing a model refactoring;
- redundancies (e.g., double occurrences of a model element with the same name);
- visual problems (e.g., overlapping model elements in a diagram);
- bad practices;
- antipatterns, i.e., misuses or violations of design patterns

In addition, these problems may either be localised in a single UML diagram, or may be caused by mismatches between different UML diagrams.

The goal is therefore to provide a general framework and associated tool support to detect and resolve such design defects. In this paper, we suggest a transformation-based approach to do this. More in particular, we propose to use graph transformation technology. We report on an experiment that we have carried out and a tool that we have developed to achieve this goal, and we discuss how our approach may be integrated into comtemporary modeling tools.

2 Suggested approach

In Figure 1 we explain the iterative process of gradually improving the quality of a design model in an iterative way. First, defects in the model are identified. As explained above, these defects can be of diverse nature. Next, resolutions are proposed, selected and applied. The user may also wish to ignore or disable certain types of defects or resolutions. This process continues until all problems are resolved or until the user is satisfied.

When trying to develop tool support for this process, it is important to decide how the design defects and their resolutions should be specified. We opted for a formal specification, because this gives us an important added value: it allows us to analyse and detect mutual conflicts and sequential dependencies between resolution rules, which can be exploited to optimise the resolution process.

The particular formalism that we have chosen is the theory of graph transformation [3, 4]. The main idea that will enable us to perform conflict and dependency analysis is the application of theoretical results about critical pairs [5], which allow us to reason about parallel and sequential dependencies between rules.

3 Graph transformation

To perform detection and resolution of model defects, the tool relies entirely on the underlying formalism of graph transformation.

The UML metamodel is represented by a so-called *type graph*. A simplified version of the metamodel, showing a subset of UML 2.0 class diagrams, statemachine diagrams

[3] In UML this is a common problem due to the lack of a formal semantics combined with the fact that some parts of the semantics are deliberately left open, which makes the models subject to interpretation.

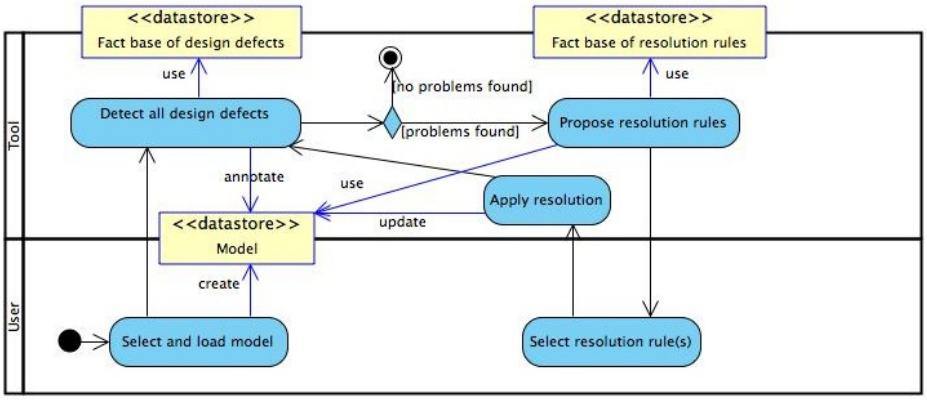


Fig. 1. UML activity diagram showing the interactive process for detecting and resolving design defects in a model.

and sequence diagrams, is given in Figure 2. The notion of design defect is incorporated explicitly in this type graph by the node type `Conflict`, which is used to identify model defects.

Every UML model will be represented internally as a *graph* that satisfies the constraints imposed by the aforementioned *type graph*. Figure 3 shows a simple example of a UML class diagram, represented as a graph model. More precisely, it corresponds to a directed, typed, attributed graph. These graph representations can be generated automatically from the corresponding UML model without any loss of information. [4]

Detection of design defects will be achieved by means of graph transformation rules. For each particular defect, a graph transformation rule will be specified that detects the defect. This is realised by searching for the occurrence of certain graph structures in the model, as well as the *absence* of certain forbidden structures (so-called *negative* application conditions or NACs). We call the defects that can be expressed as graph transformations, structural defects. Structural defects are all kinds of defects that can be expressed as a combination of the presence and/or absence of certain graph patterns.

A simple example of a detection rule is given in Figure 4. It detects the so-called *Dangling Type Reference* defect. This occurs when an `Operation` contains `Parameters` whose `type` has not (yet) been specified. The specification of this rule as a graph transformation is composed of three parts. The middle pane represents the *left-hand side (LHS)* of the rule, which is basically the occurrence of some `Operation` having a `Parameter`. The leftmost pane represents a *negative application condition (NAC)*, expressing the fact that the `Parameter` of interest does not have an associated `type`. Finally, the rightmost pane represents the *right-hand side (RHS)* of the rule, showing the result after the transformation. In this case, the only modification is the introduction

[4] An experiment along these lines has been carried out by L. Scolas as a student project.

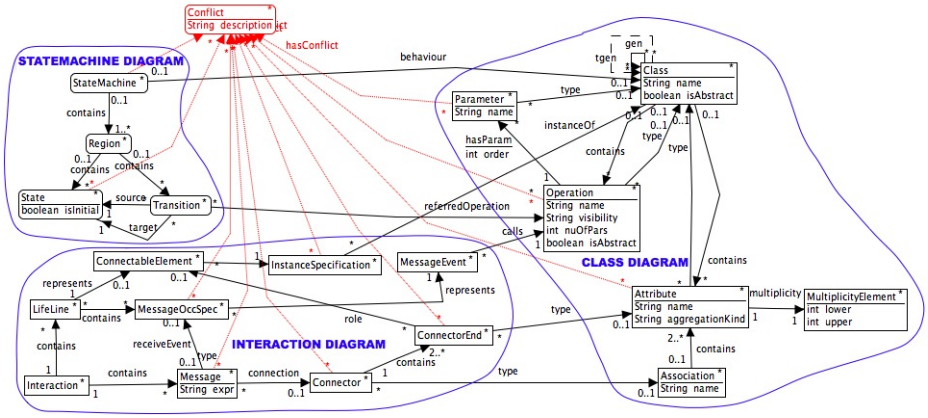


Fig. 2. Simplified metamodel for UML class diagrams, state machine diagrams, expressed as a type graph with edge multiplicities in AGG. In addition, a node type `Conflict` is introduced to represent model defects.

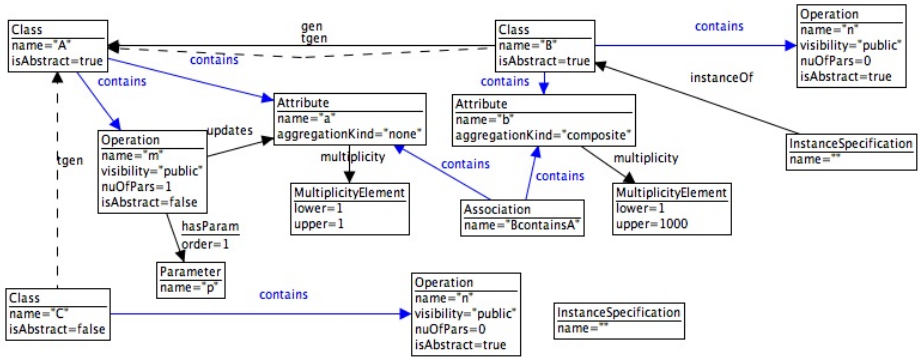


Fig. 3. Simplified UML class diagram model represented as a directed, typed, attributed graph in AGG.

of a `Conflict` node that is linked to the `Parameter` to show that there is a potential design defect.

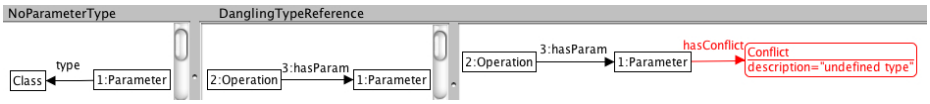


Fig. 4. Graph transformation representing the detection of a design defect of type *Dangling Type Reference*.

Given a source model, we can apply all detection rules in sequence to detect all possible design defects. By construction, the detection rules are parallel independent, i.e., the application of a detection rule has no unexpected side effects on other detection rules. This is because the only thing a detection rule does is introducing in the RHS a new node of type `Conflict` and a new edge of type `hasConflict` pointing to this node. Morever, the LHS and NAC of a detection rule never contain any `Conflict` nodes and `hasConflict` edges.

If we apply all detection rules (only one of these has been shown in Figure 4) to the graph of Figure 3, this graph will be annotated with nodes of type `Conflict`, as shown in Figure 5, to represent all detected design defects. The type of defect is indicated in the `description` attribute of each `Conflict` node. Observe that the same type of conflict may occur more than once at different locations, and that the same model element may be annotated by different types of conflicts.

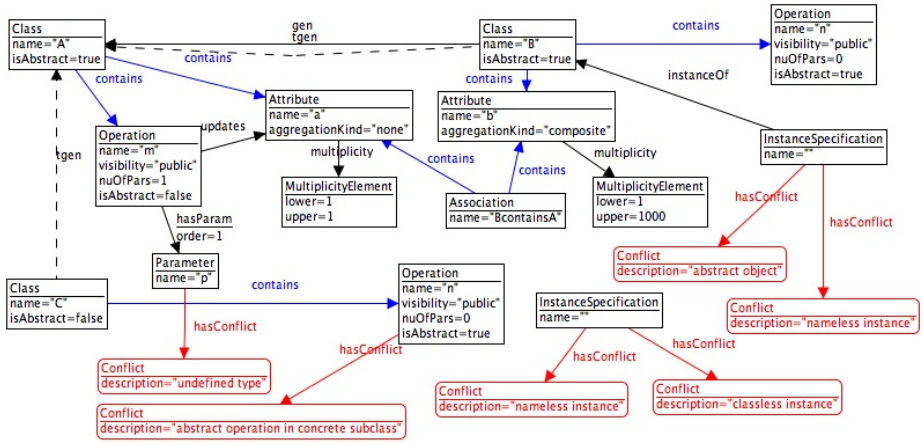


Fig. 5. Same UML class diagram model as in Figure 3, but annotated with all detected design defects.

Graph transformations will also be used to resolve previously detected design defects. For each type of design defect, several resolution rules can be specified. Each resolution rule has the same general form. On the left-hand side, we always find a `Conflict` node that indicates the particular defect that needs to be resolved. On the right-hand side, this `Conflict` node will no longer be present because the rule removes the design defect.

Figure 6 proposes three resolution rules for the *Dangling Type Reference* defect mentioned previously. The first one removes the problematic parameter, the second one uses an existing class as type of the parameter, and the third one introduces a new class as type of the parameter.

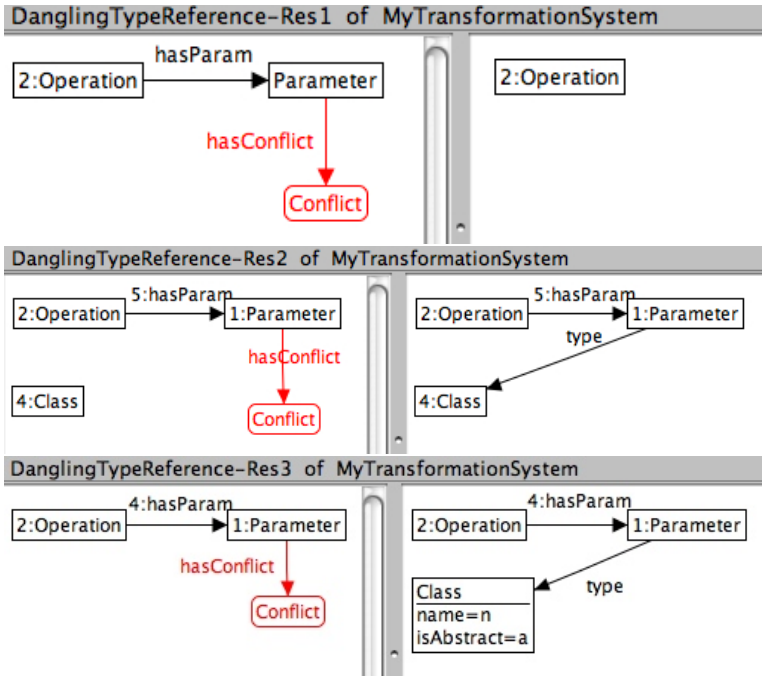


Fig. 6. Three graph transformations specifying alternative resolution rules for the *Dangling Type Reference* defect.

4 Tool support

The tool that we have selected to perform our experiments is *AGG*[5] (version 1.4), a state-of-the-art general purpose graph transformation tool [6]. We rely on the AGG

[5] See `http://tfs.cs.tu-berlin.de/agg/`

engine as a back-end, and we have developed a dedicated user interface on top of it to enable the user to interactively detect and resolve design defects [7]. This tool is called SIRP, for Simple Interactive Resolution Process. As will explained in more detail in the discussion section, integration of this tool into a UML modeling environment is left for future work.

Figure 7 shows a screenshot of the tool in action. It displays the detected design defects of Figure 3 as well as the resolution rules proposed to resolve these defects. In the screenshot, we see three resolution rules that can be selected to resolve the occurrence of the *Dangling Type Reference* defect. After selecting one of these rules, we can apply the chosen resolution, after which the model will be updated and the list of remaining design defects will be recomputed.

According to the resolution process of Figure 1, each resolution step is followed by a redetection phase. Currently, during redetection, we follow a brute-force approach, and detect all design defects again from scratch. A more optimal approach would be to come to an *incremental* redetection algorithm, thereby remembering those design defects that have already been identified in a previous phase. However, when doing this, we need to deal with a number of situations that may occur due to side effects that may impact existing model defects:

- **Orphan defects** arise when certain model elements have been removed as a result of resolving a certain design defect. In that case, some `Conflict` nodes may remain in the graph without any model element to which they refer (because the model element has been removed).
- **Expired defects** arise if the resolution of a certain design defect also resolves other design defects as a side-effect. If this is the case, there will be a `Conflict` node that points to some model element, even though the defect has already been resolved.

To address these two problems, we need to provide so-called *cleanup rules*, that remove all `Conflict` nodes that are no longer necessary. Such *cleanup rules* can be generated automatically from the detection and resolution rules.

5 Graph transformation dependency analysis

There are also other types of problems that may inevitably occur during the detection and resolution process, due to the inherently incremental and iterative nature of the conflict resolution process.

Induced defects may appear when the resolution of a certain design defect introduces other design defects as a side effect. An example is given in Figure 8. Suppose that we have a model that contains a defect of type *Abstract Object*, i.e., an instance specification (an object) that refers to an abstract class (labelled 1 in Figure 8). The resolution rule *AbstractObject-Res1* resolves the defect by setting the attribute `isAbstract` of class 1 to `false`. As a result of this resolution, the design defect called *Abstract Operation* suddenly becomes applicable. This is the case if class 1, which now has become concrete, contains one or more abstract operations.

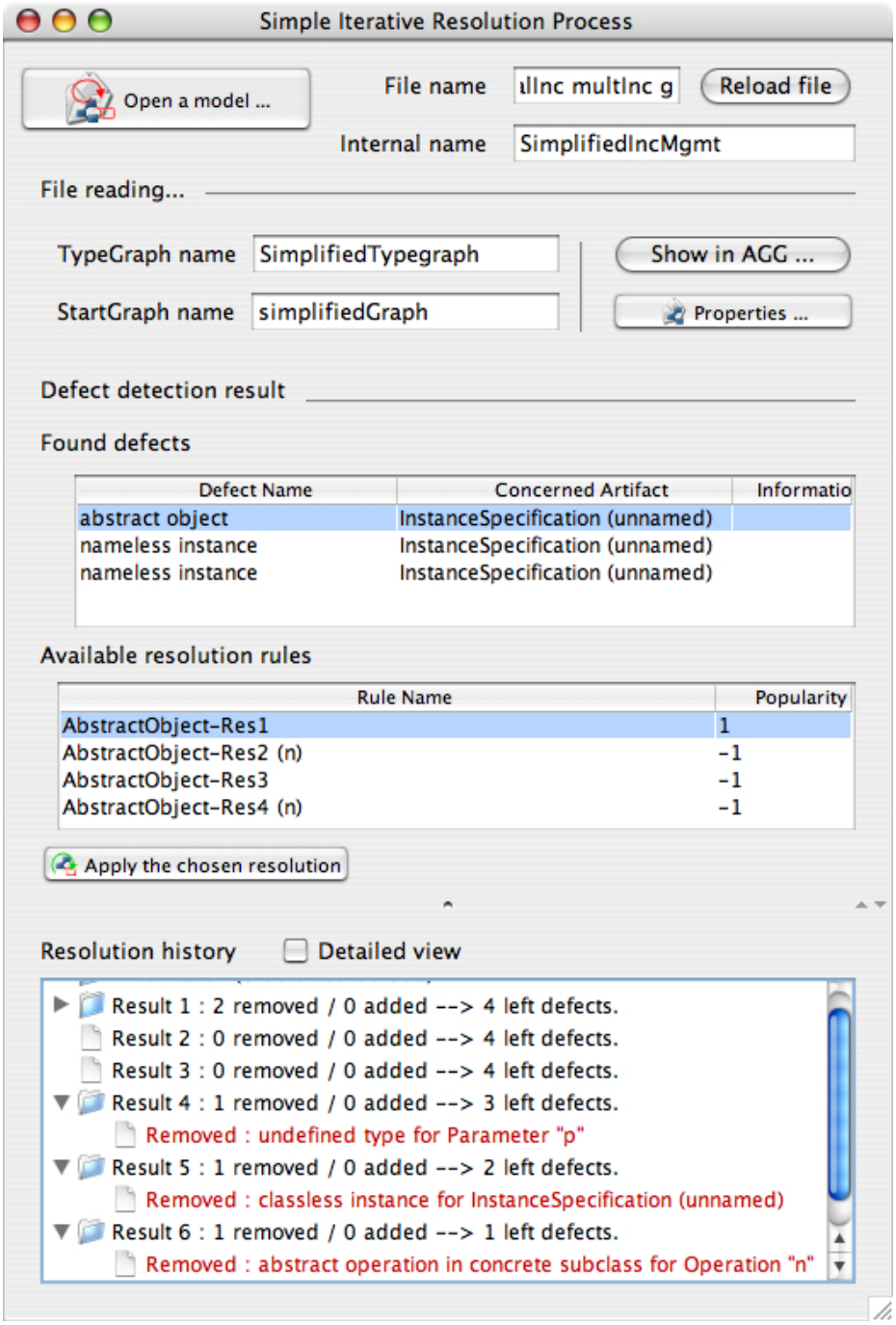


Fig. 7. Screenshot of the tool in action. Several defects have been resolved already, as shown in the resolution history. Resolution rules are proposed for each remaining defect with a certain popularity (based on whether the rule has already been applied before by the user). Selected rules can be applied to resolve the selected defect.

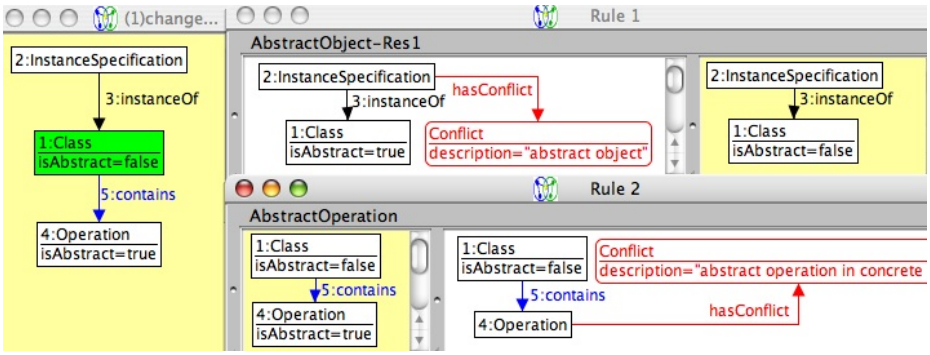


Fig. 8. *Induced defects*: Example of a sequential (causal) dependency of detection rule **AbstractOperation** on resolution rule **AbstractObject-Res1**.

Conflicting resolutions may appear when there are multiple design defects in a model, each having their own set of applicable resolution rules. It may be the case that applying a resolution rule for one design defect, may invalidate another resolution rule for another design defect. As an example, consider Fig. 9. The left pane depicts a situation where two defects occur, one of type *Abstract Operation* and *Dangling Operation Reference* respectively, but attached to different model elements. The resolution rules *AbstractOperation-Res4* and *DanglingOperationRef-Res2* for these defects (shown on the right of Fig. 9) are conflicting, since the first resolution rule sets the relation *contains* connecting class 1 to operation 2 to connecting class 4 and operation 2, whereas the second resolution rule requires as a precondition that class 1 is connected to operation 2 through a containment relation.

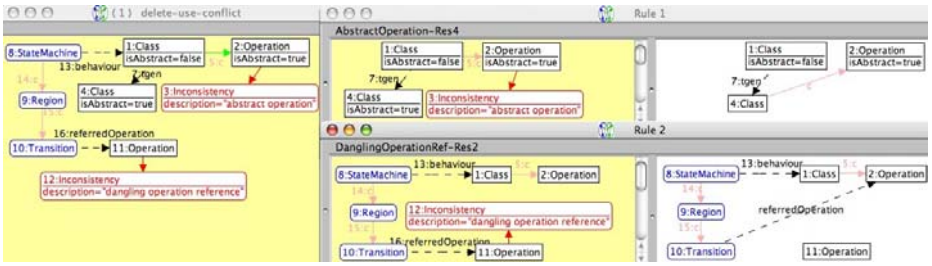


Fig. 9. *Conflicting resolutions*: Example of a critical pair illustrating a mutual exclusion between resolution rules **AbstractOperation-Res4** and **DanglingOperationRef-Res2**.

To identify and analyse the two situations explained above in an automated way, we need to make use of the mechanism of *critical pair analysis* of graph transformation rules [5, 8]. The goal of critical pair analysis is to compute all potential mutual exclu-

sions and sequential dependencies for a given set of transformation rules by pairwise comparison. Such analysis is directly supported by the *AGG* engine, so it can readily be used in our approach.

The problem of **induced defects** is a typical situation of a *sequential dependency*: a detection rule causally depends on a previously applied resolution rule. Figure 10 shows an example of a dependency graph that has been generated by AGG. Given a selection of design defects, it shows all **induced defects**, i.e., all detection rules that sequentially depend on a resolution rule. This information is quite important in an incremental resolution process, as it informs us, for a given resolution rule, which types of defects will need to be redetected afterwards.

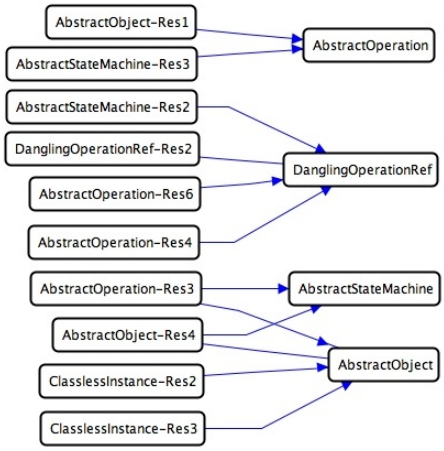


Fig. 10. Dependency graph generated by AGG showing all **induced defects**, i.e., all defect detection rules that sequentially depend on a resolution rule.

The problem of **conflicting resolutions** is a typical situation of a *parallel conflict*: two rules that can be applied in parallel cannot be applied one after the other (i.e., they are mutually exclusive) because application of the first rule prevents subsequent application of the second one. Again, the information reported in the graph is quite important during an internative resolution process, as it informs the user about which resolution rules are mutually exclusive and, hence, cannot be applied together.

Figure 11 shows an example of a conflict graph that shows a number of **conflicting resolutions** between the resolution rules for the *Abstract Operation* defect and the resolution rules of other design defects. Except for some layout issues, this graph has been automatically generated by AGG's critical pair analysis algorithm. In the Figure, we can see lots of conflicts between the resolution rules for *Abstract Operation* and the resolution rules for *Dangling Operation Reference*.

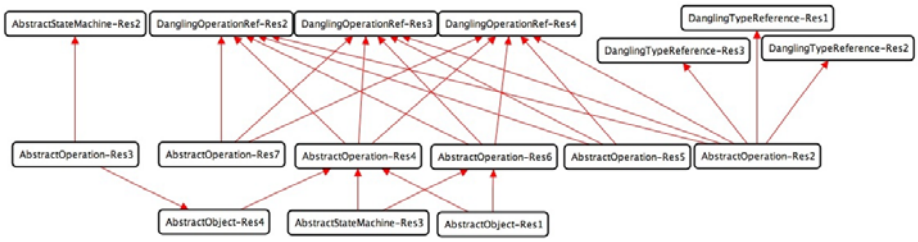


Fig. 11. Conflict graph generated by AGG showing **conflicting resolutions** (i.e., mutual exclusions) between the resolution rules for the *Abstract Operation* defect and resolution rules for another design defect.

6 Cycle detection and analysis

As illustrated in Figure 12, starting from the dependency graph, we can also compute possible *cycles* in the conflict resolution process. This may give important information to the user (or to an automated tool) to avoid repeatedly applying a certain combination of resolution rules over and over again. Clearly, such cycles should be avoided, in order to optimise the resolution process (e.g., by preventing cycles to occur).

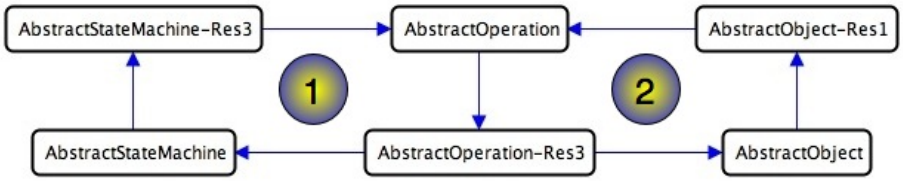


Fig. 12. Some examples of detected cycles in the sequential dependency graph.

As an example of such cycle, consider Figure 12, which represents a carefully selected subset of sequential dependencies that have been computed by AGG.[6] In this figure, we observe the presence of two cycles, both of them involving the *Abstract Operation* defect. Both cycles are of length 4, and correspond to two successive detection and resolution steps. The cycle corresponding to region 1 shows that we can repeatedly apply resolution rules *AbstractStateMachine-Res3* and *AbstractOperation-Res3 ad infinitum*. This is the case because the two resolution rules are each others inverse. Therefore, after applying one of both rules, the interactive resolution tool should not propose the other rule because it would undo the effect of the first one. The cycle corresponding to region 4 is similar to the previous one, except that it occurs between resolution rules *AbstractObject-Res1* and *AbstractOperation-Res3*.

[6] To interpret the dependency graph, the blue lines could be read as “enables” or “triggers”.

Because the sequential dependency graph can be very large, manual detection of cycles is unfeasible in practice. Therefore, we have used a small yet intuitive user interface for detecting all possible cycles in a flexible and interactive way, based on the output generated by AGG's critical pair analysis algorithm. This program has been developed by S. Goffinet in the course of a student project.

7 Discussion and Future Research

Currently, our approach has not yet been integrated into a modeling tool. The reason is that there are many mechanisms for doing this, and we haven't decided yet on which alternative is the most appropriate. The most obvious solution would be to directly integrate the proposed process into an existing UML modeling tool. *ArgoUML*[7] seems the most obvious candidate for doing this because it is open source and already provides support for design critics. It is not clear, however, how this can be combined easily with critical pair analysis since this requires an underlying representation based on graph transformation. Therefore, another more feasible approach could be to develop a modeling tool directly based on graph transformation as an underlying representation. Several such tools have already been proposed (e.g. VIATRA, GReAT, Fujaba), but none of those currently provides support for critical pair analysis. Another alternative could therefore be to build a modeling environment on top of the AGG engine. To achieve this, one may rely on the Tiger project, an initiative to generate editors of visual models using the underlying graph transformation engine [9].

The fact that the resolution of one model defect may introduce other defects is a clear sign of the fact that defect resolution is a truly iterative and interactive process. One of the challenges is to find out whether the resolution process will ever *terminate*. It is easy to find situations that never terminate (cf. the presence of cycles in the dependency graph). Therefore, the challenge is to find out under which criteria a given set of resolution rules (for a given set of design defects and a given start graph) will terminate. Recent work that explores such termination criteria for model transformation based on the graph transformation formalism has been presented in [10].

Another challenge is to try and come up with an *optimal order* of resolution rules. For example, one strategy could be to follow a so-called "opportunistic resolution process", by always following the choice that corresponds to the least cognitive effort (i.e., the cognitive distance between the model before and after resolution should be as small as possible). How to translate this into more formal terms remains an open question. A second heuristic could be to avoid as much as possible resolution rules that give rise to *induced defects* (i.e., resolutions that inadvertently introduce other defects). Yet another strategy could be to prefer resolution rules that give rise to *expired defects* (since these are rules that resolve more than one defect at once).

Another important question pertains to the *completeness* of results. How can we ensure that the tool detects all possible defects, that it proposes all possible resolution rules, and that all conflicting resolutions and sequential dependencies are correctly reported? How can we avoid false positives reported by the tool?

[7] http://argouml.tigris.org/

A related question concerns *minimality*. Is it possible to detect and avoid redundancy between detection rules and between resolution rules? Is it possible to come up with a minimal set of resolution rules that still cover all cases for a given set of detection rules?

A limitation of the current approach that we are well aware of, is the fact that not all kinds of defects and resolution rules can be expressed easily as graph transformation rules. For example, behavioural inconsistencies are also difficult to express in a graph-based way. Because of this, our tool has been developed in an extensible way, to make it easier to plug-in alternative mechanisms for detecting defects, such as those based on the formalism of description logics [11]. Of course, it remains to be seen how this formalism can be combined with the formalism of graph transformation, so that we can still benefit from the technique of critical pair analysis.

8 Related Work

Critiquing systems originate in research on artificial intelligence, and more in particular knowledge-based systems and expert systems. Rather than giving a detailed account of such systems, let us take a look at one particular attempt to incorporate these ideas into a modeling tool, with the explicit aim to critic and improve design models [12, 13]. In this view, "a *design critic* is an intelligent user interface mechanism embedded in a design tool that analyses a design in the context of decision-making and provides feedback to help the designer improve the design. Support for design critics has been integrated into the ArgoUML modeling tool. It is an automated and unobtrusive user interface feature that checks in the background for potential design anomalies. The user can chose to ignore or correct these anomalies at any time. Most critiquing systems follow the so-called ADAIR process which is sequentially composed of five phases: Activate, Detect, Advice, Improve and Record. Without going into details, our approach roughly follows the same process.

Another approach that is very related to ours is reported in [14]. A rule-based approach is proposed to detect and resolve inconsistencies in UML models, using the Java Rule Engine JESS. In contrast to our approach, where the rules are graph-based, the specification of their rules is logic-based. However, because the architecture of their tool provides a Rule Engine Abstraction Layer, it should in principle be possible to replace their rule engine by a graph-based one.

The main novelty of our approach compared to the previously mentioned ones, is the use of the mechanism of critical pair analysis to detect mutual inconsistencies between rules that can be applied in parallel, as well as sequential dependency analysis between resolution rules.

There have been several attempts to use graph transformation in the context of inconsistency management. In [15], distributed graph transformation is used to deal with inconsistencies in requirements engineering. In [16], graph transformations are used to specify inconsistency detection rules. In [17] repair actions are also specified as graph transformation rules. Again, the added value of our approach is the ability to analyse conflicts and dependencies between detection and resolution rules.

The technique of critical pair analysis of graph transformations has also been used in other, related, domains. [18] used it to detect conflicting functional requirements in

UML models composed of use case diagrams, activity diagrams and collaboration diagrams. [19] used it to detect conflicts and dependencies between software refactorings. [20] used it to improve parsing of visual languages.

An important aspect of research on model quality that is still underrepresented in literature is empirical research and case studies on the types of defects that commonly occur in industrial practice and how these can be resolved [2, 21, 22].

9 Conclusion

In this article we addressed the problem of model quality improvement. The quality of a model can be improved in an iterative way by looking for design defects, and by proposing resolution rules to remove these defects. Interactive tool support for this process can benefit from a formal foundation. This article proposed a tool based on the underlying formalism of graph transformation. Given a formal specification of the UML model as a graph (and the metamodel as a type graph), design defects and their resolutions were specified as graph transformation rules. Furthermore, critical pair analysis was used to identify and analyse unexpected interactions between resolution rules, new defects that are introduced after resolving existing defects, and cycles in the resolution process. Futher work is needed to integrate this tool ino a modeling environment.

References

1. Object Management Group: Unified Modeling Language 2.0 Superstructure Specification. http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf (2005)
2. Lange, C.F., Chaudron, M.R.: An empirical assessment of completeness in uml designs. In: Proc. Int'l Conf. Empirical Assessment in Software Engineering. (2004) 111–121
3. Rozenberg, G., ed.: Handbook of graph grammars and computing by graph transformation: Foundations. Volume 1. World Scientific (1997)
4. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of graph grammars and computing by graph transformation: Applications, Languages and Tools. Volume 2. World Scientific (1999)
5. Plump, D.: Hypergraph rewriting: Critical pairs and undecidability of confluence. In: Term Graph Rewriting. Wiley (1993) 201–214
6. Taentzer, G.: AGG: A graph transformation environment for modeling and validation of software. In: Proc. AGTIVE 2003. Volume 3062 of Lecture Notes in Computer Science., Springer-Verlag (2004) 446–453
7. Warny, J.F.: Détection et résolution des incohérences des modèles uml avec un outil de transformation de graphes. Master's thesis, Université de Mons-Hainaut, Belgium (2006)
8. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Proc. Int'l Conf. Graph Transformation. Volume 3256 of Lecture Notes in Computer Science., Springer-Verlag (2004) 161–177
9. Ehrig, K., Ermel, C., Hänsgen, S., Taentzer, G.: Generation of visual editors as eclipse plug-ins. In: Proc. Int'l Conf. Automated Software Engineering, ACM Press (2005) 134–143
10. Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination criteria for model transformation. In: Proc. Fundamental Aspects of Software Enginering (FASE). Volume 3442 of Lecture Notes in Computer Science., Springer-Verlag (2005) 49–63

11. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using description logics to maintain consistency between UML models. In: UML 2003 - The Unified Modeling Language. Volume 2863 of Lecture Notes in Computer Science., Springer-Verlag (2003) 326–340
12. Robbins, J.E., Redmiles, D.F.: Software architecture critics in the argo design environment. Knowledge-Based Systems **11** (1998) 47–60
13. Robbins, J.E.: Design Critiquing Systems. PhD thesis, University of California, Irvine (1999) Technical Report UCI-98-41.
14. Liu, W., Easterbrook, S., Mylopoulos, J.: Rule-based detection of inconsistency in UML models. In: Proc. UML 2002 Workshop on Consistency Problems in UML-based Software Development, Blekinge Insitute of Technology (2002) 106–123
15. Goedicke, M., Meyer, T., , Taentzer, G.: Viewpoint-oriented software development by distributed graph transformation: Towards a basis for living with inconsistencies. In: Proc. Requirements Engineering 1999, IEEE Computer Society (1999) 92–99
16. Ehrig, H., Tsioalikis, A.: Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In: ETAPS 2000 workshop on graph transformation systems. (2000) 77–86
17. Hausmann, J.H., Heckel, R., Sauer, S.: Extended model relations with graphical consistency conditions. In: Proc. UML 2002 Workshop on Consistency Problems in UML-Based Software Development. (2002) 61–74
18. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach. In: Proc. Int'l Conf. Software Engineering, ACM Press (2002)
19. Mens, T., Taentzer, G., Runge, O.: Analyzing refactoring dependencies using graph transformation. Software and Systems Modeling (2006) To appear.
20. Bottoni, P., Taentzer, G., Schürr, A.: Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In: Proc. IEEE Symp. Visual Languages. (2000)
21. Lange, C., Chaudron, M., Muskens, J.: In practice: Uml software architecture and design description. IEEE Software **23** (2006) 40–46
22. Lange, C.F., Chaudron, M.R.: Effects of defects in uml models – an experimental investigation. In: Proc. Int'l Conf. Software Engineering (ICSE), ACM Press (2006) 401–410

Model Driven Development of a Service Oriented Architecture (SOA) Using Colored Petri Nets

Vijay Gehlot[1], Thomas Way[1], Robert Beck[1], and Peter DePasquale[2]

[1] Center of Excellence in Enterprise Technology, Department of Computing Sciences
Villanova University, Villanova, Pennsylvania 19085, USA
{vijay.gehlot, thomas.way, robert.beck}@villanova.edu
http://ceet.villanova.edu

[2] Department of Computer Science, The College of New Jersey, Ewing, NJ 08628, USA
depasqua@tcnj.edu

Abstract. Service-Oriented Architecture (SOA) is achieving widespread acceptance in a variety of enterprise systems, due to its inherent flexibility and interoperability, improving upon the more tradition and less supportable "stovepipe" approach. The high degree of concurrency and both synchronous and asynchronous communications inherent in SOA makes it a good candidate for a Petri Nets based model driven development (MDD). Such an approach, with its underlying verification and validation implications, becomes more crucial in mission-critical applications, such as those with defense implications. This paper reports on our experience with using Colored Petri Nets (CPNs) for model driven development and quality assessment of a defense-targeted service-oriented software architecture. We identify features of CPN that have resulted in ease of adoption as a modeling tool in our present setting. Preliminary results are provided which support the use of CPNs as a basis for model driven software development, and verification and validation (V&V) for quality assurance of highly concurrent and mission-critical SOAs.

1 Introduction

Designers of enterprise architectures have embraced the Service Oriented Architecture (SOA) approach, which leverages significant advances in distributed computing and networking technologies to enable large scale interoperability [1,2]. Although the SOA approach promotes flexibility, reuse and decoupling of functionality from implementation, the inherent complexity of the enterprise class of services makes the verification and validation (V&V) of such systems difficult [3]. Specific requirements of SOA applications to net-centric Department of Defense (DoD) deployments, such as stringent service guarantees, fault tolerance and security, among others [4], coupled with the significant costs involved in fulfilling these strict requirements, suggests a need for a model driven development and quality assurance approach that can accommodate the highly concurrent nature of enterprise uses of SOA. We show how

support for hierarchical and abstraction features, concurrency, and both synchronous and asynchronous communications in Colored Petri Nets (CPNs) enable modeling real-world SOA implementations to perform V&V and quality assurance required for DoD deployments. As part of a model driven approach, the created model is also intended to be used for quality predictions.

2 Service Oriented Architectures

Service-Oriented architecture (SOA) is a distributed network architecture design approach that separates services provided from the entities that consume those services. Services communicate with each other, yet are self-contained and do not depend on the state of other services, leading to a loosely coupled architecture that, as a result of this decoupling of services, is easily reconfigurable. Generally, SOA is defined as an "enterprise-wide IT architecture that promotes loose coupling, reuse, and interoperability between systems," with the more specific view as "architectures making use of Web service technologies such as SOAP, WSDL, and UDDI... conforming to the W3C Web services architecture (WSA)." [2]

The SOA approach is attractive for enterprise systems because of its inherent flexibility and reusability and its isolation of functionality from the details of implementation. Developers of SOA providers and consumers design complex software systems using implementation-neutral interfaces, rather than less flexible, highly integrated interfaces resulting from proprietary specification and design approaches. By maintaining interoperability at the interface, developers evolve services with isolated internal implementations of which external services need not be aware. [4] The roles described in SOA are service provider, service consumer and service discovery (**Fig. 1**).

Fig. 1. Overview of roles in Service-Oriented Architecture (SOA). [4]

The service provider is responsible for producing a service, making it available to service consumers by publishing a service interface in a service registry. The service

consumer makes use of the service produced by the service provider based on the rules specified in the published interface. The service discovery component of SOA provides the service publication mechanism so that service providers can make known their service interface to service consumers.

The service interfaces published by service providers adhere to the SOA approach by decoupling implementation from definition, maintaining strict configuration management so that service consumers can seamlessly migrate from one version of a service interface to another, providing backward compatibility with existing interface versions, and allowing interface and implementation versions to evolve independently. [4]

2.1 DoD Net-Centric Enterprise Solutions for Interoperability (NESI)

The flexibility and interoperability of SOA makes it an attractive solution for enterprise services within DoD deployments. The DoD and Defense Information Systems Agency (DISA) have defined a core set of enterprise services for use in defense-related SOA systems [5], as part of an initiative called Net-Centric Enterprise Solutions for Interoperability (NESI). These Net-Centric Enterprise Services (NCES) define by NESI are a set of net-centric services, nodes and utilities for use in DoD domain- and mission-related enterprise information systems (**Fig. 2,** [4]). Development of NCES is ongoing with significant efforts currently underway in systems for various defense applications.

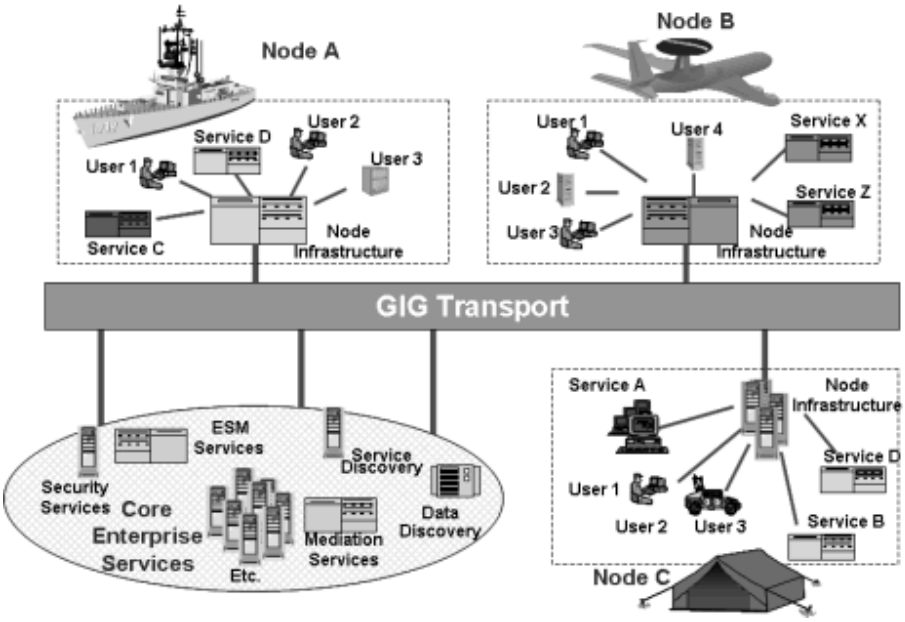


Fig. 2. Node interoperability in NESI Net-Centric Enterprise Architectures. [4]

Within SOA deployments, providers and consumers of services, when related, are collectively referred to as nodes. As illustrated in **Fig. 2**, the DoD GIG architecture describes a node as a set of information systems that form a single element in a net-centric enterprise. These nodes can include servers for web, portal, applications, and databases to provide services. To support robustness, when a node loses enterprise connectivity it should continue to serve local consumers of its services.

3 A SOA Architecture for DoD Applications

Various characteristics of SOA can be identified as necessary characteristics for broad defense applications that are otherwise not part of a traditional web-services based view of SOA. These are as follows:

- Service guarantees
- Fault tolerance
- Dynamic service discovery
- Interoperable multiple connection types
- Availability awareness
- Load balancing
- Security

A proposed general solution is to have an architecture with a mediator responsible for dynamic discovery, awareness, and load balancing. To allow interoperable multiple connection types would require a well-defined internal format and protocol with well-defined external to internal interfaces and mechanisms for internal transport and buffering. We call this architecture Service Oriented Defense Architecture (SODA).

A reference implementation of this architecture is under development by a defense contractor. Our focus is to integrate a model based approach into this software development that can be used to guide the implementation and to assess the reliability, scalability and performance of the SODA product using simulation, verification, and validation. In addition, the research goal is to provide, through modeling and analysis, feedback to the development and DoD communities to enhance their understanding of capabilities and limitations, influence architectural and technical decision making process, and set the expectations for network-centric technology architecture behaviors. The specific goals for our project are as follows:

- Gain greater understanding of the performance characteristics of multi-channel service oriented architectures.
- Achieve greater acceptance of the real world “deployability” and reliability of the multi-channel service oriented architecture, thereby accelerating “real world” legacy migrations to service-based infrastructures.
- Provide a deeper understanding of the tradeoffs that exist between performance and agility in a service-enabled environment.
- Construct a reusable set of models for researching the behaviors of large-scale deployments of service-enabled systems and the technologies that support them.

- Establish some level of benchmarking for large-scale distributed systems based on the proposed architecture.
- Create a model-based verification and validation framework for distributed systems that are based on SODA.

We discuss some details of our proposed architecture in Section 5 in conjunction with the formal model of the architecture that is being developed. We are using Colored Petri Nets (CPNs) as our modeling language. The next section gives some details of CPN.

4 Colored Petri Nets and CPN Tool

Our modeling approach is based on Colored Petri Nets (CPNs) [6]. Petri Nets provide a modeling language (or notation) well suited for distributed systems in which communication, synchronization and resource sharing play important roles. CPNs combine the strengths of ordinary Petri nets [7,8] with the strengths of a high-level programming language together with a rigorous abstraction mechanism. Petri nets provide the primitives for process interaction, while the programming language provides the primitives for the definition of data types and the manipulations of data values.

As with Petri nets, CPNs have a formal mathematical definition and a well-defined syntax and semantics. This formalization is the foundation for the different behavioral properties and the analysis methods. The complete formal definition of a CPN is given below and more details can be found in [8,9]. It should be noted that the purpose of this definition is to give a mathematically sound and unambiguous description of a CPN. In practice, however, one would create a CPN model using a tool such as CPN Tool [10]. This tool is a graphical tool that allows one to create a visual representation of a CPN model and analyze it.

Definition: A **Colored Petri Net** is a nine-tuple $(\Sigma, P, T, A, N, C, G, E, I)$, where:

(i) Σ is a finite set of non-empty **types**, also called color sets. In the associated CPN Tool, these are described using the language CPN-ML. A **token** is a value belonging to a type.

(ii) P is a finite set of **places**. In the associated CPN Tool these are depicted as ovals/circles.

(iii) T is a finite set of **transitions**. In the associated CPN Tool these are depicted as rectangles.

(iv) A is a finite set of **arcs.** In the associated CPN Tool these are depicted as directed edges. The sets of places, transitions, and arcs are pairwise disjoint, that is

$$P \cap T = P \cap A = T \cap A = \emptyset.$$

(v) N is a **node** function. It is defined from A into $P \times T \cup T \times P$. In the associated CPN Tool this depicts the source and sink of the directed edge.

(vi) C is a **color** function. It is defined from P into Σ.

(vii) G is a **guard** function. It is defined from T into expressions such that:

$\forall t \in T$: [Type(G(t)) = Boolean $\wedge$ Type(Var(G(t))) $\subseteq \Sigma$].

(viii) E is an **arc expression** function. It is defined from A into expressions such that:

$\forall a \in A$: [Type(E(a)) = $C(p)_{MS} \wedge$ Type(Var(E(a))) $\subseteq \Sigma$]

where p is the place of N(a) and $C(p)_{MS}$ denotes the multi-set type over the base type C(p).

(ix) I is an **initialization** function. It is defined from P into closed expressions such that:

$\forall p \in P$: [Type(I(p)) = $C(p)_{MS}$].

In the CPN Tool this is represented as initial marking next to the associated place.

The distribution of tokens, called **marking**, in the places of a CPN determine the state of a system being modeled. The dynamic behavior of a CPN is described in terms of the **firing** of transitions. The firing of a transition takes the system from one state to another. A transition is **enabled** if the associated arc expressions of all incoming arcs can be evaluated to a multi-set, compatible with the current tokens in their respective input places, and its guard is satisfied. An enabled transition may fire by removing tokens from input places specified by the arc expression of all the incoming arcs and depositing tokens in output places specified by the arc expressions of outgoing arcs.

CPN models can be made with or without explicit reference to time. Untimed CPN models are usually used to validate the functional/logical correctness of a system, while timed CPN models are used to evaluate the performance of the system.

The time concept in CPN is based on a global clock. The clock value represents the model time. In the timed version, each token carries a time stamp. The time stamp of a token determines the earliest (simulation) time at which the token will become available.

One aspect of CPN that is attractive for creating models of large systems is being able to create **hierarchical** CPN. Hierarchical CPN allow one to relate a transition (and its surrounding arcs and places) to a separate sub-net (called a *subpage* in CPN parlance) structure. The subnet then represents the detailed description of the activity represented by the associated transition. This allows one to build a model either in top-down or bottom up manner and also allows one to either hide or expose details as necessary. Complete detail of a hierarchical CPN and its semantics can be found in [6].

5 CPN Model of SODA

We present some details of the SODA by discussing its CPN model. At the most abstract level, a general SOA, and hence SODA, can be viewed as consisting of requests for services that are sent through some discovery/mediation/transport mecha-

nism to be processed and responses returned by providers of services. In the CPN model this is the top-level page called Top and is shown in **Fig. 3.**

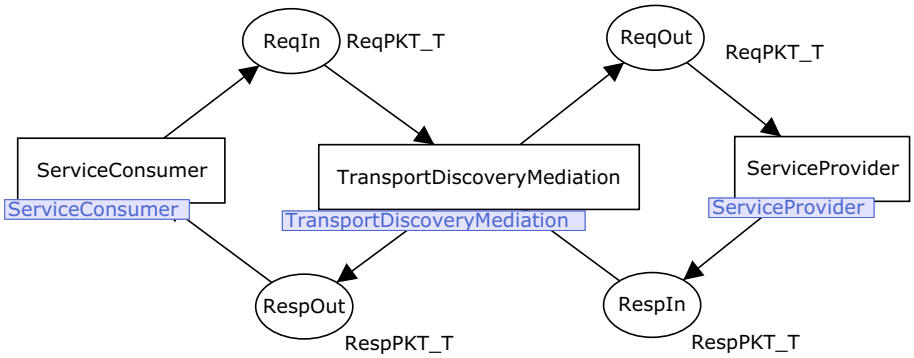


Fig. 3. Top level page in CPN model giving the most abstract view of the system.

The transport, discovery and mediation mechanism itself consists of several components. These are detailed on the CPN subpage *TransportDiscoverMediation* and are shown in **Fig. 4.** Tracing the flow of data through this net, an incoming request packet arrives via the component *Ext2IntInbound*. This component is responsible for accepting the request and possibly converting it into a desired internal format. This component is also responsible for any encryption/decryption that needs to happen as part of the request.

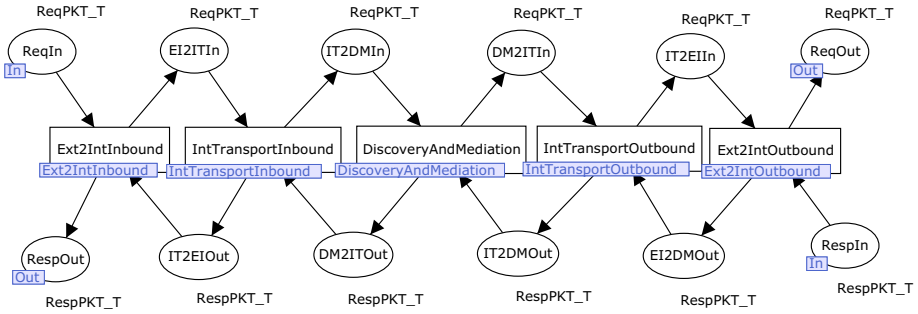


Fig. 4. The components of transport, discovery and mediation mechanism.

These different characteristics of what constitutes a request can be modeled very easily and explicitly by defining appropriate types for the associated tokens. The language for various type (or color in CPN parlance) and function declarations is CPN ML which is based on the functional programming language ML[11]. For our current purposes, a request is treated as a 3-tuple consisting of a consumer request identifier (*CID*), a connection type (*ConnType*) and type of service (*SERVICE*). The declaration of a request type (*ReqPKT*) in CPN ML is specified as:

```
colset ReqPKT =

     product CID * ConnType * SERVICE;
```

CPN supports a timed version by associating a *time-stamp* with each token. The general mechanism for this is to create *timed* color sets. For example, timed request tokens of color set, say, *ReqPKT_T*, can be defined as follows:

```
colset ReqPKT_T =

  ReqPKT timed;
```

We skip the explanation and details of the various other color sets and functions declarations for our CPN model because of space limitations.

The next component is *IntTransportInbound*. This component is responsible for essentially buffering and forwarding the request to the *DiscoveryAndMediation* component, which is the heart and the brain of this architecture. For our present purposes we focus on very simple discovery and mediation mechanism. This will get refined in the subsequent versions of the system, and this is one of the place where we hope the model to guide the implementation. Once a request has gone through service discovery and mediation, it is forwarded to outbound internal transport *IntTransportOutbound* and from there to the outbound external to internal interface component *Ext2IntOutbound*. Response packets simply follow the reverse route, as illustrated.

Using the hierarchical features of CPN, details of these individual components have been created on the associated pages. Next we present details of two of the components, namely, *Ext2IntInbound* and *DiscoveryAndMediation*.

For our present purposes, we are only focusing of one connection type, namely, http. In general though, the connection type on the service consumer side can be different from that on the service provider side and all this could be different from the internal connection type. The internal details of *Ext2IntInbound*, shown in **Fig. 5**, are as follows. It receives the service request (as a CPN timed token of type *ReqPKT_T*) from the consumer. It can then accept this request by firing the *AcceptConnection* transition. CPN provides facility to associate code segments with firing of transitions.

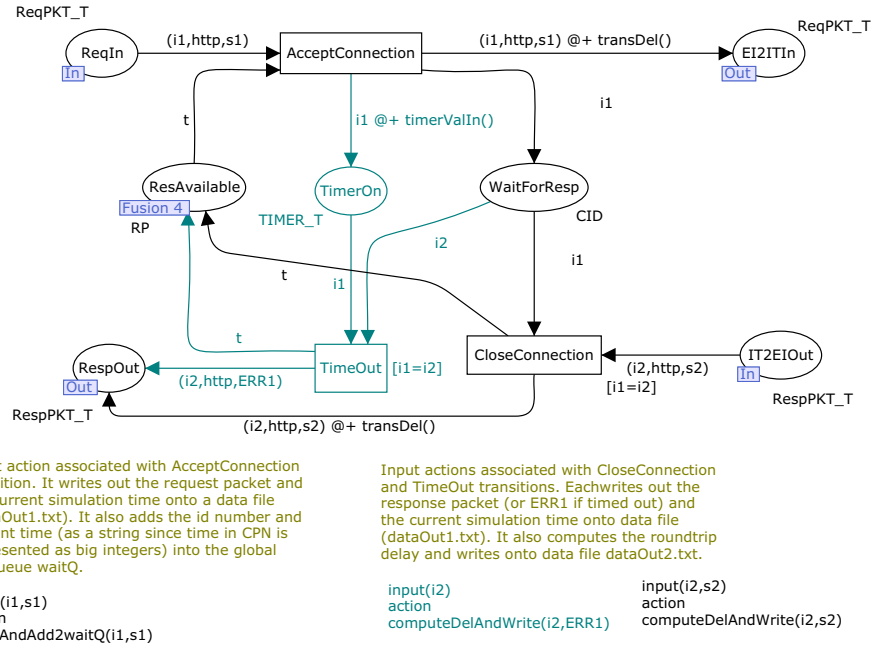


Fig. 5. Detailed net showing activities associated with Ext2IntInbound component.

Here the input action associated with the *AcceptConnection* transition is to write out the request packet and the current simulation time into a data file that can be examined later for desired properties and behavior. If there were any external to internal connection type translations involved, there would be added transitions to take care of those details here. The request is then passed to the internal transport by placing the token in the place named *EI2ITIn*. At the same time a token representing the request identifier is added to the place named *WaitForResp*.

In a real-life scenario, each connection that is open consumes some resources. Tokens in the place named *ResAvailable* represent the current number of resources available. A token from this place is removed for each firing of the *AcceptConnection* transition. This represents allocation of a resource (for example a thread from a thread pool). The connection is required to time-out if no response arrives within some specified time-out value. Thus, simultaneously a timed token is put in the place called *TimerOn*. The time stamp of this token represents the time-out value and can be set from a file by making use of input/output facilities of the CPN Tool.

The semantics of timestamp in CPN are that the associated token remains unavailable until the current simulation time becomes equal to or exceeds the timestamp value. Thus, if the response comes in, that is, a token with the correct id value and time-stamp smaller than that of the associated timer arrives, the transition *CloseConnection* fires and the response is forwarded to the consumer by placing the request in the place named *RespOut*. Otherwise, the transition *TimeOut* fires signaling expiration of the timer, and an error response is forwarded. Note that both *CloseConnection* and *TimeOut* have a guard *[i1=i2]*. This ensures that the response or the time-out is

matched with the correct request id. Finally, when either a time-out occurs or a connection is closed, the allocated resource is returned to the pool of resources. This is achieved by adding a token back into the place *ResAvailable.*

Details of the discovery and mediation are given by the net shown in **Fig. 6.** We are currently focusing on a very simple discovery and mediation. In particular, we do not account for mobility of service providers. The function *validReq : SERVICE* → *BOOL* is hard coded in that requests for certain services are considered unavailable. In particular, service *d* is considered invalid since in the current test set there are no providers for service *d.* When a request arrives in *IT2DMin*, it is checked for validity and forwarded to next component. This is achieved via firing the *transition forwardRequest* which deposits the request in *DM2ITIn*.

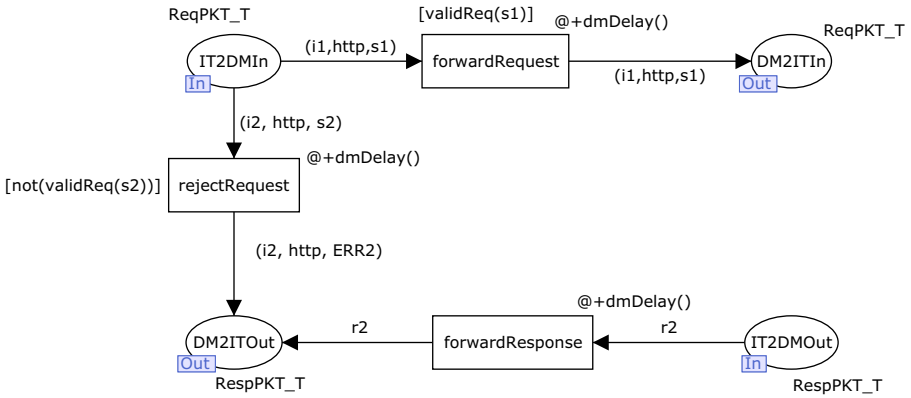


Fig. 6. Net representing details of discovery and mediation component

Note that the facility in CPN to associate data values with tokens and manipulate them or examine them and control actions based on them is a powerful one. Without such a facility it would be difficult to model requirements such as validity of requests, etc. If the incoming request is not valid, an error response is returned and this is indicated by firing of transition *rejectRequest*, which deposits an error packet in *DM2ITOut*. This component is also responsible for forwarding a response packet which is indicated by firing of transition *forwardResponse*. We skip the details of rest of the components here and discuss our verification and validation process next.

6 Quality Assurance, Verification & Validation, and CPN

From industry acceptance point of view and to have the validity of any model predictions incorporated into development, we needed to put in place a well defined verification and validation process for quality assurance. Furthermore, this verification and validation activity was to be carried out not by the modeler but by a third party, which

usually is a Quality Assurance (QA) team associated with a traditional software development process. CPN offers the following four possible analysis approaches:

- Interactive and automatic simulation
- Performance analysis
- State space and invariant analysis
- Temporal logic based analysis of state spaces

Given that a QA person may not be conversant with state space based analysis and given that usually a combination of strategies is required for a meaningful analysis, we decided to start with simulation and performance analysis. Furthermore, CPN provides a full spectrum of input/output facilities and user-defined functions so that we were able to parameterize all relevant input data and read it from data files. This also made it possible for a third party to run simulation and gather data with different input values. CPN Tool also provides an extensive collection of monitoring, performance analysis, and data logging facilities that further simplifies this task [12]. The verification and validation process and approach to integrating modeling into development that we have currently adopted is described in [13]. A simplified view relating modeling, integration, verification, and validation is given in **Fig. 7** below.

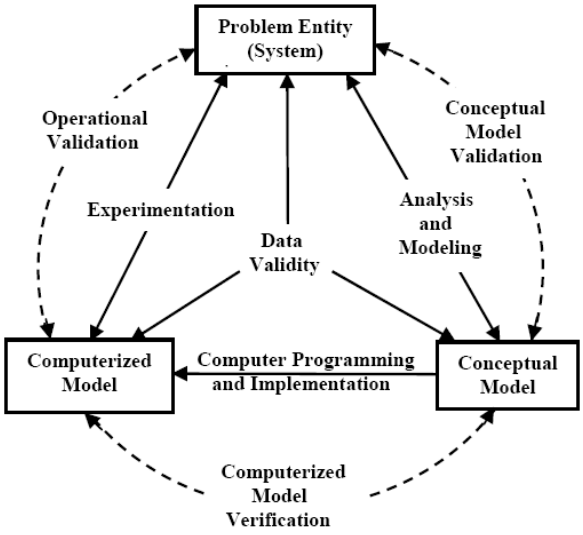


Fig. 7. Simplified modeling, integration, verification and validation process [13].

In our case, the edge labeled *Computer Programming and Implementation* gets realized as a CPN model. Furthermore, what this picture does not communicate is the incremental or spiral nature of the process. Essentially, we repeat the depicted process in each spiral. Our starting point was a base implementation. We then created a model for it. The model was subjected to verification and validation process. The preliminary results from this exercise are presented in the next section.

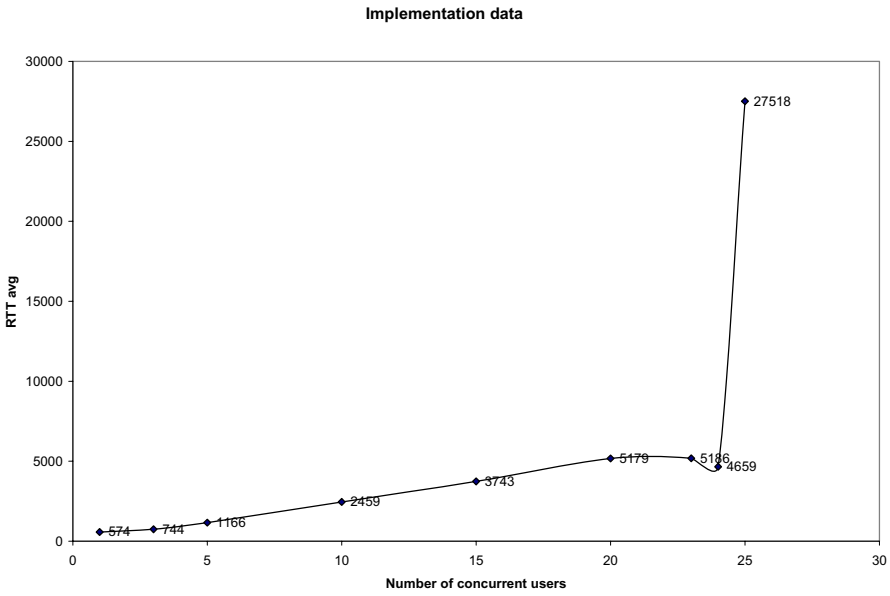


Fig. 8. Average RTT data from system implementation

7 Preliminary Results

Following the approach outlined above we carried out the verification and validation of the created model for quality assessment. To ascertain the validity of the model, we compared the behavior of the model with the corresponding behavior in the real system. For our initial validation attempt, the behavior we chose to ascertain was the performance of the system as the number of concurrent/simultaneous requests was increased. We developed an experiment in the run-time lab to measure the average round-trip time (RTT) of request-response interactions as we varied the number of simultaneous users presenting requests to the system. Performance was thus quantified as average round-trip time.

The experiment was run both on the model and on the real system. Experimental data was used to generate two graphs – one for the model and one for the real system. The shapes of the curves on these two graphs were compared to determine whether the behaviors were similar or not.

The following figures represent the data we collected from our experiments where the underlying resource pool contained 25 possible thread resources. **Fig. 8** shows the performance of the real system, while **Fig. 9** represents data from the model.

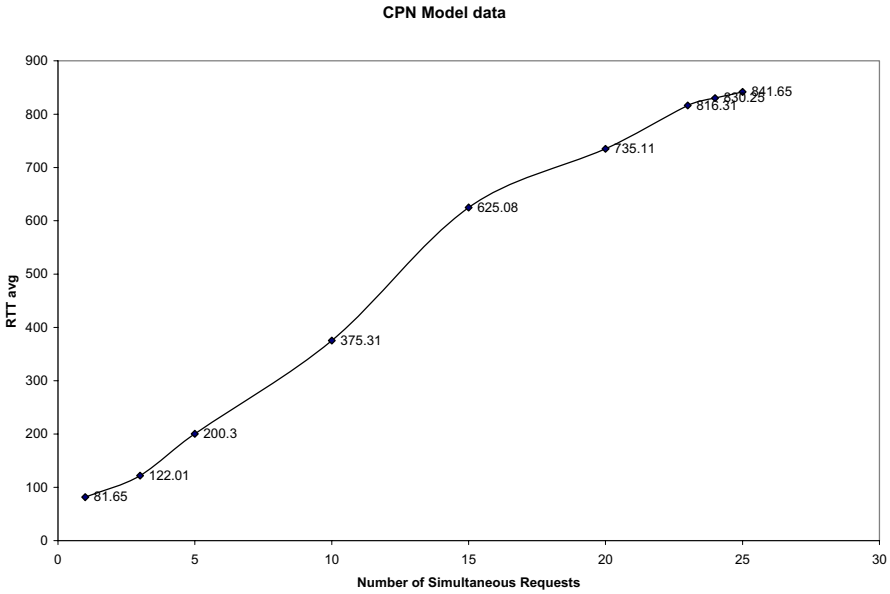


Fig. 9. Average RTT data from model simulation.

We note the following behaviors that are visible in both the system implementation and the model simulation:

- Performance decreases with increasing numbers of simultaneous requests.
- Performance bottleneck occurs when the size of the resource pool available to service requests equals the number of simultaneous requests.

However, a sharp discrepancy exists in the two behaviors when the number of concurrent request reaches the maximum resource pool size. Furthermore, the system implementation could not handle any more requests after this point was reached. The system implementation shows a sudden spike whereas the model data shows a gradual increase. This discrepancy was puzzling to us and our investigation using the CPN model found a bug in system implementation whereby threads for de-queuing operation were being allocated from the same pool as servlet pool creating a deadlock situation. This deadlock situation in the system implementation was later rectified and the results from re-verification and re-validation are given in **Fig. 10** below. It is easily seen that the two graphs show similar behavior. Ideally, these two graphs should coincide. In order for this to happen, the model needs realistic values for each of the parameters it has. However, we are currently limited in terms of what parameter values we can measure in the run-time lab on the real system. We are currently investigating approaches to such value measurements and estimation if real values cannot be measured for all model parameters.

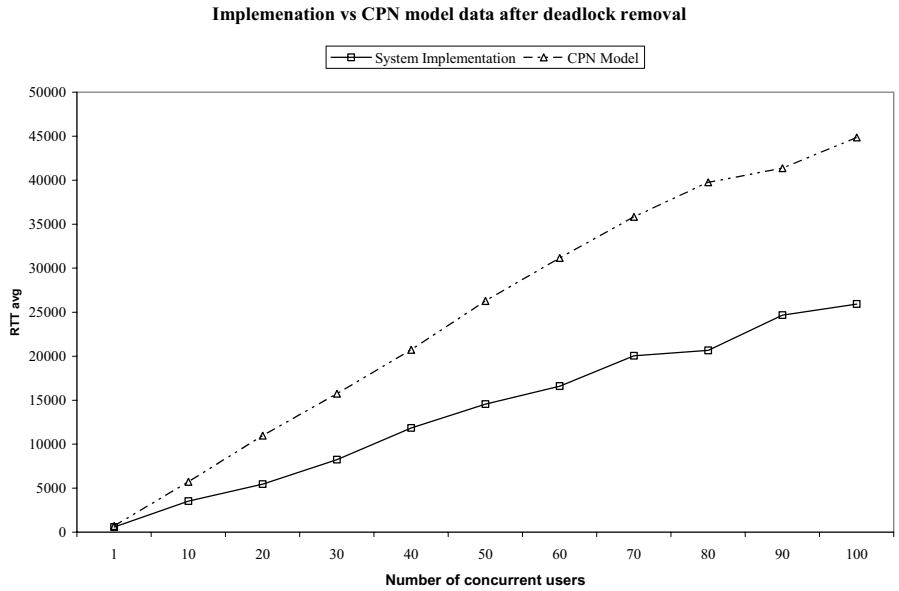


Fig. 10. Avg. RTT data from implementation and model after system deadlock was removed.

8 Conclusions and Future Work

The service oriented architecture (SOA) concept offers a framework for integration of systems and interoperability. This approach is very attractive in many business settings and is especially attractive in a defense setting where the traditional "stovepipe" approach has resulted in poor integration of systems and rendered them non-interoperable. The US DoD has an initiative called Net-Centric Enterprise Solutions for Interoperability (NESI) with the purpose to provide a service-oriented architecture solution approach for defense applications. Thus, many defense operations, including safety-critical ones, are soon to be deployed on a service oriented basis.

A model driven development based approach offers possibility of quality assessment, verification and validation, and quality prediction of such deployments. We presented a service oriented architecture and its model using Colored Petri Nets (CPNs). We illustrated aspects of CPN and the associated modeling and analysis tool called CPN Tool that have made it possible for us to integrate this approach as part of a large-scale defense software development. We also have access to a reference implementation that was used in our verification and validation process. Our preliminary analysis and results revealed a deadlock situation in the system implementation showing an early benefit of model integration in system development. Our future work includes extending the model to include other features and components of the architecture including presence and discovery mechanisms, mobility, and load balancing. Through our modeling effort we hope to guide the current system implemen-

tation and show benefits of a model driven approach in quality assessment, assurance, and prediction.

Acknowledgements: We would like to thank the Gestalt ARCES Team for their help and support. This research was supported in part by the Air Force Materiel Command (AFMC), Electronic Systems Group (ESC) under contract number FA8726-05-C-0008 . The views and conclusions contained here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of USAF, AFMC, ESC, or the U.S. Government.

References

[1] S. Anand, S. Padmanabhuni, and J. Ganesh, Perspectives on Service Oriented Architecture (tutorial). Proceedings of the 2005 IEEE International Conference on Services Computing, 2005.

[2] N. Bieberstein, S. Bose, M. Fiammante, K. Jones, and Rawn Shah, Service-Oriented Architecture Compass: Business Value, Planning, and Enterprise Roadmap. IBM Press, 2005.

[3] W. Tsai, Y. Chen, and R. Paul, Specification-Based Verification and Validation of Web Services and Service-Oriented Operating Systems. Proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 05), Sedona, 2005, pp. 139-147.

[4] Net-Centric Implementation, Part 1: Overview (Version 1.1, June 3, 2005), Netcentric Enterprise Solutions for Interoperability (NESI) project, http://nesipublic.spawar.navy.mil/docs/part1, accessed Feb. 24, 2006.

[5] Capability Development Document for Net-Centric Enterprise Services, Draft Version 0.7.15.2, April 9, 2004.

[6] K. Jensen, *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Volume 1, Basic Concepts. Monographs in Theoretical Computer Science, Springer-Verlag, 2nd corrected printing 1997.

[7] C. Girault and R. Valk, *Petri Nets for Systems Engineering*, Springer-Verlag, 2003.

[8] W. Reisig, *Petri Nets.* EATCS Monographs in Theoretical Computer Science, Vol. 4, Springer-Verlag, 1985.

[9] K. Jensen, *An Introduction to the Theoretical Aspects of Coloured Petri Nets*. In: J.W. de Bakker, W.-P. de Roever, G. Rozenberg (eds.): A Decade of Concurrency, Lecture Notes in Computer Science vol. 803, Springer-Verlag 1994, pp. 230-272.

[10] A. V. Ratzer, et al., CPN Tools for Editing, Simulating, and Analysing Coloured Petri Nets. In W.v.d. Aalst and E. Best (eds.): Application and Theory of Petri Nets 2003. Proceedings of the 24th International Conference on the Application and Theory of Petri Nets (ICATPN 2003). Lecture Notes in Computer Science, vol. 2679, Springer-Verlag, 2003, pp. 450-462.

[11] J. D. Ullman, *Elements of ML Programming*, Prentice-Hall, 1998.

[12] B. Lindstrøm and L. Wells. Towards a monitoring framework for discrete event system simulations. In *Proceedings of the 6th International Workshop on Discrete Event Systems (WODES'02)*, 2002. (Also see http://wiki.daimi.au.dk/cpntools-help/cpntools-help.wiki .)

[13] R. D. Sargent, Verification and validation of simulation models, *Proceedings of the 2003 Winter Simulation Conference*, S. Chick, P. J. Sánchez, D. Ferrin, and D. J. Morrice, eds., 2003.

A Qualitative Investigation of UML Modeling Conventions

Bart Du Bois[1], Christian F.J. Lange[2], Serge Demeyer[1] and Michel R.V. Chaudron[2]

[1] Lab On REengineering, University of Antwerp, Belgium
{Bart.DuBois,Serge.Demeyer}@ua.ac.be
[2] Dept. of Mathematics and Computer Science, Technische Universiteit Eindhoven
{C.F.J.Lange,M.R.V.Chaudron}@tue.nl

Abstract. Analogue to the more familiar notion of coding conventions, modeling conventions attempt to ensure uniformity and prevent common modeling defects. While it has been shown that modeling conventions can decrease defect density, it is currently unclear whether this decreased defect density results in higher model quality, i.e., whether models created with modeling conventions exhibit higher fitness for purpose.
In a controlled experiment[3] with 27 master-level computer science students, we evaluated quality differences between UML analysis and design models created with and without modeling conventions. We were unable to discern significant differences w.r.t. the clarity, completeness and validity of the information the model is meant to represent, nor w.r.t. the models' perceived suitability for implementation and testing.
We interpret our findings as an indication that modeling conventions should guide the analyst in identifying what information to model, as well as how to model it, lest their effectiveness be limited to optimizing merely syntactic quality.

1 Introduction

In [6], a classification of common defects in UML analysis and design models is discussed. These defects often remain undetected and cause misinterpretations by the reader. To prevent these defects, *modeling conventions* have been composed that, similar to the concept of code conventions, ensure a uniform manner of modeling [7]. We designed a pair of experiments to validate the effectiveness of using such modeling conventions, focusing on their effectiveness w.r.t. respectively (i) defect prevention; and (ii) model quality. We reported on the prevention of defects in [8]. Our study of the effect of modeling conventions on model quality forms the subject of this paper.

In the first experiment, we evaluated how the use of modeling conventions for preventing modeling defects affected defect density and modeling effort [8]. These modeling conventions are enlisted in Appendix A, and have been discussed previously in [6]. This set of 23 conventions has been composed through a literature

[3] A replication package is provided at http://www.lore.ua.ac.be/Research/Artefacts

review and through observations from industrial case studies, and concern abstraction, balance, completeness, consistency, design, layout and naming. These conventions are *formative*, in that they focus on specifying *how* information should be modeled, rather than specifying *what* should be modeled.

Our observations on 35 three person modeling teams demonstrated that, while the use of these modeling conventions required more modeling effort, the defect density of resulting UML models was reduced. However, this defect density reduction was not statistically significant, meaning that there is a (small) possibility, albeit small, that the observed differences might be due to chance.

This paper reports on the second experiment, observing differences in representational quality between the models created in the first experiment. We define *representational quality* of a model as the clarity, completeness and validity of the information the model is meant to represent. Typical flaws in representational quality are information loss, misinformation, ambiguity or susceptibility to misinterpretation, and a perceived unsuitability as input for future usages, as a.o. implementation and testing. This study investigates whether models created using common modeling conventions exhibit higher representational quality.

The paper is structured as follows. The selected quality framework is elaborated in section 2. The set-up of the experiment is explained in section 3, and the analysis of the resulting data is discussed and interpreted in section 4. We analyze the threats to validity in section 5. Finally, we conclude in section 6.

2 Evaluating Model Quality

Through a literature review, we identified three quality models for conceptual models:

- Based upon semiotic theory, different quality dimensions have been distinguished in Lindland's qualty framework, being (i) syntactic quality; (ii) semantic quality; and (iii) pragmatic quality [9]. Empirical support for the resulting quality framework has been established [11].
- Alternatively, [3] proposes a list of 24 quality attributes, operationalized in metrics. The integration between the semiotic approach of [9] and the unsystematic list approach of [3] is performed by [5].
- From design applications in computer architecture and protocol specification, [13] derived six quality criteria for semantic and pragmatic quality. These authors differentiate between external (completeness, inherence and clarity) and internal quality (consistency, orthogonality and generality) criteria. However, the criteria proposed are not operationalized nor have they been validated.

As clarity, completeness and validity are central to Lindland's framework, it is particularly well-suited for evaluating representational quality. Thus, we select this quality framework as a measurement instrument.

2.1 Lindland's quality framework

Lindland's framework relates different aspects of modeling to three linguistic concepts: syntax, semantics and pragmatics [9]. These concepts are described as follows (citing from [9]):

Syntax *relates the model to the modeling language by describing relations among language constructs without considering their meaning.*

Semantics *relates the model to the domain by considering not only syntax, but also relations among statements and their meaning.*

Pragmatics *relates the model to the audience's interpretation by considering not only syntax and semantics, but also how the audience (anyone involved in modeling) will interpret them.*

These descriptions of the concepts of syntax, semantics and pragmatics refer to relationships. The evaluation of these relationships gives rise to the notion of syntactic, semantic and pragmatic quality. We note that the effect of UML modeling conventions on syntactic quality has been the target of our previous experiment [8], and is therefore not included in this study.

In [5], Lindland's quality framework is extended to express one additional quality attribute. *Social quality* evaluates the relationship among the audience interpretation, i.e. to which extent the audience agrees or disagrees on the statements within the model.

With regard to representational quality, we are less interested in the relationship between the model and the audience's interpretation – indicated by pragmatic quality – than in the relationship between the domain and the audience's interpretation, as the former is unrelated to the information the model is meant to represent. Accordingly, we will not observe pragmatic quality, but instead introduce an additional quality attribute, *communicative quality*, that targets the evaluation of the audience's interpretation of the domain.

2.2 Measuring model quality

Lindland's quality framework evaluates the relationships between model, modeling domain and interpretation using the elementary notion of a statement. A *statement* is a *sentence representing one property of a certain phenomenon* [5]. Statements are extracted from a canonical form representation of the language, which in UML, is specific to each diagram type. An example of a statement in a use case diagram is the capability of an actor to employ a feature.

The set of statements that are relevant and valid in the domain are noted as D, the set of statements that are explicit in the model as M_E, and the set of statements in the interpretation of an interpreter i are symbolized with I_i. We say that a statement is *explicit* in case it can be confirmed from that sentence without the use of inference. Using these three sets, indicators for semantic quality (and also pragmatic quality, that we do not include in this study) have been defined that are similar to the concepts of recall and precision:

Semantic Completeness (SC) is the ratio of the number of modeled domain statements $|M_E \cap D|$ and the total number of domain statements $|D|$.

Semantic Validity (SV) is the ratio of the number of modeled domain statements $|M_E \cap D|$ and the total number of model statements $|M_E|$.

Krogstie extended Lindland's quality framework through the definition of *social quality* [5]. The single proposed metric of social quality is:

Relative Agreement among Interpreters (RAI) is calculated as the number of statements in the intersection between the statements in the interpretations of all n interpreters $|\bigcap_{\forall i,j \in [1,n]} I_i \cap I_j|$.

Similar to semantic quality, we introduce the following metrics for communicative quality:

Communicative Completeness (CC) is the ratio of the number of recognized modeled domain statements $|I_i \cap M_E \cap D|$ and the total number of modeled domain statements $|M_E \cap D|$.

Communicative Validity (CV) is the ratio of the number of recognized modeled domain statements $|I_i \cap M_E \cap D|$ and the total number of statements in the interpretation of interpreter i $|I_i|$.

Communicative completeness and validity respectively quantify the extent to which information has been lost or added during modeling.

2.3 Estimating model quality

The difficulty in applying the metrics for semantic, social and communicative quality mentioned above lies in the identification of the set of model statements (M_E), and interpretation statements (I_i). In contrast, the set of domain statements (D) is uniquely defined and can reasonably be expected to have a considerable intersection with the set of model and interpretation statements. Accordingly, we choose to estimate the sets of domain statements, model statements and interpretation statements, by verifying their intersection with a *selected* set of domain statements (D_s):

Semantic Completeness can be estimated by taking the ratio between the number of modeled selected domain statements $|M_E \cap D_s|$ and the total number of selected domain statements $|D_s|$.

Relative Agreement among Interpreters can be estimated by assessing to which extent the n interpreters agree or disagree on the selected domain statements, some of which are modeled while others are not: $|\bigcap_{\forall i,j \in [1,n]} I_i \cap I_j \cap D_s|$.

Communicative Completeness can be approximated by the ratio between the number of recognized modeled statements from the selected domain statements $|I_i \cap M_E \cap D_s|$ and the total number of modeled selected domain statements $|M_E \cap D_s|$.

Communicative Validity can be estimated by dividing the number of recognized modeled domain statements from the selected domain statements $|I_i \cap M_E \cap D_s|$ and the total number of recognized domain statements $|I_i \cap D_s|$.

Semantic validity cannot be approximated in this manner, as it requires an estimate of the set of statements that lie outside the set of domain statements ($|M_E \setminus D|$). Nonetheless, the resulting set of estimates for semantic, social and communicative quality allows to assess typical representational quality flaws as information loss (semantic and communicative completeness estimates), misinformation (communicative validity estimate) and misinterpretation (social quality estimate).

2.4 Evaluation of perceived fitness for purpose

In addition to evaluating quality differences between models composed with and without modeling conventions, we investigate the models' perceived fitness for purpose. Typically, UML analysis and design models are used as input for the activities of implementation and testing. Accordingly, we employ a questionnaire that addresses the perceived suitability of the model w.r.t. the following usages:

Comprehension – In order for UML analysis and design models to be comprehensible, the functional requirements of the software system, as well as traceability between the structural entities and these requirements should be clear.

Implementation – To support the activity of implementation, UML analysis and design models should provide information on structural entities such as packages, classes, methods and attributes, and the relationships between them as containment, inheritance, invocation and reference.

Testing – To support the activity of testing, UML analysis and design models should supply information regarding the pre- and postconditions of each method, as well as the invariants to be respected throughout execution.

3 Experimental Set-Up

Using the classical Goal-Question-Metric template, we describe the purpose of this study as follows:

Analyze UML models
for the purpose of evaluation of modeling conventions effectiveness
with respect to the representational quality of the resulting model
from the perspective of the analyst/designer
in the context of master-level computer science students

Using our refinement of representational model quality presented in the previous section, we define the following null hypotheses:

$H_{0,SeQ}$ – UML analysis and design models composed with or without modeling conventions do not differ w.r.t. *semantic quality*.
$H_{0,SoQ}$ – UML analysis and design models composed with or without modeling conventions do not differ w.r.t. *social quality*.
$H_{0,CoQ}$ – UML analysis and design models composed with or without modeling conventions do not differ w.r.t. *communicative quality*.

3.1 Experimental Design

In this study, we use a three-group posttest-only randomized experiment, consisting of a single control group and two treatment groups:

noMC – no modeling conventions. This group of subjects, referred to as the *control group* were given UML analysis and design models that were composed *without* modeling conventions.
MC – modeling conventions. The subjects in this treatment group received UML analysis and design models that were composed using the list of modeling conventions enlisted in Appendix A.
MC+T – tool-supported modeling conventions. Subjects in this treatment group received UML analysis and design models that were composed using both a list of modeling conventions and a tool to support the detection of their violation.

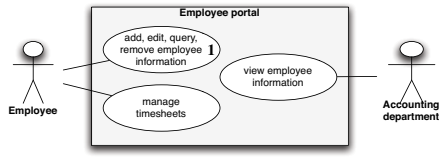
3.2 Experimental Tasks and Objects

The experiment was performed using pen and paper only. Each student was provided with (i) a hardcopy of all diagrams of a single model; (ii) a questionnaire; and (iii) a vocabulary.

The questionnaire contained a single introduction page that described the task. Another explanatory page displayed one example question and its solution, elaborating on the steps to be applied. The example question, illustrated in Table 1, asks the participant to verify whether a given UML analysis and design model confirms a given statement. As an argument for the confirmation of a statement, the participant should be able to indicate a diagram fragment dictating that the statement should hold. In case such a fragment can be found, the participant annotates the fragment with the question number.

Table 1. Example question and supporting diagram fragment

Nr	Statement	Confirmed	Not Confirmed
1	The software system should support querying employee information.	O	O



Aside this documentation, the questionnaire consisted of three parts. A *pretest questionnaire* section, asking questions about their knowledge on and experience with UML in general and various UML models in particular. The *main* part of the questionnaire asked subjects to evaluate whether a given statement was explicitly confirmed by the given model. Only two options were possible, being either "confirmed", or "not confirmed". Finally, the *posttest questionnaire* section asked for remarks. This experimental procedure was tested in a pilot study with 4 researchers.

The main part of the questionnaire allows to estimate semantic, social and communicative quality. We have identified over 60 statements that are relevant and valid in the domain, derived from the informal requirement specification for which the subjects of the first experiment composed the UML models. From this set of 60 statements, a selection of 22 statements was made, comprising the set of selected domain statementsD_s.

For each experimental group ($noMC$, MC, $MC+T$), a representative set of three UML analysis and design models was selected from the set of output models of the first experiment. The selected models serve as experimental objects, and were representative w.r.t. syntactic quality, defined as the density of modeling defects present in the model. These UML models – modeling a typical application in the insurance domain – consisted of six different types of UML diagrams used for analysis and design. The frequency of each of the diagram types in each model is provided in Table 2.

Table 2. Frequency of the diagram types in each model.

	noMC			**MC**			**MC+T**		
type	no_2	no_4	no_8	MC_2	MC_4	MC_5	$MC+T_4$	$MC+T_6$	$MC+T_{10}$
Class Diagram	6	1	6	8	1	1	11	1	5
Package Diagram	1	0	0	0	0	0	0	0	1
Collaboration Diagram	0	0	0	0	0	0	0	1	0
Deployment Diagram	0	0	0	0	0	0	1	1	1
Use Case Diagram	7	1	5	0	3	5	6	5	1
Sequence Diagram	6	26	10	3	39	14	8	56	15
total	20	28	16	11	43	20	26	23	64

The set of selected domain statements D_s can be categorized in instances of the following generic categories:

Features – The software system should support *feature X*, e.g., converting a quote into a real policy. The questionnaire contained nine feature statements. Mostly, confirmations of these statements can be found in use case diagrams, but also in class or sequence diagrams.

Concepts – *Concept X* has *aggregated concept Y*, e.g., a quote has a premium amount. The questionnaire contained eight concept statements. These statements can be explicitly confirmed in class diagrams only.

Interactions – Actor X can employ feature Y, e.g. clients can request policies. The questionnaire contained three interaction statements. These statements can be found in use case and collaboration diagrams.

Scenarios – Scenario X should incorporate *subscenario Y*, e.g., results of a client requesting quotes should contain insurance policy combinations of the insurance policy requested. The questionnaire contained two scenarios statements, that can be recognized in use case and sequence diagrams.

As the different models used synonyms for some concepts, a glossary was provided indicating which names or verbs are synonyms.

3.3 Experimental Procedure

The procedure for this experiment consisted of two major phases. First, in preparation of the experiment, the semantic quality of each selected model was assessed. Second, two executions of the experimental procedure (runs) were held to observe subjects performing the experimental task described in the previous subsection, thereby assessing the models' communicative and social quality.

Assessment of semantic quality. This assessment was performed by three evaluators, and did not require the participation of experimental subjects. The three evaluators were the first two authors of this paper, and a colleague from the first authors' research lab. After an individual assessment, conflicts were resolved resulting in agreement on the recognition of each selected domain statement in each model. This evaluation procedure provided the data to calculate the semantic completeness and semantic validity of each of the nine selected models.

Assessment of social and communicative quality. Each experimental run was held in a classroom, and adhered to the following procedure. Subjects were first randomized into experimental groups, and then provided with the experimental material. Subjects were asked to write their name on the material, to take the time to read the instructions written on an introduction page, and finally to complete the three parts of the questionnaire.

No time restrictions were placed on the completion of the assignment. When subjects completed the questionnaire, their experimental task was finished and they were allowed to leave. None of the runs lasted longer than 1.5 hours.

3.4 Experimental Variables

The independent variable subject to experimental control is entitled *modeling convention usage*, indicating whether the model was composed without modeling conventions ($noMC$), with modeling conventions (MC) or with modeling conventions and a tool to detect their violations ($MC+T$). The observed dependent variables are the estimators for semantic completeness (SC), communicative completeness (CC), communicative validity (CV) and relative agreement among interpreters (RAI), as defined in section 2.3. As these variables are all calculated as ratios, we express them in percentage.

3.5 Experimental Subjects

A total of 27 computer-science students participated in the controlled experiment. This experiment was performed across two universities in Belgium. 11 Final year MSc students from the university of Mons-Hainaut (Belgium) and 16 second-last-year MSc students from the University of Antwerp (Belgium) participated in the experiment in November and December 2005, respectively.

We evaluated the subjects' experience with the different types of UML diagrams using a questionnaire. All subjects had practical (although merely academic) experience with the diagrams required to answer the questions.

4 Data Analysis

Table 3 characterizes the experimental variables across the experimental groups.

Table 3. Statistics of the experimental variables

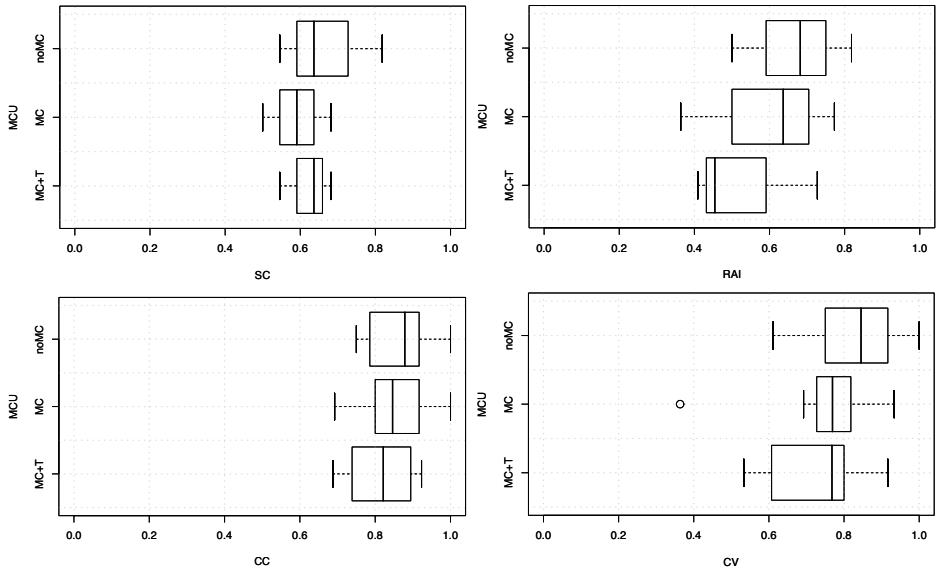
Hyp.	DV	Overall mean	MCU[1]	Mean	StdDev	Min	Max	$H(2)$	p-value
$H_{0,SeQ}$	SC	62.6%	noMC	66.7%	13.9%	54.5%	81.8%	0.4786	.7872
			MC	59.1%	9.1%	50.0%	68.2%		
			MC+T	62.1%	6.9%	54.5%	68.2%		
$H_{0,SoQ}$	RAI	59.6%	noMC	66.7%	15.6%	50.0%	81.8%	1.1556	.5611
			MC	59.1%	20.8%	36.4%	77.3%		
			MC+T	53.0%	17.2%	40.1%	72.7%		
$H_{0,CoQ}$	CC	76.9%	noMC	82.7%	14.1%	61.0%	100.0%	2.7298	.2554
			MC	74.5%	16.0%	36.0%	93.0%		
			MC+T	72.5%	13.1%	53.0%	92.0%		
	CV	85.0%	noMC	87.0%	7.9%	75.0%	100.0%	1.5235	.4668
			MC	85.9%	10.6%	60.0%	100.0%		
			MC+T	81.5%	8.9%	69.0%	92.0%		

[1] **M**odeling **C**onvention **U**sage.

Semantic Completeness (SC) – The semantic completeness of models composed without modeling conventions was somewhat higher, within a margin of 10% (see top left figure in Table 4). I.e., the models from group *noMC* described slightly more modeling domain statements. However, the *noMC* group also exhibits a larger standard deviation.

Relative Agreement among Interpreters (RAI) – There was considerable higher (about 14%) agreement among interpreters of the models composed without modeling conventions (see top right figure in Table 4). However,we also observed considerable standard deviations in Table 3 in all treatment groups.

Table 4. Variation of SC, RAI, CC and CV across experimental groups



Communicative Completeness (CC) The communicative completeness of models composed without modeling conventions was somewhat higher (around 10%) than that of models composed with modeling conventions.

Communicative Validity (CV) – The communicative validity is approximately equal between models composed with and without modeling conventions, as illustrated in in the bottom right figure in Table 4).

To verify whether the differences among experimental groups are statistically significant, Kruskal-Wallis test results are appended to Table 3. This test is a non-parametric variant of the typical Analysis of Variance (ANOVA), and is more robust with regard to assumptions about the distribution of the data, as well as unequal sample sizes ($\#noMC$=10,$\#MC$=9,$\#MC+T$=8). Moreover, the assumptions of at least an ordinal measurement level, independent groups and random sampling were also satisfied.

Table 3 indicates that the group differences concerning semantic, social and communicative quality are not statistically significant at the 90% level. Accordingly, we must accept the hypotheses stating that the UML analysis and design models composed with or without modeling conventions do not differ w.r.t. semantic, social and communicative quality.

4.1 Subjective evaluations

The questionnaire given to the experimental subjects also targeted the subjective evaluation of the suitability of the given UML analysis and design model as an input for future usage, as a.o. implementation and testing.

Questions asked concern the clarity of usage scenarios, of the design, and finally, of method specifications (see Table 5). For each of the 13 questions, subjects were asked to rate the model on a typical Likert scale (1=strongly disagree, 2=disagree, 3=neither agree nor disagree, 4=agree, 5=strongly agree). The median of each group is appended to each question. Group differences in rating were tested using the Kruskal-Wallis test, as is common for Likert scale data, and the resulting significance of group differences is discussed inline.

The questions regarding usage scenarios address the comprehensibility of the model w.r.t. functional requirements. Indifferent of whether the given model was composed with or without modeling conventions, most subjects agreed that the given model was comprehensible. While a significant difference between the three groups was remarked concerning (b1) using the Kruskal-Wallis test, with $H(2)$=6.5787, p=.03728, Dunn's multiple comparison post test indicated no pair of groups that differed significantly. Accordingly, we must accept that the models did not differ w.r.t. their perceived general comprehensibility.

The second set of questions concern the perceived suitability of the given UML model as an input for implementation. Except regarding contained packages (i1), and interactions between methods and attributes (i6), all subjects agreed that the given model supplies sufficient information. A difference between the three groups significant at the 90% level was remarked concerning (i4), with $H(2)$=4.8325, p=.08925. Once again, however, Dunn's multiple comparison post test indicated no pair of groups that differed significantly. Accordingly, we cannot state that the models differ w.r.t. the perceived suitability as an input for testing.

Finally, the last set of questions solicits the perceived suitability of the given model as an input for testing. While subjects that received a model composed without modeling conventions (*noMC* group) neither agreed nor disagreed with the availability of sufficient information, subjects that received a model composed *with* modeling conventions recognized that the model is insufficient for testing. However, these group differences were not significant.

Summarizing, we were unable to discern significant group differences w.r.t. the perceived suitability of the models as an input for implementation and testing.

5 Threats to Validity

Construct Validity is the degree to which the variables used measure the concepts they are to measure. We have decomposed representational quality, the main concept to be measured, into semantic, social and communicative quality, as well as perceived suitability for future usage, and have argued their proposed approximations.

Internal Validity is the degree to which the experimental setup allows to accurately attribute an observation to specific cause rather than alternative causes. Particular threats are due to selection bias. The selection of statements from the

Table 5. Questions regarding the perceived fitness for purpose

The UML model is comprehensible. I comprehend...
(b1) ... which usage scenarios the system should support. (medians: noMC=4, MC=4, MC+T=4)
(b2) ... by which actions each usage scenario is triggered, and which actor provides the action.(medians: noMC=4, MC=4, MC+T=4)
(b3) ... which user interactions occur during each usage scenario. (medians: noMC=4, MC=3, MC+T=4)
(b4) ... which classes and methods are involved in each usage scenario. (medians: noMC=4, MC=4, MC+T=4)
When given the assignment to *implement* the system described in the UML analysis model, I am confident that the diagrams provide sufficient information about...
(i1) ... the packages which the system contains. (medians: noMC=3, MC=3, MC+T=3)
(i2) ... the classes which each package contains. (medians: noMC=4, MC=4, MC+T=3)
(i3) ... the inheritance relationships between classes. (medians: noMC=4, MC=4, MC+T=4)
(i4) ... the attributes of each class, and their signature. An attribute's signature comprises its name and type. (medians: noMC=4, MC=4, MC+T=4)
(i5) ... the methods of each class, and their signature. A method's signature comprises its name, return type and parameter list. (medians: noMC=4, MC=4, MC+T=4)
(i6) ... the interactions between methods and attributes in the different classes. Such interactions consist of initialization, method calls and attribute references. (medians: noMC=4, MC=3, MC+T=3)
When given the assignment to *test* the system described in the UML analysis model, I am confident that the diagrams provide sufficient information about...
(t1) ... the preconditions that each method require. (medians: noMC=3, MC=2, MC+T=2)
(t2) ... the postconditions that each method guarantee. (medians: noMC=3, MC=2, MC+T=2)
(t3) ... the invariants that each method should respect. (medians: noMC=3, MC=2, MC+T=2)

domain D_s could not have introduced systematic differences, and the selection of model was performed as to be representative w.r.t. syntactic quality.

External Validity is the degree to which research results can be generalized outside the experimental setting or to the population under study. The UML analysis and design models consisted of about 27 diagrams of six different types each. Consequently, we do not consider the representativity of the UML models a serious threat to validity. Second, as the representativity of the subjects is a matter of discussion, we do not wish to generalize our results outside the context

of novice UML users. Thirdly, the set of modeling conventions was composed after a literature review of modeling conventions for UML, revealing design, syntax and diagram conventions. Our set of modeling conventions contains instances of these three categories.

Statistical Conclusion Validity is concerned with inferences about correlation (covariation) between treatment and outcome [12]. The likelihood of wrongfully concluding that cause and effect do not covary can be quantified using power analysis, which determines the sensitivity of the experimental set-up. For group differences of 1.25 times the standard deviation, our setup exhibits a 69.3% likelihood of discovering significant differences w.r.t. communicative quality and the perceived fitness for purpose of the models, and a 27.6% likelihood of discovering significant semantic and social quality differences. This means that group differences in communicative quality of around 15% are very likely to be discerned, but also, that the statistical tests for semantic and social quality have little power. Nonetheless, the considerable overlap as indicated in Table 4 does not indicate clear group differences w.r.t. semantic and social quality.

6 Conclusion

Based on the results of this experiment, we conclude that UML modeling conventions focusing on the prevention of common UML modeling defects (as reported in [6]) are unlikely to affect representational quality. In a comparison of groups of models composed with and without these modeling conventions, we did not observe significant differences w.r.t. information loss (indicated by semantic and communicative completeness), misinformation (indicated by communicative validity) nor ambiguity or misinterpretation (indicated by social quality). Moreover, no clear indications were found regarding the suitability for future usages as, a.o., implementation and testing.

We interpret our findings as an invitation to study the application of modeling conventions of a different nature. Conventions are needed that clarify which types of information are relevant to particular future model usages. Such modeling conventions might suggest the modeling of a type of information (e.g., features, concepts, interactions, scenarios) consistently in a particular (set of) diagram type(s). We hypothesize that this uniform manner of modeling different types of information is more likely to optimize semantic and communicative quality, as these types of information are the subject of their evaluation.

In other words, we argue that the optimization of the fitness for purpose of UML models requires modeling conventions that do not restrict themselves to mere properties of the model (e.g., syntax, design and layout). Rather, modeling conventions should support the modeler in identifying and consistently representing those types of information required for the model's future usage, e.g., in implementation and testing.

Acknowledgements – We would like to thank the subjects of the pilot study, the subjects of the two experimental runs and Prof. dr. Tom Mens of the University

of Mons-Hainaut (UMH), Belgium for aiding in the organization of the second experimental run.

References

[1] Ambler, S. W. (2005). *The Elements of UML(TM) 2.0 Style*. Cambridge University Press, New York, NY, USA.

[2] Berenbach, B. (2004). The evaluation of large, complex UML analysis and design models. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 232–241, Washington, DC, USA. IEEE Computer Society.

[3] Davis, A., Overmeyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., and Theofanos, M. (1993). Identifying and measuring quality in a software requirements specification. In *Proceedings of the First International Software Metrics Symposium*, pages 141–152.

[4] Genero, M., Piattini, M., and Calero, C. (2002). Empirical validation of class diagram metrics. In *ISESE '02: Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE'02)*, page 195, Washington, DC, USA. IEEE Computer Society.

[5] Krogstie, J. (1995). *Conceptual Modeling for Computerized Information Systems Support in Organizations*. PhD thesis, University of Trondheim, Norway.

[6] Lange, C.F.J. and Chaudron, M.R.V. (2006). Effects of defects in UML models - an experimental investigation. In *ICSE '06: Proceedings of the 28th International Conference on Software Engineering*, pages 401–411.

[7] Lange, C.F.J., Chaudron, M.R.V., and Muskens, J. (2006a). In practice: UML software architecture and design description. *IEEE Softw.*, 23(2):40–46.

[8] Lange, C.F.J., Du Bois, B., Chaudron, M.R.V., and Demeyer, S. (2006b). Experimentally investigating the effectiveness and effort of modeling conventions for the UML. In *O. Nierstrasz et al. (Eds.): MoDELS 2006, LNCS 4199*, pages 27–41.

[9] Leung, F. and Bolloju, N. (2005). Analyzing the quality of domain models developed by novice systems analysts. In *HICSS '05: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS'05) - Track 7*, page 188.2, Washington, DC, USA. IEEE Computer Society.

[9] Lindland, O. I., Sindre, G., and Solvberg, A. (1994). Understanding quality in conceptual modeling. *IEEE Softw.*, 11(2):42–49.

[10] Marinescu, R. (2002). *Measurement and Quality in Object-Oriented Design*. PhD thesis, Faculty of Automatics and Computer Science of the University Politehnica of Timisoara.

[11] Moody, D. L., Sindre, G., Brasethvik, T., and Solvberg, A. (2003). Evaluating the quality of information models: empirical testing of a conceptual model quality framework. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 295–305, Washington, DC, USA. IEEE Computer Society.

[12] Shadish, W. R., Cook, T. D., and Campbell, D. T. (2002). *Experimental and Quasi-Experimental Designs for Generalized Causal Inference*. Houghton Mifflin.

[13] Teeuw, B. and van den Berg, H. (1997). On the quality of conceptual models. Proceedings of the ER-97 Workshop on Behavioral Modeling and Design Transformation: Issues and Opportunities in Conceptual Modeling.

[14] Tilley, S. and Huang, S. (2003). A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. In *SIGDOC '03: Proceedings of the 21st annual international conference on Documentation*, pages 184–191, New York, NY, USA. ACM Press.

A Modeling Conventions

Table 6 enlists the modeling conventions employed in a previous experiment. These conventions were used by two of the experimental groups (MC and $MC+T$) while composing UML analysis and design models. As the resulting models were used in this experiment, it is relevant to recapitulate these conventions.

Table 6. Modeling Conventions

Category	ID	Convention
Abstraction	1	Classes in the same package must be of the same abstraction level.
	2	Classes, packages and use cases must have unique names.
	3	All use cases should cover a similar amount of functionality.
Balance	4	When you specify getters/setters/constructors for a class, specify them for all classes.
	5	When you specify visibility somewhere, specify it everywhere.
	6	Specify methods for the classes that have methods! Don't make a difference in whether you specify or don't specify methods as long as there is not a strong difference between the classes.
	7	Idem as 6 but for attributes.
Completeness	8	For classes with a complex internal behavior, specify the internal behavior using a state diagram.
	9	All classes that interact with other classes should be described in a sequence diagram.
	10	Each use case must be described by at least one sequence diagram.
	11	The type of ClassifierRoles (Objects) must be specified.
	12	A method that is relevant for interaction between classes should be called in a sequence diagram to describe how it is used for interaction.
	13	ClassifierRoles (Objects) should have a role name.
Consistency	14	Each message must correspond to a method (operation).
Design	15	Abstract classes should not be leafs.
	16	Inheritance trees should not have no more than 7 levels.
	17	Abstract classes should not have concrete superclasses.
	18	Classes should have high cohesion. Don't overload classes with unrelated functionality.
	19	Your classes should have low coupling.
Layout	20	Diagrams should not contain crossed lines (relations).
	21	Don't overload diagrams. Each diagram should focus on a specific concept/problem/functionality/...
Naming	22	Classes, use cases, operations, attributes, packages, etc. must have a name.
	23	Naming should use commonly accepted terminology, be non-ambiguous and precisely express the function/role/characteristic of an element.

OCL^2: Using OCL in the Formal Definition of OCL Expression Measures

Luis Reynoso[1], Marcela Genero[2] , José Cruz-Lemuz[2] and Mario Piattini[2]

Department of Computer Science, University of Comahue,
Buenos Aires 1400, 8300, Neuquén, Argentina
lreynoso@uncoma.edu.ar
[2] Alarcos Research Group,
Department of Technologies and Information Systems,
University of Castilla-La Mancha, Ciudad Real, Spain
{Marcela.Genero, Mario.Piattini, JoseAntonio.Cruz}@uclm.es

Abstract. Within the Object Oriented software measurement a lot of measures have proliferated during the last decades. However, most of the existent measures differ in the degree of formality used in their definition. If the measure definition is not precise enough, for instance when natural language is used, misinterpretations and misunderstanding of their intent can be introduced. Therefore, this situation may flaw the interpretation of experimental findings or even can difficult in building adequate measures extraction tools. This paper carefully describes how a set of measures that capture the structural properties of expressions specified with the Object Constraint Language (OCL) were precisely defined upon the OCL metamodel. So, we used OCL twice (OCL^2): as a language for defining measures and as a target to capture its core concepts through measures. In addition, given the relevance of models in the Model Driven Engineering (MDE) and their quality, the approach presented here could be extended for the formal definition of measures for each of the UML models.

Keywords. OCL, OCL metamodel, measures, formal definition

1 Introduction

A plethora of Object Oriented (OO) measures have been proposed from the nineties till nowadays. Intrinsic to any measure is its definition and theoretical and empirical validation. However, before addressing if the measures are theoretically or empirically valid it is important that they are "well" defined. As Baroni et al. [2] commented many difficulties arise when the measures are defined in an unclear or imprecise way:

- experimental findings can be misunderstood due to the fact that it may be not clear what the measure really captures,
- measures extraction tools can arrive to different results,

- and experiments replication is hampered.

Most of the existent measures differ in the degree of formality used in their definition. Two extreme approaches were used, informal and rigorous definitions. However none of these approaches had been widely accepted. On one hand, measures using an informal definition, such as measures defined in natural language, may be ambiguously defined. Hence, natural language may introduce misinterpretations and misunderstanding. On the other extreme, in a rigorous approach, some authors have used a combination of set theory and simple algebra to express their measures [8], [12]. This approach was not popular due the majority of members of OO community may not have the required background to understand the underpinning of the complex mathematical formalism used.

An example of how the use of natural language introduces ambiguity in the measure definition is considered in [2], referring to the measure definition of "Number of Times a Class is Reused", proposed by Lorenz and Kidd [16]. This measure is defined as the number of references to a class. We agree with Baroni et al. [2] that is not clear "What references are and how the metric should be computed, and many questions arise as: Should internal and external references be counted? Should references be considered in different modules, packages or subsystem? Does the inheritance relationship count as a reference?".

An important contribution to solve the problem of the formality degree in the measure definition is to use the Object Constraint Language [18] upon a design metamodel.

As part of our research work during the last two years, we have proposed a set of measures for OCL expressions, trying to find indicators for the understandability and modifiability of OCL expressions [20]. When we decided to formally define them we considered that the use of OCL for that purpose could have two advantages:

- The first is that OCL itself is precisely defined through metamodeling facilities, as an instance of the meta-metamodel of the OMG Meta Object Facility (MOF) [22], and the measure definition can be suitably placed at the same level (the M2 level) as the OCL definition.
- The second is a same language, OCL, is used as a formal language to define the UML and OCL semantics (at M2 Level) and is used by modelers for defining constraints on their models (at M1 Level). In fact the OCL was claimed as a language easy to use and easy to learn, and to be easily grasped by anybody familiar with OO modeling [9], [15], [24], [25]. So, the familiarity of this language can make the definition of our measures more modeler-friendly.

Thus, the approach of defining measures for OCL expressions using OCL metamodel and OCL language as the formal language allows an unambiguous definition. OCL was previously used by the QUASAR (QUantitative Approaches on Software Engineering And Reengineering) Research Group [3],[4],[5] for defining measures. However, the research group used OCL upon the UML metamodel. In our case, OCL is used as a language for defining measures for OCL expressions upon the OCL metamodel. This is why we called OCL^2 to the work presented in this paper.

In our approach when we compute the value of a specific measure an OCL expression is represented as an instantiation of OCL metaclasses. The instantiation has the shape of a tree, an abstract syntax tree (*ast*). We traverse the dynamic hierarchical

structure (the *ast*) and meanwhile we visit every element in the tree, we evaluate if each element of the tree is meaningful for the measure we want to compute. If it does, the measure is incremented otherwise it remains as it is. Due to all the measures we proposed in [20] are similarly defined, we will only explain in this paper the formal definition using OCL of the measures: the Number of Attributes referred through Navigations (NAN) and the Number of Navigated Classes (NNC).

This paper is structured as follows: Section 2 briefly introduces the measures we proposed for measuring structural properties for OCL expressions. Section 3 briefly explains the OCL metamodel and some of its metaclasses used to explain an instantiation. Section 4 describes an *ast* sample for an OCL expression and Section 5 explains the implemented strategy using a visitor pattern for traversing the *ast* and we show the formal definition of NAN and NNC measures. Finally, Section 6 concludes the paper and outlines the future work.

2 Measures for OCL Expressions

Our hypothesis is that structural properties of an OCL expression within an UML/OCL model (artifacts) have an impact on the cognitive complexity of modelers (subjects), and high cognitive complexity leads the OCL expression to exhibit undesirable external qualities on the final software product [13], such as less understandability or a reduced maintainability [7].

We thoroughly defined in [20] a suite of measures for structural properties of OCL expressions. Table 1 only introduces some of the measures we defined for measuring coupling.

Table 1. Measures for coupling within OCL expressions

Measure Acronym	Measure Description
NNR	Number of Navigated Relationships
NAN	Number of Attributes referred through Navigations
NNC	Number of Navigated Classes
WNCO	Weighted Number of Collection Operations
DN	Depth of Navigations

We defined measures for coupling within OCL expressions, because coupling is one the most complex software attribute in object oriented systems [3] and a high quality software design should obey the principle of low coupling [6], [7]. Furthermore, scanty information of object coupling is available in early stages of software development which only use UML graphical notations, and many times, many coupling decisions are made during implementation [25]. However, at early stages it would be useful the availability of more information about coupling, e.g. to decide which classes should undergo more intensive verification or validation. We believe that a UML/OCL model reveals more coupling information than a model specified

using UML only, due to the fact that OCL navigation defines coupling between the objects involved [25], and the coupled objects are usually manipulated in an OCL expression through collections and its collection operations (to handle its elements).

3 OCL Metamodel

As we previously mentioned the concepts of OCL and their relationships have been defined in the form of a MOF-compliant metamodel [18]. The benefit of a metamodel for OCL is that it precisely defines the structures and syntax of all OCL concepts like types, expressions, and values in an abstract way and by means of UML features. Thus, all legal OCL expressions can be systematically derived and instantiated from the metamodel.

The *Expression* package contains the main metaclasses of the OCL metamodel which are essential for the formal definition of our proposed measures. Figure 1 shows the core part of the *Expression* package. The basic structure in the package consists of the classes *OclExpression*, *PropertyCallExp* and *VariableExp* [18]. In OCL the concept of "property" conceptualize an attribute, method or rolename applied to an object (or OCL expression who evaluates to an object).

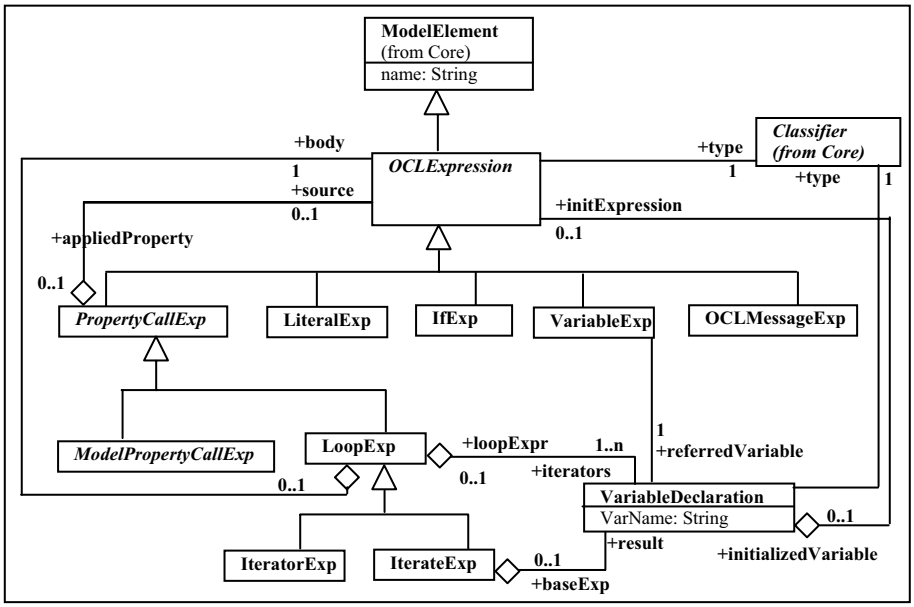


Fig. 1. Abstract syntax kernel metamodel for Expressions

This definition is consistent with the fact that: (1) each *PropertyCallExp* has exactly one source, identified by an *OCLExpression*; (2) A *ModelPropertyCallExp* (see

Figure 2) –a specialization of *PropertyCallExp*- generalizes all property calls that refer to *Features* or *AssociationEnds* in the UML metamodel [19], for instance:

- An *AttributeCallExp* is a reference to an Attribute of a Classifier defined in the UML model.
- A *NavigationCallExp* is a reference to an *AssociationEnd* or *AssociationClass* defined in the UML model.
- An *OperationCallExp* refers to an *Operation* in a *Classifier*.

This is shown in Figure 2 by the three different subtypes, each of which is associated with its own type of *ModelElement*.

Due to the fact that the OCL metamodel is composed of a set of more than thirty metaclasses we are not able to explain each of them. Nevertheless in the following section we will describe an OCL expression as an example of instantiating some of the aforementioned OCL metaclasses.

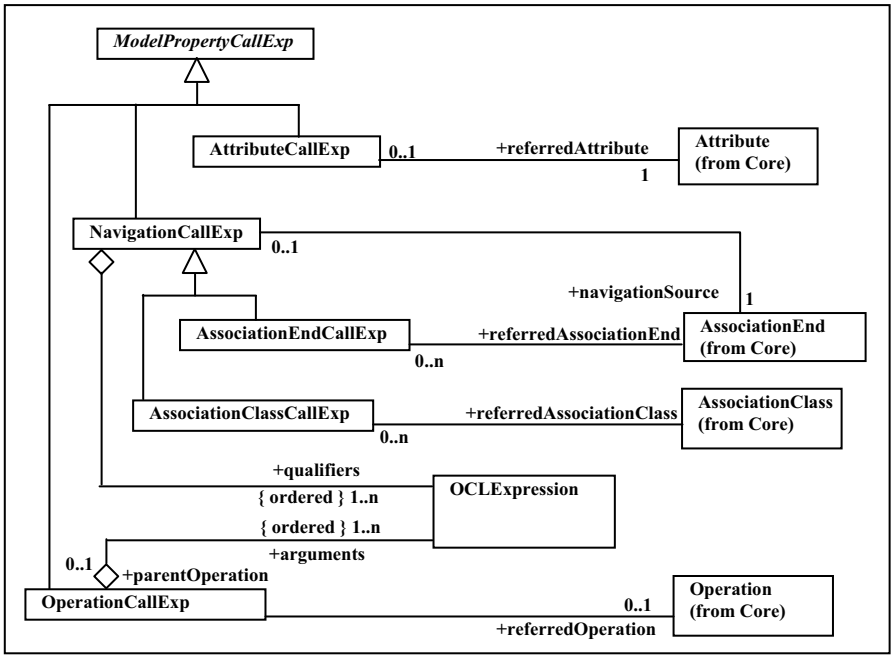


Fig. 2. Abstract syntax metamodel for *ModelPropertyCallExp*

4 A Sample of an "Abstract Syntax Tree" for an OCL Expression

The purpose of this section is to show an example of one *ast* built from an OCL expression. We choose as an example the invariant OCL expression attached to the *Flight* class of Figure 3. The meaning of the invariant expression is that a flight does

not contain more passengers than the number of seats of the airplane type associated with the airplane of the flight.

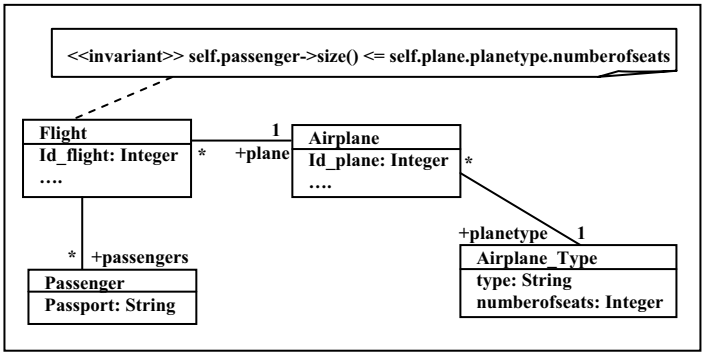


Fig. 3. OCL invariant expression in a class diagram

The basic instantiation of this fragment of the model for our example is consistent with the standard place where an invariant OCL expression occurs in the UML and OCL metamodel (see Figure 4).

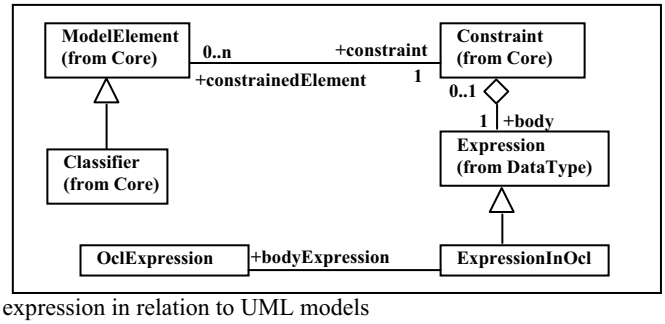


Fig. 4. OCL expression in relation to UML models

An OCL expression always constitutes the body of a Constraint object associated with one or more *ModelElement* objects. So, the instantiation includes two important objects (see Figure 5):

- a class object where its name is *Flight* (Classifier is a UML concept which represents a class, an interface, etc.)
- and a constraint object to represent an invariant constraint (see the two classes at the left top of Figure 5).

The body of the constraint will be represented by the object diagram for the *ast* of the invariant expression. The object diagram of Figure 5 basically shows an *ast* in the right part.

In order to build the tree, instances of the following OCL metaclassses have been used: *OperationCallExp, AttributeCallExp, AssociationEndCallExp, Operation, IntegerLiteralExp, VariableExp* and *VariableDeclaration* OCL metaclasses. In the tree

there is also an instance of the *Attribute* UML metaclass which constitutes the attribute referred by the *AttributeCallExp*, and three instances of the *AssociationEnd* UML metaclass.

The root of the tree of Figure 5 is the *OperationCallExp* expression, which has three branches:

- First, the source of the *OperationCallExp* is the subtree modeling the subexpression *self.passenger->size()*.
- the second branch models the referred operation, and
- the third branch represents the argument, the subtree modeling the subexpression *self.plane.planetype.numberofseats*.

In order to compute the value of a specific measure we must visit each of the tree nodes (instances of OCL metaclasses) and verify if each of them belongs to the particular metaclass we are interested to measure. The implemented strategy for visiting the elements is shown in the following section.

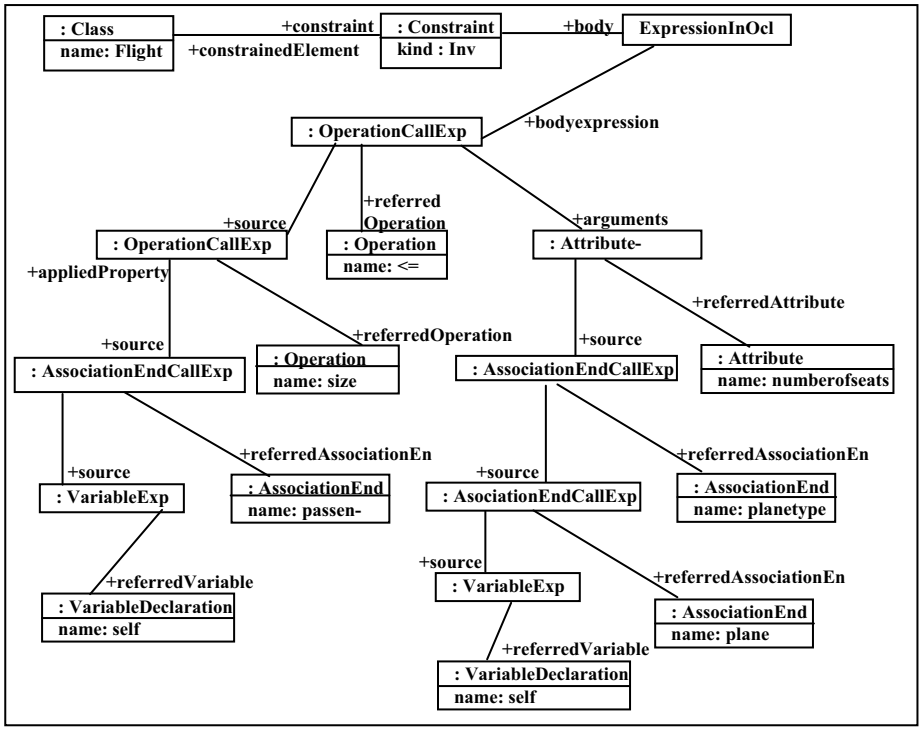


Fig. 5. *ast* built for an OCL invariant

5 Implemented Strategy

There are many operations we must define in order to compute the measures values, and these operations should be specified in many OCL metaclasses, but we do not

want to clutter the OCL metaclasses with these operations. To solve this problem we decided to use a Visitor Pattern [21]. The operations we must define are located into a separate object (a visitor). The visitor is sent to the tree root, eventually each element forwards the requests to its children and also its calls activate the visitor. The Visitor performs operations on each element. The main participants of a Visitor Pattern are:

- **Visitor:** declares a Visitor operation for each class of *ConcreteElement* in the object structure.
- **ConcreteVisitor:** implements each operation declared by Visitor.
- **Element:** defines an Accept operation that takes a Visitor as an argument.
- **ConcreteElement:** implements an Accept operation.
- **ObjectStructure:** can enumerate its elements.

A complete explanation of this pattern can be found in [11], [21]. In our case, the *Element* and *ConcreteElements* are represented by OCL metaclasses in the *Expression* package, and we will define Accept operations on them. *Visitor* and *ConcreteVisitor* are new classes introduced in our strategy to define the measures, and the *ObjectStructure* may be represented by either *Constraint* or *ExpressionInOCL* classes of Figure 4.

Figure 6 shows the basic UML design for implementing the strategy with a visitor. In this solution we also used an Enumeration UML class, *MetricAcronym*, which includes the acronym of the proposed measures (NNR, NAN, NNC, WNCO, DN, etc.)

This section is divided as follows: Subsection 5.1 shows how Accept operations were defined in the OCL metaclasses of the *Expression* Package, Subsection 5.2 shows the Visitor Class and its operations and Subsection 5.3 describes how to obtain the value of a measure. All the expressions used in this section were syntactically verified using ECLIPSE [10] and the OCTUPUS component [14] (a plug-in of ECLIPSE).

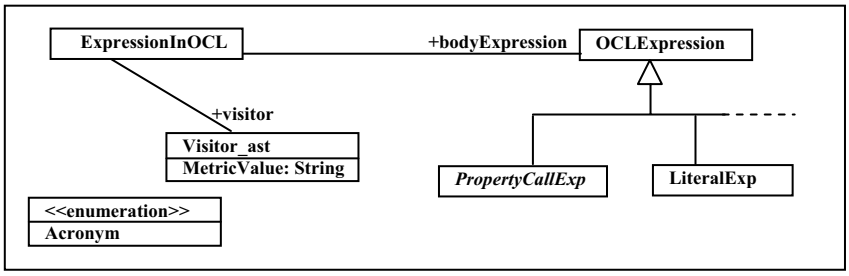


Fig. 6. Design of the implemented strategy

5.1 Implementing Accept Operations in the Expression Package Classes

Several Accept operations were implemented in the OCL metaclasses of the *Expression* Package. Their definitions include many forward operations, so, it is important to

understand the OCL metaclasses and their relationships. We will show as an example the *Accept* operations of *AttributeCallExp* and *OperationCallExp* metaclasses.

- *AttributeCallExp* subclass defines *Accept* calling the *Visitor* operation that corresponds to the class, and it calls the *Visitor* operation on its source whether the source is not empty.
  context AttributeCallExp::accept_new(v:Visitor, metricName: MetricAcronym)
  post: v^visitAttributeCallExp(self, metricName) and
  (self.source->notEmpty() implies
  self.source^accept_new(v, metricName))

- *OperationCallExp* subclass defines *Accept* calling the *Visitor* operation that corresponds to the class, and it implements *Accept* by iterating over its arguments and calling *Accept* on each of them. It also calls *Accept* operation on its source in a similar way.
  context OperationCallExp::accept_new(v:Visitor, metricName: MetricAcronym)
  post: v^visitOperationCallExp(self, metricName) and
  (arguments->size() >= 1 implies
  arguments->forAll(a | a^accept_new(v, metricName)))
  and
  (self.source->notEmpty()
  implies self.source^accept_new(v, metricName))

5.2 A Visitor Class for Obtaining the Value of OCL measures

Visitor_ast class (see Figure 7) defines the visitor attributes and operations for each class of the OCL metaclasses.

Next, we will exemplify how the visitor operations are defined in the *AttributeCallExp* and *NavigationCallExp* classes. In these visitor operations we also show how the NAN and NNC measures are computed.

NAN, the "Number of the Attributes referred through Navigations" is equal to the quantity of instances of *AttributeCallExp* where the type of its source is an *AssociationEndCallExp* or *AssociationClassCallExp*.

```
context Visitor::visitAttributeCallExp(o: AttributeCallExp,
      metricName: MetricAcronym)
post: (metricName = MetricAcronym::NAN
   and (o.source.oclIsTypeOf(AssociationEndCallExp)
   or o.source.oclIsTypeOf(AssociationClassCallExp) )
   implies importedproperties =
      importedproperties@pre ->including(o.referredAttribute.name)
```

The operation only load the name of an attribute used in a navigation in a set of attributes called *importedproperties*.

For example the *ast* of Figure 5 has only one *AttributeCallExp* instance, having a *referredAttribute* named n*umberofseats*, and this is relevant for NAN, so after the Visitor visits the *ast* for that expression, the *importedproperties* set will be equal to the set {*numberofseats}*.

ClassName: Visitor_ast
Attributes: - valueMetric: Integer; - importedproperties: Set(String); - navigatedClasses: Set(String); Methods: visitOperationCallExp(o:OperationCallExp, metricName: MetricAcronym) visitNavigationCallExp(o: NavigationCallExp, metricName: MetricAcronym) visitAttributeCallExp(o: AttributeCallExp, metricName: MetricAcronym) visitLetExp(o: LetExp, metricName: MetricAcronym) visitIfExp(o: IfExp, metricName: MetricAcronym) visitLoopExp(o: LoopExp, metricName: MetricAcronym) visitOclMessageExp (o: OclMessageExp, metricName: MetricAcronym) visitCollectionRange(o: CollectionRange, metricName: MetricAcronym) visit CollectionItem(o: CollectionItem, metricName: MetricAcronym) visitTupleLiteralPart (o: TupleLiteralPart, metricName: MetricAcronym) visitLiteralExp(o: LiteralExp, metricName: MetricAcronym) ……

Fig. 7. The Visitor Class

Whenever a Visitor accesses a *NavigationCallExp* object, it loads in a set (called navigatedClasses) the name of the classes used in navigations (whether the modeler use a navigation class) or the name of the class of the *AssociationEndCall* type (i.e. the name of the class to which the rolename references). The size of this set is used to obtain the NNC value in a similar way as we did with *importedproperties*.

```
context Visitor::visitNavigationCallExp(o: NavigationCallExp, metricName: Met-
ricAcronym)
    post:
    metricName = MetricAcronym::NNC
    implies navigatedClasses = navigatedClasses@pre->including(
    if self.oclIsTypeOf(AssociationEndCallExp)
      then
            source.oclAsType(AssociationEndCallExp).referredAssociationEnd.type.
            name
     else
             source.oclAsType(AssociationClassCallExp).referredAssociation
            Class.name endif)
```

For example, according to Figure 5, there are three *AssociationEndCallExp* instances, so the name of the classes to which the *passenger*, *plane* and *planetype role-*

names references are collected in the *navigatedClasses* attribute. So, after the visitor visits the *ast* for the OCL expression the set of *navigatedClasses* will be equal to the set {*Passenger*, *Airplane*, *Airplane_Type*}.

5.3 Explanation of How the Value of NAN and NNC Measures are Obtained

Within the *ExpressionInOCL* class many operations for obtaining the value of different measures are defined, such as the following two for obtaining the value of NAN and NNC measures:

- In order to compute the value of NAN the following operation is defined:

```
context ExpressionInOCL::value_of_NAN() : Integer
post:  self.visitor.oclIsNew()
  and self.bodyExpression^accept_new(self.visitor, MetricAcronym::NAN)
  and result = self.visitor.importedproperties->size()
```

- The size of the navigatedClasses set determines the value of NNC.

```
context ExpressionInOCL::value_of_NNC() : Integer
post:  self.visitor.oclIsNew()
    and self.bodyExpression^accept_new(self.visitor, MetricAcronym::NNC)
    and result = self.visitor.navigatedClasses->size()
```

These operations requests the creation of a new Visitor object, then send it to the root of an *ast* in order to compute the value of a measure which is specified as a parameter. In turn, each node of the *ast* forwards the Visitor allowing it to act.

6 Conclusions

Within the OO software measurement community the formal definition of measures is an important aspect that has been almost neglected. Although there is a huge amount of OO measures, the lack of formalization constitutes a serious matter. Only natural language or rigorous mathematical definitions were used, being none of them suitable and widely adopted. Our belief is that the combination of metamodeling facilities and OCL as a language for defining OCL semantics, such as in defining the UML and OCL languages, allows also unambiguous measure definition achieving both understandability and formality in their specification. We claim that the formal definition of our measures using OCL language is easy to grasp by anybody familiar with metamodeling.

The relevance of the proposed approach, the specification of measures using OCL at M2 of MOF, will become more important by the proliferation of MOF-compliant architectures and the growing field of Model Driven Engineering [17], [1], [23] paradigm. In truly model-driven software engineering the quality of the models used is of great importance as it will ultimately determine the quality of the software systems produced. In particular, it is widely believed that the system quality is highly dependent on many decisions made early in its development, specifically when artifacts and

its constraints are defined. The formal specification of measures allows the development of precise measures extraction tools for identifying the weak points of UML models and giving on the fly diagnostics about the model quality. We also believe that in the future, UML and OCL measures extraction can be translated from their formal definition to platform specific models (PSM) and from PSM to code.

Acknowledgements

This research is part of the MECENAS project (PBI06-0024) financed by "Consejería de Ciencia y Tecnología de la Junta de Comunidades de Castilla-La Mancha", the ESFINGE project (TIN2006-15175-C05-05), supported by the "Ministerio de Educación y Ciencia (Spain)" and the COMPETISOFT project (506PI0287) financed by "CYTED (Programa Iberoamericano de Ciencia y Tecnología para el Desarrollo)".

References

[1]. C. Atkinson and T. Kuhne. Model Driven Development: A Metamodeling Foundation. IEEE Transactions on Software Engineering, 20(5), 36-41, 2003.

[2] A. L. Baroni. Formal Definition of Object-Oriented Design Metrics. Master of Science in Computer Science Thesis, Vrije Universiteit Brussel, Belgium, 2002.

[3] A. L. Baroni, S. Braz, and F. Brito e Abreu. Using OCL to Formalize Object-Oriented Design Metrics Definitions. In Proc. of QAOOSE'2002, Malaga, Spain, 2002.

[4] A. L. Baroni and F. Brito e Abreu. Formalizing Object-Oriented Design Metrics upon the UML Meta-Model. In Proc. of the Brazilian Symposium on Software Engineering, Gramado - RS, Brazil, 2002.

[5] A. L. Baroni and F. Brito e Abreu. A Formal Library for Aiding Metrics Extraction. International Workshop on Object-Oriented Re-Engineering at ECOOP 2003. Darmstadt, Germany, 2003.

[6] L. C. Briand, L. C. Bunse and J. W.Daly. A Controlled Experiment for Evaluating Quality Guidelines on the Maintainability of Object-Oriented Designs. IEEE Transactions on Software Engineering., 27(6), 513-530, 2001.

[7] L. C. Briand, J. Wüst and H. Lounis. A Comprehensive Investigation of Quality Factors in Object-Oriented Designs: an Industrial Case Study. 21st International Conference on Software Engineering, Los Angeles, 345-354, 1999.

[8] S. Chidamber and C. Kemerer. A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, 20(6), 476-493, 1994.

[9] S. Cook, A. Kleepe, R. Mitchell, B. Rumpe, J.Warmer, and A. Wills. The Amsterdam Manifiesto on OCL. Clark T. and Warmer J., editors, Advances in Object Modelling with the OCL, 115-149, 2001.

[10] Eclipse Foundation, Inc. Ottawa, Ontario, Canada.Available at http://www.eclipse.org.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[12] B. Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, 1996.

[13] ISO/IEC 9126-1.2. Information technology- Software product quality – Part 1: Quality model. 2001.

[14] Klasse Objecten. OCTUPUS: OCL Tool for Precise UML Specification. Available at http://www.klasse.nl/octopus/index.html.
[15] W. Liang. UMLObject Constraint Language in Meta-Modeling. School of Computer Science. McGill University, 2002.
[16] M. Lorenz and J. Kidd. Object-Oriented Software Metrics: A Practical Guide. Prentice Hall, Englewood Cliffs, NJ, EUA. 1994.
[17] Object Management Group. MDA - The OMG Model Driven Architecture. 2002.
[18] Object Management Group. Response to the UML 2.0 OCL RfP (ad/2000-09-03), revised Submission, Version 1.6 (ad/2003-01-07). OMG document ad/2003-01-07, 2003.
[19] Object Management Group. Unified Modeling Language, V1.4, Document formal/01-09-67, OMG document formal/01-09-67, 2001.
[20] L. Reynoso, M. Genero and M. Piattini: Measuring OCL Expressions: An approach based on Cognitive Techniques. Chapter 7: Metrics for Software Conceptual Models. M. Genero, M. Piattini, and M. Calero (Eds.). Imperial College Press, UK. 161-206. 2005.
[21] L. Reynoso and R. Moore. GoF Behavioural Patterns: A Formal Specification. Technical Report Nro. 200. UNU/IIST, P.O.Box 3058, Macau, 2000.
[22] M. Richters. A Precise Approach to Validating UML Models and OCL Constraints. Monographs of the Bremen Institute of Safe Systems, 2001.
[23] B. Selic. The Pragmatics of Model-Driven Development. IEEE Software, 20(5), 19-25, 2003.
[24] J. Warmer and A. Kleppe. The Object Constraint Language. Precise Modeling with UML. Object Technology Series.Addison Wesley, 1999.
[25] J. Warmer and A. Kleppe. The Object Constraint Language. Second Edition. Getting Your Models Ready for MDA. Object Technology Series. Addison-Wesley. Massachusetts., 2003.