

Blekinge Institute of Technology
Research Report 2003:06

**<<UML>> 2003
Modeling Languages and Applications**

**Workshop on
"Consistency Problems in UML-based
Software Development II"**

**October 20, 2003
San Francisco, USA**

Workshop Materials

Editors:

Ludwik Kuzniarz
Zbigniew Huzar
Gianna Reggio
Jean Louis Sourrouille
Miroslaw Staron

Workshop organizers

Zbigniew Huzar

Department of Computer Science, Wroclaw University of Technology, Wroclaw, Poland

Ludwik Kuzniarz

Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Ronneby, Sweden

Gianna Reggio

Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Italy

Jean Louis Sourrouille

Department of Computer Science, INSA, Lyon, France

Mirosław Staron

Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Ronneby, Sweden

Program committee

Reiko Heckel

Universität Paderborn, Germany

Zbigniew Huzar

Department of Computer Science, Wroclaw University of Technology, Wroclaw, Poland

Mieczysław Kokar

Northeastern University Boston, USA

Kai Koskimies

Tampere University of Technology, Finland

Ludwik Kuzniarz

Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Ronneby, Sweden

Gianna Reggio

Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, Italy

Jean Louis Sourrouille

Department of Computer Science, INSA, Lyon, France

Table of Contents

Hassan Gomaa and Duminda Wijsekera Consistency in Multiple-View UML Models: A Case Study	1
Ludwik Kuzniarz and Mirosław Staron Inconsistencies in Student Designs	9
Thomas Huining Feng and Hans Vangheluwe Case Study: Consistency Problems in a UML Model of a Chat Room	18
C. Lange, M.R.V. Chaudron, J. Muskens, L.J. Somers and H.M. Dortmans An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs	26
Wuwei Shen, Yan Lu and Weng Liong Low Extending the UML Metamodel to Support Software Refinement	35
Jean Louis Sourrouille and Guy Caplat A Pragmatic View on Consistency Checking of UML Models	43
Bogumila Hnatkowska, Zbigniew Huzar, Ludwik Kuzniarz and Lech Tuzinkiewicz Refinement relationship between collaborations	51
Gonzalo Genova, Juan Llorens and Jose M. Fuentes The Baseless Links Problem	58
Egidio Astesiano and Gianna Reggio An Algebraic Proposal for Handling UML Consistency	62
Ragnhild Van Der Straeten, Tom Mens and Jocelyn Simmonds Maintaining Consistency between UML Models Using Description Logic	71
Robert Wagner, Holger Giese and Ulrich A. Nickel A Plug-In for Flexible and Incremental Consistency Management	78

Consistency in Multiple-View UML Models: A Case Study

Hassan Gomaa and Duminda Wijesekera

Department of Information and Software Engineering, MS 4A4,
George Mason University, 4400 University Drive, Fairfax, VA 22030.

E-mail: {hgomaa,dwijesek}@gmu.edu

Abstract

Software designs described using UML consist of multiple views describing the static and dynamic views of the system being modeled. The dynamic modeling perspective is depicted by use cases during requirement modeling and by class diagrams, statecharts and interaction (sequence and collaboration) diagrams during analysis and design. For a given software design, these multiple views must be consistent with each other. This paper describes how consistency checking between multiple views of a software design can be carried out using Object Constraint Language (OCL) constraints, and is illustrated by a detailed case study

1. Introduction

Many software specification and design methods advocate a modeling approach in which, the system under development is represented by means of multiple views. Each view, which may be either graphical or textual, presents a different perspective on the system being developed. The actual views and the way in which a system's requirements are projected onto individual views are method specific. With a multiple view approach, there are rules for mapping from the representation of an artifact in one view to a different representation of the same artifact in one or more related views. Typical problems that occur with multiple view specification and design methods are inconsistency, incompleteness, and ambiguity.

The UML notation supports multiple views. However, in general, the notation does not address the issue of mapping among the multiple views and consistency checking between multiple views. Some researchers have attempted to provide rules for mapping between different views of a UML model [7, 4]. Traditionally formal methods have been successfully used to detect inconsistency and

incompleteness in specifications. This paper describes an approach for identifying and correcting inconsistency and incompleteness across UML views, in particular use case diagrams, class diagrams, sequence diagrams, and Statecharts. The rules for describing the mapping between these different views are derived from the COMET method [9], which uses the UML notation, and are expressed as Object Constraint Language (OCL) constraints [12] enhanced with action clauses [13]. The approach is illustrated by means of a detailed case study.

2. Related Work

- **RESEARCH in UML META-MODELS:** Most meta-model level work have been done by two different organizations, the pUML group [1] and the inventors of OCL [14]. Their main trust has been to enhance the meta-model of UML so that any instance of an object from the enhanced meta-model (i.e. a modeling element) would have all other associated elements navigable from it. Meta-modeling provides a theoretical foundation for view integration in the form of providing templates, but do not address details.
- **RESEARCH at USC:** Researchers at the University of Southern [3, 4] have viewed view unification as an activity with three stages. (1) Differentiation, where mismatches are identified (2) Mapping, where overlapping or redundant elements are identified (3) Transformation that extracts and converts elements from one view to another. They have proposed a large number of rules based for each step, and has resulted in two software prototypes; the *Rose Architect* [4] and the *UML Analyzer* [16] to partially automate the process. While there are many important and key contributions made by the research effort at USC, there is no methodology or advice as to how to use them.

- **RESEARCH at IMPERIAL COLLEGE:** Researchers at Imperial College, London [6,7, 8] developed the Viewpoints framework [17]. This work has been extended recently to include inconsistent Statecharts [8]. Their work formulates the view consistency problem in terms of classical temporal logic with the closed world assumption, and toolkit support for Viewpoints. One striking difference of this work is their view that inconsistency is not necessarily a *bad* thing, and should be evaluated in light of the costs of correcting them. At the logical level, they formulate each view as an instance of a database. Consequently, each viewpoint becomes a collection of relation symbols. Constraints within a view are encoded as statements in first order classical logic. Similarly, inter-view consistency is encoded as first classical logic statements using predicates from two (or many) views involved.
- **RESEARCH at MICHIGAN STATE UNIVERSITY:** Researchers at Michigan State concentrate on View Unification in UML by providing a more formal basis to UML diagrams [2,3]. They translate UML diagrams to algebraic specifications and VHDL. Their work has not developed any specifics of how exactly view unification can be done in UML translated to these more formal languages. Their recent work has been in formalizing a UML meta-model in Promela, so that its dynamic aspects can be fed in to SPIN [15].
- **RESEARCH in FINLAND:** Two groups of researchers working on view unification in UML diagrams have developed algorithms to produce sequence diagrams to collaboration diagrams, and methods to encode sequence diagrams as statecharts. Furthermore, they have developed a prototype tool for the former [11].
- **RESEARCH in GERMANY:** View integration work at the Technical University of Berlin and at the University of Bremen addresses unifying some aspects of class, sequence and statechart diagrams in the analysis phase [18]. This work is based on a collection of *lifelines*; i.e. a sequence diagram consisting of all classes and the actor involved in interactions. Their consistency resolution is given by existence, (do the message sender and receiver in the sequence diagram have an association in the class diagram?) visibility (are the visibility parameters consistent in the send and receive messages), and multiplicity checking (are the multiplicities of sender-receiver associations correct) conditions. They check consistency by

decorating sequence diagrams with pre and post conditions of message communication methods, translating them to attribute graph grammars and checking their consistency.

3. UML and OCL

With the proliferation of notations and methods for the object-oriented analysis and design of software systems, the Unified Modeling Language (UML) has emerged to provide a standardized notation for describing object-oriented models. To be used effectively, UML must be used with a design method, such as COMET, or more generally as part of a software development process, e.g., Unified Software Development Process [10].

COMET [9] is a Concurrent Object Modeling and Architectural Design Method for the development of concurrent applications—in particular, distributed and real-time applications. Use cases are developed during requirements modeling. Object structuring criteria are used to determine the objects needed to execute each use case. The objects and their interactions are then depicted on interaction (sequence or collaboration) diagrams. COMET makes extensive use of the UML stereotype construct, which is used to distinguish among the different kinds of application classes. Thus, an application class is classified as an «entity» class, which is a persistent class that stores data, an «interface» class, which interface to the external environment, a «control» class, which provides the overall coordination for the objects that participate in a use case, or a «application logic» class, which encapsulates business logic or algorithms separately from the data being manipulated. By providing this classification of application classes, certain properties can be assigned to a specific category, for example only state dependent control classes execute Statecharts. These properties allow for more extensive consistency checking between multiple views.

4. Consistency Checking Rules for Multiple UML Views

The different views of a UML based software design need to be consistent with each other. This paper describes consistency-checking rules among the behavioral views. Example rules are:

A. Consistency Checking Rules between Use Cases and Interaction (Sequence or Collaboration) Diagrams.

1. A use case must correspond to at least one scenario described by an interaction diagram.

B. Consistency Checking Rules between Class Diagrams and Statecharts:

1. Each Statechart must correspond to a state dependent control class on a class diagram.
2. The class specification for a state dependent control class on a class diagram must have attributes for state, action, and activity, corresponding to the properties of the Statechart. The values of the actual states, events, actions, and activities that appear on a Statechart must be declared as attribute values of the respective state, event, action, and activity attributes for the state dependent control class.
3. An event on a Statechart corresponds to a method of the state control class on the class diagram.
4. Variables used to define conditions in any Statechart must be attributes of the state dependent control class on the corresponding class diagram.

C. Consistency Checking Rules between Interaction Diagrams and Statecharts:

1. Each event on a Statechart must correspond to an incoming message on the state dependent control object, depicted on an interaction diagram, which executes the Statechart.
2. Each action on a Statechart must correspond to an outgoing message on the state dependent control object, depicted on an interaction diagram, which executes the Statechart.

We assume that names of attributes used to enforce view consistency are pre-pended with a \$ sign. E.g., \$state, \$event, \$condition, \$action and \$activity. We show them with respect to a case study.

5. The Case Study

Figure 1 shows two railroad tracks with a gate, where trains can travel from left to right and on the upper lower track from right to left on the lower track. There is a highway crossing the railroad, and a gate with an arm that moves up and down prevents cars from running into trains by closing the highway when a train is in the vicinity. The railroad has sensors attached to the track at regular intervals, and when the train passes over them, distance data is passed to the train, from which the location of the train can be computed. Upon entering a region close to the gate identified as NEAR in Figure 1, the gate needs to be closed. When the train leaves the NEAR region, the gate needs to be opened

provided that there are no other trains in the NEAR region. The total length of the track is 5000 meters, and the gate is 2,500 meters from either end, and the NEAR region spans from 1000 meters before the gate to 100 meters after the gate measured in the direction of travel. The train speed is between 100 to 150 MPH outside the NEAR region, and 80 to 100 MPH inside it.

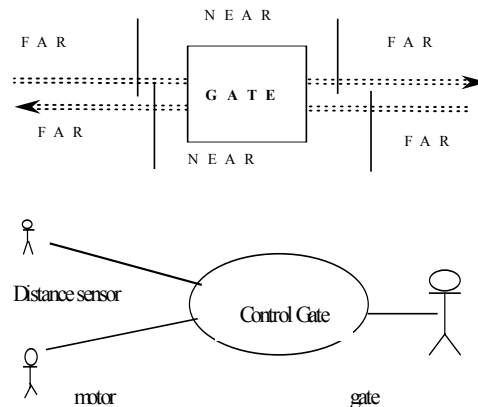


Figure 1: Two Rail Tracks with a Gate and its Use Case

5.1. Use Cases

Informally, the use case (Fig. 1) is as follows. When a train approaches the NEAR region, it must slow down and the gate must be closed. When a train leaves the NEAR region, it must speed up and request the gate to be opened, and the gate should open only if there are no trains in the NEAR region. When there are no trains within the NEAR region, the gate must remain opened.

Use Case Name: Control Gate

Summary: A train crosses a gated motorway, causing the gate to be closed when the train approaches the gate and opened when there are no trains in the vicinity of the gate.

Actors: Distance sensor, motor, gate

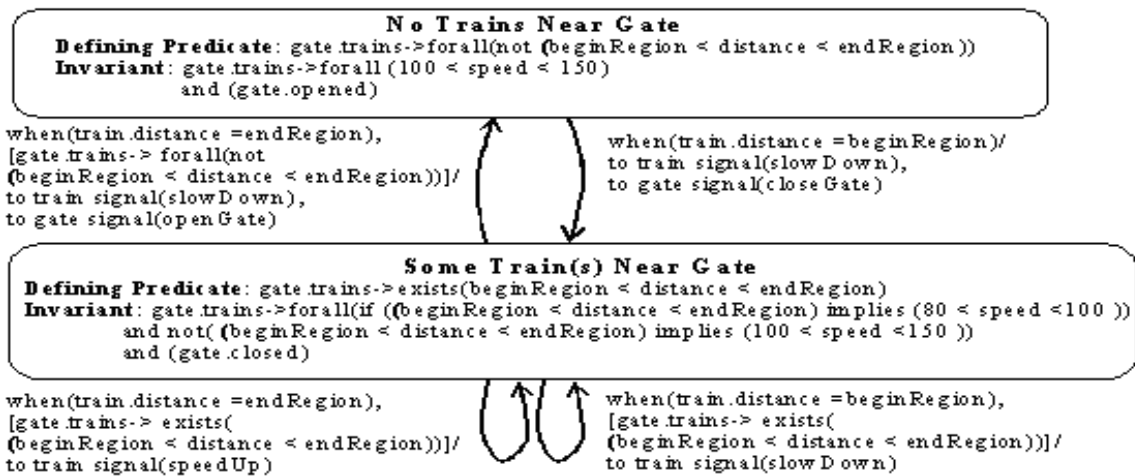


Figure 2: Statechart Description of the Use Case

Precondition: The train has traveled a distance of less than 1500 meters from start, and is traveling at a speed between 100 to 150 MPH.

Description:

The *distance sensor* sends the distance traveled by the *train* to the system.

When the distance traveled is 1500 meters, the system commands the *train* motor to slow down to a speed between 80 and 100 MPH, and requests the *gate* to be closed.

If the *gate* is not already closed, it will begin closing.

When any *train* reaches a distance of 2500 meters, the system commands it to speed up to between 100 MPH and 150 MPH and request that the *gate* be opened.

If there are no *trains* between 1500 meters and 2500 meters within the *gate*, the *gate* should be opened.

Alternatives: If a train requests an already closed gate to be opened, then the gate remains closed.

If a train requests the gate to be closed while it is being opened, then the opening procedure is halted and the gate is closed.

Post condition: The train has passed the gate and started accelerating to between 100 and 150 MPH.

This informal use case description can be specified using the Statechart given in Figure 2.

5.2. The Static Model

Figure 3 shows software classes of the design. There is one input device interface classes, which interfaces to the distance sensor, and two output device interface classes, which interface to the gate and motor actuators respectively. There are three state dependent control classes, each of which is specified by a statechart. There is one algorithm class, which encapsulates the distance computation algorithm. The Distance Sensor Interface class on the train senses the distance from the beginning of the trip, and feeds it to the Distance Computation Algorithm class of the train, which computes the distance traveled by the train. Based on its needs, the Train Controller can request the Motor Interface to speed up or slow down. The Train Controller also signals the Zone Controller that it is approaching the danger zone. The Zone Controller in turn commands the Gate Controller to open or close the gate. The Zone Controller tracks trains entering and leaving the danger zone while the Gate Controller controls the opening and closing of the gate by commanding the Gate Interface to lower or raise the gate. OCL can be used to formally specify the associations between the classes in the static model. Some examples are as follows:

- Context:** Distance Sensor Interface
Inv: self.sendDistanceTo->size = 1
- Context:** Zone Controller
Inv: self.controls->size = 1
- Context:** Train Controller
Inv: self.notifies->size = 1

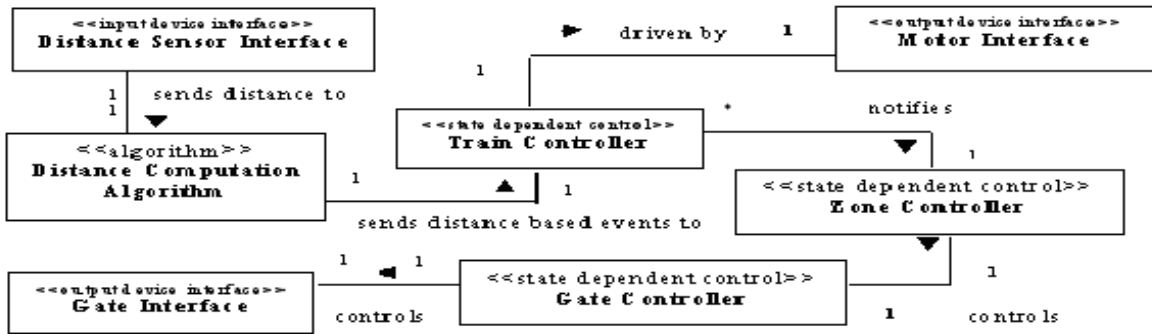


Figure 3: Class Diagram for Train Crossing

5.2. The Dynamic Model

We model object interactions with two sequence diagrams, namely for trains approaching and leaving the gate. We give the first in Figure 4. As Figure 4 shows, the *distance sensor* provides the distance traveled by the train to the *distance computation algorithm*, which in turn determines that the train is

approaching the danger zone and sends the approach signal to the *train controller*. Consequently, the *train controller* signals the *motor interface* to slow down and the *zone controller* to close the gate. The *zone controller* in response signals the *gate controller* to lower the gate. The *gate controller* requests the *gate interface* to close the gate. When the gate is closed the *gate interface* informs the *gate controller* of this occurrence.

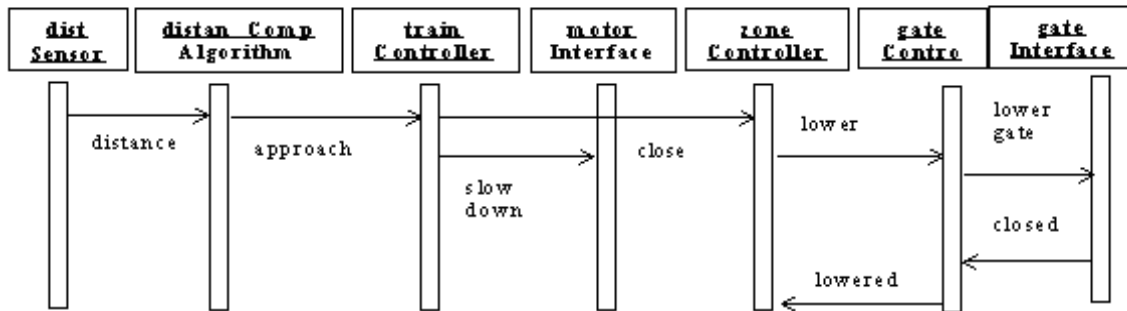


Figure 4: The First Scenario - A Train Approaching the Gate

Figure 5 shows the scenario of the train leaving the danger zone. In this scenario, the *distance Computation Algorithm* determines that the train is leaving the danger zone and sends the leave signal to the *train Controller*. Consequently, the *train Controller* signals the *motor Interface* to speed up and the *zone Controller* to open the gate. Assuming there are no other trains in the vicinity, the *zone Controller* signals the *gate Controller* to raise the gate. The *gate Controller* in response signals the *gate Interface*. When the gate is opened the *gate Interface* informs the *gate Controller* of this event. A set of OCL constraints similar to those in the “Train Approaching “ can be for

the “Train Leaving” sequence diagram. These can be specified using OCL. For example, the first can be specified as follows.

Context: distanceSensorInterface
action: if (mileStone) to
distanceComputationAlgorithm send signal(distance)

The constraints given at this stage are more detailed because there are more objects. For example, the object *system* in the use case model have now been replaced by the objects in the design level classes, consisting of the *distance Sensor Interface*, *distance*

Computation Algorithm, train Controller, zone Controller, gate Controller, motor Interface, and gate Interface. In addition, given the objects that implement the objects in the sequence diagram, we can refine those components of the *global* Statechart (that

captures all the scenarios of the application (given in Figure 3) to form the *local* Statecharts of the objects. The sequence diagram given in Figure 4 ties all incoming and outgoing events in all Statecharts together.

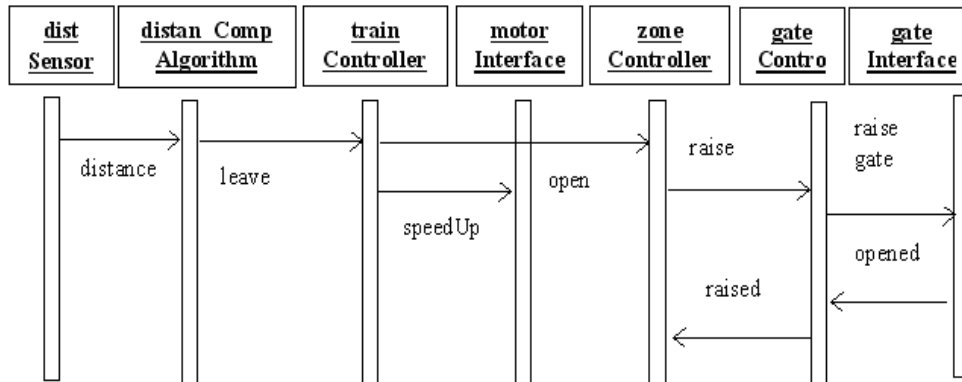


Figure 5: The Second Scenario - A Train Leaving the Gate

The constraints given at this stage are more detailed because there are more objects. For example, the object *system* in the use case model have now been replaced by the objects in the design level classes, consisting of the *distance Sensor Interface, distance Computation Algorithm, train Controller, zone Controller, gate Controller, motor Interface, and gate Interface.* In addition, given the objects that implement the objects in the sequence diagram, we can refine those components of the *global* Statechart (that captures all the scenarios of the application (given in Figure 3) to form the *local* Statecharts of the objects. The sequence diagram given in Figure 4 ties all incoming and outgoing events in all Statecharts together.

5.4. Applying Consistency Rules

Consistency rules are now described for two of the state dependent control classes that execute statecharts, the *Train Controller* and *Zone Controller* classes. Figure 6 shows the class diagram and statechart for the *Train Controller* class. The class diagram has two methods corresponding to events, *approach* and *leave*. It also has the state names *far* and *near*, and the actions *slowDown, speedUp, open* and *close*. The *Train Controller* class is specified by a statechart that has two states, FAR and NEAR. Consequently, the train controller class has a \$state attribute of enumerated type {far, near}. The state transition from FAR to NEAR take place on receipt of the *approach* event and

causes actions *slowDown* and *close*. Conversely, the NEAR to FAR transition is fired in response to a *leave* event and cause actions *speedUp* and *open* to be executed. Accordingly, the Train Controller has an \$action attribute of enumerated type {speedUp, slowDown, open, close}.

The two sequence diagram scenarios for the Control Gate use case address rule A1. The specification of the *Train Controller* class by the statechart addresses rule B1 above. The definition of the \$state and \$action attributes address rule B2. The correspondence between the *leave* and *approach* methods in the class and the events of the same name on the Statechart addresses rule B3. There are no conditions in the statechart so rule B4 does not apply in this case. The correspondence between the incoming message to the control object and the event on the statechart addresses rule D1. The correspondence between the action on the statechart and outgoing message from the control object addresses rule D2. The object *train Controller* appears in both sequence diagrams and is also an instance of the *Train Controller* class on the class diagram. This addresses Rules C1. The arrival of the *approach* and *leave* messages at the train Controller object on the sequence diagram correspond to the invocation of the methods with the same name depicted on the class diagram, corresponding to Rule C2. Consequently, the following OCL constraints are imposed on the *Train Controller* class.

context: Train Controller

Inv: (\$state = NEAR) or (\$state = FAR)
context: Train Controller::approach()
pre: \$state = FAR
post: \$state = NEAR
action: to zoneController send close()
action: to motorInterface send slowDown()
action: to self send changeState(NEAR)
context: Train Controller::leave()
pre: \$state = NEAR

post: \$state = FAR
action: to zoneController send open()
action: to motorInterface send speedUp()
action: to self send changeState(FAR)
context: Train Controller::changeState()
pre: (\$state = NEAR) XOR ((\$state = FAR))
post: ((\$state@pre = FAR) implies (\$state = NEAR))
 and ((\$state@pre = NEAR) implies (\$state = FAR))

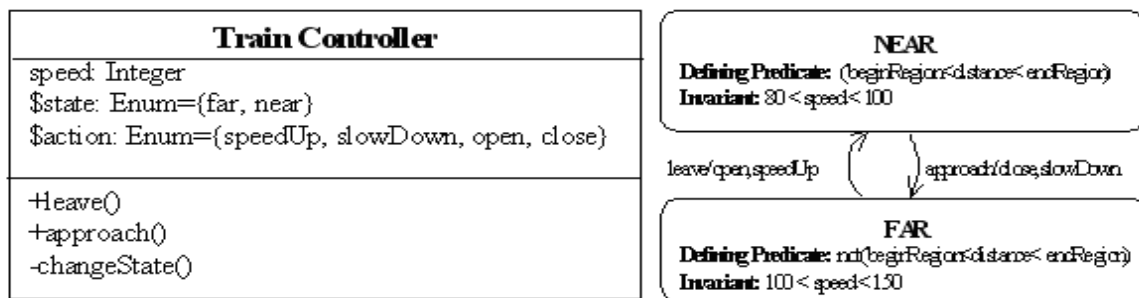


Figure 6: Train Controller Design

The *Zone Controller*, shown in Figure 7 is needed to ensure that the gates open only when there are no trains in the crossing zone. It has three states, *idle*, *processingApproach*, and *processingExit*, processes *open* and *close* events, and issues *lower* and *raise*

actions to the *gate controller*. The *gate controller* sends *lowered* and *raised* events to the *zone controller* once these tasks are completed. The *zone controller* class has an integer attribute *count* to store the number of trains in the NEAR region, as there could be a maximum of two.

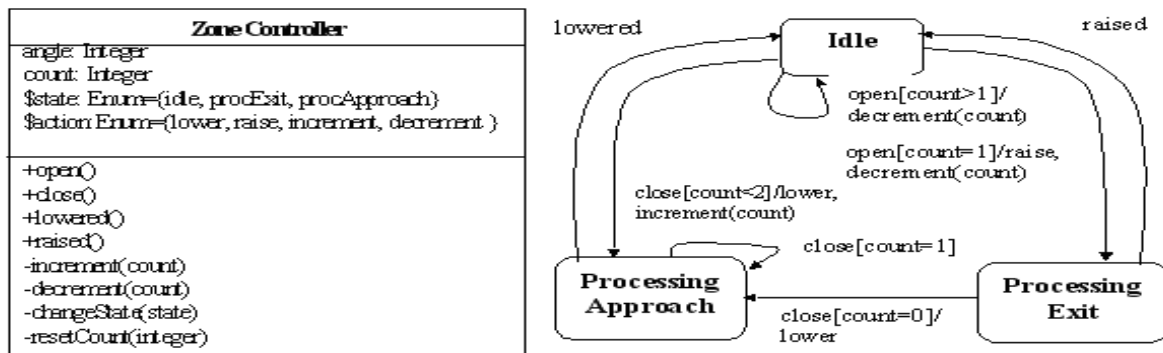


Figure 7: Zone Controller Design

The incoming events to the Statechart, *open*, *close*, *raised*, and *lowered*, correspond to the methods of the class, thereby obeying rule B3. The Statechart has conditions [count < 2], [count > 1], [count = 1], and [count = 0], that are definable from the attribute *count* in the *zone controller* class diagram, satisfying rule B4. Rule C1 is satisfied as *zone controller* object that

appears on the sequence diagrams is an instance of the *zone controller* class on the class diagram. The *open* and *close* messages from the *train controller* to the *zone controller* correspond to the *open* and *close* methods on the *zone controller* class. The *lowered* and *raised* messages going from the *gate controller* object to the *zone controller* object correspond to the *lowered* and *raised* methods in the *zone controller* class. Thus,

rule C2 is obeyed. Following example OCL constraints are implied as consequence of applying these rules.

Context: Zone Controller

Inv: (\$state = idle) or (\$state = procExit) or (\$state = procApproach) and

(\$state = idle) implies (0 <= count <3) and

(\$state = procExit) implies (count = 0) and

(\$state = procApproach) implies (0<count<3)

Context: Zone Controller::open()

Pre: (\$state = idle) and (count >= 1)

Post: (count > 1 implies \$state = Idle) and (count = 1 implies \$state = procExit)

Action: to self send decrement (count)

if (count = 1) to self send changeState (procExit)

if (count = 1) to send gateController raise()

6. Conclusions

This paper has described an approach for identifying and correcting inconsistency and incompleteness across UML views, in particular use case diagrams, class diagrams, sequence diagrams, and Statecharts. The rules for describing the mapping between these different views are derived from the COMET method, which uses the UML notation, and are expressed as Object Constraint Language (OCL) constraints enhanced with action clauses. The approach has been illustrated by means of a case study. During this research, we realized the need for some enhancements for the OCL syntax such as in the action clause proposed by [13]. The issue is that an action clause with *if <condition> to <recipient> send <operation>* is to be fired when the Boolean condition *<condition>* becomes true first. This is problematic when a frequent activity needs to be specified at a higher level without resorting to specifying details. We also noticed that the translation of information captured in Statecharts and sequence diagrams to OCL can be done in many ways, and thus calls for some methodological discipline. Secondly, as shown in examples of [13], we encountered some situations where post-conditions may need to include an action clause.

References

[1] T. Clark, A. Evans, S. Kent, S. Brodsky and S. Cook "A Feasibility Study in Rearchitecting UML as a Family of Languages using a Precise OO Meta-Modeling Approach". At www.puml.org. September 2000.
[2] B. H. C. Cheng, E. Y. Bourdeau, R. H. Richter. Bridging the Gap Between Informal and Formal Approaches to

Software Development Proc of Software Engineering Research Forum, 1995.

[3] B. H. C. Cheng, E. Y. Wang and H. Richeter, Formalizing and Integrating the Dynamic Model within OMT, IEEE Proc. of International Conference on Software Engineering, Boston, MA, May 1997.

[4] A. Egyed. "Automating Architectural View Integration in UML". Available at ESEC/FSE'99, <http://sunset.usc.edu/TechRpts/Papers/uscscse99-514.pdf>, and 1999.

[5] A. Egyed, P. B.Kruchten. "Rose/Architect: a tool to visualize architecture", *Proceedings of the 32nd Annual Hawaii International Conference on Systems Science*, 1999.

[6] S. Easterbrook and B. Nuseibeh. "Using Viewpoints for Inconsistency Management". *BCS/IEE Software Engineering Journal*, January 1996.

[7] A.Finkelsetin, J.Kramer, B.Nuseibeh and M.Goedicke. "Viewpoints: A Framework for Integration Multiple Perspectives in System Development". *International Journal of Software Engineering and Knowledge Engineering*, 2(1): 31-58, World Scientific Publishing Co., March 1992.

[8] D. Gabbay and A. Hunter. "Making inconsistency respectable I: A Logical framework for inconsistency in reasoning". *Proc.Fundamentals of Artificial Intelligence Research*, 1991.

[9] H. Gomaa, "Designing Concurrent, Distributed, and Real-Time Applications with UML", Addison-Wesley Object Technology Series, ISBN: 0-201-65793-7, August 2000.

[10] I. Jacobson, G. Booch, J. Rumbaugh, "The Unified Software Development Process", Addison Wesley, Reading MA, 1999.

[11] K. Kosikimies, T. Systa, J. Tuomi and T Mannisto, Automated Support for Modeling OO Software, IEEE Software, Jan 1998.

[12] J. Warmer and A. Kleppe, *The Object Constraint Language, Precise Modeling with UML*, Prentice Hall, 1999.

[13] J. Warmer and A. Kleppe, *Extending OCL to Include Actions*, in Proc. UML 2000 Conf., pages 440-450, Springer Verlag, 2000.

[14] A. Kleppe and J. Warmer "Unification of Static and Dynamic Semantics of UML". At www.klasse.nl/english/uml/semantics.html. July 2001.

[15] W. E. McUumber and B. Cheng "A Genral Framework for Formalizing UML with Formal languages" 23rd International Conference on Software Engineering, 2001. pp 433-442.

[16] N. Medvidovic, A. Egyed, D. S.Rosenblum. "Round-Trip Software Engineering using UML: From Architecture to Design and Back", *Proceedings of the Second International workshop on Object-Oriented Reengineering (WOOR '99)*.

[17] B. Nuseibeh, J. Kramer and A. Finkelstein, "A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification ", *IEEE Transactions on Software Engineering*, 20(10): 760-773, 1994.

[18] A. Tsiolakis "Integrating Model Information in UML Sequence Diagrams", *Electronic Notes in Theoretical Computer Science*, June 2001.

Inconsistencies in Student Designs

Ludwik Kuzniarz

Department of Software Engineering and Computer Science,
Blekinge Institute of Technology
Box 520 Soft Center
SE-372 33 Ronneby, Sweden
lku@bth.se

Mirosław Staron

Department of Software Engineering and Computer Science,
Blekinge Institute of Technology
Box 520 Soft Center
SE-372 33 Ronneby, Sweden
mst@bth.se

Abstract

The paper is a contribution to the discussion on understanding of the notion of consistency applied to the UML artifacts. It presents and discusses typical inconsistencies encountered in student designs produced within the projects done within a sample didactic development process used during the course on the introduction to object oriented software development with UML. The presented inconsistencies are supposed to be a basis for further elaboration of consistency rules for a sample didactic software development process.

1. Introduction

There is still no consensus concerning understanding of consistency [1] when the notion is applied to the UML based – the design in which the essential design artifacts are expressed in UML [2]. This applies both to what is considered as inconsistent as well as the formal definition. The paper is supposed to contribute to the discussion by showing some typical situations encountered in a sample student design which could be a basis for further formulation of what is the nature of inconsistency and some rules guiding the design process.

At first an outline of a sample didactic object oriented design process is presented. The process is based on guidelines from Unified Software Development Process (USDP, [3, 4]) customized for an introductory course on object oriented development.

The main part of the paper presents an example student design followed by an examination of the UML artifacts forming the design from the point of view of consistency between them. The examination is based on the set of rules of thumb which later can be used for more precise and formal definition of the meaning of consistency for the specified didactic process. The issues addressed concern only artifacts produced within a single iteration.

2. Definition of the Didactic UML based Process

The didactic process presented in the paper a development process used during a course on software development at Blekinge Institute of Technology. The process is based on USDP and is simplified and tailored for

the constraints of the specific in a similar way as presented in [5].

The process has the following characteristics:

- is iterative – it is arranged as a sequence of iterations driven by a set of usecases,
- is incremental – within each iteration a working version of the system is produced,
- UML-based – all major artefacts are expressed in UML.

2.1. Phases and activities

Development process is arranged as a sequence of iterations. The goal of each iteration is to produce a working version of the system. Within every iteration in the process, there is a sequence of development phases: Requirements Analysis, Analysis, Design and Implementation. And within each phase a number of activities are performed. Similarly to the Unified Process, the workload for each activity varies dependent on the iteration. For visualization of the phases, activities and related artefacts a modified notation from Rational Unified Process is been used [6].

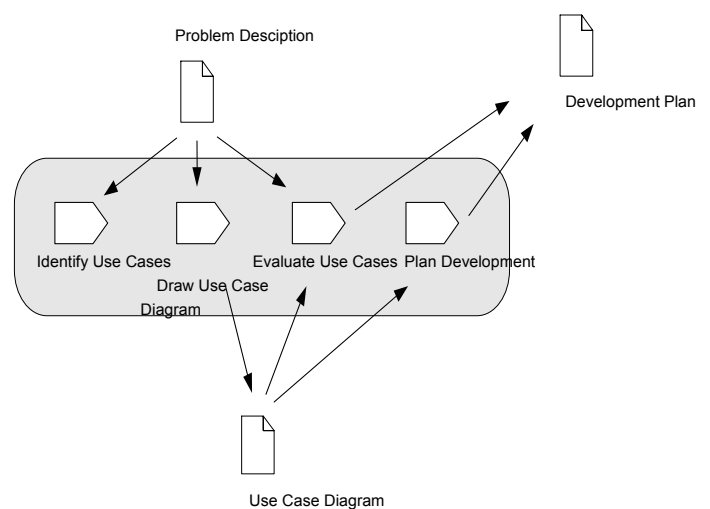


Figure 1 Activities and related artefacts in the Requirement Analysis phase

The first phase – the Requirements Analysis phase – is aimed at the identification and specification of requirements that the newly created system. Its focus is on the identification of functionality of the system based on user demands and understanding what should be done to fulfil the user expectations. Crucial business processes the domain are identified and the role of the system in these processes is defined. The definition is both in the form of UML use case diagram and the textual description of the use cases.

The starting point for the phase is a problem description and the phase ends with description of the requirements of the system. The main activity performed is Identify Use Cases.

Within this activity students examine the requirements specification within the system specification and decide upon the functionality of the system simultaneously grouping it in the use cases. This activity is done in parallel with Draw Use Case Diagram of the system. The activity Evaluate Use Cases along with the activity Plan Development based on use cases lead to creating a plan which use cases should be implemented in which iteration of the process. In the course of evaluation of use cases students have to make decisions on the importance of each use case, justify the decision and then assign each use case to the iteration. There are two main artefacts created during this phase – the Use Case Diagram and the Development Plan.

The main objective of the Analysis phase is to understand and document which elements in the real-world domain take part in the processes identified in the requirements phase and how the newly created software is involved in the processes. The specification of the processes, previously expressed in the use case description is expressed in the form of UML sequence diagrams. System operations, which encapsulate the main functionality of the system from the user perspective, are identified and the contracts for the operations are specified. This phase starts with the activity Elaborate Use Cases. In the activity the students task is to take the use case diagram and system specification as input documents and create a textual description of the use cases of the system which are to be developed in this iteration. The use case description is the input artefact for the Draw System Sequence diagram activity, during the set of system sequence diagrams is created. At this point of the process the students must also identify system operations, which correspond to the steps in the use case description. Within Performing Noun-verb Analysis and Draw Conceptual Model the students are asked to produce a conceptual model for the problem domain of the built system. Artefacts that are used during performing these activities are the system specification and system sequence diagrams. During these activities the students must also make decisions on the concepts which are relevant to this iteration of the system, based on the

system sequence diagram, development plan and use cases. The analysis phase ends with performing Elaboration of Operation Contracts, during which students express the contracts for system operations based on the requirements of the system. The contracts should be expressed in the context of a conceptual model, using the concepts, relationships and properties which are defined there.

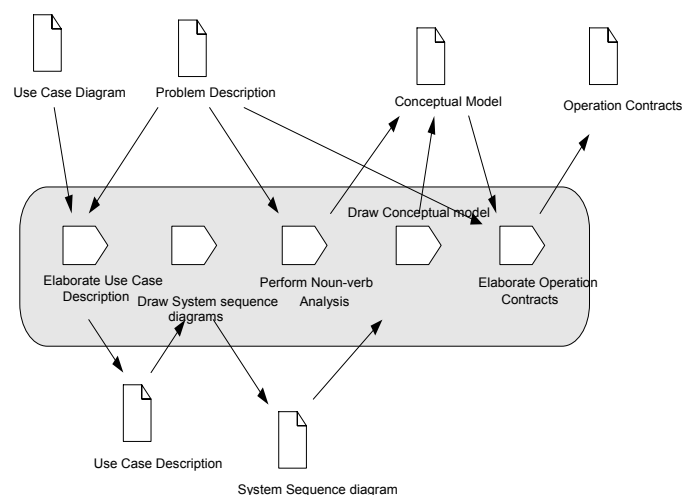


Figure 2 Activities and related artefacts in the analysis phase

In the Design phase, the emphasis is on the construction of a logical solution for the problem specified in the requirements and analysis phases. The students should design a system that works according to the contracts and provides the functionality specified in the use case description. In this phase, the application of the design patterns is stressed and precise modelling (leading to consistent, accurate models).

The design phase starts with the elaboration of the initial version of the design class diagram. One of the possible approaches is to copy the relevant concepts from the conceptual model, add types to properties of the concepts and roles for associations between the concepts. The initial version of the design class diagram is also used in the next activity – specification of the interaction diagrams for system operations. During this activity, a set of guidelines is provided for students on how to perform a good design – GRASP patterns ([7]). After the activity the design class diagram should be updated with the new operations that were required in the interaction diagrams, classes that were added to simplify the design. Afterwards the students should specify the behaviour of the system in terms of state diagrams. Firstly, they have to define a state diagram for the class which is a designated use case controller and secondly, should

specify the state diagrams for other classes in the system, which have the non-trivial behaviour.

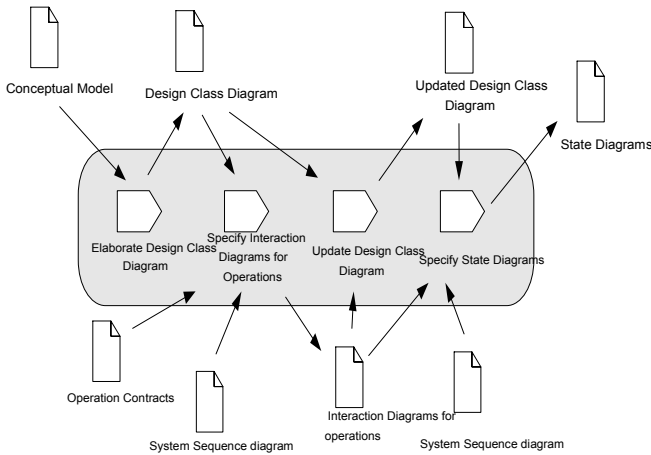


Figure 3 Activities and related artefacts in the design phase

The objective of the Implementation phase to convert the logical solution into a working system – that is translation of the logical design into a source code in a specific programming language.

2.2. Artefacts and relationships between them

Design performed in each iteration consist of a set of artefacts.

The artefacts are summarized below.

- Use case diagram -the diagram provides a graphical representation of the functionality of the system, relationships between functionalities and actors interacting with the system
- Use case description - the description forms a specification of the interaction between the system and the actor that uses the system. Consists of a set of steps (a.k.a. transactions) between the system and the actor(s).
- Domain (Conceptual) Model - the model presents the concepts from the problem domain important from the perspective of the developed system.
- System sequence diagram - one system sequence diagram for each use case specifies the steps identified in the use case description as messages exchanged between the actors and the system (also called system operations).
- Contracts for system operations - a contract for the system operation specifies the pre condition of the operation and the post condition. The contracts serve as a specification of the interface of the system.
- Interaction diagrams for system operations - each interaction diagram is elaborated for one system operation to specify how a set of objects

collaborate to provide a functionality of a given system operation.

- Design class diagram - the diagram(s) shows the static view of the system, classes used to implement the functionality, and is defined in the context of a solution domain (i.e. programming language, software environment).
- State diagram for a design class - the diagram presents the behaviour of a chosen design classes (a class having a non-trivial state machines). A special kind of such diagram is a state diagram for use case controller – a system class, which is designated as a control class for a use case.

These artefacts are produced in a certain order and are mutually dependent and influential. Figure 4 presents the sequencing of the artefacts within a single iteration. Unidirectional arrows indicate the sequence of creation of artefacts; bidirectional arrows indicate that the artefacts should be updated continuously based on each other.

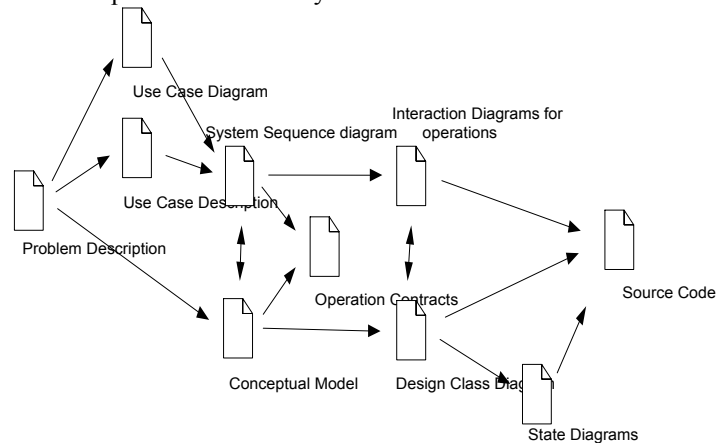


Figure 4 Sequencing between artifacts.

The sequencing of the artefacts is important from the perspective of consistency between artefacts. Each artefact should be consistent with its predecessors and successors.

3. Example Student Design

The design is inconsistent in several places. It is presented here with the short description of the artefacts, while the inconsistencies are presented in the subsequent chapter.

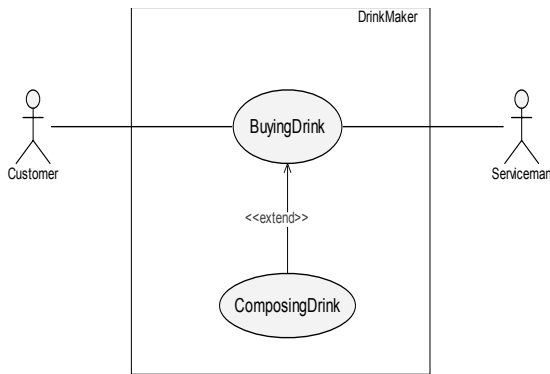
3.1. Problem Description

From the DrinkMaker you can obtain hot drinks, coffee and tea. To buy a drink you have to choose the type of drink (tea or coffee), then the price of the chosen drink is displayed. Then you pay and the appropriate drink is delivered. The chosen drink is delivered on request after inserting money. The DrinkMaker accepts

the coins 1, 5 and 10 SEK. The coins should be inserted to the special coin slot. After inserting a coin a total amount is reported. At any time you can cancel the transaction and obtain the money back. The DrinkMaker is maintained by a service man who is called when the machine runs out of drink ingredients. In the machine there is a hot water tank and two containers, one for the tea and one for the coffee.

Drinks can be standard or composed by the client. If the client decides to compose the drink she/he can choose the ingredients from which the drink is made. For the coffee drink the possible ingredients are sugar, milk, additional coffee powder. For the tea drink the choice is : sugar, lemon and mint. Every ingredient has a price. When the client has made a choice for the ingredient the composed drink is priced and the sum to be paid is display.

3.2. Use Case Diagram



3.3. Development Plan

The development plan consists of the evaluation of use cases for the system and then planning the development.

Use Case	Importance	Justification
BuyingDrink	Most important	The use case provides the main functionality of the system, should be developed first.
Composing Drink	Less important	The use case is the extension of the main use case of the system. It specifies some additional functionality, which is not used in all cases of system usage.

Table 1 Importance of the use cases of the system for the development plan.

Based on the importance of the use cases presented (Table 1) the development process is planned as follows:

- Iteration one: BuyingDrink
- Iteration two: ComposeDrink

It is clear that the use case BuyingDrink should be developed prior to the ComposingDrink use case, since this is a more important use case and the ComposingDrink is the use case that extends it (a part of the main use case).

Because of the scope of the paper only the first iteration is presented and discussed.

3.4. UseCase Description

Use case Name: BuyingDrinks

Main flow.

	Actor Action	System Response
1	Customer chooses drink	
2		System displays price of the chosen drink
3	Customer inserts coins one by one	
4		For each inserted coin, system validates the coin and displays amount left to pay for the drink
5	On the completion of inserting coins Customer chooses to deliver drink	
6		System composes the drink, delivers the drink and gives change.

Table 2 Description of use case buying drink

Alternative flow

Step 2. If there is not sufficient components for the chosen drink system send appropriate message to the ServiceMan and the usecase is stopped.

Step 4. If the inserted coin is not accepted it is returned to the Customer.

Step 6. If the inserted money is not enough system display information and Usecase realisation goes to step 3.

At steps 1,3,5 Customer can choose to cancel the transaction – if so then system returns inserted money and the usecase is stopped.

3.5. System Sequence Diagram

System sequence diagram for the main flow of the BuyingDrink Use Case is presented on Figure 5.

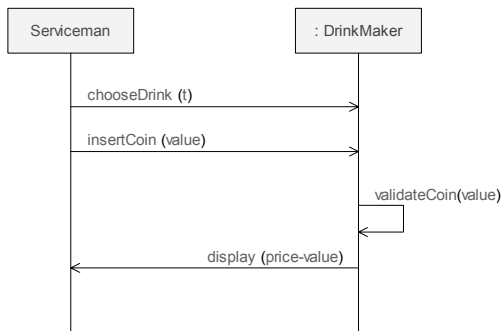


Figure 5. System sequence diagram for the use case BuyingDrink

3.6. Conceptual Model

The conceptual model for the first iteration can look as follows:

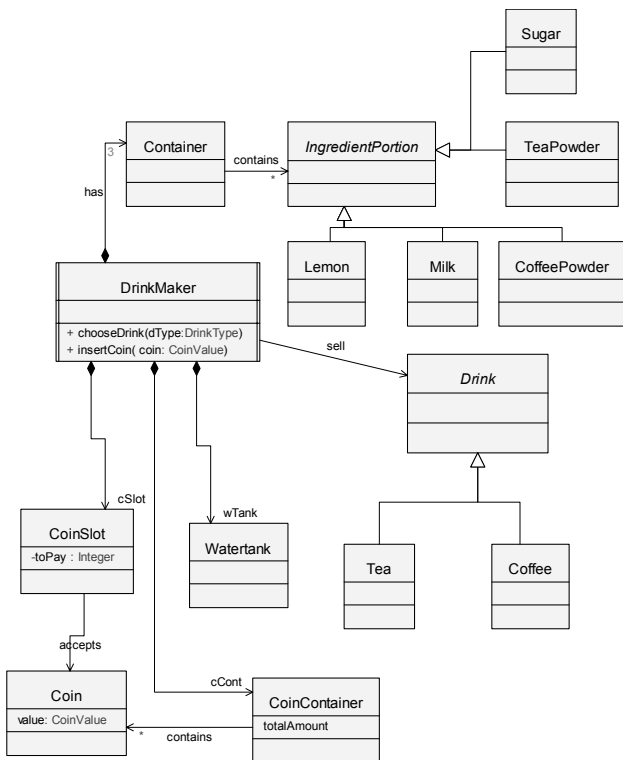


Figure 6 Conceptual model of DrinkMaker

3.7. Contracts for System Operations

DrinkMaker::chooseDrink (dType:Drink)

pre: true
post: chosenDrink = dType

DrinkMaker::insertCoin(coin:Coin)
pre: chosenDrink != null
pre: coin.value = 1 or coin.value = 5 or coin.value = 10
post: cCont.totalAmount = cCont.totalAmount@pre + coin.value

DrinkMaker::prepareDrink()
pre: cSlot.toPay = 0
post: instance of Drink was created

3.8. Sequence Diagram

Specification of the operation
DrinkMaker::chooseDrink(dType:DrinkType)

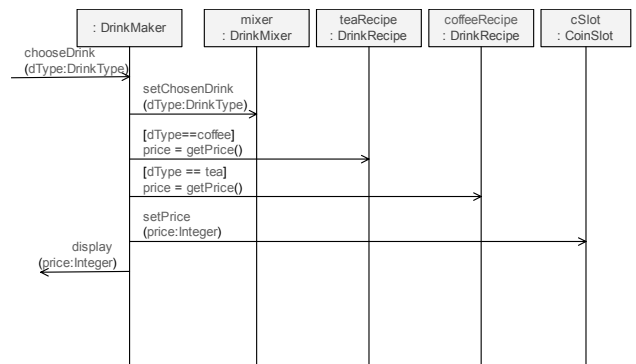


Figure 7 Sequence diagram for operation chooseDrink(...)

Specification of operation
DrinkMaker::insertCoin(coin:CoinValue)

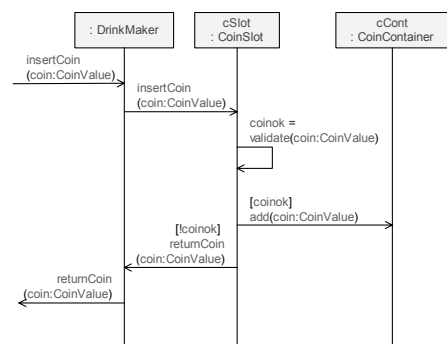


Figure 8 Sequence diagram for operation insertCoin(...)

Specification of operation
DrinkMaker::prepareDrink()

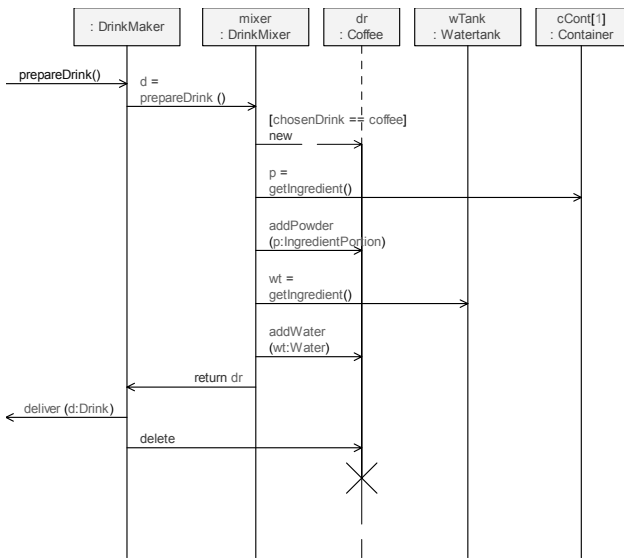


Figure 9 Sequence diagram for operation prepareDrink()

State machines for Drink class and DrinkMaker class (use case controller).

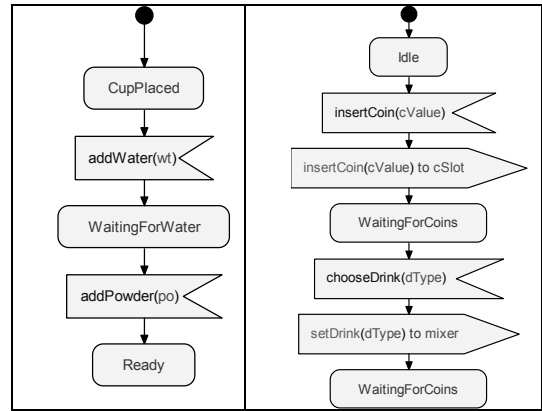


Figure 11 Statechart diagrams for the Drink class (left-hand side) and DrinkMaker class (right-hand side)

3.9. Design Class Diagram

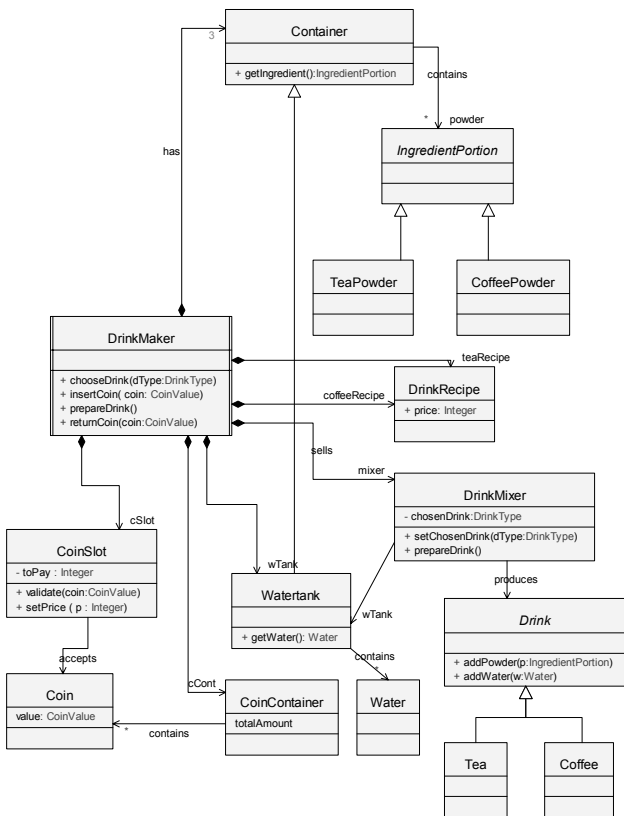


Figure 10 Design class diagram of DrinkMaker

3.10. Statechart Diagrams

4. Example Inconsistencies in the presented design

After completing the activities within a phase all artifacts are examined to see whether two of them might express contradictory properties of the system. The examination is based on the set of rules of thumb which later can be used for more precise and formal definition of the meaning of consistency for the specified didactic process. Subsequent sections include example situations which are considered as inconsistencies. The examination is done by checking the pairs of artifacts. The pairs are formed according to the relationship between types of artifacts shown on figure 4.

4.1. Use Case diagram and Use Case description

The actor shown on the use case diagram is not the same actor that takes part in interaction defined in system sequence diagram and use case description. This is not the case in the system presented in the paper, but is a common mistake in the student designs.

4.2. Use Case Descriptions and System Sequence Diagrams

During the analysis phase, where the system requirements are being expressed with help of UML, there should be a transition from the Use Case description to the system sequence diagram. However, this transition does not always lead to a consistent model when done manually. The following part of system sequence

diagram was created for the BuyingDrink Use Case, although it is not consistent with it.

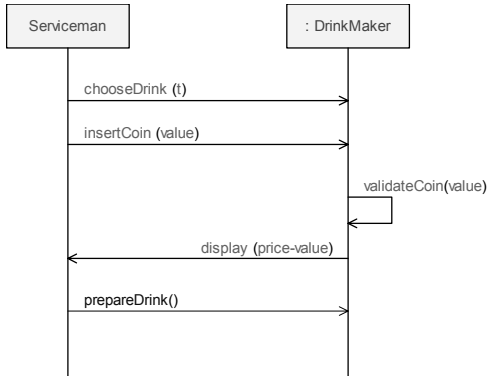


Figure 12 System sequence diagram inconsistent with the Use Case description

These are the main inconsistencies in the diagram

- the actor from the system sequence diagram (Serviceman) is not the same actor as specified in the interaction in the Use Case Description
- not all steps in the Use Case Description correspond to messages in the System Sequence Diagram. The missing steps are: 2 (displaying price), 5 (choosing to prepare drink) and 6 (drink delivery)
- extension point for the extending use case – ComposingDrink is missing
- iteration symbol for inserting coins is missing

The inconsistencies can lead to the improper design of the whole system and may result in failing the acceptance tests, since the designed system (according to the system sequence diagrams) is not what the customer expected (according to the use case description).

4.3. System Sequence Diagram and Conceptual Model

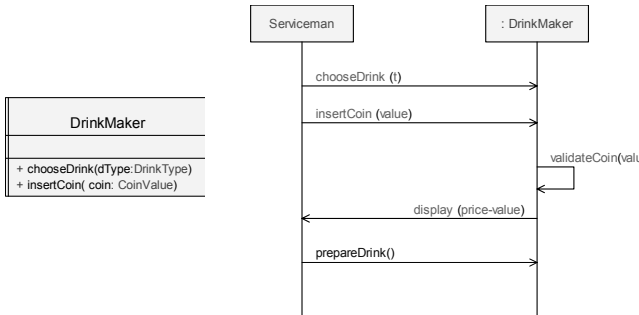


Figure 13 Inconsistency between the conceptual model (represented by the DrinkMaker class) and system sequence diagram

4.4. Contracts and Conceptual Model

An important feature of the contract is that it is expressed in the context of conceptual model. However, the designs do not always follow this principle, which leads to inconsistencies between the conceptual models and operation contracts. These inconsistencies question the rationale of the contracts for operations. For instance, the following figure presents such inconsistency.

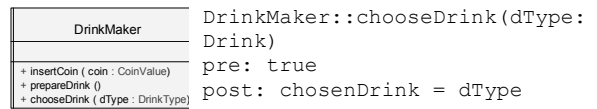


Figure 14 Inconsistency between the conceptual model and an operation contract

As it is shown in the contract the post-condition of chooseDrink(...) operation states that the value of attribute chosenDrink(...) of a current instance of the DrinkMaker class is equal to the value of the parameter defined in the conceptual model on the left-hand side of the figure.

This kind of inconsistencies in designs can lead to failing the integration tests since the contracts are used in other components of the system to ensure the proper integration. If the contract cannot be checked (as in the case of the above example), then there is no guarantee that the contract is met.

4.5. Interaction diagram for operation and design class diagram

Missing elements in the design class diagram, which are depicted according to the sequence diagram, are quite recurring inconsistency. The following two diagrams show a sample situation.

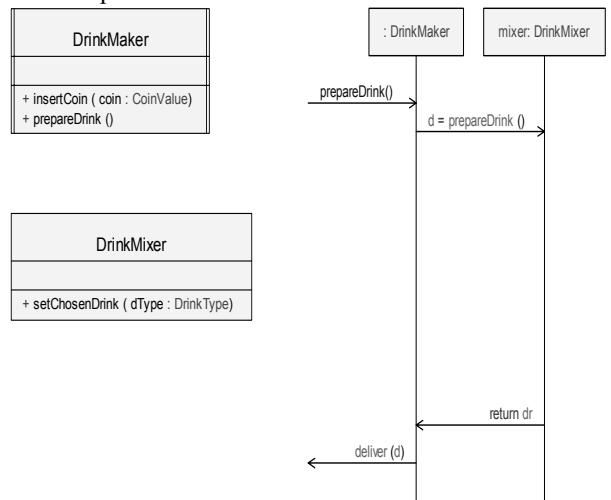


Figure 15 Inconsistency between the design class diagram and the sequence diagram for operation

The design class diagram shown on the left-hand side shows two classes – DrinkMaker and DrinkMixer, which do are not related with an association relationship. On the contrary, there is a message sent between the DrinkMaker and DrinkMixer on the sequence diagram on the right-hand side of the figure. What is more, the message that is sent over a link which is an instance of the missing association – prepareDrink(...) is also missing in the design class diagram.

The inconsistency presented in this example can lead to errors in code generation or lead to failing the unit tests. What is more, most of the code generators cannot generate the code based on sequence diagrams, but based on class diagram only. In such situations the developer is responsible for writing the code based on the sequence diagrams. However, these errors can be easily discovered during the implementation phase, since they most probably result in compilation errors.

4.6. State diagrams and interaction diagram for operation

A common mistake is to develop the state machine for a class, which is not consistent with the desired design of the interaction diagram for operation. For instance, the following diagram presents a situation where the state machine for the drink class does not accept the desired interaction as designed on the sequence diagram.

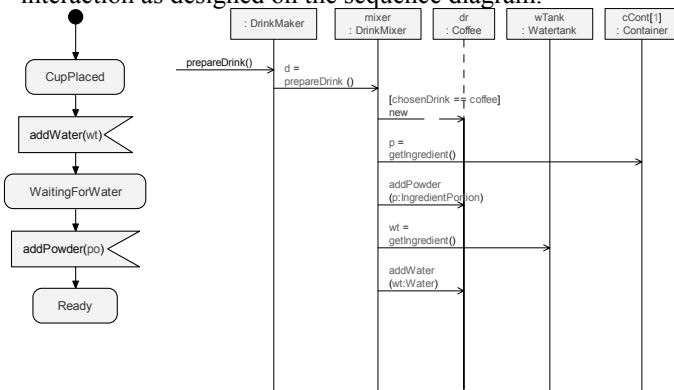


Figure 16 Inconsistency between state machine for the Coffee class and the desired sequence of messages in the interaction diagram for this class

The desired sequence of messages to the instance of a Coffee class is addPowder; addWater, while the accepted sequence of messages by the state machine is: addWater; addPowder.

Such inconsistencies may result in improper implementation of the class, since the state machine of the class, specifying its behaviour clearly does not accept the sequence of messages required in the sequence diagram. It may result in failing the unit tests for the subsystem, this class is part of.

4.7. State diagram for a use case controller and system sequence diagram

The inconsistency between the state diagram for the use case controller – the DrinkMaker class and the system sequence diagram for buying drink in messages chooseDrink(...) and insertCoin(...).

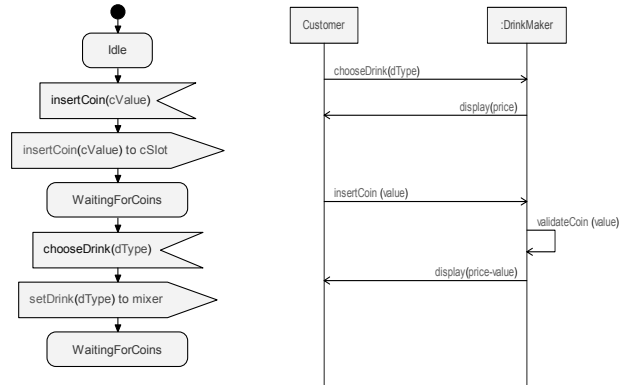


Figure 17 Inconsistency between state machine of the use case controller and the interaction between the system and the actor in this use case.

The state machine for the DrinkMaker does not accept the sequence of messages depicted on the system sequence diagram. The state machine accepts sequence: insertCoin, chooseDrink, while the system sequence diagram requires interaction sequence: chooseDrink, insertCoin. What is more, there is a missing step in the state machine – the displaying of the price.

This inconsistency can lead to the improper implementation of the system and can results in failing both the unit tests for the class and the acceptance tests for the system (since the class is a Use Case controller).

5. Final remarks

In the paper an investigation of the artifacts expressed in UML and produce within a single iteration of a sample development process has been done. Several situation which can be considered as inconsistent were pointed out. These situations have been indicated based on a subjective but ‘common sense’ judgement. From the perspective of the sample process it seems that a set of rules would help in preventing such rather common errors in design. Such rules should relate the following elements and address the following example situations:

- Use case diagram and use case description – i.e. actor(s) interacting with the system in a use case description should be the actor(s) that are defined in the use case diagram.
- Use case description and system sequence diagram - i.e. each step in a use case description

should have a corresponding message in the system sequence diagram for this use case.

- System sequence diagram and conceptual model – i.e. messages (system operations) used in System Sequence Diagram should be defined for the class encapsulating the system in the Conceptual Model.
- Contracts and Conceptual Model – i.e. elements used in the pre and post conditions of the contracts should be defined in the conceptual model (contracts should be defined in the context of conceptual model).
- Interaction diagram for operation and design class diagram – i.e. links used in the interaction diagrams should be instances of the existing associations defined in the design class diagram.
- State diagrams and interaction diagram for operation – i.e. the sequence of messages that is defined for the interaction diagram for the operation in interaction diagram should be a subsequence of a sequence of messages acceptable by state machines for classes that take part in this interaction.
- State diagram for a use case and system sequence diagram – i.e. sequence of messages shown on system sequence diagram of a use case should be acceptable by the use case controller, which is responsible for handling the use case interaction.

As a contribution to the more general understanding of consistency the identified informal consistency rules might be a starting point to what should be formalised as a part of general understanding of consistency from the perspective of a didactic process.

References

1. Kuzniarz, L., et al. *Workshop on Consistency Problems in UML-based Software Development*. in <<UML>> 2002. 2002. Dresden: Blekinge Institute of Technology.
2. Object Management Group, O., *Unified Modeling Language Specification v. 1.4*. 2002.
3. Booch, G., *Unified Software Development Process*. 1998: Addison-Wesley.
4. Arlow, J. and I. Neustadt, *UML and the unified process : practical object-oriented analysis and design*. 2002, London ; New York: Addison-Wesley.
5. Robillard, P.N., P. D'Astous, and P. Kruchten, *Software engineering process with the UP/Edu*. 2003, Boston: Addison Wesley.
6. Kruchten, P., *The rational unified process : an introduction*. 2nd ed. Addison-Wesley object technology series. 2000, Reading, MA: Addison-Wesley. xviii, 298 p.
7. Larman, C., *Applying UML and patterns : an introduction to object-oriented analysis and design and the unified process*. 2nd ed. 2002, Upper Saddle River, NJ: Prentice Hall PTR. xxi, 627 p.

Case Study: Consistency Problems in a UML Model of a Chat Room

Thomas Huining Feng and Hans Vangheluwe
Modelling, Simulation and Design Lab
McGill University
<http://msdl.cs.mcgill.ca/>

Abstract

This article describes a case study, where a model of a chat room application is built from initial requirements. UML class diagrams, sequence diagrams and statecharts are used in different stages of the development process. Consistency problems are identified and methods, most notably simulation, are proposed, to ensure consistency between some aspects of the models. We focus on intra-consistency, the consistency among artifacts within a given model.

1 Introduction

The development process of a software system is usually divided into steps, in which different UML diagrams are involved. As modelled systems become more and more complex, consistency problems become more prominent. Two types of problems are apparent. The first, intra-consistency problems, are concerned with consistency among artifacts within a given model. The second, inter-consistency problems, are concerned with consistency between different models evolved during the course of the software development process. In the sequel, we focus mostly on intra-consistency.

Various formal methods have been explored in the literature to automatically check consistency and discover design problems. In the following sections, steps of the development process of a chat room model are studied. The case is inspired by a project report on “Executable UML” by Geir Melby at Agder University College (<http://fag.grm.hia.no/ikt2340/year2002/>). Potential consistency problems in the model are shown. For some of them, automatic consistency checking methods are proposed.

In this case study, a model of a chat room application is developed from requirements. A protocol specifying communication between clients, a manager object, and chat rooms is given in section 2, and treated as initial requirements. Section 3 studies a possible class design.

It defines interfaces conforming to the protocol. The sequence diagrams in section 4 further refine and illustrate the inter-component communication, consistent with the class design. Section 5 uses statecharts to further specify the application’s behaviour. This specification can be either simulated or executed in real-time in our SVM (Statechart Virtual Machine) environment. In section 6, consistency of simulation traces with the protocol specification is discussed. Section 7 concludes this case study.

2 The Communication Protocol

The chat room application to be built features a client-server configuration. Clients try to connect to random chat rooms. After a client is accepted by a chat room, it sends messages to its chat room. The chat room broadcasts each message so that every client connected to it, except the sender, receives a copy.

A specific, simplified use case is described below:

- There are 5 clients and 2 chat rooms in the system. Initially, the clients are not connected. They try to connect to a random chat room every 1 to 3 seconds (uniformly distributed). The requested chat room instantaneously receives the request (no network delay, and reliable communication are assumed).
- A chat room accepts at most 3 clients. It accepts a connection request if and only if its capacity is not exceeded.
- The requesting client receives an acceptance or rejection notice immediately.
- A client must be accepted by a chat room before it may send chat messages.
- When connected, a client sends random messages to the chat room it is connected to every 1 to 5 seconds (uniformly distributed). The chat room immediately receives the messages. It takes 1 second to process a

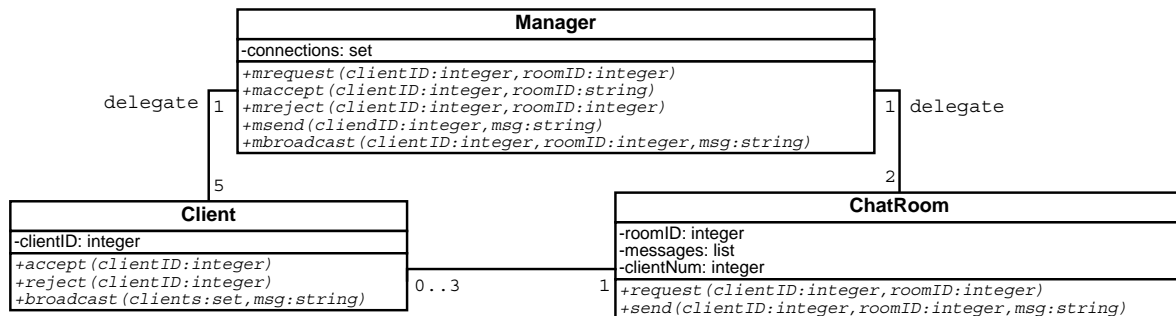


Figure 1: Class design

message and broadcast it to all the clients connected to it, except the sender.

- The clients instantaneously receive the broadcast.

For simplicity, disconnection is not discussed.

3 The Class Design

According to the above specification, two classes are obviously required: `Client` and `ChatRoom`. At this early stage in the development process, so that no user intervention is needed for a simulation, we explicitly model client behaviour (connection requests and chat messages) as a random process. Later, this model of the client will be replaced by real human clients interacting with the software. When the simulation is started, 5 instances of `Client` and 2 instances of `ChatRoom` are initialized.

A singleton class `Manager` is added. The `Manager` acts as a mediator and relays all the communication between components. This central control facility helps to intercept all the messages passed in the system, with which correctness of the model can be checked.

Figure 1 shows the UML class diagram featuring the three classes.

- A `ChatRoom` provides two methods to handle incoming events. `request` handles incoming requests, each of which has parameters `clientID` and `roomID`. The `ChatRoom` sends back an acceptance or rejection notice to the sender with a global ID `clientID`. It also uses the `roomID` parameter to decide whether the request is sent to itself or to another chat room¹. The `send` method receives a `msg` sent by client `clientID`. This `msg` will be broadcast 1 second later.

¹As UML statecharts (pre UML 2.0) are used in later steps, which do not allow one to specify the receiver of an event, all the chat rooms receive the same request even if only one of them will handle it.

- Methods `accept` and `reject` of `Client` handle incoming acceptance and rejection notices. Parameter `clientID` is used to identify the target client. When a `Client` receives a broadcast event, it checks if itself is in the set of `clients`. If so, message `msg` is printed to the output.

- The `Manager` relays connection requests, acceptance and rejection notices, messages sent from clients and broadcasts from chat rooms. For example, when it receives broadcasts from chat rooms, the three parameters tell it the original sender (client) of the message, the broadcaster (chat room) and the message string. It then sends the message to all the clients connected to this chat room, with the exception of the original sender.

Though this API definition is not functional, the behavior behind the interface is easily understood. Checking its consistency with the protocol is however difficult or even impossible because of the following reasons:

- Behavior is hidden behind the interface, which can only be interpreted by human understanding.
- The protocol is specified in natural language, which a program cannot easily process.
- For a well-defined system there can be a number of interface designs. They may differ substantially. For example, in this design a `Manager` class is used to intercept communication. Another design may use `RequestManager` to intercept requests, acceptances and rejections, and use `MessageManager` to intercept messages and broadcasts. Yet another design may not use any manager at all.

4 Sequence Diagrams

The sequence diagrams discussed in this section bring the design to a lower level of abstraction (higher level of

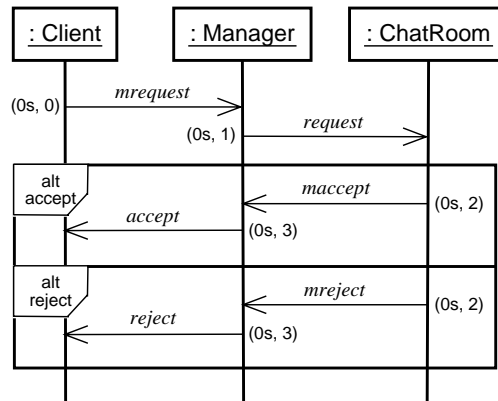


Figure 2: Sequence diagram of the request pattern

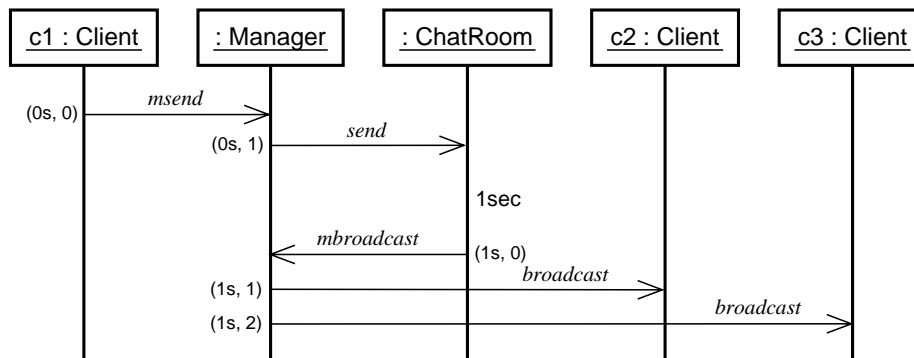


Figure 3: Sequence diagram of the message pattern

detail) than the class diagram. The sequence diagrams must clearly reflect the interaction between components.

4.1 Timing

Timing issues make conversion of the protocol into sequence diagrams and later statecharts difficult. In the protocol description, more than one action can happen at the same time, though they may be causally related. For example, a chat room should not send an acceptance or rejection before it receives a request, though time is not advanced. Hence, “request at time 1; accept at time 1” in the output trace is correct, while “accept at time 1; request at time 1” is not.

One possible solution is to use a tuple (t, s) to represent time, where t is float-point time in seconds and s is an integer sequence number. In this way, the correct output can be written as “request at time $(1.0s, 0)$; accept at time $(1.0s, 1)$ ”. The sequence “accept at time $(1.0s, 0)$; request at time $(1.0s, 1)$ ” is incorrect.

4.2 The Request and Message Patterns

The request pattern is shown in Figure 2. A Client first invokes the `mrequest` method of the Manager. The Manager then relays the request by calling `request` of a ChatRoom. The ChatRoom immediately responds and calls back the `maccept` or `mreject` method of the Manager. The requesting Client then receives the relayed reply from the Manager.

Figure 3 shows the message pattern where a random message is generated and passed in the system. Note that the ChatRoom deliberately delays 1 second after it receives a request. No other time delay is shown in the two sequence diagrams.

4.3 Consistency with the Class Diagram and the Protocol

Consistency with the class diagram can be easily checked by collecting all the method calls that a component receives. For example, in the request pattern, the Manager

receives `mrequest`, `maccept` and `mreject`. In the message pattern it receives `msend` and `mbroadcast`. These five methods are defined in the class diagram, and no other public methods are defined. As parameters are not shown in the sequence diagram, there is no need to check their parameters. This checking process can be automated.

Consistency with the protocol can be partially checked. One can easily see that according to the request sequence diagram, if a chat room receives a request at time 0, it accepts *or* rejects the client at time 0. The absolute values of the two times are not important. Important is that the reply is sent back at exactly the same time. In this way the designer can check “what should happen at a certain time” manually. If the rule-based approach discussed later is used, limited automatic checking is also possible.

Note how basic sequence diagrams are unable to express what should *not* happen at a certain time or in a certain period. For example, it is implied in the protocol that a chat room does not accept or reject a client without a request. This information is missing in the sequence diagrams. This may introduce design errors into the model, and affect the correctness of later development steps.

Another possibility for an erroneous design is due to the semantics of sequence diagrams. For example, in the request pattern, the sequence diagram describes: *if* a client sends an `mrequest`, *then* the manager sends a request without time advance, *then* the chat room sends an `maccept` or `mreject` without time advance, *then* the manager sends `accept` or `reject` accordingly. Unfortunately, an inert client that does not send any request, which is obviously a problem in the system, can not be detected by checking the sequence diagram. In the worst case, no client tries to connect, and thus the system halts forever.

To compensate for the loss of information, designs in other UML formalisms are needed which do not completely depend on sequence diagrams, or the sequence diagram formalism must be extended. An excellent example of the extension and use of sequence diagrams are Live Sequence Charts, as described by Harel [1].

5 Statechart Design

Statecharts are used to implement the behavior behind the class definitions. They can be executed in our SVM (Statechart Virtual Machine) [2] [3], an interpreter for an extended statechart formalism written in Python.

5.1 SVM Conventions

Before the statechart designs can be easily understood, some SVM conventions must be introduced beforehand.

SVM interprets models in the extended statechart formalism. New features are added. Though expressiveness is not enhanced², ease of use is greatly improved.

Component-based design is possible in SVM, though original statecharts are not modular. This is necessary for the chat model, where components such as clients and chat rooms are designed separately, but work together in the final system. Components are reused by importation. A larger component imports (an instance of) a smaller one into one of its states. The result is as if all the (hierarchical) states and transitions of the imported component were directly written inside that state.

SVM models are written in text files. *Macros* are a concept introduced in SVM. Macros are defined in the `MACRO` section of an SVM source file. Once defined, they can be used in brackets throughout the text file. For example, with `PREFIX=state` defined, `[PREFIX]` can be used to literally substitute string “state”, and thus `[PREFIX] 1` is equivalent to `state1`.

Some of the predefined macros are used later on. `[EVENT(event, param)]` raises an event. It carries a string event and an optional list `param` as the parameters that travels along with it. `[PARAM]` is used to retrieve parameters of the event being handled. It is usually used in the guard or output of a transition. `[DUMP(msg)]` prints debugging messages to the screen or records them in a text file.

Macros also serve as parameters when a component is imported. The importing component may *redefine* some or all of the macros originally defined in the imported component, including predefined macros. As a continuation of the previous example, if the importing component specifies `PREFIX=mystate` as an importation parameter, `[PREFIX] 1` within the imported component is interpreted as `mystate1` instead.

It is easy to show that these extensions do not increase the expressive power of statecharts.

5.2 The Chat Room Model in the Extended Statechart Formalism

Components `Client`, `ChatRoom` and `Manager` are designed in separate statecharts. As Figure 4 shows, model `Chat` imports five instances of `Client`, two instances of `ChatRoom` and one `Manager`. Each instance of the same type has a unique ID parameter. Instances of different types can have the same ID since their sets of acceptable events are disjoint. This model can be simulated or executed in the SVM environment.

The `Client` component is shown in Figure 5. Initially, it is in the `nochat` state. It repeatedly tries to connect to the

²More extensions can be made to enhance expressiveness, but checking the correctness of a model becomes much more difficult.

chatroom via the manager by raising an `mrequest` event every 1 to 3 seconds (uniformly distributed), until the request is accepted (the `accept` event is received). `uniform` is a Python function which returns a random real number in a range, and `randint` returns a random integer. The event's first parameter gives the client's unique ID. The event's second parameter gives the destination chatroom (randomly chosen from 1 or 2). Then, the client moves to state `connected` and starts sending messages and receiving broadcasts. Since parameters of events are sent as a list, `[PARAMS][0]` gives access to the first parameter, and so on. Note that when the content between square brackets is not a macro name or a Python index, it is a guard as defined in the original statechart formalism [4].

User-defined macro `[ID]` gives a unique ID to each Client. Its definition `ID=0` implies that the default value is 0. It is changed by the importing component (the Chat model in this case) to a unique number. ID of a component is important. Since the whole system can be viewed as one large statechart after importation, all broadcast events are received by every orthogonal component. Thus, the only way to send an event to a specific client is to give the receiver's ID in the parameter list. Each client checks if its ID matches before handling an event.

Compared to the Client component, ChatRoom is much more complicated. It uses a list `messages[ID]` to queue incoming messages. This means every chat room with a unique ID has its own queue. (For `ID=0`, `messages[ID]` is equivalent to `messages0`.) If a message comes when it is busy processing a previous message (it takes 1 second), the new one is added to the list. The time when the message is received is also recorded so that even if a message is queued, its processing time is still 1 second since its arrival.

The Manager component simply relays messages. Function `rec_comm(client, room)` records a connection in a list when a chat room accepts a client. `get_clients(room, client)` looks up the list and returns all the clients in chat room `room`, except `client`. `get_room(client)` returns the room ID for `client`.

The message queue of chat rooms and the connection list of manager are examples of variables. They help to record the state of the model. Strictly speaking this is also an extension to original statecharts, where states must be explicitly specified. The discussion of variables is outside the scope of this case study.

5.3 Consistency with the Class Diagram

This component-based design should strictly conform to the class design in Figure 1. Otherwise, a component may send an event to a receiver, who cannot handle it. Or,

the sender may provide less parameters than required. The result can be a fatal run-time error.

A program can be written which automatically checks sender-receiver consistency of all the method calls. Not how at the code-level this might be checked by a type-checker and/or linker. For example, Manager accepts event `maccept`. This means it provides method `maccept` in its class definition. In the guard and output of the transition that handles this event, `[PARAMS][0]` and `[PARAMS][1]` are used, so it requires *at least* two parameters. The checker then looks through the whole Chat model and finds that this method is only called (asynchronously) by the ChatRoom component. The call `[EVENT("maccept", [[PARAMS][0], [ID]])]` provides exactly two parameters (`[PARAMS][0]`³ and `[ID]`). The checking of this call is successful.

Similarly, the consistency of all the method calls in the model can be checked against the class diagram.

6 Consistency Checking by Model Execution

The Chat model is simulated or possibly executed in real-time (needed in case of a human in the loop) by the SVM interpreter. The output produced in the execution is dumped to screen and a text file. As mentioned above, human intervention is not needed if all user interaction is explicitly modelled. The output trace is the only means by which we validate the execution. Consistency of the trace with all the design artifacts discussed above must be checked.

Consistency with the class diagram was studied in the previous section. The checker formally checks the statechart design. Model execution is not needed.

Consistency with the sequence diagrams is checked by validating the output trace of experiments. Although correctness can in many cases not be proved (as it would require the exploration of a large or possibly infinite state-space of possible behaviours), *confidence* in the final product is greatly increased.

Consistency with the statecharts is implied provided that the SVM execution environment is correct.

Proving consistency with the original protocol is not easy, because it contains much more information than the sequence diagrams does. It is also hard to be processed by a checker program.

³ `[PARAMS][0]` here refers to the first parameter of event `request`, which is handled by the ChatRoom component. This parameter is further passed on in event `maccept` as the latter's first parameter.

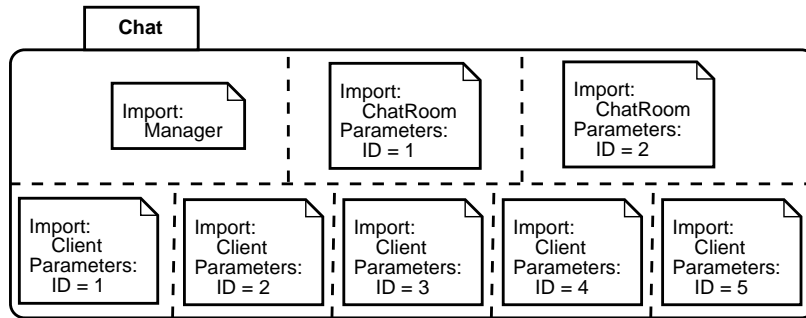


Figure 4: Chat model

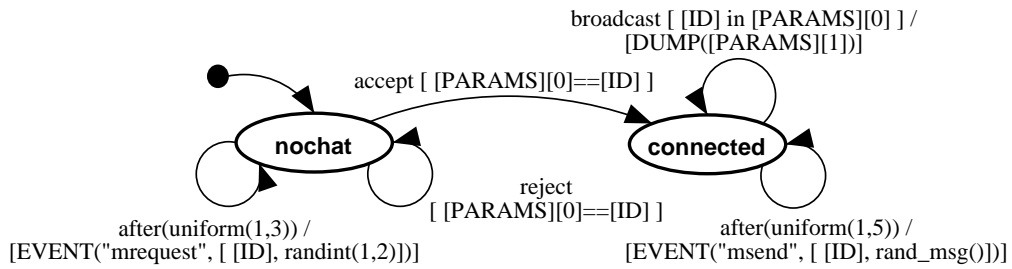


Figure 5: Client component

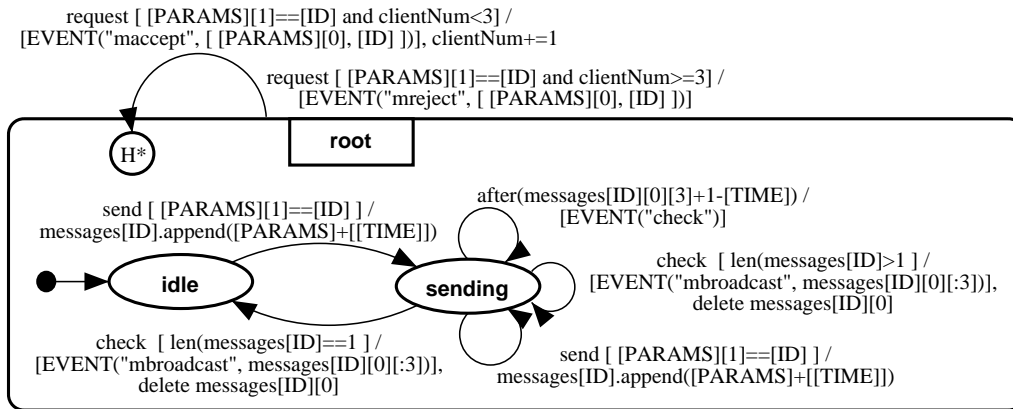


Figure 6: Chat room component

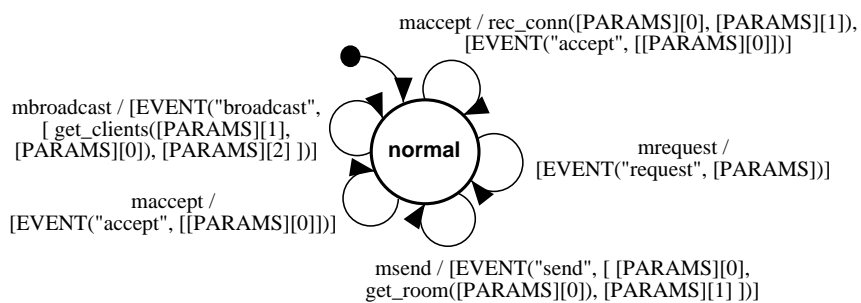


Figure 7: Manager component

6.1 Output Trace

[DUMP(msg)] macro is used to record messages msg in a file, until the execution is finished (either automatically or by manual control by the debugger). Each message consists of three parts: the time written as a tuple (t,s), the sender or receiver with its unique ID, and the message body. The following is taken from the output:

```

.....
CLOCK: (10.5s,0)
Client 0
Says "Hello!" to ChatRoom 1
.....
CLOCK: (11.5s,0)
ChatRoom 1
Broadcasts "Hello!" to all clients except
  Client 0
.....
CLOCK: (11.5s,2)
Client 1
Receives "Hello!" from Client 0
.....

```

This output is produced by the manager, which has access to all relevant information in the communication. At time 10.5 a message is sent by the client with ID 0. According to the protocol, chat room 1 broadcasts this message after 1 second. Another client (client 1), which is also connected to chat room 1, receives the broadcast at the same time. The original sender of the message from is also shown.

6.2 Consistency with the Sequence Diagrams

Consistency with the sequence diagrams can be checked in a rule-based approach. A set of rules are defined and written in a text file. A checker reads the file and checks if the output trace satisfies every rule.

Regular expression are extended to describe rules. A rule consists of four parts: pre-condition, post-condition, guard (optional) and counter-rule property (optional). *Pre-condition* is a regular expression used to match a part of the output trace. It, combined with the *guard* (a boolean expression), defines when the rule is applicable. When it is applicable and the *counter-rule property* is false, the *post-condition* (another regular expression) must be found in the output; if counter-rule is true, the post-condition must *not* be found.

For example, the following rule expresses the fact that the sender of a message does NOT receive the broadcast after 1 second:

pre-condition	CLOCK: \((\d+\.{0,1}\d*)s, (\d+\.{0,1}\d*)\)\nClient (\d+)\nSays "(.*?)" to ChatRoom (\d+)\n
post-condition	CLOCK: \((\d+\.{0,1}\d*)\)\nClient [(\d)]\n Receives "[(\d)]" from Client [(\d)]\n
guard	[(\d+)]<50
counter-rule	true

In the pre-condition, five expression *groups* are defined in parentheses. They are numbered 1 to 5. Group 1 matches the floating-point time. Group 2 matches the sequence number. They constitute a time tuple. Group 3 matches the integer client ID of the sender. Group 4 matches the message, which is an arbitrary string. Group 5 matches the chat room which the sender is in.

In the post-condition, [(...)] contains an expression, where values of groups can be cited with their index numbers behind “\”. Thus, [(\d+)] is the value of the first group plus 1. [(\d)] is equal to group 3. More about the regular expressions used can be found in [5].

Suppose the execution stops at simulated time 50. The checking should not exceed time 50. Without additional conditions, if a message is sent to a chat room at time 49.5, the checker would expect a corresponding broadcast at time 50.5. To cope with this, a guard [(\d+)]<50 is added. This tells the checker that the rule is applicable only when the value of group 1 (floating-point time) plus 1 is less than 50.

Since a client should not receive its own message, this is a counter-rule.

6.3 Consistency with the Protocol

It is difficult, if not impossible, to prove the model is completely consistent with the protocol. The protocol, also regarded as a set of requirements, is described in natural language. Its interpretation is the main obstacle for the development of an automatic checker.

One may argue that the protocol can be transformed into a set of rules. With the rule-based method described above, consistency with the protocol can be checked. However, it is hard to transform the complete meaning of the protocol into a formal representation, which is easily processed by a program. Obvious facts implied in the protocol and common knowledge are usually lost. As an interface between human beings and computer programs, a natural language processing technique is required.

In this case study, a series of steps are used to achieve the final, executable design. Information is lost while converting a design into another in a different formalism. Checking between intermediate steps does not guarantee the correctness of the final product.

On the one hand, checking intermediate steps is not strong enough. On the other hand, it is extremely hard to check the model directly against the original protocol. “How to prove the correctness of a final design” is the last and largest open question in this case study.

7 Conclusion

A concrete example is discussed in this case study. An executable model is developed from initial requirements. Steps are gone through and designs at different levels of abstraction are studied. A component-based approach is chosen to make the model modular and manageable. A class diagram defines the interface of components. Sequence diagrams formalize the communication and make automatic checking possible, though they only partly illustrate the requirements. The component-based model is modelled in the extended statechart formalism. This model is directly interpreted by the SVM execution environment.

Development of the chat room model gives rise to a series of consistency problems. For some of them, automatic checking is successfully applied.

1. Consistency between the *sequence diagrams* and the *class diagram* is checked. A checker verifies all the required methods are correctly specified in the interface.
2. Consistency between the *statecharts* and the *class diagram* is also checked in a similar way. The sender of an event always provides enough parameters to the receiver.
3. Consistency between the *statecharts* and the *sequence diagrams* is checked with a rule-based checker. Regular expressions are extended to specify pre-conditions, post-conditions, guards and counter-rule properties.

However, other consistency problems remain unsolved.

1. Consistency between the *class diagram* and the *protocol* (initial requirement) is not checked. Design flaws may be discovered in later steps or be hidden in the final product.
2. Consistency between the *sequence diagrams* and the *protocol* is only checked manually. Though the sequence diagrams are just a formalization of the protocol, it is not easy to check their correctness with a program.
3. It is even harder to check the consistency between the final design in *extended statecharts* and the *protocol*. This checking is necessary, as information is lost in intermediate steps.

Attention must be paid to these open questions which mostly pertain to inter-consistency. It is believed that consistency checking should be an integral part of the development process and of software development tools.

References

- [1] D. Harel and R. Marelly. Specifying and executing behavioral requirements: The play-in/play-out approach. Technical Report MSC01-15, The Weizmann Institute of Science, 2001.
- [2] Thomas Feng. An extended semantics for a Statechart Virtual Machine. In A. Bruzzone and Mhamed Itmi, editors, *Summer Computer Simulation Conference. Student Workshop*, pages S147 – S166. The Society for Computer Modelling and Simulation, July 2003. Montréal, Canada.
- [3] Thomas Feng. Statechart Virtual Machine (SVM), 2003. MSDL, McGill University, <http://moncs.cs.mcgill.ca/people/TFeng/?research=svm>.
- [4] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
- [5] Python 2.2.3 documentation, May 2003. <http://www.python.org/doc/2.2.3/>.
- [6] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [7] Michael von der Beeck. A structured operational semantics for UML statecharts. *Software and Systems Modeling*, 1(2), 2002.
- [8] Jim Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

An Empirical Investigation in Quantifying Inconsistency and Incompleteness of UML Designs

C. Lange¹, M.R.V. Chaudron^{1*}, J. Muskens¹, L.J. Somers^{1,2}, H.M. Dortmans²

¹ Technische Universiteit Eindhoven, Department of Mathematics and Computing Science,
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands

² Océ Technologies, P.O. Box 101, 5900 MA Venlo, The Netherlands

*M.R.V.Chaudron@tue.nl

Abstract

The UML is becoming the de-facto notation for software engineering projects. It is a common hypothesis that incompleteness and inconsistency allowed by UML are a source for problems in the software development process. However, it appears that many adequate software systems are built with the use of UML. This raises the question as to what degree inconsistency and incompleteness in UML designs impact software engineering projects. For instance, are there typically many or few inconsistencies in a design? Which type of inconsistency occurs most often? Do the types of inconsistencies that are present in a design change in the course of the design process? If so, how?

To investigate these questions, we have developed a number of techniques for analyzing UML designs. In this paper, we present the results of our study on inconsistency and incompleteness in large industrial systems.

1. Introduction

Software systems are described using multiple views. These views are partially overlapping. UML is a collection of diagramming techniques for describing these different views. Describing software systems using multiple diagrams that contain overlapping information introduces the risk for inconsistencies.

Often it is impossible to conclude whether diagrams are inconsistent or incomplete. Consider the example presented in Figure 1. In the message sequence chart (MSC), an instance of class A invokes a method Z on an instance of class B. However, in the class diagram, Class B does not have a method Z. Is the MSC inconsistent with the Class diagram or is the Class

Diagram incomplete with respect to the MSC? Hence, we consider incompleteness to be strongly related to inconsistency.

Inconsistency and incompleteness allowed by UML can be sources of problems in software development. This paper describes initial results of our research into the severity of the inconsistency and incompleteness problem and the results of quantitative assessment of a number of industrial systems.

We will put related work into perspective in section 2. In section 3 we will define the notions of consistency and completeness, define their relationship and give classifications of different types of inconsistencies. The classifications are illustrated with examples. Our approach to the problem of detecting design faults and some examples of inconsistency rules and metrics will be given in section 3. The results of analysing inconsistency in industrial cases are presented in section 4. Our conclusions and ideas for future work are summarized in section 5.

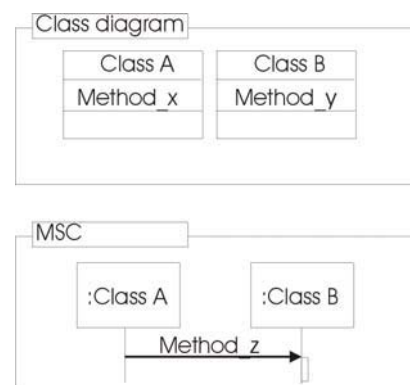


Figure 1. Inconsistency or incompleteness?

2. Related Work

There main areas of related work are: software product metrics and consistency analysis of UML.

The work on software product metrics attempts to quantify quality attributes of software design. The work on software product metrics has been oriented at the level of source code. In recent years, some work on metrics has been applied at the level of software design [3] [2] and software architecture [10]. However, this has mainly been a migration of existing code-level metrics to the level of UML. This plain migration fails to acknowledge that software architecture and design differ from code in that they exhibit inconsistency and incompleteness.

Metrics for inconsistency and incompleteness might be good predictors for the quality of the resulting systems. The research described in this paper is the first attempt known to the authors that proposes the use of inconsistency metrics.

Of course, the problem of consistency is not limited to UML-based approaches and is also manifest in, for instance the RM-ODP method [1]. Our focus is on UML-based approaches.

The area of consistency in UML can broadly be divided into the complete approaches and the partial approaches. The complete approaches aim to define a fully formal semantics for the complete UML notation. Notable initiatives in this direction are the ‘precise UML’ project [12] and the ‘Omega’ project [11].

Partial approaches select a subset of the UML notation, typically related to one view such as the process view, and proceed to develop a well-defined meaning for that subset with the aim of automatically identifying inconsistencies in this partial design. Starting point for literature on this line of research are the workshop on consistency problems in UML [17].

The partial approaches can be divided into two principle types: a formal approach that is concerned with mapping (parts of) UML designs onto formal methods in order to give a meaning to the UML constructs. The second is a design-oriented approach. Here emphasis is on modelling the UML in order to analyse properties of the design. In this approach UML is modelled using a meta-model or using OCL [18]). Subsequently, consistency rules for actual UML designs are defined in terms of the meta-model.

The formal approaches attempt to map partial designs that describe system dynamics such as Sequence Charts and State charts onto operational formalisms such as process algebra’s and Petri-Nets. For example, Engels et. al. [5] have presented consistency checking for UML State chart diagrams by

providing a mapping onto CSP. McUumber and Cheng provide a mapping of UML onto Promela [15]. An example of a formal approach that formalizes the functional/structural view of UML designs using Z is described by [14].

In the design-oriented approach, Hnatkowska et. al. [8] present a number of consistency rules that are defined in terms of the OCL. Also our approach falls in this category. We use a subset of the UML meta-model and formulate consistency rules in terms of that meta-model. As we will show later in the paper, this approach supports the detection of inconsistencies as well as the computation of design metrics and heuristics, while it does not require the more complex machinery of fully formal methods.

3. Our approach for identifying inconsistencies and incompleteness

In this section we first introduce our conception of inconsistency and incompleteness. Subsequently, we explain the scope within software development where we want to be able to identify inconsistency and incompleteness. The next sections give a meta-model for UML designs and a formal set of inconsistency and incompleteness rules.

3.1 Concepts

Faults in UML designs can be of different nature. A basic type of design faults is concerned with the well-formedness of diagrams. An example of a well-formedness rule is that all classes in a class diagram must contain a name. More complex fault may occur due to the overlap of information in different diagrams. Inconsistency and incompleteness faults fall in this category of faults.

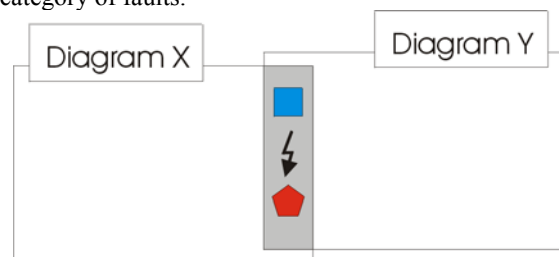


Figure 2. Inconsistency.

A software design describes a set of possible solutions. If diagrams of a design specify solutions that imply contradictory design decisions, then there is an *inconsistency* between these diagrams (illustrated by figure 2). Any software system will correspond to at most one of these diagrams. If this inconsistency is

unresolved, this may lead to misunderstandings and integration problems. Hence one of the diagrams needs to be adapted.

Consistency defines the soundness of a design. In addition we address completeness of a design. Completeness of a design is concerned with the fact that the presence of information in some diagram requires the presence of other information in another part of the design (illustrated by figure 3). For instance, if there is a use-case that describes some system functionality, then there should also be a (collection of) class(es) that provide(s) this functionality. If some information that we can deduce from available diagrams is not present in a design, then there is an *incompleteness* in the design.

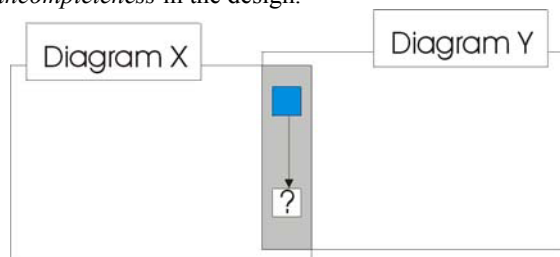


Figure 3. Incompleteness.

Another conception of completeness is to consider whether all the requirements of a system are met by a design. This notion of completeness is concerned with the traceability of development and is very relevant to system development. Validating this type of completeness automatically may be possible to some degree for the functional properties. However, for non-functional properties this becomes extremely difficult. Instead, we shall use inconsistency and incompleteness in way defines earlier: relating to contradicting or missing information that can be deduced from a collection of UML diagrams that form the description of a single system.

3.2 Scope

Figure 4 depicts a number of important artefacts that are produced in a software development project and the relations between them. The figure distinguishes the three levels: requirements, design and implementation. The figure shows that each requirements is related to one or more use cases. Each use case is associated with a number of scenario's (or sequence charts). Each sequence chart uses one or more classes and a class may have a state chart. At the implementation level there should be code for each class in the UML diagram. The scope of our research are the artifacts that are modelled using UML (between the horizontal

lines). Hence we exclude the relations with requirements and implementation artefacts. The main reason for this is that automated checking the consistency and completeness of the relation between requirements and design and between design and implementation require very different techniques or may even turn out to be infeasible without significant human contributions.

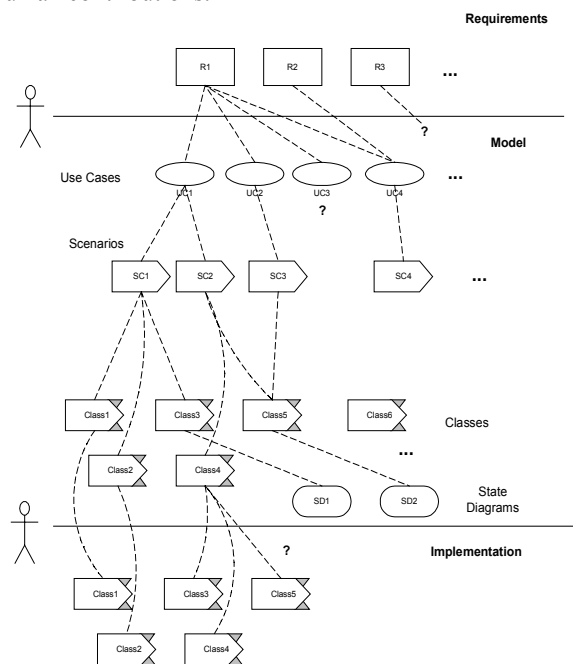


Figure 4. Scope of our research.

Another conception of completeness is to consider whether all the requirements of a system are met by a design. This notion of completeness is concerned with the traceability of development and is very relevant to system development. Validating this type of completeness automatically may be possible to some degree for the functional properties. However, for non-functional properties this becomes extremely difficult. Instead, we shall use inconsistency and incompleteness in way defines earlier: relating to contradicting or missing information that can be deduced from a collection of UML diagrams that form the description of a single system.

3.3 Relational Meta-Model

As a basis for the definition of rules and metrics we use a meta model that describes the elements of a UML design and their interrelationships. An proximate candidate is the UML meta model. This model is a well-defined meta model for the UML specified by the

Object Management Group [OMG] using the Meta Object Facility (MOF).

The interchange format for the UML is XMI, whose specification is based on the UML meta model specification. The specifications of both, the XMI and the UML meta model are very extensive and cover much more information than needed for defining our rules and metrics.

Our analysis focuses on the four most widely used types of diagrams: class diagrams, state chart diagrams, use case diagrams and message sequence charts (MSC). These are a subset the UML, therefore the subset of the meta model depicted in Figure 5 suffices for our purpose. Our meta model is essentially a relational data model.

The meta model incorporates the central concepts of the addressed diagram types. In the use case diagram the use cases are central and have relations to actors and scenarios (message sequence charts). The MSCs consist of messages and objects. This diagram type has a relation to the class diagrams, where the classes are obviously the central concept. Classes may be described by state machines. The relations between classes and actors (inheritance, associations and development) are omitted in figure 5 for the sake of clarity. (The elements of the tables (as it is a relational meta model all entities represent tables) are omitted to keep the figure simple.

The table definitions are given in the appendix).

In the next section, we present examples of rules and metrics for the three categories well-formedness, consistency and completeness. We express rules and metrics in a form of relational algebra.

3.4 Well-formedness

Individual diagrams must satisfy certain basic soundness restrictions, i.e. well-formedness rules. Here we present some examples these rules. For each example of our rules we give a brief explanation and the definition in terms of the relational meta model.

No abstract Leaf Classes. Leaf classes are classes without any subclasses. Leaf classes should not be declared abstract. The purpose of abstract classes is to hand down abstract functionality to subclasses, therefore abstract leaf classes are conflicting. The only exception of this rule is if the described system is a framework that is supposed to provide abstract functionality for subclassing.

$$(\forall c \in C : (\exists m \in M : (c.idc, m.idm) \in CM))$$

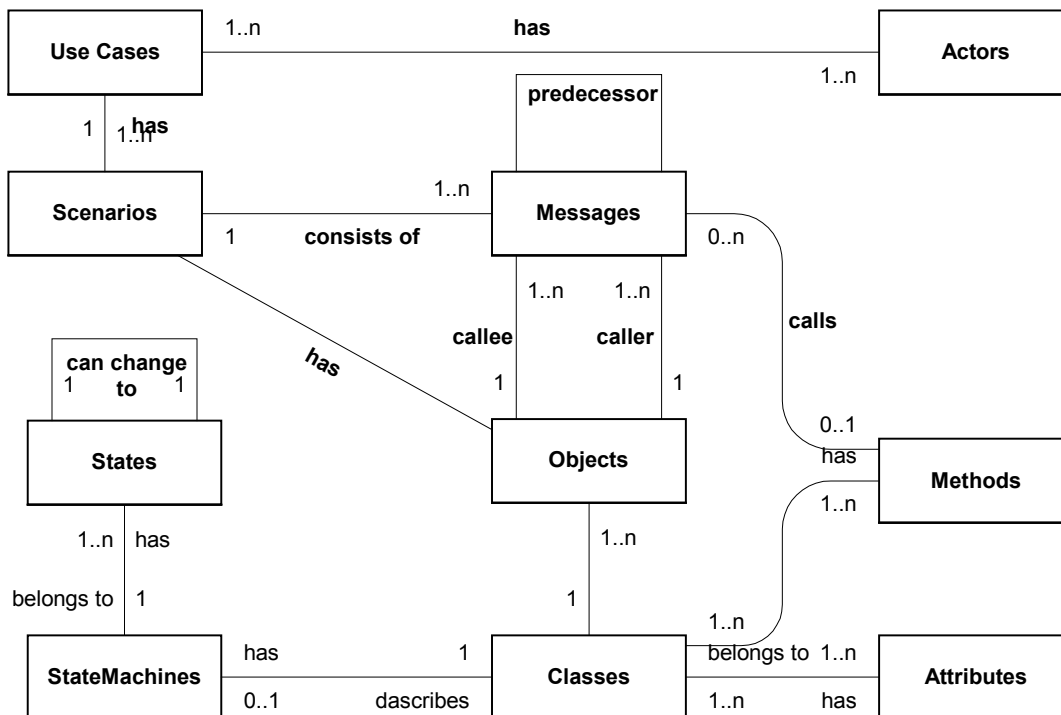


Figure 5. Relational meta model.

Private attributes. A class' attributes should be declared private. Otherwise the concept of encapsulation does not hold.

$$(\forall c \in C : (\forall a \in A \wedge (c.idc, a.ida) \in CA : a.visibility = 'private'))$$

Data records. Classes with only attributes and no methods are data records in disguise and therefore they do not comply with the paradigm of object-orientation.

$$(\forall c \in C : (\exists m \in M : (c.idc, m.idm) \in CM))$$

Objects must have a name. Each object in a scenario must have a name. A named object is much more expressive and understandable than an unnamed one. Especially in the case that several objects of the same type occur in one scenario it is essential to name objects.

$$(\forall o \in O : o.name \neq \perp)$$

3.5 Inconsistency

Consistency is compliance between different views or diagrams of a design. The various diagrams of a design contain overlapping information. This overlapping information is a possible source of inconsistencies, as the overlapping information might not 'match'. For two diagrams to be consistent, they must at least satisfy certain consistency rules, which will be described here.

Messages must correspond to methods. Each message received by an object in a scenario must correspond to a method in the object's class interface.

$$(\forall e \in E : (\exists m \in M : e.method = m.idm))$$

Messages only between related classes. In class diagrams, messages should only be passed between classes that are related to each other by means of associations of dependencies. This law corresponds to the Law of Demeter in object-oriented programming ("talk to your friends only").

$$(\forall e \in E : (class(e.caller), class(e.callee) \in As \vee e.caller = e.callee))$$

Methods must be called in MSC. The (public) methods provided by a class in a class diagram must be called in message sequence charts to depict the interaction of classes. Non-called methods are either questionable in their purpose or their (interactive) functionality is not sufficiently described in the design.

$$(\forall m \in M : (\exists e \in E : e.method = m.idm))$$

When we consider Figure 5, we observe that there is also consistency needed between different abstraction levels (hence different phases of the project).

For the analysis of the inter-abstraction-level-consistency we introduce the following metaphors.

Leaf. With the term leaf we depict the situation that an architectural element exists in a specific level, but there is no element of a later phase (or more detailed abstraction) which is related to it. In Figure 5 the use case UC3 is an example for this situation. A leaf clearly indicates a dead end and therefore a "forgotten branch" that results in not implemented requirements.

Orphan. An orphan is the opposite of a leaf: an architecture element that occurs at some level of the design and is not related to any earlier or higher abstraction level. Of course, on the requirements level, by definition we only have orphans. If we have an orphan (like Class6 in Figure 5) we have a lack of documentation, as it is not clear why the architecture element is created.

Commonality, overlapping. We can observe that the requirements R1 and R2 have a common use case UC4. This situation depicts an overlapping in the requirements. UC4 is possibly too coarse and might be divided in a common part and two separate parts.

Synonym, homonym, matching. Especially when we contemplate inter-abstraction-level consistency of the same diagram type we have to deal with matchings. Refinement is an example, where one high level element matches several lower level elements. A matching is established by relating elements of different abstraction layers with each other. If this is done by naming conventions (or usage of other identifiers) the relations can be analyzed by an automated tool. The automated analysis is problematic in situations where we have to deal with synonyms and homonyms. A synonym is the usage of different terms for the same entity, which results in an undetected match. The opposite is the homonym, where the same term is used for different entities. In this situation match is detected, that does not exist (i.e. is not intended to be a match).

3.6 Incompleteness

We approach completeness in a similar way as we have approached consistency: if two diagrams of a design are overlapping, their overlapping part is complete if all elements of one diagram in the part have a matching counterpart in the overlapping part of the other diagram.

Here we present some examples of our rules and metrics that help identifying incomplete spots, i.e.

elements in overlapping parts of diagrams that do not have a counterpart in the other diagram.

Classes must occur in MSCs. If a design contains a class, that does not occur as objects in a message sequence chart, the class is either redundant (under the assumption that the entire functionality is described in MSCs) or the interaction of the design's classes is not completely described using MSCs.

$$(\forall c \in C : (\# s \in S : (\exists e \in E : e.ids=s.ids \wedge (class(e.caller)=c \vee class(e.callee)=c))) > 0)$$

Classes must have methods. A violation of this rule is also a violation of the object-oriented paradigm, in particular the concept of encapsulation. A class without methods cannot interact with other classes and is therefore not complete. To make the class complete the designer has to define the class' methods and describe the interactions in message sequence charts.

$$(\forall c \in C : (\exists m \in M : (c.idc, m.idm) \in CM))$$

Highly dynamic classes must be described by a state diagram. State diagrams describe the internal dynamic behaviour of a class. In most situations it is not necessary to define state diagrams for all classes, as most classes don't show a very dynamic behaviour. We introduce the metric "dynamicity" that measures the number of incoming and outgoing messages for each class in all message sequence charts of the design. Because messages are usually indications for state transitions, this method helps identifying classes with a very dynamic behaviour. For a design to be complete the identified classes must be described by a state diagram.

$$(\forall c \in Dyn(X) : (\exists d \in D : d.idc = c.idc) \text{ where } Dyn(X) = \{ c \mid c \in C \wedge X \leq (\# e \in E : (\exists o \in O \wedge c.idc = o.idc : o.ido = e.caller)) + (\# e \in E : (\exists o \in O \wedge c.idc = o.idc : o.ido = e.callee)) \}$$

We do not expect that any project requires that all these rules hold. However, counting the number of inconsistencies and incompletenesses can give an indication of the progress or quality of the design.

4. Experimental Validation

To validation our ideas we are currently applying our techniques to a number of industrial case studies. We are especially interested in investigating whether the observed characteristics are bound to a specific domain, or attribute of the development process, or whether characteristics are general and applicable in

different situations. Therefore we have chosen three different suppliers for or cases. We will briefly introduce the various designs:

Design A1 and A2. These are two large scale (108 and 168 classes) industrial systems from our first supplier. A1 and A2 are two design-level models of different subsystems of a single version of a system. They are developed by different teams.

Design B. This design is provided by another industrial partner from a different application domain. The design is a high level analysis model of a large scale industry project with 34 classes.

Design C. This design is supplied by a group of post-graduate M.Sc. students in advanced software engineering. The design is a design-level model of a project the group performed at an industrial company. This design describes a prototyping framework. Its size in terms of classes is 75.

Table 1 gives an overview of the some basis size metrics of the cases.

Table 1. Basic size metrics.

	<i>A1</i>	<i>A2</i>	<i>B</i>	<i>C</i>
<i>Use Cases</i>	0	21	32	11
<i>Objects</i>	200	544	297	103
<i>Classes</i>	108	168	34	45
<i>Methods</i>	340	406	1	142
<i>Messages</i>	613	853	705	210
<i>Class Associations</i>	161	254	30	48

4.1 Similarities in Metrics Results of Different Designs

In section 3 the three categories of design-faults we are dealing with were given. Here we present the results of the four case studies for each of the presented categories.

4.1.1 Well-formedness

The rules in this category address the well-formedness of diagrams.

In message sequence charts the understandability of the object's roles is emphasized by assigning names to the objects (this is especially important if several instantiations of one and the same class occur in a message sequence chart). In the industrial cases A1, A2 and B this rule is strongly violated, only the student project has a reasonable result with only one fifth of the objects without name.

Table 2. Well-formedness rules.

	<i>A1</i>	<i>A2</i>	<i>B</i>	<i>C</i>
Objects without name	52.00%	61,58%	91.92%	25.24%
Classes without methods	60.19%	51.19%	100.00%	20.00%
Interfaces without methods	8.82%	9.38%	N/A	60.00%
Abstract classes in MSC	5.56%	0.60%	0.00%	0.00%
Public Attributes	67.23%	5.08%	0.00%	0.00%

According to the object-orientation paradigm, classes and interfaces are entities that provide data and functionality, therefore they should provide methods. In the industrial projects at least 90% of the interfaces have methods. In contrast, in cases A1 and A2 more than half of the classes do not have methods.

The rule “Abstract classes should not be instantiated in message sequence charts” is adopted from object-oriented programming. In UML designs, this rule is not always used. The results show that in spite of that the developers are modeling conforming to this programming-level rule.

Attributes contain the data of objects. According to a basic concept of object-oriented programming – encapsulation – the data should only be accessed using the object’s methods, therefore attributes should be declared private. Except for A1 all cases adhere to this rule. It is remarkable, that there is such a big difference in adherence to this rule in one and the same project (A1 and A2).

4.1.2 Inconsistency

Table 3. Inconsistency rules.

	<i>A1</i>	<i>A2</i>	<i>B</i>	<i>C</i>
Messages without Name	0.00%	0.00%	0.28%	0.00%
Messages without Method	58.73%	7.62%	100.00%	27.14%
Messages between unrelated Classes	71.94%	41.03%	77.73%	81.90%

The rules presented here indicate inconsistencies in UML designs. A message in an MSC without a name is an inconsistency that could also be considered as an incompleteness (refer to the example given in the introduction). None of the case studies violate this rule.

The rule “messages without method” addresses the fact, that messages in an MSC must correspond to methods in the class diagram. Here the ideal case is to get a low score, which means that most messages nicely correspond to the methods. Case B does not make use of methods, therefore it is obvious that it does not conform to this rule. Again we have a remarkable discrepancy between two cases of the same project (A1 and A2).

The same is valid for the third rule: messages should only be passed between related (in class diagram associated or dependent) classes. This rule is the object-oriented Law of Demeter shifted to design level. As it is a very strict rule, it is not very surprising, that three of the cases violate this rule in about 75% of all cases. Therefore it is remarkable that A2 has such a positive score.

4.1.3 Incompleteness

Table 4. Incompleteness rules.

	<i>A1</i>	<i>A2</i>	<i>B</i>	<i>C</i>
Classes not called in MSC	61.11%	59.52%	35.29%	42.22%
Interfaces not called in MSC	100.00%	87.50%	100.00%	70.00%
Methods not called in MSC	67.65%	77.59%	N/A	40.14%

The rules in this section deal with the completeness of a design. The purpose of message sequence charts is to illustrate the interaction of the system components. Therefore it is desirable that all classes defined in the design are incorporated in at least one MSC to depict its functional interaction. Hence, the metric „classes not called in MSC“ should tend to zero. We observe that in the best case, still 35% of all classes are not incorporated in MSCs. We have a strong similarity between A1 and A2 (from the same project).

For interfaces a similar discussion applies as for abstract classes: in object-oriented programming they cannot be instantiated, but in UML modelling it is allowed. The results however show, that the developers of the design stick to the rule from object-oriented

programming. This is not very surprising, as the system eventually will be implemented in an object-oriented programming language.

Case B has no methods, this is why the last rule does not apply to it. The rule says that (public) methods should be called in MSCs. If not, either the MSC is incomplete, or the class diagram specifying the method is overspecifying functionality (which is eventually not needed for the system). The designers of the student project C adhere as expected more to this rule than the industrial designers.

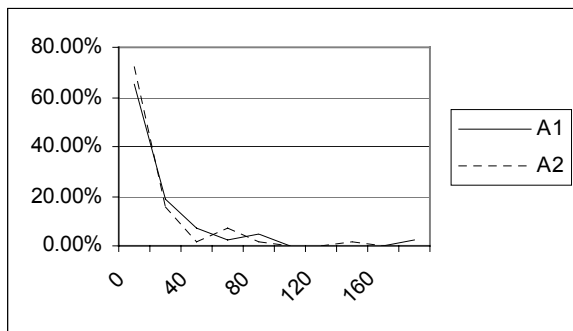


Figure 6. Design A1 & A2.

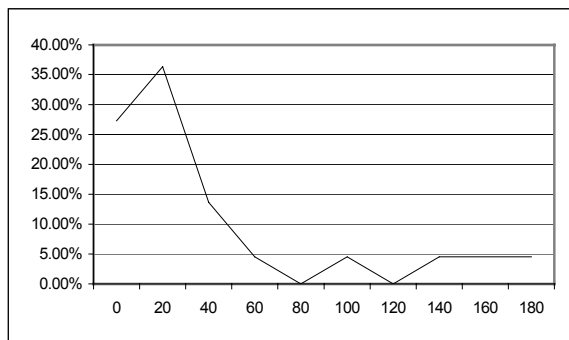


Figure 7. Design B.

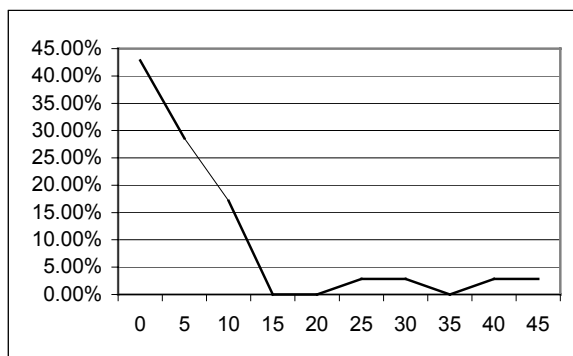


Figure 8. Design C.

4.2 Dynamicity Threshold

In addition to strict completeness rules, we also consider completeness heuristics. For example: „Classes with high dynamicity must be described by a state machine“. We calculated the dynamicity as described in section 3 and analyzed the distribution of dynamicity metrics over all classes of a design.

Figures 6, 7 and 7 display the distribution of dynamicity of all five designs. The dynamicity is given on the x-axis and on the y-axis the percentage of classes is given. The diagram shows the percentage of classes with a given dynamicity.

We observe that there are strong similarities between the distributions of five designs. The first, is the large number of classes with a low dynamic behavior. In the middle of the dynamicity-scale, the curve tends to reach 0 (marked by the arrow) and in the higher dynamicity values, there is a (relatively small) peak of classes with a highly dynamic behavior. The heuristic that we propose is that classes with a higher dynamicity value than the dip of the curve must be described by a state diagram. The threshold above which one would like to apply this rule, can either be identified by visualizing the curve as we did here or using a statistical formula.

5. Concluding Remarks

In this paper we presented techniques for the quantitative assessment of inconsistency and incompleteness in UML designs. Using these techniques we performed a number experiments based on industrial case studies. From these experiments we observe that:

- Quantifying inconsistencies and incompleteness provides insight into the use of UML.
- The absolute number of inconsistencies in UML designs is quite large (although no reference numbers have been established yet).
- The types of inconsistencies appear strongly related to the habits and conventions used by the designers.
- In industrial practice, designs are moved into implementation stage while there are still significant numbers of inconsistencies in a design.
- Design C shows the least inconsistencies. Possible factors for this are:
 - o This case is developed with lesser time pressure than regular industrial projects.
 - o This case is developed by people who learned UML at university whereas the other cases are designed by people who learned UML in practice.
- The degree of inconsistency of a design should be seen in the context of the completeness of a design.

- The cases show a remarkable similarity in distribution of dynamicity of classes.

For future work we aim to investigate to what degree inconsistencies are a ‘bad thing’; i.e. whether inconsistencies actually have a negative effect on software quality? To this end, we will research the relation between inconsistencies in modules of large systems and the number of problem reports for that module.

Acknowledgements

We would like to thank G. Florijn, J. Warmer, G. Schouten, F. Jacobs and J. Xu for useful discussions.

6. References

- [1] E. Boiten, H. Bowman, J. Derrick and M. Steen, *Viewpoint consistency in Z and Lotos: A case study*. In Proceedings of 4th Symposium of Formal Methods Europe, LNCS 1313, pp. 644-664, 1997
- [2] L.B.V. Basili and W. Melo, *A validation of object oriented design metrics as quality indicators*, IEEE Tr. SE, Vol.22, no. 10, pp. 751-761, 1996
- [3] S. Chidamber, and C. Kemerer, *A Metrics Suite for Object-Oriented Design*, IEEE Tr. On SE, vol. 20, no. 6, pp. 476-493, 1994
- [4] Workshop on Consistency Problems in UML-based Software Development, October 1, 2002, Dresden, Germany, <http://www.ipd.bth.se/uml2002/>
- [5] G. Engels, J. H. Hausmann, R. Heckel. S. Sauer. *Testing the Consistency of Dynamic UML Diagrams*. IDPT 2002, Pasadena, California. June 2002.
- [6] Fenton, N.E. and S.L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*, 2nd ed., ITP Press, 1997
- [7] B. Henderson Sellers, *Software Metrics*, Prentice-Hall, 1996.
- [8] B. Hnatkowska, Z. Huzar, J. Magott, *Consistency Checking in UML models*, 4th Int. Conf. on Information Systems, Modeling ISM'01, 2001.
- [9] J. Muskens. *Software Architecture Analysis Tool*. Master's Thesis, Eindhoven University of Technology, Faculty of Mathematics and Computing Science, April 2002.
- [10] L. Nenonen, J. Gustafsson, J. Paakki, A. I. Verkamo, *Measuring object-oriented software architectures from UML diagrams*; In: *Proc. 4th Intl. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, France, June 2000, 87-100.
- [11] *Correct Development of Real-Time Embedded Systems* Project <http://www-omega.imag.fr/>
- [12] *The Precise UML Group*. <http://www.puml.org/>
- [13] K. Scott. *UML Explained*. Addison Wesley, 2001.
- [14] M. Shroff and R.B. France, *Towards a formalization of UML Class structures in Z*, Proc. of the 21st COMPSAC conference, pp. 646-651, IEEE CS Press, 1997.
- [15] W.E. McUumber and B.H.C. Cheng, *A General Framework for Formalizing UML with Formal Languages*, IEEE, 2001.
- [16] *Object Management Group*. <http://www.omg.org>
- [17] L. Kuzniarz, G. Reggio, J. L. Sourrouille, Z. Huzar. *Workshop Materials: Workshop on Consistency Problems in UML-based Software Development*. UML 2002. Blekinge Institute of Technology (Ronneby, Sweden), Research Report 2002:6
- [18] J. Warmer and A. Kleppe, *The Object Constraint Language: Precise Modelling with UML*, Addison-Wesley, 1999

Extending the UML Metamodel to Support Software Refinement

Wuwei Shen, Yan Lu, and Weng Liong Low
Department of Computer Science, Western Michigan University
{wwshen,ylu,wllow}@cs.wmich.edu

Abstract

With modern software development being a complicated process, refinement has become an unavoidable process in software development. However, supporting refinement process during software development has not received much attention in the research community. In this paper we present a set of rules to support software model refinement based on expanding the UML metamodel. We use a stereotype, which is one of the UML extension mechanisms, to represent these refinement rules. These rules represented by stereotypes can be easily included in the UML metamodel, and so helping software developers find some inconsistencies between two models can be achieved when a model is compared with the UML metamodel. An ATM example is illustrated in this paper to show how the refinement rules work.

1 Introduction

With the Unified Modeling Language (UML) [5] becoming a standard object-oriented modeling language, it has become a trend for software developers to deploy UML to design a software system. Thus, UML has aroused a lot of research attention about how to effectively apply UML to software development. Most research work [1, 3, 6, 7] is based on support of one software model from different views during software development. But refinement plays an important role in modern software development. No software system can be produced in a single step from problem analysis to implementation through design phase. More precisely, a software development process has become more iterative. In view of the importance of refinement, this paper will present a way to extend the UML metamodel to support model checking at different levels during software development.

UML is defined in terms of a four-layer architecture. The M0-layer defines a specific information model, which is an instance of a software model (M1-layer) designed by software developers. Any M1-layer model is regarded as an instance of the UML metamodel, i.e. an M2-layer model

in the four-layer architecture. The UML metamodel is defined by a set of class diagrams together with some well-formedness rules so any M1-layer model can be checked for syntactic correctness. It is the UML metamodel that makes it possible for software developers to find some errors in one model.

However, software development is an iterative process and one model only represents a software development status at one time. Only the lowest-level model representing a software system written in some object-oriented language is the final goal for software developers. It always takes many refinement steps for software developers to derive a final model. In other words, with the process going on, one model will be replaced by another refined model. Checking whether a refined model satisfies the previous level model has become an important issue in the software research community. In particular, an automatic tool to support a refinement process is of great interest to software developers. Unfortunately, the UML metamodel does not deal with any refinement process. In this paper, we will tackle this problem by extending the UML metamodel.

Although there are many diagrams in UML, class diagrams play an elementary role in a software model and they give a ground model for a software system. Our support for software refinement is based on class diagrams. Since more details are considered during software development as a refinement process continues, a refined class diagram has a more complicated structure than its predecessor which serves as a backbone for refinement. In this paper, we will present a set of rules which can be used to check whether a refined class diagram satisfies its predecessor diagram. Therefore, software developers can reconsider their software models if the consistency between two models is not satisfied.

In order to make a tool support software refinement, we extend the UML metamodel by introducing a set of rules for class diagram refinement. There are two ways to extend the UML metamodel. One is called lightweight built-in extension, which extends UML model elements with new semantics using stereotypes, tagged values and constraints instead of introducing some new elements in the metamodel. The

other is called heavyweight built-in extension where new elements are added to the metamodel to achieve new semantics. Since there are many inconsistencies in UML and the introduction of new elements can bring about more inconsistencies, we decide to use lightweight built-in extension to represent our refinement rules.

There is some research work about class diagram transformation [3, 7]. Compared with these existing research work, we concentrate on the transformation based on forward engineering. As a software refinement process continues, more requirements from the problem domain are considered and so more elements are added to a class diagram. Therefore, some inconsistencies can be introduced, and helping software developers find these inconsistencies between two models at the two consecutive levels has become extremely important. To achieve this goal, we extend the UML metamodel based on a set of refinement rules. Extension to the UML metamodel can also benefit tool developers to build a such tool supporting refinement model checking.

The paper is organized as follows. Section 2 briefly introduces the UML extension mechanism and proposes a set of refinement rules. An ATM example is shown in Section 3 to illustrate the application of the refinement rules. We draw some conclusion in section 4.

2 UML Extension Mechanism and Refinement Rules

Before introducing the refinement rules, we briefly introduce the UML's main extension mechanism, i.e. stereotypes.

2.1 UML Extension Mechanism: Stereotypes

Although UML is a very complicated modeling language used to design different views of a software system, it is impossible for software developers to deploy UML to all different domain-based software systems. However, UML does provide a mechanism to have different domain-based system developers extend the UML notation using the extension mechanism. Among the techniques used in the extension mechanism, stereotypes are the most powerful way to extend the UML metamodel. A stereotype can be defined by either using the pre-defined *stereotype* to show the dependency between a new defined stereotype and its base class or listing a table where a stereotype, its base class and some other optional constraints are given.

Since the stereotype dependency clearly shows a newly defined stereotype, we prefer this graphical notation to the tabular structure to give new stereotypes. According to [2], a new user-defined stereotype consists of the name, description, tagged values and constraints besides the graphical no-

tion. In the following we follow this pattern to extend the UML metamodel.

2.2 Rules for Refinement

Software refinement means that more and more details will be added to a given set of software elements over the time. It is similar to using a magnification glass to look over an object. With software development going further, more information about a software system becomes visible and close to software developers. However, checking whether a lower-level model still satisfies its higher-level model is becoming an interesting topic.

Although a software model consists of several different views, each of which is represented by one or more diagrams in UML, the static view, especially represented by a class diagram, plays an important role in software development and it provides a basic structure for a software system.

A class diagram in UML provides some basic information for a software model. It describes a static structure of a software system. In a class diagram, there are two kinds of elements, i.e. class and relationship. A relationship consists of generalization, association and dependency. An association can be further divided into a unidirectional association, bidirectional association, aggregation, or composition.

Based on the structure of a class diagram, refinement of a class diagram can be divided into two categories. One is class refinement and the other relationship refinement. Most research work deals with them separately. However, we think that a relationship refinement in a class diagram plays a main role and in most cases the class refinement can be inferred from the corresponding relationship refinement. In other words, if we dive into a relationship between two classes in a class diagram, then we can add more relationships and classes, sometimes called helper classes, to the original class diagram. Our refinement rules are based on relationships in a class diagram.

The relationship refinement in a class diagram can result in a mapping between a single relationship in a higher-level class diagram and more than one relationship in a lower-level class diagram, denoted by a one-to-many mapping. However, we treat a one-to-many mapping as a set of one-to-two mappings in order to make it easy to check the inconsistency problem between models at two levels.

While some research work investigated model transformation by presenting some rules in their own notation, we think the best way to present these rules for inconsistency checking is to embed these rules into the UML metamodel. The UML metamodel defines the syntax and semantics for representing a software system using a variety of diagrams in UML so it is possible for software developers to find some errors in their software system. As one kind of errors in software development, the inconsistency problem

between models at two different levels should be addressed in the UML metamodel. During software development, any inconsistency between two levels of diagrams should be reported to software developers as early as possible. Therefore, the inconsistency problem should be checked at the same time when a diagram is compared with the UML metamodel.

As mentioned before, there are two kinds of extension mechanisms in UML. In view of the importance of making the fewest changes to UML, we adopt the lightweight built-in extension mechanism to represent our rules. As a continuous effort to find errors in a software model, we compare class diagrams at two different levels during software development with the extended UML metamodel, trying to find as many errors as possible. In the following we give these rules based on the lightweight built-in extension mechanism.

The lightweight built-in extension includes defining a stereotype, constraint and tag value. A stereotype is a user-defined metaelement which is used to extend an existing UML metaelement, called a base class. Since a stereotype spans two different levels in a model system, we need a special mechanism to declare how a stereotype is related to a UML metaelement. In this paper, we use a graphical notation, i.e. using a special dependency called `<< stereotype >>` to show a stereotype and its base class.

1. Generalization: At level one, we assume that class A is a subclass of class B. Then at level two which is a refined level of level one, the generalization can be refined into two generalizations with the helper class C. But the relationship between the class A and B at level two should still be kept. Fig. 1 shows the rule for generalization abstraction.

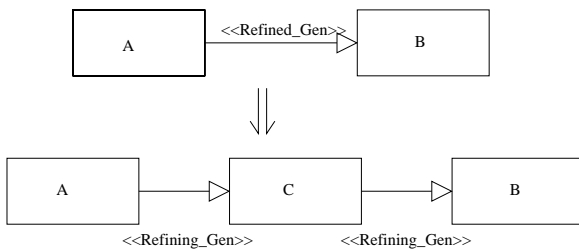


Figure 1. A rule for generalization abstraction.

Fig. 2 gives the extended UML metamodel to support generalization abstraction by defining two new stereotypes: **Refined_Gen** and **Refining_Gen**.

Description: The stereotype **Refined_Gen** is used to represent a generalization, defined in a higher-level class diagram, which will be refined in the next level class diagram.

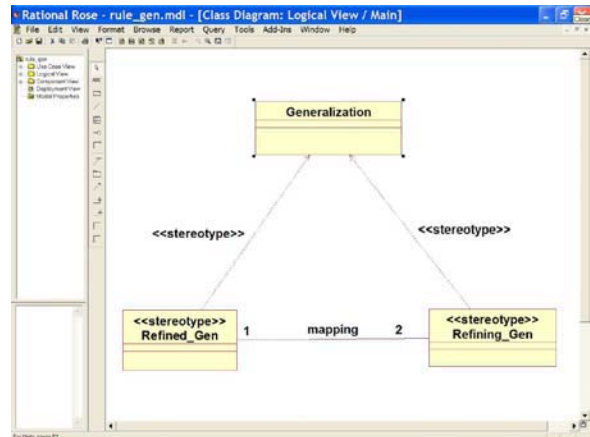


Figure 2. Stereotypes for generalization abstraction based on the rule shown in Fig 1.

Constraint: The stereotype **Refined_Gen** should satisfy the restrictions between itself and its corresponding refining generalizations.

- The refined generalization should be refined to a generalization followed by another generalization, and
- The child of the refined generalization should correspond to the child of one of the refining generalizations, and
- The parent of the refined generalization should correspond to the parent of the other refining generalization.

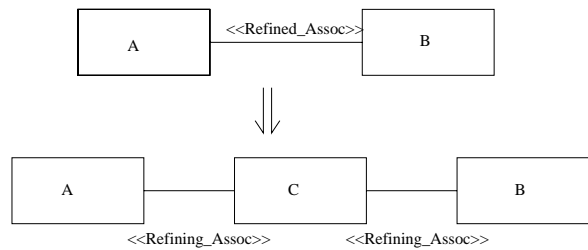


Figure 3. A rule for association abstraction.

Then the OCL constraint for Stereotype **Refined_Gen** is as follows:

$self.mapping \rightarrow exists(x,y|x \langle \rangle y \text{ and } self.child = x.child \text{ and } self.parent = y.parent \text{ and } x.parent = y.child).$

Description: The stereotype **Refining_Gen** is used to denote a generalization defined in a lower-level class

diagram which is used to refine a generalization in its higher-level class diagram.

Constraint: Since the constraint is related to the stereotype Refined_Gen and already given in Refined_Gen, there is no constraint for the stereotype Refining_Gen.

2. Bidirectional Association: At level one, we assume that there is an association between class A and B. The rule is shown in Fig. 3.

To represent the above rule, we define stereotypes as shown in Fig. 4.

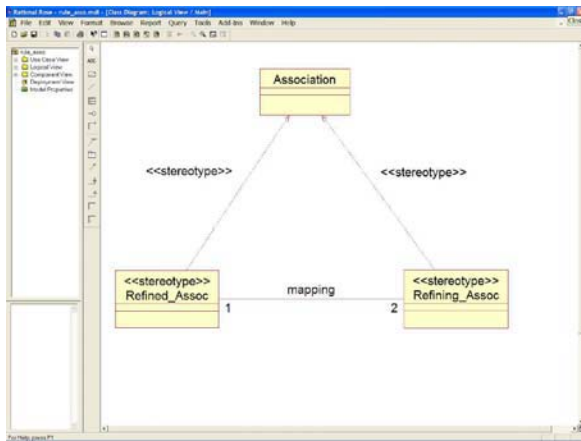


Figure 4. Stereotypes for bidirectional association abstraction based on the rule shown in Fig. 3.

*Description:*The stereotype Refined_Assoc is used to represent an association defined in a higher-level class diagram which will be refined in the lower-level class diagram.

Constraint: The stereotype Refined_Assoc should satisfy restrictions between itself and its corresponding refined associations.

- The refined association should be refined to an association followed by another association, and
- The two end classes on the refined association correspond to the two classes at both ends of the two refining associations, and
- Both refining associations are bidirectional associations.

Then the OCL constraint for Stereotype Refined_Assoc is as follows:

$self.associationEnd \rightarrow forall(p | self.mapping.associationEnd \rightarrow exists(x | x.classifier = p.classifier))$ and $self.mapping \rightarrow forall(x, y | x \langle \rangle y$ and $x.associationEnd \rightarrow exists(a$

$| y.associationEnd \rightarrow exists(b | a.classifier = b.classifier)$ and $self.associationEnd \rightarrow collect(classifier) \rightarrow excludes(a.classifier))$ and $self.mapping.associationEnd \rightarrow forall(z | z.isNavigable = true)$

Description: The stereotype Refining_Assoc is used to denote an association, defined in a lower-level class diagram, which is used to refine an association in its corresponding higher-level class diagram.

Constraint: Since the constraint is related to the stereotype Refined_Assoc and already given in Refined_Assoc, there is no constraint for the stereotype Refining_Assoc.

3. Unidirectional Association: In this case we only consider a unidirectional association. At level one, we assume that there is a unidirectional association from class A to class B. There are several cases which can refine the unidirectional association. Fig. 5 presents the first case and we define stereotypes in Fig. 4.

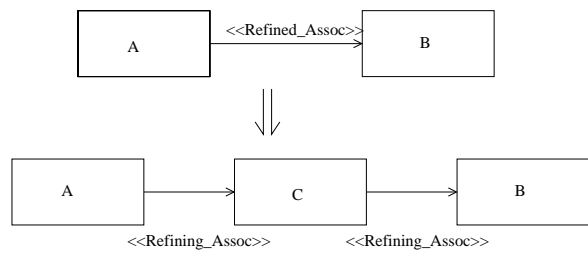


Figure 5. A rule for unidirectional association abstraction.

Description: The stereotype Refined_Assoc is the same as the stereotype Refined_Assoc defined for the bidirectional association.

Constraint: Besides the constraint given in the bidirectional association, we should add the restriction on navigable property for association ends, which is shown in the following.

- The refined association should be refined to a unidirectional association followed by another unidirectional association, and
- The class at the refined association at the navigable end should correspond to the class at one of the refining associations at the navigable end, and
- The class at the refined association at the non-navigable end should correspond to the class at the other refining association at the non-navigable end.

The the OCL for Stereotype Refined_Assoc is as follows:

```
self.associationEnd->forall(p|self.mapping.associationEnd->exists(x|x.classifier=p.classifier and x.isNavigable=p.isNavigable)) and self.mapping->forall(x,y|x<>y and x.associationEnd->exists(a|y.associationEnd->exists(b|b.classifier=a.classifier and self.associationEnd->collect(classifier)->excludes(a.classifier) and b.isNavigable<>a.isNavigable)))
```

Description: The stereotype Refining_Assoc is the same as the stereotype Refining_Assoc defined in the bidirectional association.

Constraint: No constraint for the stereotype Refining_Assoc.

Fig. 6 denotes the second rule to refine a unidirectional association and Fig. 7 gives the declarations for stereotypes.

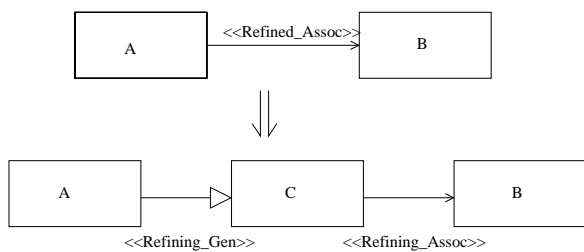


Figure 6. Another rule for unidirectional association abstraction.

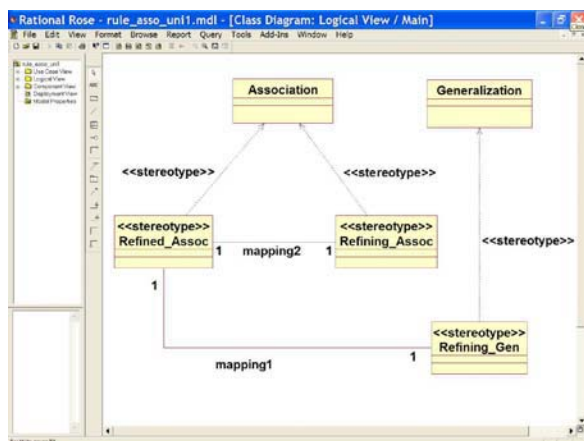


Figure 7. Stereotypes for unidirectional association abstraction based on the rule shown in Fig. 6.

Description: The stereotype Refined_Assoc is the

same as the stereotype Refined_Assoc defined in the section of bidirectional association.

Constraint: Any refined association in a higher-level class diagram should satisfy the following conditions:

- The refined association should be refined to a generalization followed by a refining association, and
- The child of the generalization should correspond to the class in the refined association at the navigable end, and
- The class at the refining association at the non-navigable end should correspond to the one at the refined association at the non-navigable end.

Then the OCL constraint for Stereotype Refined_Assoc is as follows:

```
self.associationEnd->exists(x|x.isNavigable=false and self.refining_Assoc.associationEnd->exists(y,z|y<>z and y.isNavigable=false and x.classifier=y.classifier and self.associationEnd->collect(classifier)->excludes(z.classifier) and self.associationEnd->exists(a|a.isNavigable=true and self.refining_Gen.child = a.classifier and z.classifier = self.refining_Gen.parent)))
```

Description: The stereotype Refining_Assoc is used to refine an association in a higher-level class diagram.

Constraint: There is no constraint for the stereotype Refining_Assoc.

Description: The stereotype Refining_Gen is used to refine an association in a higher-level class diagram.

Constraint: There is no constraint for the stereotype Refining_Gen.

Due to space, other rules related to a unidirectional association are also skipped here and interested readers are referred to [4].

4. Association With an Composition: At level one, we only consider the case where class B has an composition and class A does not. There are two cases which can refine this association. The first rule for the association composition is shown in Fig. 8 and the corresponding stereotype declaration is shown in Fig. 4.

Description: The stereotype Refined_Assoc is used to represent a refined association with an composite end in a higher-level class diagram.

Constraint: The constraint for an association composition is similar to the one for the unidirectional association, but we replace the navigable property with the composition property.

- The refined association should be refined to an association followed by another association, and

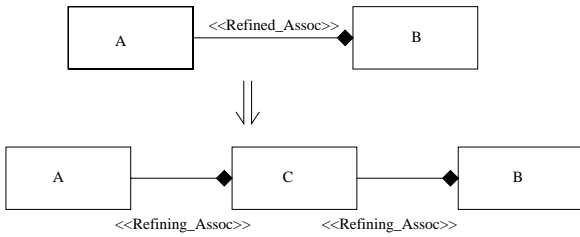


Figure 8. A rule for association composition abstraction.

- The class at the refined association at the composition end should correspond to the class at one of the refining associations at the composition end, and
- The class at the refined association at the non-composition end should correspond to the class at the other refining association at the non-composition end.

The the OCL constraint for Stereotype Refined_Assoc is as follows:

```
self.associationEnd->forall(p|self.mapping.associationEnd->exists(x|x.classifier=p.classifier and x.aggregation=p.aggregation)) and self.mapping->forall(x,y | x <> y and x.associationEnd->exists(a | y.associationEnd->exists(b|b.classifier=a.classifier and ((b.aggregation=#composite and a.aggregation=#none) or (a.aggregation=#composite and b.aggregation=#none)))) and self.associationEnd->collect(classifier)->excludes(a.classifier))
```

The second rule for composition refinement is shown in Fig. 9.

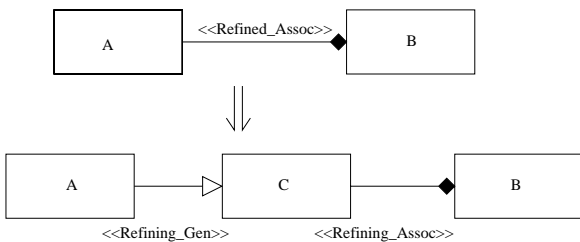


Figure 9. Another rule for association composition abstraction.

The stereotype declaration for the second rule is given in Fig. 7. To reflect the composition end at an association, we replace the navigable property with an composition. Due to space, we omit the OCL constraint and all other rules are also skipped. Interested readers are referred to [4].

3 Case Study: ATM Example

In this section, an ATM example is shown to illustrate an application of the refinement rules presented above.

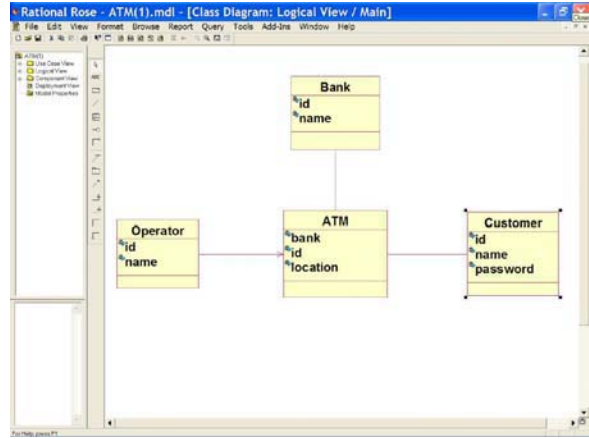


Figure 10. An abstract class diagram for the first (highest) level model.

In the highest level, we can abstract the description as follows:

An ATM machine will serve one customer at a time. Based on a customer requests, the ATM will communicate with the bank's computer over an appropriate communication link. Also the ATM will allow an operator to start and stop the service for customers.

Based on the above informal description, we find four major classes and a class diagram can be designed in Fig. 10.

Now let us refine the association relationship between the class Customer and ATM based on the following problem description:

The ATM machine has a magnetic stripe reader for reading an ATM card. A customer will be required to insert an ATM card and enter a personal identification number. The customer will then be able to perform one or more transactions through a console connected to the ATM machine.

Fig. 11 is the refined class diagram. In this diagram, we add class CardReader, Card and Console as helper classes. When we apply our rules, there is no inconsistency which can be found.

Now we consider another relationship between class ATM and Customer. It is the transaction that makes ATM machine provide a service to a customer.

The ATM machine has a slot for depositing envelopes, a dispenser for cash.

A transaction should be one of the following services provided by the ATM machine:

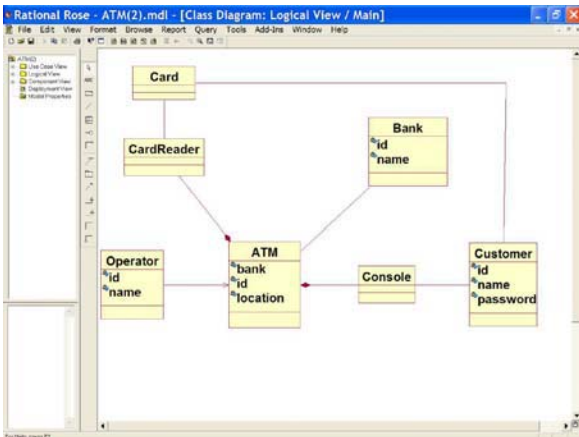


Figure 11. An abstract class diagram for the second level model.

- *Withdrawal:* A customer must be able to make a cash withdrawal from any suitable account linked to the card.
- *Deposit:* A customer must be able to make a deposit to any account linked to the card, consisting of cash and/or checks in an envelope. The customer will enter the amount of the deposit into the ATM, subject to manual verification when the envelope is removed from the machine by an operator.
- *Transfer:* A customer must be able to make a transfer of money between any two accounts linked to the card.
- *Inquiry:* A customer must be able to make a balance inquiry of any account linked to the card.

Fig. 12 is the refined class diagram based on the above description. We introduce a helper class Transaction to refine the previous relationship between class Customer and ATM. No inconsistency can be found based on our rules.

We further refine the association between the class Console and Customer as follows:

A console of the ATM consists of a keyboard and a display window. A customer can interact with the ATM machine through the display window and the keyboard. After the ATM machine verifies the ATM card and PIN number, the ATM machine will display all services available to the customer through the display window. A customer can start her transaction by pressing some key in the keyboard. The ATM machine will perform a transaction after she chooses some service based on the keyboard input. The customer can stop a transaction at any time by pressing ESC in the keyboard.

Fig. 13 is the refined class diagram. Again there is no inconsistency which can be reported according to our rules.

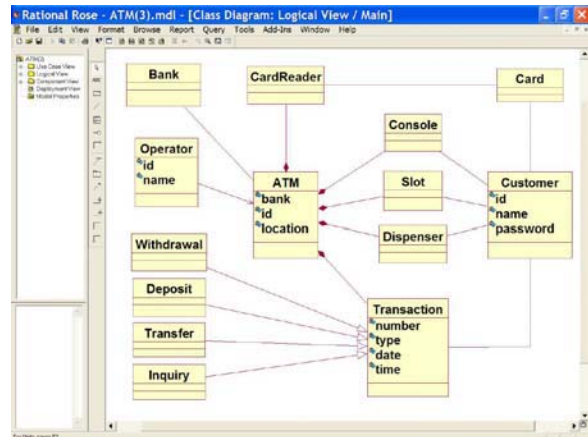


Figure 12. An abstract class diagram for the third level model.

The association between the class Operator and ATM can be further refined based on the following description.

An operator can start or stop the service for customers by using a key-operated switch.

The association between the ATM machine and Bank can also be refined based on the following description.

The ATM will communicate each transaction with the bank. A card can have several accounts whose types can be divided into checking and saving accounts. The ATM will send the corresponding data to the bank based on different services.

- *Withdrawal:* Check the account information, if there is enough money in the appreciate account then approval is issued and cash is dispensed.
- *Deposit:* A customer can deposit money to the ATM machine through an envelope. Also, the customer will enter the amount of deposit into the ATM machine, subject to manual verification when the envelope is removed from the machine by an operator. Temporal account information will be updated and real account information will not be updated until an operator checks the real account deposited by the customer.
- *Transfer:* A customer can move money from an account to another linked to a card.
- *Inquiry:* A customer can inquire any account linked to a card.

Fig. 14 is the refined class diagram based on the above description. But an inconsistency is reported because the unidirectional association between class Operator and ATM has been replaced by two bidirectional associations with the

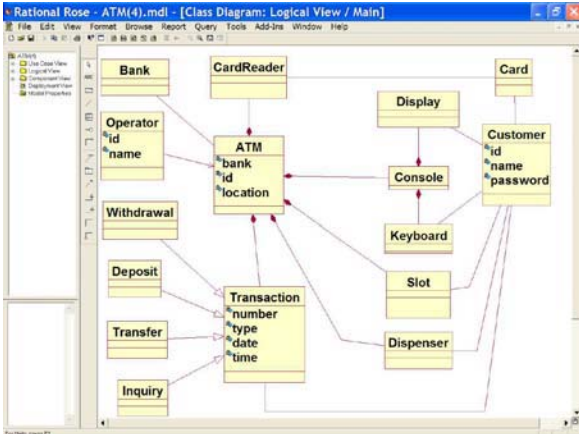


Figure 13. An abstract class diagram for the fourth level model.

helper class Switch. Some changes should be made in the model to keep the consistency.

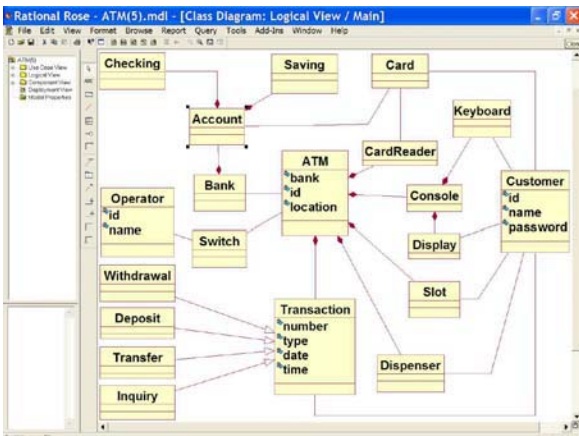


Figure 14. An abstract class diagram for the fifth (lowest) level model.

4 Conclusion

Refinement has become an inevitable step in software development. Keeping the consistency between any two consecutive models is crucial for software developers when they design a software system. Although the UML metamodel provides a way to check the syntactic correctness for a software model, it does not propose any consistency checking between two models. In this paper, we propose a set of rules based on class diagram refinement, especially

about the relationship refinement, to support model checking for software refinement.

Our contribution is not only to discover a set of rules to refine a relationship in a class diagram but to extend the UML metamodel using stereotypes to denote these rules. Since these rules are embedded into the UML metamodel, it is possible for UML tool developers to support model checking for software refinement in their tool. When software developers design a model for a software system, our mechanism can help the developers check whether the model satisfies its predecessor model besides checking the syntactic correctness of the model itself.

As an effort to help software developers find errors during the early phase of software development, we will build a tool to support these rules. We are expecting to explore more rules based on other UML diagrams to support model refinement. Moreover, we will look for some industry-sized applications to test our tool.

References

- [1] David Akehurst and Stuart Kent. A Relational Approach to Defining Transformations in a Metamodel. In *UML 2002-The Unified Modeling Language: Model Engineering, Concepts and Tools*, volume 2460 of *LNCS*, pages 243–258. Springer, 2002.
- [2] Jim Conallen. *Building Web Applications with UML*. Addison-Wesley, Reading, MA, 2000.
- [3] A. Egyed and N Medvidovic. A Formal Approach to Heterogeneous Software Modeling. In *Proceedings of the 3rd International Conference on Foundational Aspects of Software Engineering (FASE)*, Berlin, Germany, 2000.
- [4] Yan Lu, Weng Liong Low, and Wuwei Shen. Extending the UML Metamodel to Support Software Refinement(Extended Version). Technical report, Dept of CS, Western Michigan University, Aug, 2003.
- [5] OMG Unified Modeling Language Specification, version 1.3, June 1999.
- [6] Wuwei Shen, Kevin Compton, and James Huggins. Formalising UML state machines for model checking. In *Proceeding of 26th annual international computer software and applications conference, Oxford University*, pages 147–152, IEEE Computer Society, August 2002.
- [7] Jon Whittle. Transformation and Software Modeling Language: Automating Transformations in UML. In *UML 2002-The Unified Modeling Language: Model Engineering, Concepts and Tools*, volume 2460 of *LNCS*, pages 227–242. Springer, 2002.

A Pragmatic View about Consistency Checking of UML Models

Jean Louis SOURROUILLE Guy CAPLAT
INSA, Bat. Blaise Pascal
F69621 Villeurbanne Cedex, France
sou@if.insa-lyon.fr caplat@if.insa-lyon.fr

Abstract

Since the UML semantics is not formally defined, it is not possible to check model semantics exhaustively. This work gathers practical considerations from a UML user about model consistency. The UML language constraints are compared with the usual notions of syntactic and semantic rules to identify the main issues within an actual modeling process. Translation of UML models into formal languages as well as code generation is examined. Then actions to improve the consistency checking of models are proposed, from changes in the UML definition to practical modeling methods.

1. Introduction

During usual development cycles, models are successively built at a lower and lower abstraction level until the required level for coding is reached. Everybody knows for a long time that the cost of errors found early in the development cycle is vastly lesser than the cost of errors found in later steps such as coding or test. Therefore to verify UML models as soon as possible is a practical requirement. Numerous works are related to this question and search for a satisfactory response (see *Consistency Problems in UML-based Software Development* [18]).

Let $m(S)/f$ the model of the system S in the formalism f . In short m/UML is any model in the UML. To answer the question “Is m/UML consistent?” the UML syntax and semantics should be completely specified, i.e., there exists a model $m(UML)/F$, where F is a formal language. The UML metamodel MM is the description of the language, for instance using the MOF [9]: $MM = m(UML)/MOF$ (unlike the UML document [17] specifies, a metamodel is not a model of a model but a model of a formalism). The current metamodel MM is neither precise nor formal. As a result, it is not possible to check definitely UML models. This is the reason why numerous works aim to define a formal or precise metamodel MM (see precise UML [11]).

With a description of MM in a formal language F , all the wished checks could be done. This approach that seems theoretically perfect comes up against many practical obstacles: Is it possible to formally define UML? At which abstraction level? Is it advisable to do that? Otherwise, how to deal with checks whose usefulness is well known? This work tackles these issues and attempts to find practical and immediately usable solutions.

2. Consistency constraints and languages

The Fig. 1 (from [15]) recalls important relationships between the main notions in modeling. The term *Model* should be understood in the broad sense of a set of expressions describing a system from various points of view, not in the UML sense. A *System* is a part of a real or imagined domain. A model is an *Abstraction* of the system. Models are expressed in a *Formalism*, i.e., a set of primitive notions, represented by a concrete syntax (language of forms), whose semantics induces constraints on models. Notice that the modeled domain adds constraints, e.g., a Human cannot (still) have two fathers, leading to the need of the notion of *Constraint* among the set of abstract primitives.

2.1. Consistency and refinement

During the development process, different views of the system are produced. These views should be properly related to each other in order to form a consistent description of the system, even if they are at different level of abstraction. In other words, parts of models are built more or less asynchronously and consistency has to be checked within and between different temporal views on the system under construction.

These two types of consistency are named horizontal and vertical consistency [18], and intra-model and inter-model consistency in [7] (where a model is a part of the whole system model). *Horizontal* consistency means that all the created model elements respect all the syntactic and semantic UML rules. As all the model elements belong to

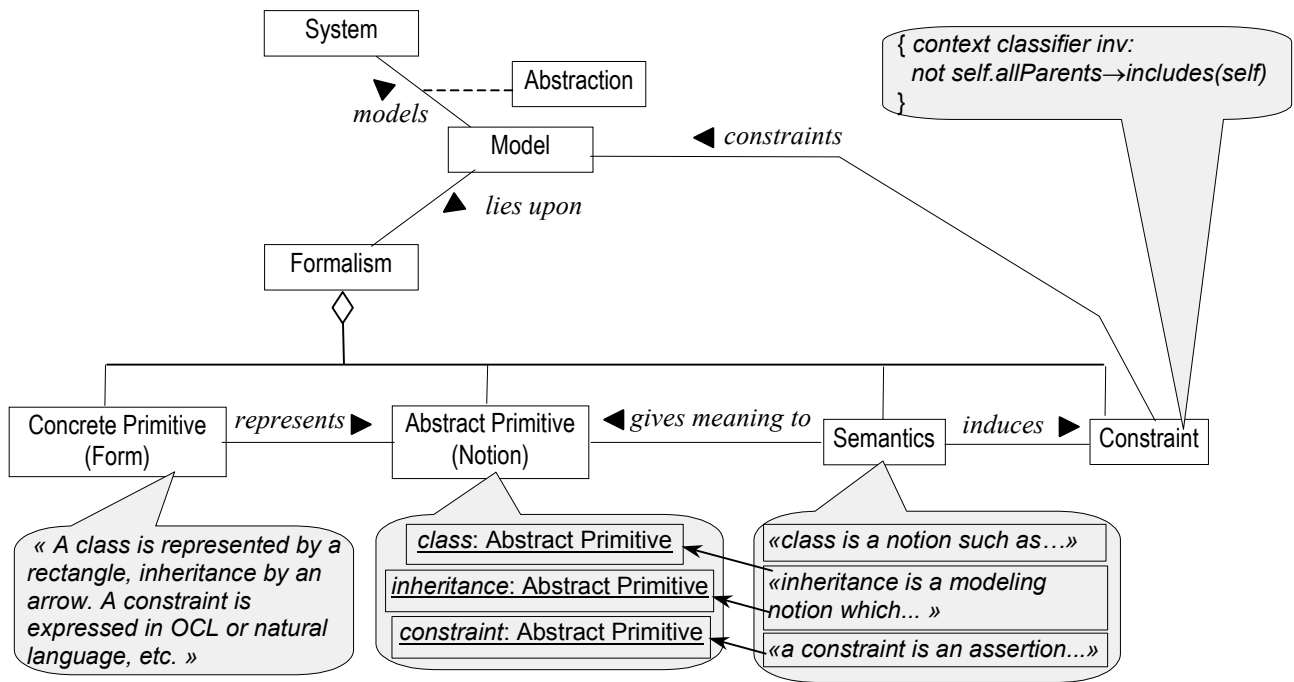


Figure 1. System, model and formalism: a basic model

the same system model, i.e., a hierarchy of UML packages – even if they appear within various diagrams – horizontal consistency is a *property* of a model. *Vertical* consistency is related to models at different abstraction levels, most of the time due to refinement. In this case we want to know to what extent a refined model *conforms to* a given origin model. Hence vertical consistency is a *relationship* between models. As refinement add details and transforms models, the “conform to” relationship cannot be an equivalence. In the best case it may be a *preorder*, i.e., the refined model “does at least the same” or “owns at least the same properties”, but generally the relationship is more complex. For instance when a “two-classes, one-association” diagram is refined into a “three-classes, two associations” diagram, it is not easy to conclude that the refined model conforms to the previous one using general rules.

Horizontal and vertical consistencies are very different: on the one hand incompleteness and contradictions should be found, while on the other hand a relationship “conform to” with fuzzy contours should be enforced. These verifications are very different and the methods and tools will also be different. In the sequel *we only tackle the horizontal consistency*, i.e., we assume that all the model elements have a compatible abstraction level.

2.2. Constraints

Several constraint classifications have been proposed. For instance we have given in [15] a classification based

on levels, from the *paradigmatic level* (the semantics of abstract primitives) to the *implementation level* (setting unspecified details). In [2], *Astesiano and Reggio* cite two orthogonal classifications, horizontal vs. vertical and syntactic vs. semantic, that lead to four potential cases.

In this work, we start from three aspects of formal languages particularly interesting for the UML. First, a formal language should provide syntactic rules to decide whether or not any expression is valid. Second, language constructions are associated with notions in a semantic domain. In this semantic domain, an expression can be true or false independently of the fact that its syntax is valid. Finally, it is usual to consider that a language is formal when all the expressions in this language can be directly translated into a programming language, which is formal in essence. These three aspects will be further detailed.

2.3. Syntactic and semantic rules

Programming languages usually distinguish the syntax, the static semantics, e.g., in «if expr then» expr is a *boolean*, and the dynamic semantics, e.g., in «t[i]», i is in the range declared for the array t. The same distinction applies in the UML. A typical syntactic constraint is “An Association consists of at least two AssociationEnds, each of which represents a connection of the association to a Classifier”. As this constraint is specified in the metamodel class diagram, it has no OCL form. Syntactic constraints are concerned with instances also, for example

"an abstract class cannot be instantiated". A typical constraint of static semantics is "Circular inheritance is not allowed". It is expressed in OCL by the rule: "not self.allParents->includes(self)". Semantic constraints are also described in the *UML Semantics* section [17] using natural language, for instance "A constraint attached to a stereotype must not conflict with constraints on any inherited stereotype, or associated with the baseClass". Dynamic semantics constraints are checked at run time, but since UML is not executable, the constraints cannot be checked directly.

Making a difference between syntactic and static semantics rules does not have any interest, especially as the distinction greatly depends on the language definition [15]. These rules are both defined formally and checked statically. For greater convenience, we call *syntactic constraint* any rule expressed formally and that can be checked statically, and *semantic constraint* any non-formal rule (as in [6]).

In this classification, semantic constraints are related to the semantics of the modeling domain and are described in the UML metamodel. Usually, constraints related to the modeled domain are also called "semantic constraints", but they are described within models. In the first case constraint checking comes down to check a property (semantically correct) of a model, while in the latter a "conform to" relationship is checked (the model conforms to the modeled system).

3. UML semantics

3.1. UML semantic domain

The semantic domain from which the UML language takes its meaning is the modeling domain, not to confuse with the modeled system domain in which a UML model takes its meaning (e.g., in [10] there is no distinction). UML valid expressions should have a meaning in the modeling world. This semantic domain does not lie on a set of formally described rules, and is not observable. Hence it seems that all the constraints related to the semantic domain have to be checked manually. What can be done? Theoretically there is an infinity of ways to cover a domain with a set of primitives. In practice, numerous modeling languages have been defined, and basic notions with very close meaning have stood out, for instance class/concept/entity or relationship/association. That means that there are universal notions in the modeling domain. UML is an example of convergence for it gathers points of view from numerous modeling languages while unifying the notions. However, although these notions have often very close meanings, they are not always interpreted in the same way. Since the semantics induces constraints (Fig. 1), an agreement on the

semantics is needed to express constraints. Everybody agrees about the rule that forbids cycles in the inheritance relationship. But there is no agreement on the way attributes with the same name should be inherited in the case of multiple inheritance. To come back to consistency checking of UML models, the issue is: which modeling semantics do we agree?

3.2. Semantics and interpretation

The *interpretation* is the mechanism by which sense is given to a model, but commonly the result of this mechanism is also called an interpretation. In a formal language, an expression is associated with only one interpretation in the semantic domain. In the UML, the semantics of the notions is often not precise and incomplete. Hence several interpretations can be associated with UML expressions. Moreover, the UML claims that it is an universal language implementation-independent, and interpretations are intentionally kept open: "Although the intent is to define the semantics of state machines very precisely, there are a number of semantic variation points to allow for different semantic interpretations that might be required in different domains of application".

Contrary to all appearances, the many interpretations are not a drawback but rather an advantage. In a modeling approach, the developer first delimits a set of a priori acceptable interpretations. As far as his knowledge of the system increases, details are added and interpretations are removed. In some cases the development is done keeping several paths open to alternative solutions between which the choice will be done later. For instance in the MDA approach [8], the PIM (Platform Independent Model) is completed before mapping to the PSM (Specific) implementation platform. To sum up, at the beginning of a software development, there is no advantage for the UML to be a formal language, quite the opposite. The abstraction level should remain high not to be overburdened with details and to reduce the cost of changes. However, the problem to overcome is: whatever the set of licit interpretations, the community of readers should agree about the content of this set. The problem is now well identified: *the aim is not to avoid multiple interpretations, but to define precisely the set of licit interpretations.*

For instance here is the way events are processed in a state machine: "Events are dispatched and processed by the state machine, one at a time. The order of dequeuing is not defined, leaving open the possibility of modeling different priority based schemes". Several interpretations are possible, but the set of licit interpretations is known. Compared with formal languages, the UML requires defining the wished interpretation before implementation, for instance the way the event queue is managed. As a

result, the translation of the UML model into code is an essential step: choices that are done along the many dimensions of the interpretation space should be consistent.

3.3. Consistency definition

Any licit interpretation should enforce the set of rules and constraints specified in the UML metamodel. *A model is inconsistent when it has no interpretation.* A model implementation has only one interpretation.

All the licit interpretations are not compatible, e.g., as long as a class is not marked *abstract*, instances may be created, but adding the « abstract » stereotype will forbid all the interpretations that include instances of the abstract class. More subtly, the choice of the dequeuing order will reduce the number of ways to write the interaction sequence to get a given result. In the former case, the illicit interpretations are found by enforcing syntactic constraints, while in the later case the user has to examine carefully the sequence of interactions. The next section considers the various ways to check the constraints.

4. Constraint checking

There are many proposals in the literature to check constraints in order to detect inconsistencies. This section attempts to assess the feasibility and the interest of typical proposals in present software development with present (imperfect) tools and present (imperfect) UML.

4.1. Syntactic constraints

A lot of works have been done about syntactic constraints. There is no theoretical problem to get syntax with the wanted properties and to verify that the OCL rules are consistent (see *Ritchers and Gogolla* [12][13]). The UML syntax is likely to improve, but first the existing rules should be enforced. As in usual languages, *tools must automatically check syntactic constraints.* Unfortunately, this is not quite true for most of the tools. The first condition on the way to model consistency is to provide tools that check all the syntactic rules. To complete this aspect, *tools must implement the whole UML.* This is more important that it seems because users don't know when an expression is not implemented and when it is forbidden.

When a syntactic rule is violated, the model is inconsistent since it is not valid. In practice, *users should check manually an additional set of rules* depending on the used tool: this is a strange and unnecessary burden that is not accepted in any computer science language.

4.2. Semantic constraints

By definition, these constraints are checked manually by the developer. What can be done to ease this work? First the number of semantic constraints should be reduced. Second the set of licit interpretations should be precisely defined to avoid misunderstanding about models within a team. The approach is clear but to bring it into play is not simple:

- To precisely delimit interpretations (domain semantics) using expressions in natural language is difficult since all the licit/illicit interpretations should be listed,
- Natural language reading depends on numerous factors that are difficult to master (cultural aspects, etc.)

4.2.1. Reducing the number of semantic constraints.

Some constraints are semantic due to the lack of expressive power of the UML description language. If the constraints below could be specified with OCL, they would become syntactic:

- “*The type associated with a tag definition is either the name of a UML metaclass, including elements of the DataType package, or an instance of the DataType metaclass or one of its descendants*”: level M2 not accessible
- “*The data value of a tagged value must conform to the data type specified by the “tagType” attribute of the tag definition*”: requires an OCL function that converts a string name into a corresponding metatype.

To reduce the number of semantic constraints in the next versions of UML, a first action is to *increase the expressive power of OCL.* Thus the new or rewritten constraints will be formal and syntactic (see also 4.6.5.). Second, it is possible to *better define the semantics:* the constraint “*A guard should not have side effects*” is written in OCL: “self.transition->stateMachine->notEmpty implies post: (self.transition.stateMachine->context = self.transition.stateMachine->context@pre)”, where *context* is the *ModelElement* associated with the *StateMachine*. For many readers, the constraint written in natural language is much more precise and complete than the OCL description!

4.2.2. Semantic constraint checking. To define all the UML licit interpretations in the modeling domain in such a way that everybody agrees is not possible, at least in the short term. Not to wait an unlikely way out, the proposed solution is to *define in a group or community the set of licit interpretations.* This choice may induce additional constraints on UML models, but these models still conform to the UML. Examples are: “each use case must be associated with a scenario described using a sequence diagram”, “there is only one trigger signal in a scenario”,

“any trigger signal comes from an external actor or a clock”, “any asynchronous message must be sent to an active object”, etc. In fact, these rules are very close to style guides used in programming languages, and thus very easy to introduce within teams that get used to software engineering practices. The main advantage of this solution is the little work to do since only the interpretations relevant to the needs of the community must be defined. Moreover, unchecked syntactic rules can easily be added, as well as constraints related to the development process. To stay in the UML world, constraints can partly be described into a profile and linked to steps in the development process. Novice users will have trouble reading constraints written in OCL, but expressions in natural language such as above are so restrictive that they are often simple, and tools will (soon?) check these constraints automatically. A way to increase automatic checks is given in 4.6.5.

Finally, why do we need to check semantic constraints? Practically, the serious errors that students do when designing UML models are not related to UML itself but to the knowledge of the modeling domain. The error Fig. 2a will be easy to find and could be checked automatically. The error Fig. 2b is very serious and very frequent. The model seems to conform to the modeled domain semantics but in fact it does not always. To reduce the inconsistencies in the UML model, a *fundamental condition is to know its semantic domain*, i.e., the modeling domain. When users know how to express current patterns, they automatically avoid using forbidden expressions.

Remark. In programming languages, all the expressions that may be invalid are rejected, even when it can be proved by a deep manual analysis of the code that they are always valid. This policy is adapted for syntactic checks, but semantic checks have to detect errors that are sure only: among all the possible interpretations, many are mutually exclusive. For instance, to send an asynchronous message to an object that is not declared as active is not an error, but a team may decide that it is an error when the receiver object is declared as passive. Thus, adding details reduces the set of licit interpretations.

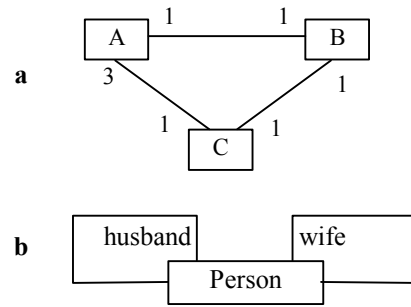


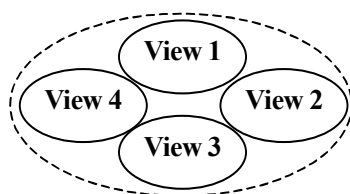
Figure 2. Modeling errors.

4.3. Multi-views

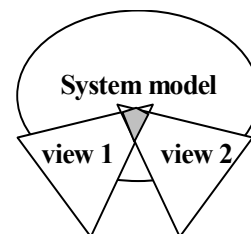
The UML provides several diagrams to describe the modeled system from several points of view such as structural or dynamic. This multi-view representation is often seen as a source of inconsistencies but it is not as obvious as it seems. Assuming that a model is the merging of several views or models (Fig. 3a), then multi-view consistency is an issue. But in fact a system model is a hierarchy of UML packages, and the views are simply particular representations of this unique model (Fig. 3b). The common data shown in the views are the same. When a class is represented in several views, it has the same properties since it is the same *ModelElement* in the UML model. In fact only graphical properties are specific of the views.

4.3.1. Multi-view errors. To illustrate this fact, assume a diagram including all the classes of a model, *CDialog* and *CLibrary* Fig. 4a, and an interaction from a *User* with a *Click* that triggers the sending of the message *Borrow* to *library:Library* Fig. 4b. There are incompleteness and inconsistencies errors in this sequence diagram:

- Does the object *dlg:CDialog* exist? It is possible, but then the class diagram is incomplete,
- The object *library:Library* does not exist: there is likely an error in the name of the class *Library*; on the other hand, the only object of class *CLibrary* that an object of class *CDialog* sees is named *lib*.



a: the system model is the merging of the views



b: views are representations of system parts

Figure 3. Multi-views interpretation

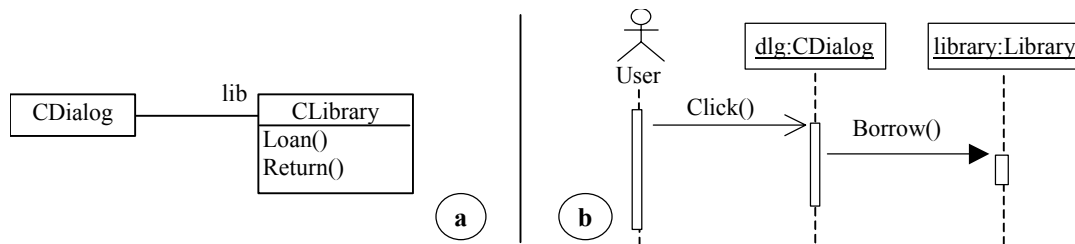


Figure 4. Multi-view inconsistencies.

- There is no event named *Borrow* in the namespace of *CLibrary* therefore there is an error ; it is likely a confusion with *Loan*.

These errors are found when comparing the views, but they are simple syntactic errors, for example any instance should be associated with a class and *Library* cannot exist. It is not possible for a model to be always syntactically correct. That explains the presence of *Library*, but a simple syntactic check will detect the problem: the tool has a great importance to limit the errors.

When the UML metamodel supplies the connections between all the *ModelElements* involved in a semantic dependency relationship, there is no multi-view issue. Otherwise, the metamodel should be enhanced to always supply, browsing through relationships, a path between all the *ModelElements* linked or dependent. We can reasonably assume that the UML metamodel enforces or will enforce this property.

Then and again, the remaining and recurrent problem is to express the constraints to allow automatic checking. Some semantic constraints such as the link between the state/transitions and the sequence diagram are difficult to express in either OCL or natural language. A solution is to describe some constraints using specific languages that provide tools for automatic check (see formal languages below). Another practical way in critical situations is to write a specific program that checks the constraint directly on the model (see code generator below).

4.3.2. Merging models. When a team develops a project, each developer may design independent models. In this case models should be merged as in Fig. 3a. In practice the merging is done manually since many decisions are required, and the team has to adopt a rigorous policy to minimize the merging work.

4.4. Dynamic semantic constraints

In this direction, the UML is similar to the other languages and the same solutions apply. It is possible to use properties of the target language, for instance in C++ a class constructor can be declared private to avoid creation of instances. Some code generators introduce

extra code to check properties, for instance in the state machine when “multiple outgoing transitions emanate from a common choice point, if no guards are true after the choice point has been reached, the model is ill formed”.

4.5. Translation of the model into code

It may seem strange to mix semantic constraints and code generation, but there is a strong link. Generally a programming language is viewed as a formal language, i.e., with only one interpretation. On the other hand a UML model is associated with several interpretations. To translate a model into code requires choosing one interpretation among all the possible ones. This choice is done by the code generators of tools and is guided by the value of a set of parameters. Therefore in practice, the tool freezes the semantics of the model. Moreover, the semantics depends on the target language, for instance *abstract class* or *private attribute* do not have the same meaning in all programming languages.

Again, at least for models translated into code, the degree of freedom for the meaning of the UML notions is useful. As a drawback, developers give the same meaning to both the UML notions and the target language notions (the target language is a parameter of some tools). This use is questionable but very difficult to avoid. Moreover, code generators only translate a subset of UML, and developers know how to reduce their implementation work. Here the principle of the MDA could apply, keeping the model (PIM) independent of its implementation as long as possible.

4.6. Translation of UML models into a formal language

Since UML is not a formal language, numerous works attempt to translate, at least partly, UML models into an existing formal language.

4.6.1. Translation of UML expressions into formal language. The entire model or a subset is translated into a formal language $m/UML \rightarrow m'/F$. This language F is

chosen according to the properties to check, and the properties of the model m in UML will be deduced. This process, named “translational approach” in [2], is very appealing for it allows reusing existing tools and skill. This approach is widely proposed to check UML models (F can be B , *object Z*, *CSP*, see UML’2002 [16]). For instance Engels & al. in [4][5] perform $m/\text{UML} \rightarrow m'/\text{CSP}$, where m represents only statecharts. Then checks are done using standard CSP tools.

Going deeper, this approach is not as appealing as it seems. First the translation $m/\text{UML} \rightarrow m'/F$ with F formal is done. The, at code generation time, a new translation $m/\text{UML} \rightarrow m''/E$ will be done where E is the execution context (language plus OS). During the translation $m/\text{UML} \rightarrow m'/F$, the many interpretations of m have been reduced to only one in m' since F is formal. During $m/\text{UML} \rightarrow m''/E$ a single interpretation has also be chosen since E has only one interpretation. It is now clear that this approach has a sense only when the same interpretation of the UML model is chosen during the two translations. For instance when $F=\text{CSP}$ and $E=\text{C++|Vx-Works}$, the two translations of m should behave the same.

As a result, to check a UML model when E is unknown has limited usefulness. Only “universal” properties such as deadlock can be searched for, but even in this case the implemented model m''/E should behave the same than the tested model m'/F (e.g., dequeuing order), a property that is difficult to prove. When the OS is involved, for instance through threads or scheduling, the proof is even more difficult to find.

Generally, papers advocating this approach are implementation independent. That means that the developer is faced with the usual problem: does the implementation conform to the specification? In the translation $m/\text{UML} \rightarrow m'/F$, m' is in fact the specification with the wished properties. For the implementation $m/\text{UML} \rightarrow m''/E$, there is to prove that m'' actually implements m' . This problem is very costly and very difficult to solve. It is tackled essentially in critical domains for which the cost to find an error is very lesser than the cost of the error itself. In [19] the UML model and the B specification are kept in parallel during refinement steps. To be sure that the implementation conforms to the specification, the B model can be refined until code is reached, but UML becomes useless.

4.6.2. UML profiles for translation. To solve the above problem, the general solution is to guide the translations $m/\text{UML} \rightarrow m'/F$ and $m/\text{UML} \rightarrow m''/E$ to force the choice of the same interpretation. In the UML framework, that means to define for each couple (F,E) a profile with additional constraints. This solution is limited because the choice of the interpretation cannot be entirely guided by constraints, and because code generators do not provide

enough flexibility. At the end, the solution may be to write a code generator.

4.6.3. Translation of UML models into a formal and executable language. For the translational approach to be more useful, a way is to do a unique translation $m/\text{UML} \rightarrow m'/F\&E$ where $F\&E$ is both the formal language and the execution context. In this case there is no need to prove that the implementation conforms to the specification. According to this principle, the *Esterel* language (principles in [1]) do the translation $m/\text{UML} \rightarrow m'/\text{Esterel-C++}$, and the properties of m' are the same in the formal language *Esterel* and in the generated code. For instance it is possible to prove that the lift cannot move when the door is open. In fact, *Esterel* uses a totally synchronous language interpretation of UML that is not a licit interpretation according to the metamodel. However, this approach is really appealing and provides executable UML as well as proved properties.

4.6.4. Translation of UML model into logical algebraic specifications. The translation of UML models into logical algebraic specifications is proposed in [2]. Again a precise definition of the UML semantics is required to check semantic consistency, and moreover checking poses several technical problems: “As a consequence, a method for helping detect all possible inconsistencies is not feasible”. Finally they adopt an approach very close to the one proposed in this paper, reducing the permitted interpretations to allow consistency checking.

4.6.5. Translation of UML models into a rule-based language. In [15] we have proposed the translation of the UML model into the expert system *Sherlock* [3], i.e., $m/\text{UML} \rightarrow m'/\text{Sherlock}$. Unlike translations to formal languages, it keeps all the interpretations. *Sherlock* is a rule-based formalism that provides logical deduction chains and inferences. Since its expressive power is greater than the OCL one (the level M2 is known), this solution reduces the number of semantic constraints.

Our above proposal requires the licit interpretations of UML to be defined for a community. Constraint checking can be done manually, but this tool allows us to check automatically a large part of them. Constraints are stored into a rule base that defines the semantics of the modeling domain. Compared with constraint description in a UML profile, this approach is currently more powerful because the tools that supply OCL constraint checking are still rare. Moreover, it provides diagnosis and suggestions while OCL can only return *true* or *false*. In the case of Fig. 2, it is possible to ask the user some precision: “when *Person A* is the *husband* of *Person B* then is *B* the *wife* of *A*?”.

5. Conclusion

Apparently there is no way to check UML model consistency since the semantics of UML is not formally defined. The semantic domain of the UML is the modeling domain, and in this domain UML expressions may be interpreted in several ways since the meaning of the notions is not frozen. This is not actually a problem, the main issue being to define precisely all the licit interpretations. As it is neither possible nor desirable for all the users to be in agreement, the proposed approach is to define the licit interpretations for a group or community. This solution boils down to add constraints that can be gathered into a UML profile. The checks can be done either manually or partly automatically when a tool is available. Several profiles can be defined according to the application domain or the community.

At code generation time, an interpretation is chosen among all the licit ones, thus the semantics of the UML is frozen. This aspect is to take into consideration to define the semantics for a given context, especially as code generators do not always specify the UML interpretation they have chosen.

Checking manually model consistency adds some burden to the developer. To translate UML models into formal languages seems appealing to automate the checks. However, this translation induces the choice of an interpretation that plays the role of a specification, and later the developer will have to verify that the implementation conforms to the specification. This is the weakest point of the translational approach, and a promising approach is to provide a translation of the UML model into a formal language that is also executable.

To ease consistency checking, both UML and tools are to improve and here are some directions. In UML: (i) to increase the expressive power of the meta-metalanguage in order to reduce the number of semantic constraints; (ii) to improve the precision of the metamodel to better define licit interpretations; (iii) to create a path between all the model elements involved in a semantic dependency relationship. In tools: (i) to check all the UML syntactic constraints and to implement the entire UML; (ii) to provide OCL constraints checking; (iii) to generate the code for a larger subset of UML.

References

[1] C. André, M-A. Peraldi-Frati, J-P. Rigault, "Integrating the Synchronous Paradigm into UML: Application to Control-Dominated Systems", UML 2002, LNCS 2460, pp.163-178
[2] Astesiano E., Reggio G., "An Algebraic Proposal for Handling UML Consistency", Workshop on Consistency Problems in UML-based Software Development, 2003, //www.ipd.bth.se/consistencyUML/

[3] G. Caplat, "Sherlock Environment", available at //servif5.insa-lyon.fr/chercheurs/gcaplat/
[4] G. Engels, JM. Küster, L. Groenewegen, R. Heckel, "A Methodology for Specifying and Analyzing Consistency of ObjectOriented Behavioral Models", FSE, ACM Press, 2001
[5] Engels G., R. Heckel, JM. Küster, Luuk Groenewegen, "Consistency-Preserving Model Evolution through Transformations", UML 2002, LNCS 2460, pp. 212-226
[6] D. Harel, B. Rumpe, "Modeling Languages: Syntax, Semantics and All That Stuff", TR MCS00-16, The Weizmann Institute of Science, 2000. Available on www.cs.york.ac.uk/puml
[7] B. Hnatkowska, Z. Huzar, L. Kuzniarz, L. Tuzinkiewicz, "A systematic approach to consistency within UML based software development process", RR-2002-6, pp16-29, available at [18]
[8] "Model Driven Architecture", Document number ormsc/2001-07-01, Architecture Board ORMSC 2001
[9] MOF, "Meta-Object Facility Specification", version 1.4, 2002, www.omg.org
[10] Naumenko A., Wegmann A, "A Metamodel for the Unified Modeling Language", UML 2002, LNCS 2460, pp. 2-17
[11] Precise UML site, http://www.cs.york.ac.uk/puml/
[12] M. Richters, M. Gogolla. "OCL - Syntax, Semantics and Tools". In *Advances in Object Modelling with the OCL*, LNCS 2263, 2001, 43-69
[13] M. Richters, M. Gogolla, "Validating UML Models and OCL Constraints", UML 2000, LNCS 1939, 265-277
[14] J.L. Sourrouille, G. Caplat, "Checking UML Model Consistency", Workshop on Consistency Problems in UML-based Software Development, RR-2002-6, pp1-15, available at [18]
[15] J.L. Sourrouille, G. Caplat, "Constraint Checking in UML Modeling", SEKE'02, ACM-SIGSOFT, pp217-224
[16] UML'2002, LNCS 2460, Springer Verlag, 2002
[17] UML, "OMG Unified Modeling Language Specification", Version 1.5, 2003
[18] Workshop on "Consistency Problems in UML-based Software Development", in L. Kuzniarz, G. Reggio, J.L. Sourrouille, Z. Huzar editors, RR-2002-6, available at http://www.ipd.bth.se/uml2002/RR-2002-06.pdf
[19] R. Marcano, N. Levy. "Using B formal specifications for analysis and verification of UML/OCL models", 2002, in [18]

Refinement relationship between collaborations

Bogumila Hnatkowska
Computer Science Division,
Wroclaw University of Technology,
Wybrzeze Wyspianskiego 27,
50 370 Wroclaw, Poland
B.Hnatkowska@pwr.wroc.pl

Zbigniew Huzar
Computer Science Division,
Wroclaw University of Technology,
Wybrzeze Wyspianskiego 27,
50 370 Wroclaw, Poland
Z.Huzar@pwr.wroc.pl

Ludwik Kuzniarz
Department of Software Engineering and Computer Science,
Blekinge Institute of Technology
PO Box 520 Soft Center
SE-372 33 Ronneby, Sweden
lku@bth.se

Lech Tuzinkiewicz
Computer Science Division,
Wroclaw University of Technology,
Wybrzeze Wyspianskiego 27,
50 370 Wroclaw, Poland
L.Tuzinkiewicz@pwr.wroc.pl

Abstract

The paper presents a constructive, structural definition of refinement relationship between collaborations. Both the relationship and the artifact were chosen as representatives of more general concepts from a software development process. The definition has a characteristic of being constructive and structural and thus enabling practical application and tool support.

1. Introduction

Software development is usually a long and complex process including several phases and a number of activities. During the process a number of artefacts are produced. The artefacts can be grouped to form models of the system, which present a view of the system from certain perspective and on a certain level of abstraction. The phases and models produced are usually constructed in certain order and models should be related in a certain way to each other. The basic relationship between models is that they describe the same ‘thing’ on different levels of abstraction and from different point of view or taking into consideration different aspects. Development process can be viewed as sequence of model transformations. The transformations can be of the following type: adding more details (refinement), extending or reducing, translation – description in a new formalism.

UML specification document introduces an attempt to address the problem definition of relationship between artefacts expressed in the language by introducing four stereotyped abstraction relationships between artefacts: «derive», «realize», «trace» and «refine». The stereotypes are rather informally explained; their definitions are very general and not precise. But they highlight a set of interesting concepts. The most interesting and useful from practical point of view seems to be the idea behind the refine relationship. The paper contains an attempt to define the relationship in a more systematic and formal way. To approach the problem two significant assumptions has been made. The first concerns the model or the elements between which the refinement relationship will be defined. Collaborations have been cho-

sen as representative element. The second restriction concerns the process. The relation is defined for a top-down refinement process applied to collaborations. This means that an activity in the process is a refinement activity, which for a given collaboration produces a more detailed refined collaboration.

The paper is organized as follows. First the notion of collaboration is defined and the intuitions behind the concept of refinement used are explained. The following two sections contained specification of the refinement relationship divided into two parts – structural and behavioural refinement. Final section contains conclusions.

2. Collaboration

Collaborations are used to describe behaviour of a system or a use case or an operation [5]. The description is given by means of:

- collaboration diagrams at specification level that specify admissible structures of ensembles of objects, and
- collaboration diagrams at instance level that specify admissible communication scenarios.

Collaboration *Coll* we present as a pair:

$$Coll = \langle SpecD, INSTD \rangle$$

where *SpecD* is a collaboration diagram at specification level, and *INSTD* is a finite family of collaboration diagrams at instance level. Each collaboration diagram at instance level $InstD \in INSTD$ is consistent w.r.t. *SpecD*.

Definitions of collaboration diagrams at specification and instance level as labelled directed multigraphs are formally defined in [4]. Definition of consistency of a collaboration diagram at instance level to a given collaboration diagram at specification level is given in [2], [3].

In the sequel, both kinds of collaboration diagrams will be used informally according to the official UML notation given in [5].

Collaboration diagram at specification level describes a structure of an ensemble of roles that define participants needed for a given set of purposes. Vertices of the dia-

gram represent roles of classifiers (classes and actors), and arcs represent association roles and generalization relations.

Collaboration diagram at instance level describes an interaction – a partially ordered set of stimuli. Vertices of the diagram represent instances of classifier roles, and arcs represent instances of association roles (links) that conform to respective classifier roles and association roles at a collaboration diagram at specification level. Additionally, arcs are labelled by lists of messages sent along arcs. All messages in the diagram are partially ordered.

For simplicity, we assume that:

- there are binary associations only,
- instances and links may be created only during execution,
- all links are two-way navigable,
- the messages are linearly ordered; therefore each message has a unique ordering number.

In further, we will use the following notation: an interaction represented by a collaboration diagram at instance level $InstD$ will be noted by $interaction_{InstD} = s_1, s_2, \dots, s_n$ for $n > 0$, where $s_i = \langle sender_i, op_i, receiver_i \rangle$, where $sender_i$ and $receiver_i$ are instance roles that send and receive, respectively, the stimulus op_i ($i = 1, \dots, n$). By definition:

$$\begin{aligned} s_i.sender &= sender_i, \\ s_i.stimuli &= op_i, \\ s_i.receiver &= receiver_i. \end{aligned}$$

3. Concept of refinement

Refinement relation *refine* is defined between two collaborations $Coll_1$ and $Coll_2$, where

$$Coll_i = \langle SpecD_i, INSTD_i \rangle \text{ for } i = 1, 2$$

provided $card(INSTD_1) \leq card(INSTD_2)$.

An illustration of the general structure of *refine* relation is given in Fig. 1.

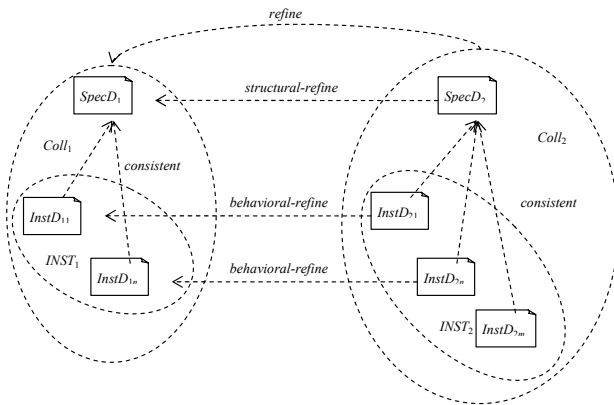


Fig. 1. A structure of a refinement relationship between collaborations

The structural refinement is defined as a sequential composition of elementary mappings on collaboration diagrams at specification level.

The idea of elementary mapping is the following. First, some elements $core_1$ are selected from $SpecD_1$ for replacement. The elements that are possible for the selection and the new elements that can replace them are defined later in Section 4. Relation core-mapping as depicted in Fig. 2 represents the replacement. Having decided which elements are to be replaced the problem how to join the new elements $core_2$ to the diagram arises. The new elements have to maintain the context of selected elements in the diagram $SpecD_1$. The context is expressed in terms of a set of generalizations and association roles that have exactly one endpoint relevant to the selected elements $core_1$ in $SpecD_1$. The relation *context-mapping* in Fig. 2 represents merging the new elements with the association roles from the context. It means that the endpoint of an association role (or generalization) that was connected to an element in $core_1$ must be switched to other element from $core_2$.

So, realization of elementary mapping is understood as composition of two relations: *core-mapping* and *context-mapping* (Fig. 2).

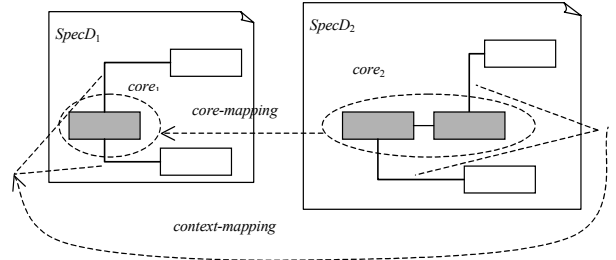


Fig. 2. A structural refinement between collaboration diagrams at specification level

Behavioural refinement is established in the following way. A correspondence between diagrams $INSTD_1$ and $INSTD_2$ is defined that each $InstD \in INSTD_1$ matches with exactly one $InstD' \in INSTD_2$. The behavioural refinement between matching diagrams is defined as a relation between interactions represented by the two diagrams – see Section 5. The diagrams in $INSTD_2$ that are not matched with diagrams in $INSTD_1$ represent new interactions that are added by developer to collaboration within refinement process.

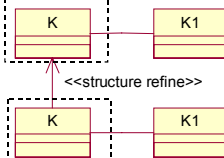
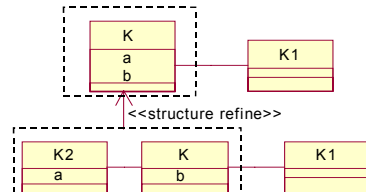
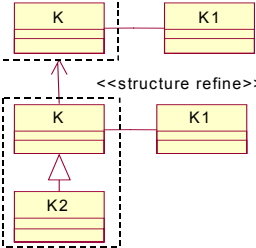
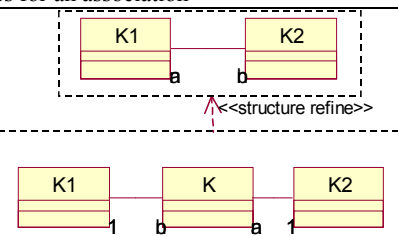
4. Structural refinement

The structural refinement is defined as a sequential composition of elementary mappings on collaboration diagrams at specification level.

Two diagrams $SpecD_1$ and $SpecD_2$ at specification level are in structural refinement relationship if there exists a sequence of elementary mappings g_1, \dots, g_k , for $k > 0$, such that:

$$g_k(g_{k-1}(\dots(g_1(SpecD_1))\dots)) = SpecD_2$$

Table 1.

No	Mapping core	Kind of mapping	Result of mapping	Examples/Notes
1	Empty	Adding a class	Class	You can always add a class to your diagram
2	Class	Adding a class property (attribute or operation)	Class	You can always add a property to a class 
3	Class	Modifying a class attributes: <ul style="list-style-type: none"> Changing attribute type declaration (adding a type or changing type) Attributes splitting 		Changing a type in attribute declaration: $a: T_1 \leftarrow \ll \text{structure refine} \gg a: T_2$ or $a \leftarrow \ll \text{structure refine} \gg a: T$ Attributes splitting: $a[: T] \leftarrow \ll \text{structure refine} \gg a_1[: T_1], \dots, a_k[: T_k]$ where '[']' means an option
4	Class	Modifying a class operations: <ul style="list-style-type: none"> Changing operation declaration (adding or specifying operation parameters) Operations splitting 		Changing an operation declaration: $op_1(a_1[: T_1], \dots, a_k[: T_k]) \leftarrow \ll \text{structure refine} \gg op_1(a_1[: T_1], \dots, a_n[: T_n])$ where a_i is unchanged for $i \leq k$ and $k \leq n$, '[']' means an option Operations splitting: $op(a_1: T_1, \dots, a_k: T_k) \leftarrow \ll \text{structure refine} \gg op_1(b_1: X_1, \dots, b_k: X_k), \dots, op_j(c_1: Y_1, \dots, c_m: Y_m)$ where $\{a_1: T_1, \dots, a_k: T_k\} \subseteq \{b_1: X_1, \dots, b_k: X_k\} \cup \dots \cup \{c_1: Y_1, \dots, c_m: Y_m\}$
5	Class	Splitting a class into two classes with association (1-1) between them	Two classes with association between them	
6	Class	Introducing a successor of a class	Two classes with generalization relationship between them	
7	Two classes	Adding an association	Two classes +association	You can always add an association between two classes
8	Two classes +association	Modifying of association	Two classes +association	You can constraint the multiplicities, add name and name of roles for an association
9	Two classes with association between them	Introducing a intermediate class	Three classes with two associations	

Each elementary mapping g_i operates on some elements (*core*) and their surrounding (*context*). All identified elementary mappings with explanation notes are gathered in Table 1.

For most kind of mappings their context becomes unchanged. The only interested is the case no. 5 – splitting a class into two classes with association between them. In that case context mapping is ambiguous, but limited to both resulting classes. In that situation the context mapping is beyond the designer decision.

5. Behavioral refinement

Let $InstD \in INST_1$ from collaboration $Coll_1$ and $InstD' \in INST_2$ from collaboration $Coll_2$ be two corresponding collaboration diagrams at instance level – see Fig. 1, and

$$interaction_{InstD} = s_1, s_2, \dots, s_i, \dots, s_n$$

be the message sequence representing the $InstD$ diagram.

We consider behavioural refinement between the diagrams $InstD$ and $InstD'$ by analysis of two cases.

The first case deals with situation when the collaboration diagrams at the specification level are the same, i.e. $SpecD_1 = SpecD_2$. In this case new message sequences α_i ($i = 0, \dots, n$) may be inserted in any place of the interaction $interaction_{InstD}$ provided that the new extended sequence:

$$interaction_{InstD'} = \alpha_0, s_1, \alpha_1, s_2, \dots, s_i, \alpha_i, \dots, s_n, \alpha_n$$

is consistent to the collaboration diagram at specification level.

The second case deals with a situation when the collaboration diagrams at the specification level are changed, i.e. $SpecD_1 \neq SpecD_2$. In this case we regard below the changes of collaboration structure numbered by 4, 5, 9 in the Table 1 (Section 4), because only these changes influence on the change of behaviours.

- Substituting of operation op in class K for a set of operations $\{op_1, \dots, op_k\}$ (no. 4 in Table 1). In this case the diagram $InstD'$ may be represented by

$$interaction_{InstD'} = s_1, s_2, \dots, s_{i-1}, \alpha, s_{i+1}, \dots, s_n$$

where

$$\alpha = s_1', s_2', \dots, s_k'$$

The diagrams $InstD$ and $InstD'$ are in behavioral refinement relationship if the following condition holds:

$$\begin{aligned} \text{if } & s_i.receiver = :K \text{ and } s_i.stimuli = :K.op \\ \text{then } & (s_i.sender = s_1'.sender) \text{ and } (s_1'.receiver \\ & = :K) \text{ and } (s_i.receiver = s_k'.receiver) \end{aligned}$$

where $:K$ means an object of the class K .

- Substituting class K for two classes K_1 and K_2 (no. 5 in Table 1). In this case the diagram $InstD'$ may be represented by

$$interaction_{InstD'} = s_1', s_2', \dots, s_k'$$

Depending on the role of an object $:K$ in the $interaction_{InstD}$, the following three subcases should be taken into consideration to guarantee the behavioural refinement relationship between the diagrams $InstD$ and $InstD'$:

1. For the object $:K$ being both a sender and receiver of a message (self message), the condition is required:

$$\begin{aligned} \text{if } & (s_i.sender = :K) \text{ and } (s_i.receiver = :K) \\ \text{then } & (s_i'.sender \in \{:\underline{K}_1, :\underline{K}_2\} \text{ and } s_i'.receiver \\ & \in \{:\underline{K}_1, :\underline{K}_2\}) \end{aligned}$$

2. For the object $:K$ being a sender only, the condition is required:

$$\begin{aligned} \text{if } & (s_i.sender = :K) \text{ and } (s_i.receiver \neq :K) \\ \text{then } & (s_i'.sender \in \{:\underline{K}_1, :\underline{K}_2\} \text{ and } s_i'.receiver \\ & = s_i.receiver) \end{aligned}$$

3. For the object $:K$ being a receiver only, the condition is required:

$$\begin{aligned} \text{if } & (s_i.receiver = :K) \text{ and } (s_i.sender \neq :K) \\ \text{then } & (s_i'.receiver \in \{:\underline{K}_1, :\underline{K}_2\} \text{ and } s_i'.sender \\ & = s_i.sender) \end{aligned}$$

- Substituting an association between classes K_1 and K_2 for class K with two associations with classes K_1 and K_2 (no. 9 in Table 1).

If at least one message pertaining to the substituted association exists then modification of message sequence is necessary to maintain diagrams' consistency at instance level and at specification level.

The diagrams $InstD$ and $InstD'$ are in behavioral refinement relationship if each message s_i from $interaction_{InstD}$ such that $(s_i.sender = :\underline{K}_1$ and $s_i.receiver = :\underline{K}_2)$ or $(s_i.sender = :\underline{K}_2$ and $s_i.receiver = :\underline{K}_1)$ is replaced by the a sequence of messages

$$\alpha = s_1', s_2', \dots, s_k',$$

in $interaction_{InstD'}$ that the following condition holds:

$$\begin{aligned} & (s_i.sender = s_1'.sender) \text{ and } \exists j \bullet ((1 < j \leq k) \text{ and} \\ & (s_i.receiver = s_j'.receiver) \text{ and } (s_i.stimuli = \\ & s_j'.stimuli)) \text{ and } (s_k'.sender = :K) \end{aligned}$$

In the consideration above, we have assumed that no message in any iteration may be removed. The existing message may be replaced with sequence of new messages. It is also allowed to insert new message into an existing sequence messages.

6. Case study

The aim of the section is to illustrate the usage of refinement relationship.

We will consider one of the functions – called “Add a team member” – of a hypothetical software system used by a sport coach to assign a sportsman to a given team. This system function is specified, following the USDP (RUP) methodology [6], [7], by a use case – see Fig. 3 – with some short description. The use case realization is modeled by collaboration. The change of the use-case description, the business rules, and designer decisions may result in the change of the collaboration during analysis and further workflows of USDP (RUP).

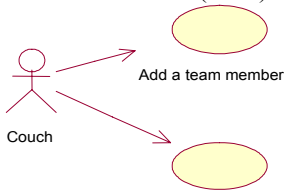


Fig. 3 – Use case diagram for a hypothetical software system

Three steps of collaboration refinement will be presented. The collaborations $Coll_1$ – $Coll_4$ are the subsequent versions of the use case “Add a team member” realization. The relationships between their elements are presented in Fig. 4.

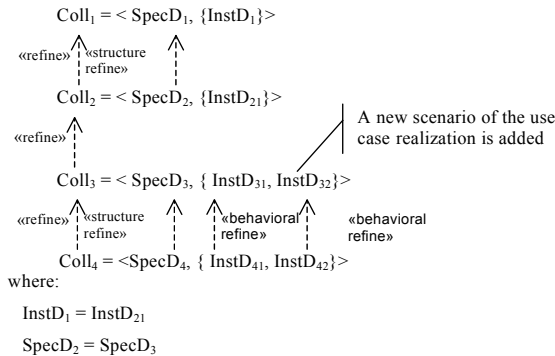


Fig.4 The idea of the refinement relationship between collaborations in the example

First version of the collaboration – $Coll_1$

The first version of the use case description is presented below.

Use Case Description – ‘Add a team member’ – version 1

The use case assigns a member with a team. A team couch initiates the use case. He or she chooses the team, and provides the name and address of a team member. System looks for the member. System creates relationship between the team and the member.

The appropriate collaboration diagram at specification level $SpecD_1$ is shown in Fig. 5, and the diagram at instance level $InstD_1$ – in Fig. 6. The following business rule from the business modeling stage was derived: ‘A team may have many members, and a member may belong to many teams’.

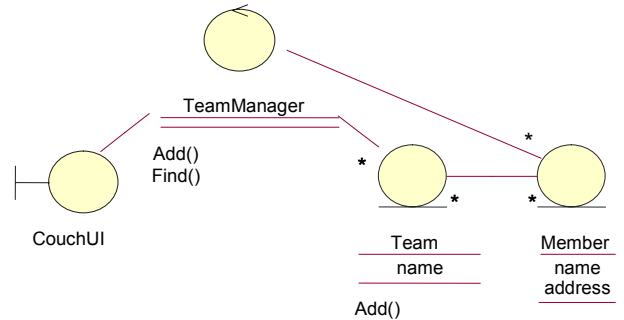


Fig. 5 Collaboration $Coll_1$ – specification level $SpecD_1$

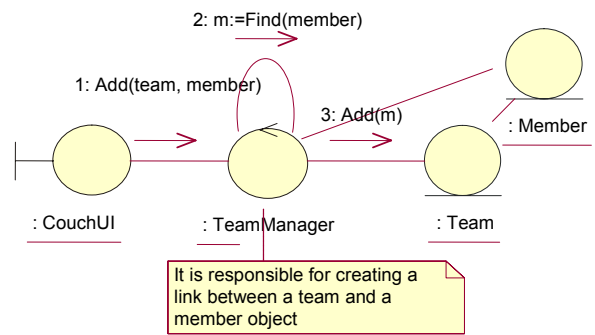


Fig. 6 Collaboration $Coll_1$ – instance level $InstD_1$

Second version of the collaboration – $Coll_2$

The collaboration designer decides to introduce a new class *Address*. The structure of the collaboration – $SpecD_2$ – is presented in Fig. 7. The use case behavior was unchanged.

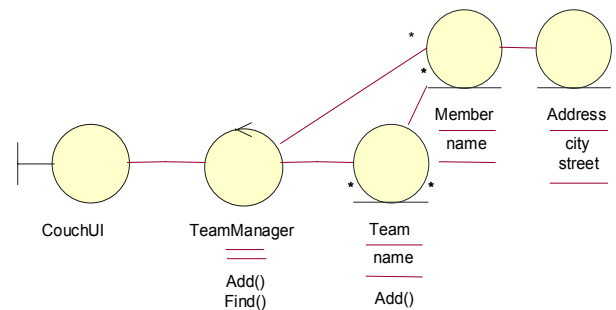


Fig. 7 Collaboration $Coll_2$ – specification level $SpecD_2$

Third version of the collaboration – Coll₃

The use case description is extended by a new scenario.

Use Case Description – Add a team member – version 2 (the changes of the use-case description are written with italic)

The use case assigns a member with a team. A team couch initiates the use case. He or she chooses the team, and provides the name and address of a team member. System looks for a member. *If the member does not exist system creates it.* System creates relationship between the team and the member.

The new collaboration diagram at instance level – *InstD₃₂* – is presented in Fig. 8. The diagram shows that an object of the control class *TeamManager* creates a *Member* class object.

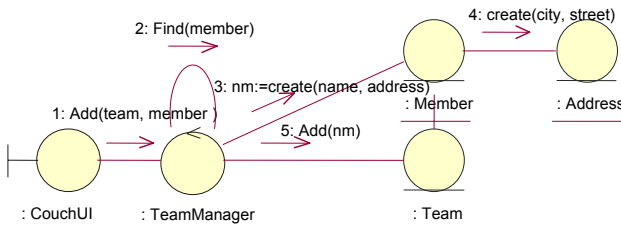


Fig. 8 Collaboration *Coll₃* – instance level *InstD₃₂*

Fourth version of the collaboration – Coll₄

There are two reasons for next collaboration refinement:

- (a) Change of the use case description

Use Case Description - Add a team member – version 3

The use case assigns a member with a team. A team couch initiates the use case. He or she chooses the team, provides the name and address of a team member, *and the date of enrollment the member to the team.* System looks for a member. If the member does not exist system creates it. System creates relationship between the team and the member *and remembers the date of the enrollment.*

- (b) Change of the business rule: “A team has many members, and a member may belong to 1-5 teams”

The use-case designer decides to introduce a new intermediate class *Membership* (between classes *Team* and *Member*), which is responsible for remembering the membership of each sportsman to the team. There is no message interchanged between *Team* class object and *Member* class object, so we can simply insert new messages between the existing ones.

The new version of the collaboration diagram at specification level *SpecD₄* is presented in Fig. 9, and the new version of the collaboration diagram at instance level for the introduced scenario *InstD₄₂* – in Fig. 10.

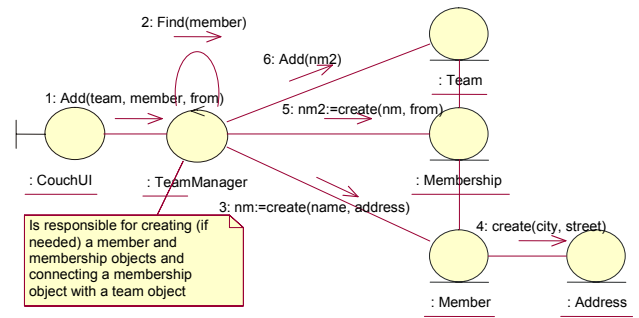


Fig. 9 Collaboration *Coll₄* – specification level *SpecD₄*

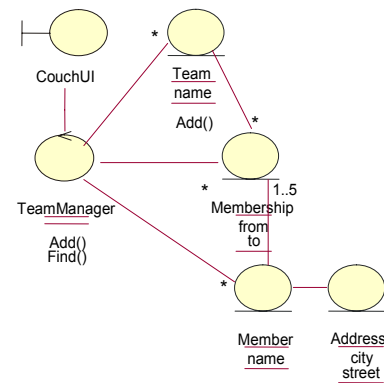


Fig. 10 Collaboration *Coll₂* – instance level *InstD₄₂*

7. Summary

The paper presents an attempt to define refinement relationship between collaborations. The definition reflects the notions of horizontal and vertical consistency [1]. Namely, horizontal consistency corresponds to the consistency between collaboration diagram at specification and instance levels within a given collaboration, while vertical consistency corresponds to structural and behavioural consistency between collaboration diagrams from two subsequently elaborated collaborations.

An important characteristic of the approach use to define the relationship is that the definition is constructive and structural. This characteristic makes the definition more practical and user oriented. What is also important the definition can be a basis for construction of a tool which can be used as a sort of an aid during top-down refinement activity checking whether transformations performed on a model are legal with respect to refinement relation or providing suggestions on possible transformations.

The definition is still not complete and formal but seems to be a good starting point for further improvement and formalisation. To be of practical usage the definition of the relationship should also be extended on other types of artefacts produced during the development process but

it seems that the presented approach has the required extensibility property.

References

1. G. Engels, J.M. Kuestler, L. Groenewegen, *Consistent Interaction of Software Components*, Proceedings of IDPT, 2002.
2. B. Hnatkowska, Z. Huzar, L. Tuzinkiewicz: *The Information Base for Analysis Model Deriving*. Proceeding of International Conference ISM'02, Hradec nad Moravici, Czech Republic 2002, pp. 95-102.
3. B. Hnatkowska, Z. Huzar, L. Tuzinkiewicz: *Class Identification in Business Modelling*. Foundations of Computing and Decision Sciences, vol. 27, no. 4, 2002, pp. 211-225.
4. B. Hnatkowska, Z. Huzar, L. Tuzinkiewicz: *Consistency Checking within Collaboration*, (in Polish) accepted by KKIO 2003.
5. OMG Unified Modeling Language Specification – version 1.4. Rational Software Corporation, September 2001.
6. L. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
7. P. Kruchten, *The Rational Unified Process. An Introduction*. Addison Wesley Longman Inc., 1999.

The Baseless Links Problem

Gonzalo Génova, Juan Llorens, José M. Fuentes
Computer Science Department, Carlos III University of Madrid
Avda. Universidad 30, 28911 Leganés (Madrid), Spain
{ggenova, llorens}@inf.uc3m.es, jmiguel@ie.inf.uc3m.es

Abstract

The UML Standard does not consistently follow the rule that every link is an instance of an association, leading to consistency problems in the metamodel itself, as well as in user models. The proposed solution reaffirms this rule, implying a slight change to the metamodel, and gives guidelines for CASE tools.

1. Introduction

One of the rules that are expected to be fulfilled in a well-formed UML model is that every instance has a type (in some cases, an instance can have more than one type). If this rule is not satisfied, we can say that there exists an intra-consistency problem within the model.

A link between objects is a special kind of instance, namely, an instance of an association between classifiers. However, this principle is not clearly and consistently followed in the UML Standard [2]. As we are going to see, there are common modeling situations in which, apparently, there are links that are not instances of associations (thus, *baseless links*).

2. Is every communication link an instance of an association?

Consider the example in Figure 1(a): there is a class diagram with a one-way association from class `Owner` to class `Bank`, and another one-way association from class `Owner` to class `Account`. In Figure 1(b) there is a collaboration diagram where an object of class `Owner` sends a message to an object of class `Bank` containing an object of class `Account` as argument, and the `Bank` object uses this `Account` object to send it a message: the owner object communicates its bank to close its account. In UML this interaction is usually modeled using a stereotyped `«parameter»` link from the `Bank` object to the `Account` object. Is this a true link, or is it only a

graphical fiction? Does this link require an association between the `Bank` and `Account` classes? If not, does this mean that the link is not an instance of any association? This question is far from having been clarified, as recent research demonstrates[3]².

The Standard is rather contradictory in this respect, since it gives two different solutions to this problem:

- *Sometimes a message does not use a communication link.* After stating that a message instance (a.k.a. stimulus) “uses a link between the sender and the receiver for communication”, the Standard acknowledges some *special situations* in which this communication link may be missing: “if the receiver is an argument inside the current activation, a local or global variable, or if the stimulus is sent to the sender instance itself” [UML, p. 2-114]. Therefore, the link `myBank→myAccount` in Figure 1(b) would be a *fiction*, and no association is required between `Bank` and `Account`.
- *Sometimes a link is not an instance of an association.* The Standard defines *five standard stereotypes* for `LinkEnd` (`«global»`, `«local»`, `«parameter»`, and `«self»`, in addition to the redundant `«association»`) to handle those same special situations [UML, p. 2-103], where we find communication without associations. Therefore, the link `myBank→myAccount` in Figure 1(b) would be a *true link*, but a link that is not derived from the existence of an association between `Bank` and `Account`, but from “other circumstances”. Again, no association is required between `Bank` and `Account`.

¹ We quote, by section and page numbers as usual, version 1.4 of the UML Standard. Thus, “[UML, p. 2-114]” means “[2] sect. 2, p. 114”. Version 1.5 has not introduced any changes regarding the subject of this paper, and version 2.0 is not yet approved and available to the general public.

² See also the contributions to The Precise UML Group mailing list [4] during the years 2000-2001 under the subjects “Links & messages”, “Link as instance, tuple, path”, “Sets and bags”, and “Dependencies and associations”, where the authors played an active role. Other similar problems involving compound navigation expressions and stereotyped `«self»` links are described in a recent paper by the authors [1]. We are not aware of any proposal to UML 2.0 that addresses this issue.

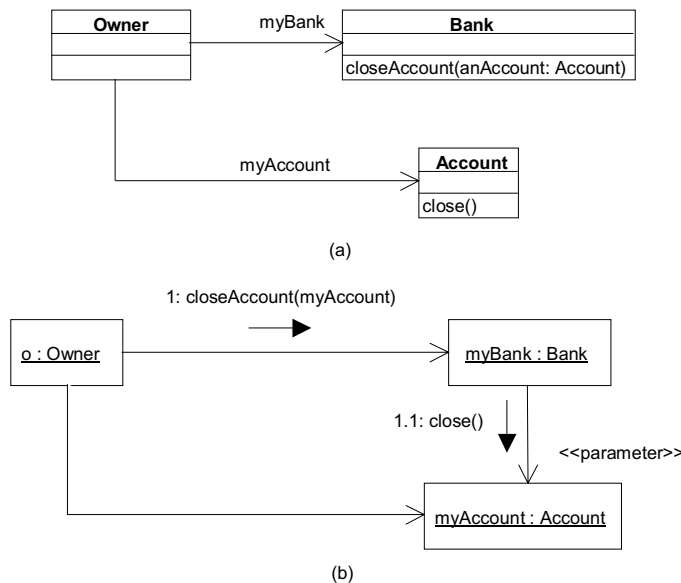


Figure 1. (a) Class diagram with classes Owner, Bank and Account, and two associations among them. (b) Collaboration diagram using a stereotyped `<<parameter>>` link without any existing association Bank→Account

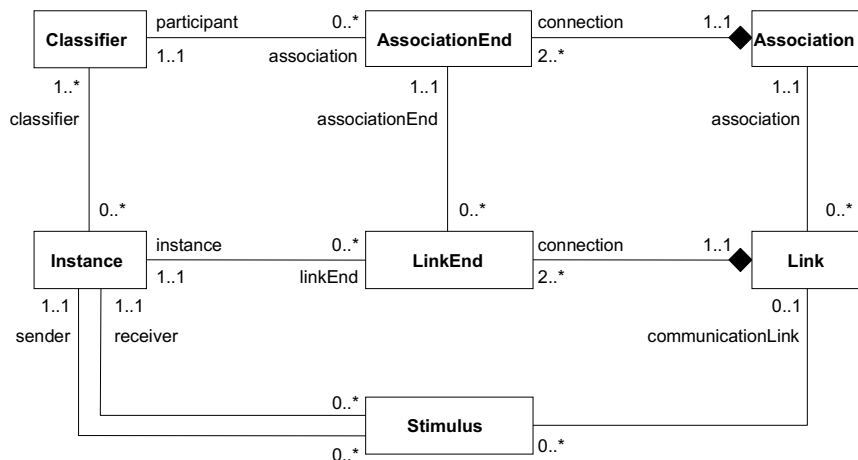


Figure 2. Metamodel of communication links extracted from Figures 2-6, 2-16 and 2-17 in the Standard

The first solution is consistent with the statement that a link *is* an instance of an association, represented in the metamodel by a mandatory `Association` that specifies the `Link` (multiplicity 1..1 on the role `Link.association`, see Figure 2), and it is also consistent with the statement that the link is *optionally* used by a message for communication (multiplicity 0..1 on the role `Stimulus.communicationLink`): sometimes the message uses a link (which is an instance

of an association), and sometimes the message does not use any link (so no association involved).

The second solution (stereotyped links) breaks the principle that every link is an instance of an association, and it contradicts the first solution (optional links): if the communication link is optional, what is the sense of defining these special stereotyped links? Nevertheless, this solution is consistent with the common representation of interactions in collaboration diagrams, where a message *always* uses a link.

None of these two solutions is satisfactory. If links are optional, what is the representation of a message that is sent through a missing link in a collaboration diagram? The idea of a fictitious link does not seem a good one. In a previous paper we rejected optional communication links and supported the idea of links that are not instances of associations [1]. However, we are not satisfied with this solution, since a link, like an object, is a “concrete thing” (an instance); thus, a link, like an object, requires a “type” that specifies its features; the type of an object is a class, and the type of a link should be an association that specifies, among other features, the classes of the linked objects, the navigability and changeability of the links, etc. If a link had no type, we would not be able to describe which its properties are or how it is supposed to behave³. Therefore, we will try to find a conceptually better solution that avoids “baseless links”.

3. Association and link stereotypes in the UML Standard

Some authors have tried to distinguish between two disjoint subtypes of associations to solve this problem, “static associations” and “dynamic associations” [3], but we consider that this distinction is not adequate, since in object orientation every association has both static and dynamic features.

If static/dynamic is not an adequate classification of associations, how can we distinguish “normal” associations from other kinds of associations that seem relevant in modeling? UML has five predefined stereotypes for links ends (in the metamodel, `LinkEnd` metaclass) which are supposed to solve the dynamic associations issue, that is, how an instance can communicate with another instance without any existing (static) association between the respective classes. The five stereotypes specify different ways in which an instance is “visible” [UML, p. 2-103]:

- `«association»`: the instance is visible via association.
- `«global»`: the instance is visible because it is in a global scope relative to the link.
- `«local»`: the instance is visible because it is in a local scope relative to the link.
- `«parameter»`: the instance is visible because it is a parameter relative to the link.

³ There are some object-oriented *untyped* programming languages, such as Eiffel or Smalltalk, and even though it is completely legitimate the use of UML to model systems implemented in these languages, UML itself is strongly typed, likewise languages such as Java.

⁴ Note how the Standard is imprecise in using the concept of visibility in these definitions. Instead of “visible”, it should say “accessible”.

- `«self»`: the instance is visible because it is the dispatcher of a request.

For association ends (in the metamodel, `AssociationEnd` metaclass), we have the same five stereotypes, although their definitions are slightly different [UML, p. 2-24]. It is worth to copy them here and compare with the preceding ones, which are rather obscure:

- `«association»`: specifies a real association; default and redundant option, although it can be used for emphasis.
- `«global»`: the target is a global value known to all elements, rather than an actual association.
- `«local»`: the relationship represents a local variable inside the procedure, rather than an actual association.
- `«parameter»`: the relationship represents a procedure parameter, rather than an actual association.
- `«self»`: the relationship represents a reference to the object that owns the operation or action, rather than an actual association.

The intention of the Standard in defining these five stereotypes is not very clear. On the one hand, it seems that we should have a *coherence rule*, in the sense that a link end having a certain stereotype implies the same stereotype for its corresponding association end; but the Standard does not impose this restriction. On the other hand, the four stereotypes `«global»`, `«local»`, `«parameter»`, and `«self»` are apparently intended to give a kind of access that is not properly derived from an association, but from other circumstances, supporting the statement that there are communication links which are not instances of associations; that is, a stereotyped link end would correspond to no association end, not even to one having the same stereotype. But this would contradict the suggested coherence rule, and make the stereotypes unnecessary for association ends (that is, they would be required for link ends only)⁵.

Maybe this paradox is due to a careless writing of the Standard, rather than to a true inconsistency. We may suppose that the intention was to state that every link is an instance of an association, but there are “special” links that are not instances of normal associations, but special *implicit* associations, which do exist without need of being declared in the model, although the modeler can declare them in favor of clearness.

⁵ We have already mentioned in Section 2 another contradiction in the Standard, when it states that the communication link is not necessary in certain special situations (the same ones in which these stereotypes are defined) [UML, p. 2-114].

4. A proposed solution

In this Section we briefly propose a solution that reaffirms the principle that every link is an instance of an association, and uses association and link stereotypes consistently.

Associations

- Every association must be declared in a well-formed model, in order to specify its features and to avoid interactions that might be inconsistent with the rest of the model.
- It is not necessary that every association appears in a class diagram. It is enough that the association is represented in the underlying model.
- The different kinds of associations are distinguished by the stereotypes applied.

Links

- In a well-formed model, every link is an instance of an association.
- During the initial phases of a model's development, it is legal to represent links without specifying its association.
- A link bears the same stereotype as its association.
- Every stimulus or message instance requires a communication link.

This proposal implies a correction in the multiplicity of the `Stimulus.communicationLink` role [UML, p. 2-98] (change from 0..1 to 1..1).

5. Conclusions

CASE tools could be designed to help modelers in avoiding the baseless links problem. Probably, it is not convenient, in general, to require in a model that every link that appears in an object or collaboration diagram must correspond to an association in a class diagram. In our approach, every link is an instance of an association, but you don't need to show every association in a diagram: it is enough that these associations are represented in the underlying model, even though they do not appear in any diagram. It is convenient that the tool allows (even requires) the specification of the corresponding associations, likewise a class is specified for every object⁶: in this way you avoid that the

association properties remain unspecified or that the modeler specifies an interaction that is inconsistent with the rest of the model.

References

1. Gonzalo Génova, Juan Lloréns, Vicente Palacios. "Sending Messages in UML", *Journal of Object Technology*, vol.2, no.1, Jan-Feb 2003, pp. 99-115.
2. Object Management Group. *Unified Modeling Language Specification*, Version 1.4, September 2001 (<http://www.omg.org/>).
3. Perdita Stevens. "On the Interpretation of Binary Associations in the Unified Modelling Language", *Journal on Software and Systems Modeling*, 1(1):68-79 (2002).
4. The Precise UML Group (<http://www.cs.york.ac.uk/puml/>).

⁶ In certain development phases of a software project, especially in the initial ones, it is convenient not to specify an object's

class, and in the same way the specification of a link's association should not be mandatory. CASE tools can enable or disable this feature as it is convenient for the modeler.

An Algebraic Proposal for Handling UML Consistency

Position Statement

Egidio Astesiano - Gianna Reggio
DISI, Università di Genova - Italy
{astes,reggio}@disi.unige.it

Abstract

In this paper, we look at the consistency problems in the UML in terms of the well-known machinery of classical algebraic specifications. Thus, first we review how the various kinds of consistency problems were formulated in that setting. Then, and this is the first contribution of our note, we try to reduce, as much as possible, the UML problems to that frame. That analysis, we believe, is rather clarifying in itself and allows us to better understand what is new and what instead could be treated in terms of that machinery. We conclude with some directions for handling those problems, basically with constrained modelling methods that reduce and help precisely individuate the sources of possible inconsistencies.

1 Introduction

In this paper, we look at the consistency problems in the UML in terms of the well-known machinery of classical (logical) algebraic specifications. Within the UML context “consistency” bears roughly the same meaning of logical consistency; indeed the UML artifacts/documents are organized in “models” that correspond to logical specifications. However the nature of those artifacts is, at least apparently, so different from the traditional specifications used in the formal method community that even the definition of consistency is somewhat controversial. Among the sources of difficulty we can mention: the nature of the notation, that is visual and not directly defined adopting inductive techniques; the UML multiview approach; the use of UML artifacts throughout a software development process typically consisting of many phases, in which the same kind of document may have different meanings.

Confronted with the consistency problems in the UML, first during some work on UML semantics [8, 9] and in the development of a UML-based method [5, 6], because of our long acquaintance with algebraic development techniques, we have been tempted to look at them trying to borrow the

classical frame of logical-algebraic specifications, also for understanding and isolating the possible novelties. Indeed, in that framework one could define concepts and problems in a rather rigorous way, though of course that does not mean to solve all consistency problems, many of them being of uncomputable nature. However, among the many related valuable research attempts at dealing with consistency in UML (consider, e.g., [1] also for further references) that view is not explicit; though it appears to be underlying some treatment, notably in [7], where some rather clean informal definitions have a clear logical origin.

The purpose of this paper is first of all informative and exploratory, to establish a link between the work and terminology in the UML world and in the logical-algebraic setting. To achieve that purpose we first recall the basic setting of logical-algebraic specifications in Sect. 2, avoiding many technicalities, in a simple language adopting some conceptual object-oriented notation (a subset of UML); and, in Sect. 3, we propose a view of the UML setting in terms of the classical concepts of the logical-algebraic setting. On the basis of the new setting, we then propose, in Sect. 4, a first attempt at exploiting and learning from the outlined correspondence to deal in practice with consistency issues in the UML framework.

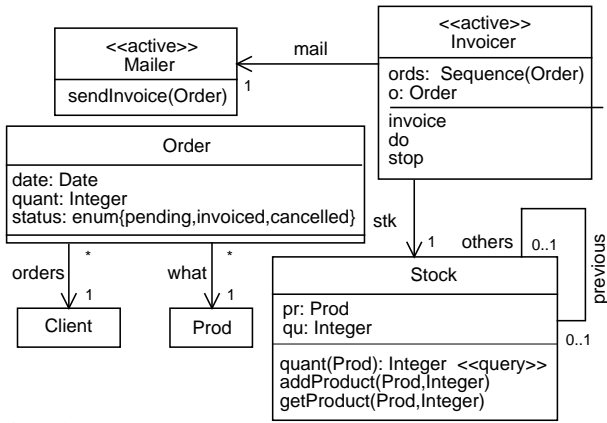
Here, by UML we intend UML 1.3 as presented by the official standard specification version 1.3, [10]. For the logical-algebraic approach we mention [11, 2].

In the paper we will use as example a simple UML model presented in Fig. 1 2, 4 and 3.

2 The Logical-Algebraic Specification Case

2.1 The Logical-Algebraic Framework

The logical-algebraic framework may be presented in a very refined way using the categorical language; here to make smoother the transition to UML and also to be understandable by the wider audience of UML specialists, we present it following the metamodelling style typical of the



Class invariants

context S: Stock inv: S.qu >= 0

context I: Invoicer inv: I.stk.previous = {}

Operation pre/postconditions

getProd(p:Prod,q:Integer)

pre: self.quant(p) >= q

post: self.quant(p) = self.quant@pre(p) - q

addProd(p:Prod,q:Integer)

post: self.quant(p) = self.quant@pre(p) + q

Figure 1. UML Class Diagram with Constraints

```

getProd(p:Prod,q:Integer) method:
  if (self.pr = p) {self.qu = self.qu - q}
  elseif (self.others <> {})
    {self.others.getProd(p,q)}
  else {null}
quant(p:Prod):Integer method:
  if (self.pr = p) {return self.qu}
  elseif (self.others <> {})
    {return self.others.quant(p)}
  else {return 0}
  
```

Figure 2. Methods Associated with Operations of the Passive Class Stock

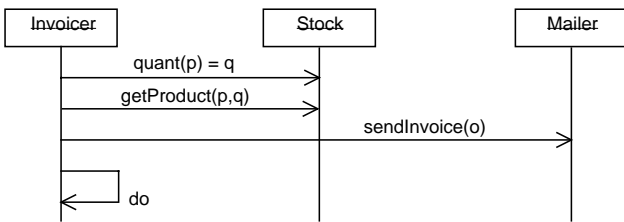


Figure 3. Sequence Diagram Showing a Successful Order Invoice

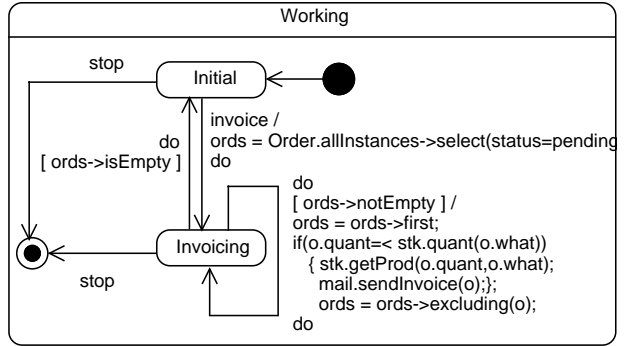
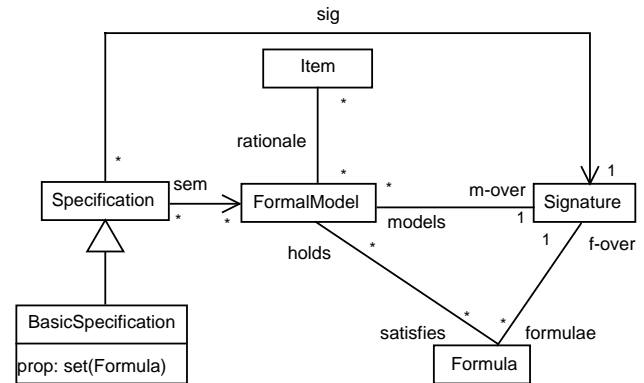


Figure 4. Statechart Defining the Behaviour of the Active Class Invoicer



General

context F: Formula inv: F.holds.m-over = F.f-over

context FM: FormalModel inv:

FM.satisfies.f-over = FM.m-over

context S: Signature inv:

S.models.includesAll(S.formulae.holds)

Well-Formedness Rules

context BSP: BasicSpecification inv:

BSP.prop.f-over = BSP.sig

Semantics

context SP: Specification inv:

SP.sig.models.includesAll(SP.sem)

context BSP: BasicSpecification inv:

BSP.sem = BSP.prop.holds

Figure 5. Logical-Algebraic Specification Framework: Basic Specifications

UML and the concept of specification method of [4]. In Fig. 5 we summarize the essential structure with the comments below, and [exemplify them for the case of first-order specification of partial abstract data types, see [2]]. Notice below the different terminology from the one of the UML: the UML models roughly correspond to the specifications here; while the (formal) models here correspond to the basic semantic structures.

- The **Item** are the specified elements [abstract data structures/types].
- The **FormalModel** are formal structures corresponding to the specified items [many-sorted partial algebras].
- The association **rationale** describes how the formal models give an abstract and precise representation of the specified items [it is easy to see how many-sorted algebras formally models data structure: carriers = sets of values, algebra functions = data operations and predicates]; see [4] for more significative examples of rationales.
- The formal models are classified by their static structure, that in the logical-algebraic case is usually defined by a **Signature**; in general a signature is a list of the *constituent features* of the formal models [many-sorted first-order signatures; in this case the constituent features are sorts, operations and predicates].
The association, whose association ends are **m-over** and **models** links the formal models with the signatures describing their structure [each many-sorted algebra is built over a many-sorted signature].
- A **Formula** is a description of a property of interest about the formal models. More precisely, a formula describes a property that is sensible only for the formal models having a given structure represented by a signature; thus each formula is built over a signature (the association with ends **f-over** and **formulae** links signatures with the formulae built over them) [first-order formulae over a many-sorted signature].
- The association, whose ends are **holds** and **satisfies**, defines when a formula holds on a formal model/a formal model satisfies a formula. Clearly, this relationship is sensible only when the linked formal models and formulae are built over the same signature, see the constraints in Fig. 5 [the usual interpretation of first-order formulae].
- A **Specification** is characterized by a signature (**sig**) and by a set of formal models, its semantics (**sem**); such models are all over

the signature of the specification (constraint $SP.sig.models.includesAll(SP.sem)$).

- A **BasicSpecification** is a specification that consists exactly of a signature and of a set of formulae over it, and determines the set of formal models satisfying all such formulae, constraint $BSP.sem = BSP.prop.holds$. The well-formedness constraint $BSP.prop.f-over = BSP.sig$ requires that the formulae of a basic specification are built over its signature.

Typically, an algebraic specification language offers together with **construct** to present basic specifications several ways to structure complex specifications, each one given by a combinator which builds new specifications from existing ones, such as **sum** or **union**, **reveal** and **rename**. Here for lack of room we consider only basic specifications.

2.2 Consistency in Logical-Algebraic Specifications

In the logical-algebraic setting the consistency problems are defined along the following lines, where we use some current terminology found in the literature about UML.

First of all we define the so-called *syntactic consistency* that is called in the formal methods/programming language community *syntactic/static correctness* or *static semantics*.

- A basic logical-algebraic specification is *syntactically consistent* whenever it consists of a set of well-formed formulae over its signature, determine by the association **f-over** of Fig. 5 (see the constraint $BSP.prop.f-over = BSP.sig$ in the same figure).

Usually in the algebraic setting the syntactically consistent basic specifications are defined in an inductive/constructive way, that is by defining directly by induction the set of all the correct formulae over a signature, and of all the correct (basic) specifications, instead of qualifying which elements in a larger set correspond to correct formulae, see, e.g., [11, 2] for inductive definition of first-order formulae.

For the semantic consistency, also in the logical-algebraic world we have the distinction between consistency of one specification (or intra-specification consistency) and vertical consistency (or inter-specifications consistency) of one specification w.r.t. another one.

- A logical-algebraic specification **SP** is (*horizontally*) *semantically consistent* (standard terminology *consistent*) iff its semantics, defined by the association **sem** of Fig. 5, is not empty ($SP.sem \neq \{\}$).

Horizontal semantic inconsistencies in the logical-algebraic case are due to the fact that a specification includes some formula that implies the negation of another

of its formulae (including the case of a formula that implies its negation).

Notice that a semantically consistent specification is not always a sensible specification of some data structure. Consider, for example, the case of a specification whose all models are isomorphic to the trivial algebra (the one whose carries have exactly one element and the interpretation of operations and predicates is the obvious one); this is consistent, but *in most of the cases* it is not what the specifier intended. Unfortunately, it is not possible to define in general this kind of specifications, they depend on the particular setting, and cannot fully banned (sometimes they are really wanted, e.g., the specification of a token requires a unique sort with just one element). However, from the methodological point of view, it may be useful to define the class of such specifications, which we name *pseudo-inconsistent*, and perhaps to introduce techniques to detect them.

The problem of vertical consistency, concerning two algebraic specifications, one *implementing* the other, is handled in a very careful way. In this setting, we speak of vertical consistency relative to a given relationship between the structures of the two specifications due to Sannella and Wirsing (see [11]), given as a function mapping specifications into specifications. For lack of room here we do not consider here vertical consistency.

3 The UML Framework

Here we try to provide a framework for the UML corresponding to the previous one for algebraic specifications. After some preliminary considerations we give a schematic view of the possible correspondence. Our presentation is sketchy in the sense that we concentrate on the basic underlying ideas; indeed a too detailed treatment here would be senseless, since the UML is evolving; the version considered here 1.3 [10]¹ is now going to be replaced by a rather different one, UML 2.0.

Here we consider UML at the level of the abstract syntax, which is as it is presented by the metamodel in the *Semantic* chapter of the official standard [10]; reference to the corresponding visual diagrams (as presented in the *Notation* chapter of [10]) will be added to help relate what we present with the current view of UML.

Recall that in the following we use the terminology of Sect. 2.1, along the schema of Fig. 5.

The UML constructs to structure models (specifications) is the package, thus UML models without packages may be considered as basic specifications, following the terminology of the previous section.

In this section we will explore the analogy with the algebraic specifications and argue that the role of basic spec-

¹To be precise the last version is UML 1.5, but there are no big differences with 1.3.

ifications can be played in the UML by the notion of *UML basic model*.

UML Item

UML models are meant to describe real-world systems, as software systems, information systems, business organizations; and thus these are the elements of *Item*.

UML Signature

In the UML, neither in the metamodel, nor in the associated notation, there is an obvious construct that may play the role of the signature in the algebraic specification case. However, if we look carefully at the various (model) elements building a UML model we may discover that many of such model elements just state which *entities* will be used in the UML model to describe the system (giving their name and their kind/type), for example classes, operations and attributes. We call *structural* such model elements.

Now, a UML model made only of structural model elements may be considered a kind of signature, since it defines the structure of the modelled system; such models will be called *signature diagrams*.

A *structural model element* may be

- * a classifier of the following kinds: class (distinguished in active and passive), datatype, interface, use case, actor and signal; clearly it will be equipped with its own particular features (e.g., attributes, operations and signal receptions for classes), but without any form of semantic constraints (e.g., the *isQuery* annotation for an operation, or the specification part for a signal reception);
- * an association, but without any semantic attribute (e.g., multiplicity and changeability)²;
- * a generalization relationship, but without any predefined constraint (only the subtype aspect matters at the structural level).

A *signature diagram* is a UML model built only of structural model elements satisfying some well-formed constraints, as

- all classes have different names,
- all attributes of a class have different names,
- all operations of a class have different names,
- the type of an attribute is either an OCL type, or the name of a class appearing in the diagram,

– . . .

²We do not include aggregation/composition in the signature diagrams, because essentially they are just a normal associations plus some constraints concerning the creation/termination of the aggregated/composed objects and those of their subparts.

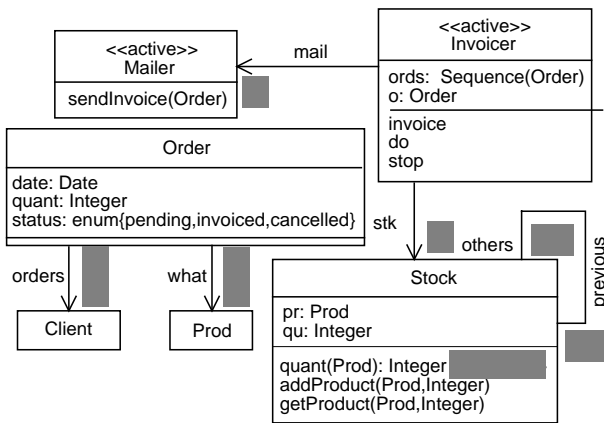


Figure 6. The signature diagram of the example UML model

Here for simplicity we have expressed the well-formedness constraints on signature diagrams by using the natural language, but they may be expressed precisely as OCL formulae.

We report in Fig. 6 the signature diagram for our running example of UML model. To help make clear the difference with the class diagram of Fig. 1 we have shadowed the parts not belonging to the signature diagram.

Notice that our signature diagram may be visualized as a particular class diagram, and that any development method based on the UML requires to provide at least a static view of the modelled system, which is a class diagram. Thus our proposal is not peculiar, but it just makes explicit the underlying splitting between the static/structural part of a UML model and the dynamic/behavioural/semantic part.

It is possible to define our signature diagrams in a very precise way at the metamodel level; we just need to introduce a new metaclass corresponding to the signature diagrams, to slightly modify the metaclasses corresponding to the structural model elements (e.g., by dropping the multiplicity from the associations) and to redefine the metaclass corresponding to a complete model (now it is an aggregate of one signature diagram and of several semantic model elements).

UML FormalModel

There is not a standard choice for the formal models for the UML, because no official formal semantics is available. Many, mostly partial, proposals may be found in the literature; the important point is that the chosen formal models should be structures able to accommodate all the aspects supported by the UML models.

Just for having one at hand to be used as reference for explanatory purposes, we mention our proposal of [8, 9],

where we advocated the use of what we call UML-formal systems. They are sufficiently general to be used also for UML 2.0 we believe.

A *generalized labelled transition system* is a 4-uple $(STATE, LABEL, INFO, \rightarrow)$, where $STATE$, $LABEL$ and $INFO$ are sets and $\rightarrow \subseteq INFO \times STATE \times LABEL \times STATE$. A transition $(i, s, l, s') \in \rightarrow$, represented as $i : s \xrightarrow{l} s'$, describes a possibility/capability of the modelled system to pass from a situation represented by s into another one represented by s' , where l describes the interaction with the external world during such transition, and i some additional information on the transition (e.g., moves of the subcomponents).

The *UML-formal systems* are a particular class of generalized labelled transition system (see [9]).

We have then to match each signature diagram with the corresponding UML-formal systems. This can be done quite simply, because the UML-formal systems of [9] include a description of the static structure of the UML model to which they give semantics. For example, they define the names available in such model and classify them in class names, attribute names, operation names and so on.

UML Formula

Surprisingly or not, the UML constructs playing the role of formulae in our setting will be particular model elements (the non-structural ones) that state properties of the *entities* used in the modelled system. We name them *semantic model elements*, and list them below.

constraints including the implicit ones (i.e., those embedded in the definition of structural model elements, as association multiplicities and signal reception specifications);

methods in the UML official standard [10] they are considered as class features, but they just define the semantics of operations; thus they have no structural effect, but restrict the formal models to those where the interpretation of some operation matches the one described by the method itself;

state machines visually presented as statecharts, they fix the behaviour of the instances of classes, or of an operation, or of a use case; thus they restrict the formal models to those where the behaviour of such elements is the one described by the state machine.

collaborations visually presented as sequence or collaboration diagrams, impose restrictions on the possible interactions (exchange of messages, i.e., operation calling) among some objects used to model a system, and so they constrain the behavior of such objects.

activity graphs visually presented as activity diagrams, impose restrictions on the causal relationships among facts happening in the associated entity (an operation, the whole system, a part of the system, a use case, ...) and so they constrain the behavior of the mentioned entities.

Then, we need to define when a semantic model element is built (well-formed) over a signature diagram (association f-over of Fig. 5).

As we have pointed out repeatedly considering the current status of the UML, it is not worthwhile to present here in detail the well-formedness conditions for all constructs. Still it is worthwhile to illustrate that kind of statement w.r.t. some basic points and significant examples. Thus we single out the well-formedness of a state machine, as a paradigmatic case of UML semantic model element.

A state machine SM is well-formed over a signature diagram ΣD iff it fulfills the following rules

- Well-Formedness Rules from UML official specification (not depending on ΣD); e.g., “A final state cannot have any outgoing transitions.”
- Well-Formedness Rules depending on ΣD , mainly about elements of the SM containing expressions/actions; e.g., for each transition of SM
 - the (OCL) expression of the guard is well-formed w.r.t. ΣD , the context class and the parameter list of the event, and has type Bool;
 - the event is well-formed w.r.t. ΣD and the context class, which depends on the kind of event (call event: the operation is an operation of the context class; change event: the (OCL) expression is well-formed w.r.t. ΣD and the context class, and has type Bool; ...)
 - the action of the effect is well-formed w.r.t. ΣD , the context class and the parameter list of the event.

Notice that to check whether an OCL expression or an action is well-formed w.r.t. ΣD , a class and a possibly empty list of typed free variables, and to find the type of a correct expression are the basic ingredients to define not only when a state machine is well-formed, but also for the other semantic model elements.

For what concerns the association holds/satisfies, [9] defines when a UML-formal system is in agreement with an OCL constraint and a state machine. We have made also some feasibility studies for what concerns collaborations and activity graphs, whose results confirm that it is possible to define when a UML formal system is in accord with one of these UML constructs.

The *formulae* in our running UML model are obviously the diagrams of Fig. 4 and 3, the methods of Fig. 2, the ex-

PLICIT constraints in Fig. 1 and the implicit ones in the class diagram in the same figure (as the association multiplicity).

UML BasicModel

The basic specifications in the UML framework are the UML models without packages, which we name *UML basic model*; indeed, the package is the basic and unique UML construct to structure a model. A UML basic model may be rearranged to explicitly present a signature diagram and a set of semantic model elements. The semantics of a UML basic model (association sem) is, then, defined as for the algebraic specifications, that is the set of the UML formal systems over its signature diagram that *satisfies* all its semantic model elements.

Notice that a UML model without packages defined following the UML metamodel of [10] may be automatically transformed into the corresponding basic model, as already hinted before. D

4 Dealing with Consistency in the UML Framework

4.1 Defining Consistency

Using the various ingredients of the UML framework defined in Sect. 3 we can now define the various kinds of consistency in the same way as for the logical-algebraic case of Sect. 2.

In the algebraic case we have that a basic specification is syntactically consistent (statically correct) whenever its signature is well-formed and all its formulae are well-formed over such signature.

For the UML case we can state that a UML basic model UBM , essentially a pair $(\Sigma D, \{SME_1, \dots, SME_k\})$ with ΣD signature diagram and for $i = 1, \dots, k$, SME_i semantic model element, is *syntactically consistent (well-formed/statically correct)* iff ΣD is well-formed and for $i = 1, \dots, k$ SME_i is well-formed over ΣD (defined by the association f-over, see Sect. 3).

Notice that in this way we do not need to define for all kinds of semantic model elements when they are pairwise syntactically consistent, just as in the logical-algebraic framework, we never had to define which pair of logical formulae are mutually syntactically consistent. Thus the technique that we have proposed is quite modular/scalable; indeed our definition of syntactic consistency can easily extended whenever the UML is extended; for each extension you have just to enlarge/restrict the set of the structural elements and of the semantic model elements, and if new kind of elements are added, just define the new well-formedness conditions w.r.t. the signature diagram.

The horizontal semantic consistency of the UML basic models is defined as in the algebraic case.

- A UML basic model *UBM* is *semantically consistent* iff its semantics, a set of UML-formal systems, is not empty ($UBM.sem \langle \rangle \{ \}$).

Notice that this definition, obviously, requires to have at hand a UML formal, or at least quite precise, semantics.

We have thus given the general precise definition, but now we have to analyse the UML models to look for the possible causes of semantic inconsistency. In the algebraic case the only causes for inconsistency are the presence in the specification either of an unsatisfiable formula or of two mutually contradictory formulae, although to check it in the general case is an undecidable problem. Instead, we find that in a UML model there are many different causes of inconsistency and of very different nature. Here, we list some of the most relevant ones:

- a pre/postcondition is in contradiction with a method definition for the same operation;
- a pre/postcondition is in contradiction with an activity graph associated with the same operation;
- a pre/postcondition is in contradiction with a state machine associated with the same operation;
- a precondition on an operation is in contradiction with a state machine or an activity graph including a call of such operation;
- an invariant constraint is in contradiction with a state machine for the same class;
- a collaboration including a role for class C is in contradiction with a state machine for class C.
- ...

Notice, that several cases are quite subtle depending on which is the chosen semantics for the UML constructs; for example, it is quite hard to decide when two collaborations including a role for a class C are contradictory. If we assume that the semantics of a collaboration is to present a possible execution/life cycle/... of the modelled system, then two collaborations will never be in contradiction.

The semantics of the UML plays a fundamental role to discover the possible kinds of inconsistency. Moreover, such semantics should also help express in a precise (also if not formal way) the reason for the possible inconsistencies.

Quite surprisingly, the actual semantics of pre/postconditions does not produce inconsistent models, but just pseudo-inconsistent ones (see Sect. 2.2). Recall that the semantics of the pre/postconditions associated with an operation of [10] is precisely intended as follows:

“postcondition: a constraint that must be true at the completion of an operation.”

“precondition: a constraint that must be true when an operation is invoked.”

Thus, with this semantics a pre/postcondition does not constrain the associated operation, but just its usage; for example, an operation that will be never called satisfies any pre/postcondition.

Some other cases of pseudo-inconsistencies are:

- an unsatisfiable invariant constraint on class C (it holds on trivial UML-formal systems where there are no instances of class C);
- two invariant constraints are contradictory (as before);
- two preconditions (postconditions) for an operation are contradictory (it holds on trivial UML-formal systems where the operation will be never called).

4.2 Checking Semantic Consistency

The list of all possible causes for semantic inconsistencies in a basic UML model seems to be very long, and a very careful analysis is needed to complete it (see Sect. 4.1). Furthermore, each kind of inconsistency poses a different kind of technical problems, from classical satisfaction problems in the first-order logic or in the Hoare logic, to check whether a sequence is a possible path in a transition tree, or if a transition tree is in agreement with a partial order. As a consequence, a method for helping detect all possible inconsistencies is not feasible. Moreover, the semantics inconsistencies are based on the UML semantics that may change, e.g., in semantics variation point, or because we are using a UML profile. On the other side, a development method based on the UML in general uses only models having a particular form (for example, some construct can be never used, or used only in a particular context, or used only in particular form, e.g., state machines only associated with active classes).

Thus, to survive with (horizontal) semantic inconsistencies we propose to design development methods based on the UML with the following characteristics.

- First of all, the method should require to produce UML models with a precise syntactic structure. Such structure must also guarantee the syntactical consistency of those models.

For example, the method may require that

- for each operation of a class there is either a pre/postcondition or a method definition, but not both;
- a use case is complemented by a set of sequence diagrams, and that any of them represents an alternative scenario;
- at most one invariant is associated with each class and at most one pre/postcondition is associated with an operation.

The problem of checking whether a model has the required syntactic form should be computable, and thus

it should be possible to develop tools to perform such check.

- The intended (formal) semantics of the UML models produced following the method should be defined.
- The UML models having the particular form required by the method and the chosen semantics should be analysed w.r.t. consistency.

Thus, it should be possible to factorize the checking of the consistency into a precise list of subproblems. If the possible causes of inconsistency are too many or too subtle, perhaps the method needs to be revised, by making more stringent the structure of the produced models, or perhaps the troubles are due to the chosen semantics.

- For each inconsistency subproblem detected in the previous activity,
 - it should be described rigorously for the developers in terms of UML constructs without formalities [if you know it, then you can avoid it].
 - guidelines helping detect its occurrences should be developed, using the available techniques, such as automatic tools, inspection techniques, check lists, sufficient static conditions,

As an experiment, we have applied the approach proposed in this subsection to analyse a method for requirement specification based on UML [5, 6]. Such method requires that the UML models presenting the requirements to have a precise form, and to be, obviously, statically consistent. The proposed approach seems to be quite effective in this case; indeed, the possible causes of semantical inconsistencies have been found explicitly, and they are not too many, because such method requires to produce UML models having a very tight structure, and makes precise the semantics of any used constructs. For what concerns the support to detect the various possible inconsistencies we are extending the tool ArgoUML³. with many new critiques, each one signalling either an inconsistency (for example, those concerning the static aspects) or a possible cause of inconsistency (e.g., a postcondition for an operation of a class with an invariant on the same class) with an explanation of the reason. One of the most problematic point, concerning semantic consistency, is to check whether a state machine is in contradiction with a sequence diagram; to help this check we are trying to use a prototyping tool [3].

The above experiment shown the fundamental importance of defining a precise explicit semantics of the used UML constructs. For example, in the method a state machine is used to define the behaviour of each use case, but

³<http://argouml.tigris.org/>

its semantics is different from the original one. Indeed, following the new semantics a state machine describes A SET of the possible lives of the instances of the context class, whereas the original semantics states that a state machines describe ALL such lives. Thus, two state machines with the new semantics can never be in contradiction.

5 Conclusions

Consistency is a really big problem in practical UML-based software development. Here, contrary to the vast majority of the current literature on the subject, we have taken an unorthodox approach, starting from the experience we have gained in many years of involvement with the logical-algebraic techniques. As an exploratory experiment, we have presented the problem and then tried to propose a framework for handling the consistency problems of the UML inspired by the framework of the logical-algebraic specifications. Admittedly at first sight the proposal may look naive; perhaps most people will be surprised by seeing a state machine playing the role of a formula; and indeed one may wonder what can be the benefit of that analogy. But, if we exploit that analogy to build a UML framework for consistency, then we can forget the terminology (formulae, formal models, signatures, etc.) and handle the consistency issues knowing exactly what should be done and thus also what we are not able to do; for example, because we do not have at hand a precise semantics. In particular we have a setting where to locate, with merits and limits, the proposed approaches.

Our analysis is preliminary and incomplete. Still, we believe that it shows the feasibility of the following, even for the new UML versions to come:

- a precise and sensible way to define the various kinds of inconsistencies (static, semantic intra-model, semantic inter-models);
- a workable method for detecting the static inconsistency (in other communities just known under the name of static correctness), that is quite modular and scalable;
- a possible methodological approach to cope with semantic intra-model inconsistency, especially with approaches that are, in our own terminology, “well-founded” and use “tight structuring”.

As aside remarks, we believe that some ADT concepts and frames are still useful to provide clarification and practical guidance; but also new problems are appearing, such as the syntax presentation of visual multiview notations, the aspect currently handled in the UML by metamodeling.

References

- [1] Consistency Problems in UML-based Software Development: Workshop Materials. In L. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar, editors, *Consistency Problems in UML-based Software Development: Workshop Materials*, Research Report 2002-06, 2002. Available at <http://www.ipd.bth.se/uml2002/RR-2002-06.pdf>.
- [2] E. Astesiano, B. Krieg-Brückner, and H.-J. Kreowski, editors. *IFIP WG 1.3 Book on Algebraic Foundations of System Specification*. Springer Verlag, 1999.
- [3] E. Astesiano, M. Martelli, V. Mascardi, and G. Reggio. From Requirement Specification to Prototype Execution: a Combination of a Multiview Use-Case Driven Method and Agent-Oriented Techniques. In *Proc. SEKE 2003*. ACM Press, 2003.
- [4] E. Astesiano and G. Reggio. Formalism and Method. *T.C.S.*, 236(1,2), 2000.
- [5] E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proc. SEKE 2002*. ACM Press, 2002.
- [6] E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications: Complete Version. In *Proc. of Monterey Workshop 2002: Radical Innovations of Software and Systems Engineering in the Future. Venice - Italy, October 7-11, 2002.*, LNCS. Springer Verlag, Berlin, 2003. To appear. Available at <ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAl103f.pdf>.
- [7] G. Engels, J. Kuester, and L. Groenewegen. Consistent Interaction of Software Components. In *Proceedings of IDPT 2002*, 2002.
- [8] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML Active Classes and Associated State Machines – A Lightweight Formal Approach. In T. Maibaum, editor, *Proc. FASE 2000*, number 1783 in LNCS. Springer Verlag, Berlin, 2000.
- [9] G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In H. Hussmann, editor, *Proc. FASE 2001*, number 2029 in LNCS. Springer Verlag, Berlin, 2001.
- [10] UML Revision Task Force. *OMG UML Specification 1.3*, 2000. Available at <http://www.omg.org/docs/formal/00-03-01.pdf>.
- [11] M. Wirsing. Algebraic Specifications. In J. van Leeuwen, editor, *Handbook of Theoret. Comput. Sci.*, volume B. Elsevier, 1990.

Maintaining Consistency between UML Models Using Description Logic

Ragnhild Van Der Straeten
Vrije Universiteit Brussel
System and Software Engineering Lab
Pleinlaan 2
1050 Brussels, Belgium
E-mail: rvdstrae@vub.ac.be

Tom Mens
Service de Génie Logiciel
Université de Mons-Hainaut
6, Av. du Champs de Mars
7000 Mons, Belgium
E-mail: tom.mens@umh.ac.be

Jocelyn Simmonds
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussels, Belgium
E-mail: jsimmond@dcc.uchile.cl

Abstract

A software design is often modelled as a collection of UML diagrams. There is an inherent need to preserve consistency between these diagrams. Moreover, through evolution those diagrams get modified leading to possible inconsistency between different versions of the diagrams. State-of-the-art UML CASE tools provide poor support for consistency maintenance. To solve this problem, an extension of the UML metamodel enabling support for consistency maintenance and a classification of inconsistency problems is proposed. To achieve the detection and resolution of consistency conflicts, the use of description logic (DL) is presented. By means of a number of concrete experiments in Loom, we show the feasibility of using this formalism for the purpose of maintaining consistency between (evolving) UML models.

1. Introduction

A software design is typically specified as a collection of UML diagrams [14]. Because different aspects of the software system are covered by many different UML diagrams, there is an inherent risk that the overall specification of the system becomes inconsistent and as such it is necessary to check the consistency between related UML diagrams. Especially in the context of design evolution, it is necessary to ensure that the overall consistency is preserved. Hence, it is important to provide a means to detect and resolve the in-

consistencies between related UML diagrams and models.

A first type of consistency, indicating consistency between different models within the same version, is called *horizontal consistency*. *Evolution consistency* indicates consistency between different versions of the same model. *Vertical consistency* indicates consistency between a model and its successor refinements. However, vertical consistency will not be treated in this paper. [12]

Unfortunately, current-day UML CASE tools provide poor support for maintaining consistency between (evolving) UML models. This results in less maintainable and comprehensible models.

To counter this problem, there is first of all a need to specify the consistency between (evolving) models in a formal and precise way. The current UML metamodel [14] provides poor support for consistency preservation and software evolution, e.g. versions are not supported. It is shown how such support can be integrated in the UML metamodel with only some minor additions.

Based on the different kinds of inconsistencies observed between UML models, a *classification of inconsistencies* is proposed. To be able to detect and resolve inconsistencies, both a formal specification of model consistency and a formal reasoning engine relying on this specification is needed. Therefore, in this paper we propose to use the formalism of description logic (DL) [2].

DL is a two-variable fragment of first-order predicate logic that offers a classification task based on the subconcept-superconcept relationship. In most description logics, this classification task is decidable and com-

plete. While the satisfiability problem is undecidable in first-order logic, most DLs have decidable inference mechanisms. These inference mechanisms allow one to reason about the consistencies of knowledge bases specified by DLs. As such these mechanisms enable the identification and resolution of consistency problems.

As description logic tool we chose *Loom* [13] because of its extensive query language and associated production rule system. This allows one to specify UML models, their evolution, consistency rules and also design improvements in a straightforward way.

In the next section the developed UML profile for model consistency is explained. Before introducing the running example used in this paper in section 3, a possible classification of inconsistencies is proposed in section 2. A motivation for the use of description logic is given in section 4. Section 5 outlines the UML profile we developed for model consistency. Section 6 discusses some experiments and section 7 gives a summary of related work. We conclude in section 8.

2. Classification of Inconsistency Conflicts

For the sake of presentation, we deliberately confine ourselves to three kinds of UML diagrams: class diagrams, sequence diagrams and state diagrams. In this section, we propose a two-dimensional classification of inconsistency conflicts that can be observed between these (evolving) UML diagrams.

The first dimension indicates whether structural or behavioural aspects of the models are affected. Structural inconsistencies arise when the structure of the system is inconsistent with respect to existing behaviour. They typically appear in class diagrams which describe the static structure of the system. Behavioural inconsistencies arise when the behaviour of the system is inconsistent with respect to existing behaviour or definitions. They typically appear in sequence and state diagrams which describe the dynamic behaviour of the system.

The second dimension considers the type of affected model. For this purpose, class diagram, sequence diagram and state diagram are classified following the four layers of the meta-tower of the MDA standard [3], i.e. **Instance**, **Model**, **Meta-Model** and **Meta-Meta-Model**. A class diagram belongs to the **Model** level because the model elements it represents (more specifically, classes and associations) serve as definitions for instances (more specifically, objects, links, transitions and events) in sequence and state diagrams which belong to the **Instance** level. Remark that only sequence diagrams representing a CollaborationInstanceSet are considered. Conflicts can occur at the **Model** level, between the **Model** and **Instance** level, or at the **Instance** level. The classes of observed conflicts are

listed in Table 1. Because of space limitations, only the *dangling (inherited) association reference* conflict which belongs to the *instance definition missing* class of conflicts is detailed in the next section. The explanation of the other *instance definition missing* conflicts can be found in [18] together with the explanation of the *incompatible behaviour* conflicts. All conflicts mentioned in Table 1 are detailed in [15].

Instance definition missing occurs when an element definition does not exist in the corresponding class diagram(s). This class of conflicts represents structural conflicts between class, sequence and state diagrams because the structure of the software system as specified in the class diagram is incomplete or incompatible with respect to existing instances. These conflicts can be caused by removing elements from a diagram or having not yet included the necessary element(s). This class of conflicts represents the following conflicts:

- *Classless instance* arises when an object in a sequence diagram is the instance of a class that does not exist in any class diagram.
- *Classless statechart* arises when the state diagram is associated to a class that does not exist in any class diagram.
- *Dangling (inherited) feature reference* arises when a stimulus, event, guard or action references an attribute or operation that does not exist in the corresponding class (or its ancestors).
- *Dangling (inherited) association reference* occurs when a link in a sequence diagram is related to a non-existing association in a class diagram. This includes inherited associations that are lost when inheritance links between classes are removed. This conflict can also be caused by the removal of existing associations or by the omission of the necessary associations when creating the class diagram.

Concrete examples of *dangling (inherited) association reference* are given in the next section.

3. Running Example

In this section, a running example is introduced that will be used throughout the paper. The automatic teller machine (ATM) example¹ is rather small but sufficiently complex to illustrate our ideas. Figure 1 shows part of a class diagram of our ATM example. This class diagram contains a *ATM* class together with its subclass *PrintingATM* and all necessary classes, associations and operations to be able to represent communication with an ATM. The sequence diagram

¹adopted from <http://www.math-cs.gordon.edu/local/courses/cs211>

	Behavioural	Structural
Model-Model		dangling (type) reference inherited association conflict
Model-Instance	incompatible definition	instance definition missing
Instance-Instance	invocable behaviour conflict observable behaviour conflict incompatible behaviour conflict	disconnected model

Table 1. Two-dimensional inconsistency conflict table

shows the execution of a withdraw transaction performed on an ATM enabling printing receipts. Starting from these diagrams we will explain the experiments we performed.

In the UML model consisting of the diagrams in Figure 1, two examples of the *inherited association conflict* can be demonstrated. The first example appears when the inheritance relationship marked as A in Figure 1 is removed. As a result the *PrintingATM* class does not inherit the association between the *ATM* and *CashDispenser* classes. As such the links C and D of the sequence diagram in Figure 1 refer to an *illegal* association. The second example appears when the association between the *ATM* and *CashDispenser* classes, marked as B in the class diagram is deleted. The links C and D refer to a non-existing association. All these experiments are detailed in section 6.

4. Description Logic

Description Logics (DLs) are a family of knowledge representation formalisms. Those formalisms allow us to represent the knowledge of the world by defining the *concepts* of the application domain and then using these concepts to specify properties of individuals occurring in the domain. The basic syntactic building blocks are atomic concepts (unary predicates), atomic roles (binary predicates) and individuals (constants). The expressive power of the language is restricted. It is a two-variable fragment of first-order predicate logic and as such it uses a small set of constructors to construct complex concepts and roles.

The most important feature of these logics is their reasoning ability. This reasoning allows us to infer knowledge that is implicitly present in the knowledge base. Concepts are classified according to subconcept-superconcept relationships, e.g. *PrintingATM* is an *ATM*. In this case, *PrintingATM* is a subconcept of *ATM* and *ATM* is the superconcept of *PrintingATM*. Classification of individuals provides useful information on the properties of individuals e.g., if an individual is classified as an instance of *PrintingATM*, we infer that it is also an *ATM*. Instance relationships may trigger the application of rules that insert additional facts into the knowledge base e.g., the specification of a rule stating that all *Withdraw* transactions debit

an *Account*, has as result that an individual known to be a *Withdraw* transaction, is also known to debit an *Account*. The classification reasoning task is one of the main reasons why we resort to DL.

Another important feature of DL systems is that they have an open world semantics, which allows the specification of incomplete knowledge. Due to their semantics, DLs are suited to express the design structure of the software application. For example, Calí *et al.* [4] translated UML class diagrams to the description logic *DLR*.

Several implemented DL systems exist (e.g., Loom, Classic, and so on). We have selected the *Loom* system for carrying out our experiments because it offers reasoning facilities on concepts and individuals for the DL *ALCQRIFO*. This logic extends the basic description logic *ALC* with qualified number restrictions on roles, inverse roles, role hierarchy and nominals. Its distinguishing feature from other DL systems, is the incorporation of an expressive query language for retrieving individuals, and its support for rule-based programming. This makes it possible to specify additional necessary conditions for individuals which are explicitly mentioned and are derived to be instances of a certain defined concept. For example, consider the following query:

```
;gives values to "the-prev-ver" role of
class concept instances
(do-retrieve (?c1 ?c2 ?et ?m1 ?m2)
 (:and
 (Class ?c1)
 (Class ?c2)
 (EvolutionTrace ?et)
 (Supplier ?et ?m1)
 (In-namespace ?c1 ?m1)
 (Client ?et ?m2)
 (In-namespace ?c2 ?m2)
 (:same-as (name ?c1) (name ?c2))))
 (tellm (the-prev-ver ?c2 ?c1)))
```

This query searches all pairs of classes of which the second class (*c2*) belongs to the next version (*m2*) of the model (*m1*) to which the first class (*c1*) belongs. If those classes have the same name, we can conclude that the second class is the next version of the first class. To make this relation

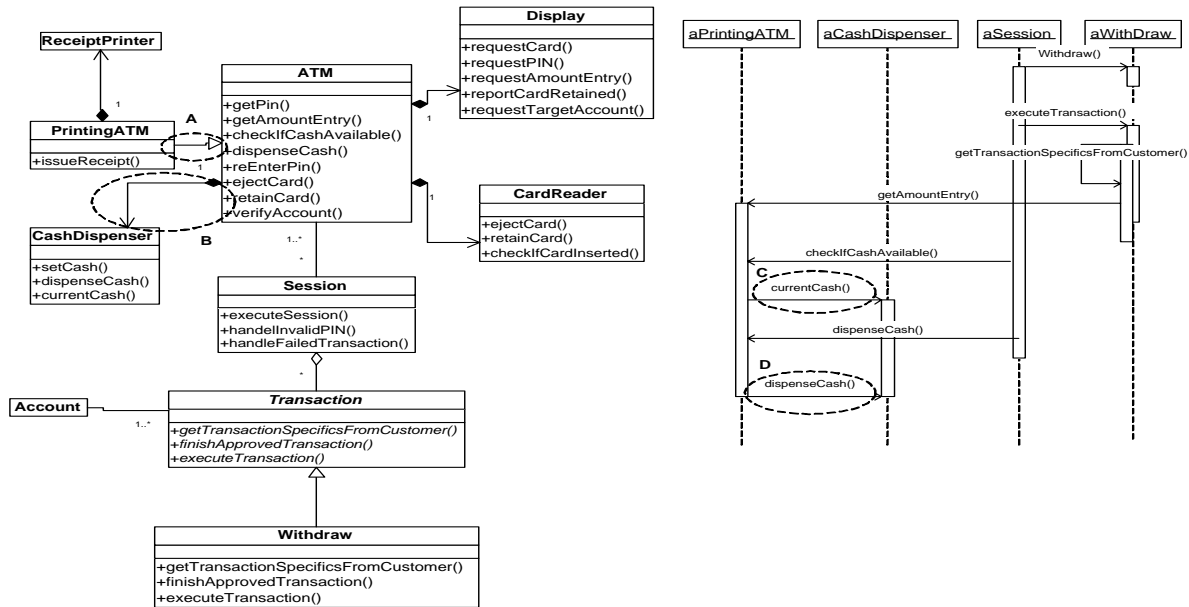


Figure 1. Class and sequence diagram ATM Example

explicit, a relation `the-prev-ver` is established between the first and the second class.

5. UML

To be able to support consistency and evolution of UML models, we need to develop a UML profile for model consistency. This UML profile contains a subset of the UML metamodel. This subset is then further extended to support evolution.

5.1. UML Profile

Because only class, sequence and state diagrams are considered, our UML profile consists of subsets of the *Core*, *Model_Management*, *Common_Behaviour*, *Collaborations* and *State_Machines* packages of the UML metamodel.

To express model evolution and model consistency, we need to extend our subset of the UML metamodel. In our UML profile, horizontal and evolution consistency can be expressed by defining stereotypes for the *Trace* metaclass: *HorizontalTrace* and *EvolutionTrace*.²

We also need a notion of *VersionedModel*, which is a stereotype for the *Model* metaclass. It adds a tag-value pair (`version,Integer`) to denote the model version.

²The term *vertical consistency* also exists, but it is not treated here. It is used to specify the relationship between a model and a refinement of this model that includes more specific details. In the UML, the *Refinement* relationship, which is a stereotype of the *Abstraction* metaclass, can be used for this purpose.

To specify the kind of models that can be related by horizontal or evolution consistency traces, the *Model* metaclass in the package *Model_Management* is stereotyped to distinguish between primitive models and composite models. *PrimitiveModel* is a stereotyped *Model* that can be specialised (stereotyped) further into *ClassModel*, *SequenceModel* and *StateModel* (representing a class diagram, sequence diagram and state diagram, respectively). *CompositeModel* is a stereotyped *Model* that is a container of *VersionedModels* all belonging to the same version. In order to keep track of the models belonging to a *CompositeModel*, a tag-value pair (`vmodel, Set(VersionedModel)`) is introduced. For *VersionedModel*, a tag-value pair (`container, CompositeModel`) is needed to refer to the *CompositeModel* it belongs to.

5.2. Translation of UML Profile into DL

Our UML profile is translated in *Loom* in terms of atomic concepts and roles as well as more complex descriptions that can be built from them with concept constructors. As an example we give the translation into *Loom* of the meta association *vmodel*. Meta associations are translated into *Loom* as roles between concepts. The association *vmodel* between a *CompositeModel* and a *VersionedModel* is translated into the role `vmodel` with as domain the concept *CompositeModel* and as range *VersionedModel*.

```
;Relation compositeModel-versionedModel
(LOOM:defrelation vmodel
:domain CompositeModel
```

```

:range VersionedModel)
(LOOM:defrelation container
  ;container is the inverse of vmodel
:is (:inverse vmodel))

```

UML metaclasses are translated into *Loom* concepts. As an example, the translation of the metaclass *CompositeModel* which is a stereotyped *Model* is given:

```

;Concept COMPOSITEMODEL
(LOOM:defconcept CompositeModel
:is (:and VersionedModel
  (:all vmodel VersionedModel))
:in-partition $VersionedModel$)

```

In the same way all the other classes, associations and attributes in the UML metamodel are translated into *Loom*. The OCL well-formedness rules of our UML profile are translated into logic rules.

For our current experiments, we manually translate the UML models into DL format. However, we are currently working on an automatic translation of UML models (exported from Poseidon in XMI 1.2 format) using XSLT. To this extent, we intend to use the SAXON XSLT processor tool (saxon.sourceforge.net).

The modeling elements of the user-defined class, sequence and state diagrams are specified as instances of the appropriate classes, association and attributes of the UML metamodel. This guarantees the consistency of the user-defined model elements with the UML metamodel. As an example, the *ATM* class is represented by the instance *ATM-1.0* of the concept *Class*. Furthermore, different properties for *ATM-1.0* are specified, e.g. this class has the operations *getPin()* and *getAmountEntry()* presented by *getPin-1.0* and *getAmountEntry-1.0* which are instances of the concept *Operation*. The complete translation of the metamodel into *Loom* code can be found in [15].

```

(create 'ATM-1.0 'Class)
(tellm (:about ATM-1.0
  (name ATM)
  (Has-feature getPin-1.0)
  (Has-feature getAmountEntry-1.0)
  (Is-parent-of PrintingATM-1.0)
  (IsAbstract false)
  (In-namespace Class-Diagram-1.0)))

```

6. Experiments

To carry out our experiments, the diagrams of Figure 1 are manually translated into *Loom*. To detect and resolve inconsistencies between models *Loom*'s query processor is used. Due to space limitations only important fragments of the developed *Loom* predicates are shown. All predicates for all the inconsistencies described in section 2 can be found in [15].

6.1. Inherited association conflict - illegal associations

To detect links that reference illegal associations due to the fact that the inheritance link between two classes is removed, the following predicate is used:

```

(defun illegal-link (?link ?assoc)
  (let* ((?list
    (retrieve (?class1 ?class2 ?class3 ?class4
      ?assocEnd1 ?assocEnd2
      ?stim ?obj1 ?obj2)
    (:and
      (assoc-assocEnd ?assoc ?assocEnd1)
      (assoc-assocEnd ?assoc ?assocEnd2)
      (has-participant ?assocEnd1 ?class1)
      (has-participant ?assocEnd2 ?class2)
      (link-stimulus ?link ?stim)
      (received-by ?stim ?obj1)
      (sent-by ?stim ?obj2)
      (instance-of-class ?obj1 ?class3)
      (instance-of-class ?obj2 ?class4))))
    (if (equalp NIL (related ?list)
      (format t "~S association exists,
        but not available through inheritance"
          ?assoc)
      )))
  ))

```

First of all, the classes related through the association *?assoc* and the classes linked by this association in the sequence diagram are collected in the variable *list?*. The predicate *related* checks if those classes are related by inheritance. If this is not the case, we conclude that the association exists but is not available through inheritance. If this predicate is applied to the example where the inheritance link between the *ATM* and *PrintingATM* classes is removed, the obtained results indicate that, even though the association between the *ATM* and *ClassDispenser* (*CashDispenser-ATM-1.0*) exists in the class diagram, the *PrintingATM* class no longer inherits this association and cannot reference it.

```

|I|CASHDISPENSER-ATM-1.0 association exists,
but not available through inheritance
NIL

```

6.2. Inherited association conflict - non-existing associations

To detect links that reference non-existing associations, the following query is used:

```

(do-retrieve (?link ?assoc)
  (:and
    (Link ?link) ;is ?link a link
    (Link-association ?link ?assoc)
    ;does ?link have as corresponding
    ;association ?assoc
    (In-namespace ?assoc NIL))
  ))

```

```

        ;?assoc does not exist in any
        ;diagram
(format t "~S association does not exist
        in any class diagram" ?assoc))

```

This predicate is applied to the example where the association between the *ATM* and *CashDispenser* classes (*CashDispenser-ATM-1.0*) is removed and it automatically detects that the association *CashDispenser-ATM-1.0* does not exist anymore.

```

|I|CASHDISPENSER-ATM-1.0 association does
not exist in any class diagram
NIL

```

For most of the detected inconsistencies we also provide rules to automatically resolve them. For example, it is possible to add the non-existing association between the classes *ATM* and *CashDispenser* with or without user interaction.

We applied all the above mentioned rules to the example of section 3. This enables us to detect and resolve all inconsistencies as specified in that section. Other, more extensive experiments are performed in [15].

7. Related work

Finkelstein *et al.* [9] explain that consistency between partial models is neither always possible nor is it always desirable. They suggest to use temporal logic to identify and handle inconsistencies. Grundy *et al.* [11] claim that a key requirement for supporting inconsistency management is the facilities for developers to configure when and how inconsistencies are detected, monitored, stored, presented and possibly automatically resolved. They describe their experience with building complex multiple-view software development tools supporting inconsistency management facilities. Our DL approach is also easily configurable, by adding, removing, of modifying logic rules and facts in the knowledge base.

A wide range of different approaches for checking consistency has been proposed in the literature. Engels *et al.* [7] motivate a general methodology to deal with consistency problems based on the problem of protocol statechart inheritance. In that example, statecharts as well as the corresponding class diagram are important. Communicating Sequential Processes (CSP) are used as a mathematical model for describing the consistency requirements. This idea is further enhanced in [6, 8] with dynamic meta modeling rules as a notation for the consistency conditions because of their graphical, UML-like notation. Model transformation rules are used to represent evolution steps, and their effect on the overall model consistency is explored.

Ehrig and Tsiolakis [5] investigate the consistency between UML class and sequence diagrams. UML class diagrams are represented by attributed type graphs with graphical constraints, and UML sequence diagrams by attributed

graph grammars. As consistency checks between class and sequence diagrams only existence, visibility and multiplicity checking are considered. In [17] the information specified in class and statechart diagrams is integrated into sequence diagrams. The information is represented as constraints attached to certain locations of the object lifelines in the sequence diagram. The supported constraints are data invariants and multiplicities on class diagrams and state and guard constraints on state diagrams. Fradet *et al.* [10] use systems of linear inequalities to check consistency for multiple view software architectures. Finally, note that consistency of models should not be confused with consistency of a modeling language. UML has been formalized within rewriting logic and implemented in the Maude system by Ambrosio Toval and his students [1, 16]. Their objectives are to formalize UML and transformations between different UML models. They focus on using reflection to represent and support the evolution of the metamodel.

8. Conclusion

In this paper we propose and validate an approach to detect and resolve inconsistencies between different versions of a UML model, specified as a collection of class diagrams, sequence diagrams and state diagrams. For research purposes, we restrict ourselves to a significant subset of the UML metamodel.

The formalism used is description logic, a decidable fragment of first-order predicate logic. More specifically, we use the *Loom* knowledge representation tool to formally specify UML models as a collection of concepts and roles.

Logic rules are used to detect and to suggest ways to resolve inconsistencies. Based on a simple but illustrative example, we illustrate the feasibility of the approach. Until now, we only use small examples. The question remains if our approach remains feasible for larger models.

Obviously, a lot of future work remains to be done. We will investigate how the formal properties of DL can help us to prove interesting properties about consistency between UML models. We need to further automate the consistency maintenance process, by using a description logic tool as a basic engine for a UML CASE tool (such as Poseidon), and providing feedback about the detected inconsistencies to this CASE tool. We need to incorporate other kinds of UML diagrams (such as collaboration diagrams and activity diagrams). We also need to translate the corresponding OCL well-formedness rules. We need to extend our ideas to deal with co-evolution and consistency maintenance between different levels of abstraction, more specifically, source code and UML models. This idea, which is also explored in [19] will allow us to provide better formal support for the round-trip engineering and model-driven architecture process.

References

- [1] J. Alemán, A. Toval, and J. Hoyos. Rigorously transforming UML class diagrams. In *Proc. 5th Workshop Models, Environments and Tools for Requirements Engineering (MENHIR)*, 2000. Universidad de Granada, Spain.
- [2] F. Baader, D. McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.
- [3] J. Bézivin and N. Ploquin. Tooling the MDA framework: a new software maintenance and evolution scheme proposal. *Journal of Object-Oriented Programming (JOOP)*, 2001.
- [4] A. Calí, D. Calvanese, G. De Giacomo, and M. Lenzerini. Reasoning on UML class diagrams in description logics. In *Proc. of IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development (PMD 2001)*, 2001.
- [5] H. Ehrig and A. Tsiolakis. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *ETAPS 2000 workshop on graph transformation systems*, pages 77–86, March 2000.
- [6] G. Engels, J. Hausmann, R. Heckel, and S. Sauer. Testing the consistency of dynamic UML diagrams. In *Proc. Sixth International Conference on Integrated Design and Process Technology (IDPT 2002)*, June 2002. Pasadena, CA, USA.
- [7] G. Engels, R. Heckel, and J. M. Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In M. Gogolla and C. Kobryn, editors, *Proc. Int'l Conf. UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, number 2185 in Lecture Notes in Computer Science, pages 272–286. Springer-Verlag, October 2001. Toronto, Canada.
- [8] G. Engels, R. Heckel, J. M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *Proc. Int'l Conf. UML 2002 - The Unified Modeling Language. Model Engineering, Concepts, and Tools*, number 2460 in Lecture Notes in Computer Science, pages 212–227. Springer-Verlag, October 2002. Dresden, Germany.
- [9] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. In *European Software Engineering Conference*, pages 84–99, 1993.
- [10] P. Fradet, D. Le Métayer, and M. Périn. Consistency checking for multiple view software architectures. In *Proc. Int'l Conf. ESEC/FSE'99*, volume 1687 of *Lecture Notes in Computer Science*, pages 410–428. Springer-Verlag, 1999.
- [11] J. C. Grundy, J. G. Hosking, and W. B. Mugridge. Inconsistency management for multiple-view software development environments. *IEEE Transactions on Software Engineering*, 24(11):960–981, 1998.
- [12] L. Kuzniarz, G. Reggio, J. Sourrouille, and Z. Huzar. Consistency problems in UML-based software development: Workshop materials. Research Report 2002-06, 2002. Available at <http://www.ipdt.bth.se/uml2002/RR-2002-06-.pdf>, Blekinge Institute of Technology, October 2002.
- [13] R. MacGregor. Inside the LOOM description classifier. *SIGART Bull.*, 2(3):88–92, 1991.
- [14] Object Management Group. Unified Modeling Language specification version 1.5. formal/2003-03-01, March 2003.
- [15] J. Simmonds. Consistency maintenance of UML models with description logics. Master's thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium and Ecole des Mines de Nantes, France, September 2003.
- [16] A. Toval and J. Alemán. Formally modeling UML and its evolution: a holistic approach. In S. Smith and C. Talcott, editors, *Formal Methods for Open Object-based Distributed Systems IV*, pages 183–206. Kluwer Academic Publishers, 2000.
- [17] A. Tsiolakis. Semantic analysis and consistency checking of UML sequence diagrams. Master's thesis, Technische Universität Berlin, April 2001. Technical Report No. 2001-06.
- [18] R. Van Der Straeten, T. Mens, J. Simmonds, and V. Jonckers. Using description logic to maintain consistency between UML-models. In *Proc. 6th International Conference on the Unified Modeling Language*, 2003.
- [19] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. A UML extension for automating source-consistent design improvements based on refactoring contracts. In *Proc. 6th International Conference on the Unified Modeling Language*, 2003.

A Plug-In for Flexible and Incremental Consistency Management

Robert Wagner¹, Holger Giese², Ulrich A. Nickel¹
Software Engineering Group
Department of Computer Science
University of Paderborn
Warburger Straße 100
D-33098 Paderborn, Germany
[wagner|hg|duke]@upb.de

Abstract

The problem of inconsistency detection and management is central to the development of large and complex software systems. Although sufficient support for model consistency is a crucial prerequisite for the successful and effective application of CASE tools, the current tool support is rather limited. In this paper we present a flexible and incremental consistency management realized in the Open Source UML CASE tool Fujaba. The consistency management is highly configurable and can be adapted individually to user, project or domain specific demands. For the specification of syntactical consistency rules we provide a built-in formalism based on graph grammars.

1. Introduction

Nowadays, the modeling of large object-oriented software systems incorporates many informal and semi-formal notations, with the Unified Modeling Language (UML) [19] being the most prominent example. The UML provides different diagrams describing the system under construction from different, partly overlapping view points, e.g. class diagrams for the static structure and state charts for the behavior of the system. This separation of concerns on the one hand reduces the complexity of the overall specification, but on the other hand the increasing number of used notations very often leads to a wide

range of inconsistencies [9, 10]. For example, syntactical inconsistencies violating the well-formedness of models, behavior inconsistencies between different diagrams [7] or inconsistencies during refinement of diagrams [5]. Of course, the general problem of inconsistency is much broader and comprises a large number of disciplines. For a survey we refer to [17] and for a research agenda to [8].

Meanwhile, there are many commercial and Open Source UML modeling tools available. These tools promise increasing development productivity and a gain in quality of the system under development, e.g. by automated code generation from the design models. However, the built-in consistency management of many tools is not satisfactory.

One reason for this is, that tools often try to enforce consistency and do not allow temporal inconsistencies during development. In [1, 17], Nuseibeh et al. emphasize the importance of tolerating inconsistencies and propose a conceptual framework for inconsistency management which allows inconsistencies to be ignored, deferred, circumvented, ameliorated, or resolved.

Another reason is, that most tools with consistency checking support only consistency checks on user demand [18, 20]. In those tools, the whole specification is checked even if no changes were made. However, starting from a consistent state, inconsistencies are introduced by model changes. Hence, a smarter approach will monitor model changes and check only the fraction of the model which was modified. Thus, the effort spent on consistency checking will be

¹. This work has been supported by the DFG grant GA 456/7 ISILEIT as part of the SPP 1064.

². This work was partly developed in the course of the Special Research Initiative 614 -- Self-optimizing Concepts and Structures in Mechanical Engineering -- University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

reduced.

Another problem concerns the flexibility of consistency rules being checked. In most CASE tools, the consistency checks being performed are rather static and predefined as they are hard coded into the tool [15]. Thus, new consistency rules neither can be added nor can existing consistency rules be adapted to special user, enterprise, project, target language, or domain specific demands. However, during large projects you will never obtain a complete set of rules covering all relevant inconsistencies. In fact, the set of consistency rules will be expanded and refined through the whole lifecycle of a project [17]. Thus, for a tool developer it becomes infeasible to identify all consistency rules in advance.

In this paper we present a plug-in for a flexible and incremental consistency management realized within the FUJABA TOOL SUITE [3]. FUJABA itself is an Open Source UML CASE tool project. It was started by the software engineering group at the University of Paderborn in fall 1997 and has a special focus on code generation from UML diagrams resulting in a visual programming language. Hence, consistency management was an important issue from the beginnings since consistent specifications are a required prerequisite for an error-free implementation.

The structure of the paper is as follows: In the next section we will present the architecture of our consistency management plug-in. We will discuss in more detail how a more appropriate consistency management should look like and how the previously discussed techniques are realized in our plug-in. Section 3 explains the requirements to be fulfilled when specifying a consistency rule for our plug-in. This is accompanied by an example of the built-in consistency rule specification based on graph-grammars. Section 4 gives an overview about different applications for our consistency management and outlines how domain specific consistency rules are employed. Finally, we conclude and outline future work.

2. Architecture

As outlined in the introduction, sufficient support for model consistency is a crucial prerequisite for the successful and effective application of CASE tools. The current support of this requirement is, however, rather limited. Either the consistency is enforced and therefore often hinders the systematic development of

appropriate solutions by rather tedious restrictions or consistency can only be checked on request and then usually results in a large number of more or less syntactical errors. What is instead required is a more flexible and incremental consistency management.

To understand how a more appropriate consistency management should look like, we first have to understand the different phases of (1) detecting a possible inconsistency, (2) checking that is indeed an inconsistency, and (3) resolving the inconsistency w.r.t. a given consistency rule. The general lifecycle of a model modification w.r.t. consistency is depicted in Figure 1.

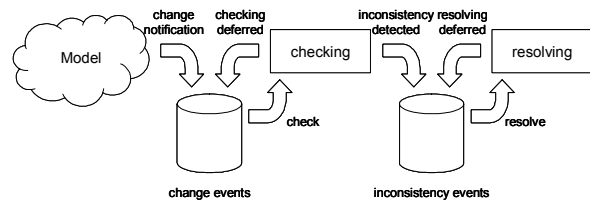


Figure 1. Consistency lifecycle of a model modification

To detect a possible inconsistency, most tools usually simply wait for the user command to check all rules. A better idea realized within the presented plug-in is to observe the model modifications and employ an on demand consistency management approach that reacts on different change events or specific user commands in a flexible manner. The incremental checking algorithms employed in the plug-in only study that fraction of the system which has been modified and thus often the effort to check consistency can be drastically reduced.

For each change event we check whether it results in a modification that is inconsistent. The change events are processed by analysis engines which usually simply report detected inconsistencies via related events. In more complex cases multiple analysis engines may cooperate via higher level change events which permit to decompose the detection of inconsistencies into a sequence of less complex steps. If possible, automated checking mechanisms are employed. If, however, only user inspection can decide whether we have a real inconsistency or not, the consistency management plug-in keeps track of the required manual inconsistency checks and informs the user.

For the final step of resolving an inconsistency, also automatic as well as manual handling is supported. We decide to not combine these two last phases, as a proper consistency management system must tolerate that a

detected inconsistency remains in the model until it is resolved in a later stage of the development when the required context information is available.

For the automated detection, checking and repair mechanisms, we provide a built in formalism which employs graph grammars. Additionally, external tools as for instance model checkers can be integrated for checking and repair. Therefore, different tools which transform the model into a specific semantic domain and then employ their special checking or repair algorithm can be integrated (see Section 4).

As the different employed techniques can show rather different run-time complexity, we permit to configure for each rule whether automatic checking and/or repair is executed automatically or on demand.

Starting from a consistent model, inconsistencies can only be introduced by model changes. In our tool, models are changed by executing user commands from predefined pull-down or pop-up menus. In general, a user command is not an atomic operation and usually consists of more than one modification to the meta-model instance. Thus, inconsistencies during the execution of a user command are not unusual and are often temporary. A consistency check after each single modification will produce unnecessary overhead. Moreover, the consistency management will inform and confuse the engineer with false positive inconsistency notifications.

To tackle this problem, the automatic consistency checking within our plug-in is not executed until a user command is completed. During the execution of a user command all references to the modified objects are stored in a set. Additionally this set is marked with the command's name responsible for the modification, i.e. the cause of the modification. This allows us to specify some consistency rules and/or repairing actions which will be not executed after this special command. Once the user command complete, the set is stored in the change event queue and is ready for being checked.

The consistency checking runs in the background and processes each change event, i.e. each set of modified elements, in a sequential manner. Nevertheless, the background processing of change events can result in problems due to the parallel user interactions. To overcome this problem the user interactions and the automatic checking are synchronized using a coordination strategy which gives user interactions a higher priority to not hinder the engineer. However, the consistency checking of a rule cannot be interrupted at any given point of time since

the analysis results may become invalid after a user interaction. Therefore the checking mechanism is interrupted only after a consistency rule was completely checked and before a new rule is executed. To guarantee a short response time for user interactions, the execution time of consistency rules has to be short. Consistency rules with a long execution time will block up the CASE tool and should only be executed on demand or in parallel in the background.

If an inconsistency is detected, it can be resolved either automatically or manually. In the case of automatic resolution of inconsistencies, some consistency rules and repair actions may introduce new inconsistencies. Even worse, contradictory consistency rules and repair actions can result in non terminating chain of change events, checking activities, inconsistency events, repair activities and change events. To prevent such a cyclic execution and to detect repair actions which introduce inconsistencies, the algorithm was extended and works in two phases.

In the first phase each necessary repair activity is executed. During a repair, all elements modified by the appropriate repair action are stored and it is noticed which consistency rule caused the repair action to be executed.

In the second phase, the consistency rules are executed once again for the previously stored elements collected during the preceding repair. If an inconsistency is detected, it is checked whether this inconsistency already occurred during the preceding checking and repair phase w.r.t. the current consistency rule. If the inconsistency occurs for the first time this means, that a repair action is faulty. Otherwise, i.e. if the inconsistency was already repaired in the first phase, a contradictory rule introduced the inconsistency once again. In both cases a further repair action is not executed. Instead, the user is informed about the detected problems and has to resolve the conflict in the consistency rule(s) and repair action(s).

In Figure 2 the part of the consistency management system architecture relevant for the user is summarized. The rules are organized in catalogues and different categories, which permits to support domain or project specific consistency management system configurations as discussed in the following subsection. The system employs a given configuration and binds the rule catalogue and its analysis engines at run-time into the FUJABA CASE tool. These rules are then applied to the currently developed model with the beforehand outlined execution options.

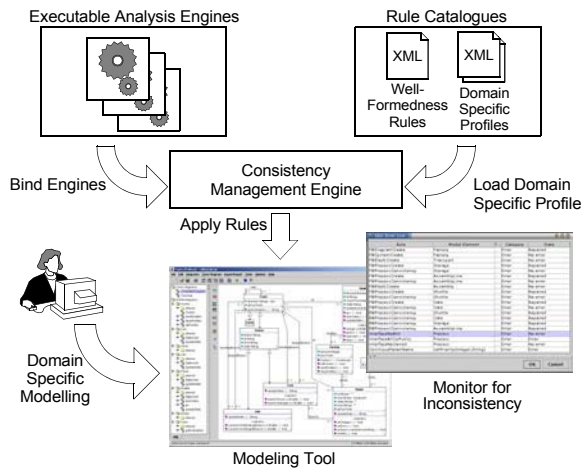


Figure 2. Using the consistency management tool

For example, consistency rules and categories can be activated or deactivated by the user on demand. If a category is deactivated, the entries of the category are not considered and thus all the rules within a category are not checked. This way, the engineer can adjust the consistency checking to her/his current needs.

With another setting the user can specify whether or not she/he will be informed immediately about any detected inconsistencies. If the immediate notification option is selected, the system informs the user as soon as an inconsistency is detected by displaying an information dialogue. In most cases, especially for minor inconsistencies, the option will be deactivated in order to not hinder the engineer.

If an inconsistency is detected, it can be resolved either automatically or manually. This behavior can be controlled by the repair option. If no repair action is specified or the automatic repair is disabled, the developer is informed about the unresolved inconsistency and must resolve it manually. This way, the automatic resolution of inconsistencies can be also circumvented by the user if required.

The architecture relevant for the administration of the consistency management system is visualized in Figure 3. The plug-in permits to first develop the required consistency rules with a UML extension based on graph grammars as outlined in the following section. Additionally the rules can be organized in form of rule catalogues. Thus, catalogues for different domains, projects or even process phases can be developed. Such pre-defined catalogues can then be loaded on demand or at start-up of the plug-in to

configure the FUJABA CASE tool to the project specific needs of the developer and its current activity. The engineer may switch on demand between different profiles containing different consistency rules, e.g. one profile for the analysis and another for the design life-cycle.

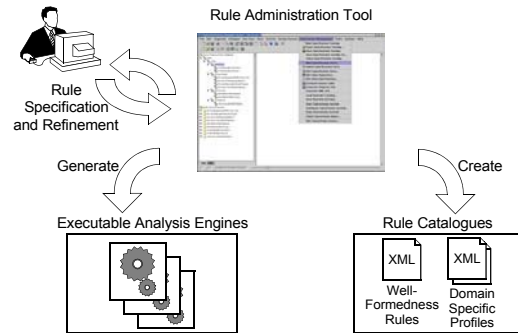


Figure 3. Rule administration in the consistency management tool

3. Rule Specification

The Unified Modeling Language (UML) is defined by the abstract syntax of the underlying meta-model [19]. We will use a simplified fragment of this logical model as a running example.

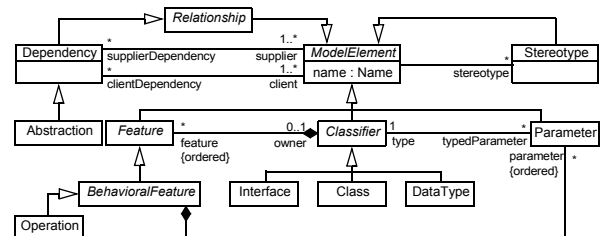


Figure 4. Simplified fragment of the UML meta-model

Figure 4 shows the simplified fragment of the meta-model concerning class diagrams. To simplify the meta-model fragment we have omitted some classes in the original inheritance hierarchy, e.g. the meta-classes *Stereotype* and *Classifier* inherit directly from the *ModelElement* meta-class and not - as in the original definition - from the *GeneralizableElement* meta-class. We have also omitted classes not needed for our example, e.g. *Association* and *StructuralFeature*.

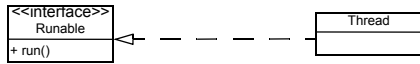


Figure 5. Inconsistent UML model

In Figure 5, a sample class diagram is shown. The diagram contains the interface *Runnable* and the class *Thread* which are connected by a realization dependency. Hence, the *Thread* class has to realize the operation *run()* from the interface *Runnable*. Note that this does not hold for our example since the class *Thread* does not define an appropriate operation *run()*.

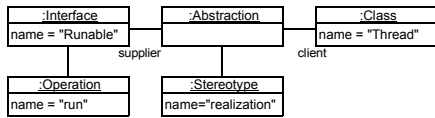


Figure 6. Meta-model instance

In Figure 6, our sample class diagram is represented as a meta-model instance using the simplified meta-model fragment from Figure 4. The object of type *Class* represents the *Thread* class. The instances of type *Interface* and *Operation* represent the *Runnable* interface and its operation *run()*. The realization dependency is represented by an instance of *Abstraction* and an associated *Stereotype* instance named *realization*. The *Interface* and the *Thread* have links to the *Abstraction* object. The former acts in a supplier role whereas the latter acts in the client role.

Every instance of a meta-class has to conform to the abstract syntax specified by the meta-model. To obtain a syntactically correct model, additional well-formedness rules have to be specified. In the UML, well-formedness rules are notated as Object Constraint Language (OCL) expressions.

```

context Classifier inv:
  self.specification.allOperations->
  forAll(interOp|self.allOperations->
  exists(op|op.hasSameSignature(interOp)))
  
```

Figure 7. Well-formedness rule as OCL expression

A relevant well-formedness rule of the UML states that for each *Operation* in a specification realized by the *Classifier*, the *Classifier* must have a matching *Operation* [19]. The corresponding OCL expression is depicted in Figure 7. Note, that our example in Figure 5 violates this well-formedness rule because the realizing class *Thread* does not declare a counterpart for the specified interface operation *run()*.

```

context Classifier:
  specification : Set(Classifier);
  specification = self.clientDependency->
  select(d|
  d.ocIsKindOf(Abstraction)
  and d.stereotype.name = "realization"
  and d.supplier.ocIsKindOf(Classifier))
  .supplier.ocAsType(Classifier)
  
```

```

context Classifier:
  allOperations : Set(Operation);
  allOperations = self.allFeatures->
  select(f|f.ocIsKindOf(Operation))
  
```

```

context BehavioralFeature:
  hasSameSignature(b:BehavioralFeature):Boolean;
  hasSameSignature(b) =
  (self.name = b.name) and
  (self.parameter->size = b.parameter->size) and
  Sequence{1..(self.parameter->size)}->
  forAll(index:Integer|
  b.parameter->at(index).type =
  self.parameter->at(index).type and
  b.parameter->at(index).kind =
  self.parameter->at(index).kind)
  
```

Figure 8. Additional operations used in the OCL rule

Although the presented OCL expression seems to be quite small it has to be mentioned for clarity that some additional operations are needed. They are depicted in Figure 8. The operation *specification* yields the set of *Classifiers* that the current *Classifier* realizes. The operation *allOperations* results in a set containing all *Operations* of the *Classifier* itself and all its inherited *Operations*. The operation *hasSameSignature* checks if the argument has the same signature as the instance itself.

The main drawback of the OCL is the fact that it is not an operational language. Hence, the OCL lacks the ability to modify object structures which is a required prerequisite for automated resolving of inconsistencies.

The modification of object structures is a well known application for graph-grammars [21]. Hence, we have decided to use graph rewriting rules both to specify the properties to be checked and the resolution actions for detected inconsistencies. The idea of using graph-grammars for the visual specification of consistency rules and appropriate repair actions is not new [2, 6]. We rely on these concepts and show how they can be applied in a CASE tool (cf. [24]).

Let us return to our example and see how the well-formedness rule from Figure 7 can be expressed with a graph-grammar. The consistency rule depicted in Figure 9 contains the specification of a counter example, i.e. it specifies an inconsistent situation in the meta-model instance that is not allowed.

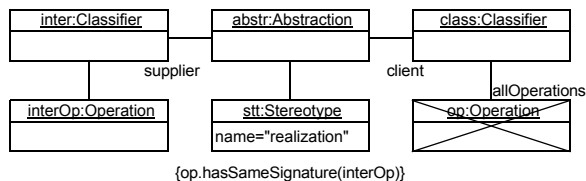


Figure 9. Inconsistency rule expressed as a graph-grammar

Compared to the object-structure of our concrete example in Figure 6 the specified rule is more general. It conforms to the OCL well-formedness rule since it describes the general situation between specification and realization classifiers of different kinds. Thus, this rule also holds for our example where the specification classifier is of type *Interface* and the realization classifier is of type *Class*.

In a graph-grammar based consistency check, a graph rewriting system will search for the pattern and as soon as a match has been found an inconsistency will be reported to the user.

The consistency checking is performed by applying the graph-grammar rule as follows: After a change event has occurred, the applicability of the rule is checked by looking for a match of the modified model element in the specified graph-grammar. If the element can be mapped to one of the graph-grammar nodes, the remaining structure is searched for. If there is no such a match in the consistency rule, or if the remaining structure cannot be found, the consistency rule is not applicable for this element.

If a match for the whole structure is found, all negative application conditions (NAC) [12] have to be checked. They are specified by cross out nodes and express that some parts must not exist for a rule to be applied. The NAC node may be accompanied by an additional condition specifying the forbidden node in more detail.

In our example the additional condition *op.hasSameSignature(interOp)* express that only those *Operation* nodes have to be examined which have the same signature as the operation *interOp* defined in the specification classifier. If no matching operation *op* in the realizing classifier is found, i.e. no operation having the same signature as the operation *interOp* defined in the specification classifier, an inconsistency is found. Otherwise, either the rule was not applicable or there is already an operation *op* in the realizing classifier which realizes the operation *interOp* from the specification classifier.

The repair activity is only executed if automatic inconsistency resolving is activated. In our example the repair activity is simply performed by creating and adding the missing *Operation* instance *op* to the realization classifier and assigning it the signature from the operation *interOp* defined in the specification classifier. It is also specified using a graph grammar and is depicted in Figure 10.

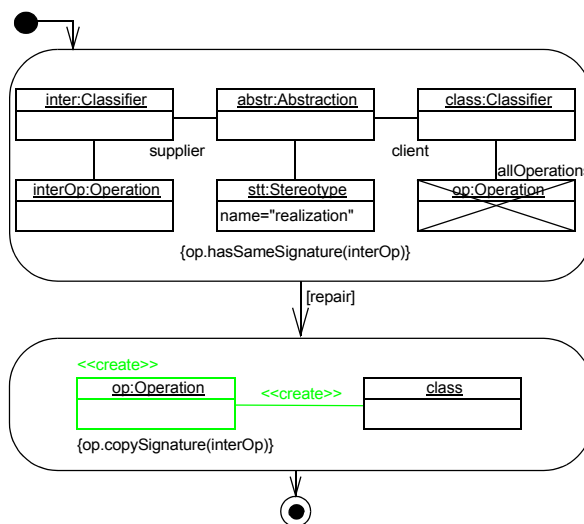


Figure 10. Repair action for automatic inconsistency resolution

Handling inconsistencies may be as simple as adding or deleting some information from the description. However, in some cases resolving inconsistencies immediately is not always possible or desired [1, 17]. If, for example, the user deletes a realization operation it will be not very clever to recreate the deleted operation just to ensure consistency. Often additional information is needed from the engineer. Therefore, the automated repair may remain unspecified.

In our opinion, visual expressions are more readable than their textual counterparts. Of course, it is rather a matter of taste and education whether a visual representation is more readable than a textual OCL expression or the other way round. Therefore, our consistency checking architecture is not bound to a particular rule specification technique. Thus, it is also possible to integrate a checking mechanism which employs OCL expressions (cf. [4]).

4. Applications

As already mentioned, we implemented the described consistency management within the FUJABA TOOL SUITE. Therefore, we were able to evaluate our results in practical use, namely in the ISILEIT project. It deals with the agent-oriented specification and modelling of distributed flexible production control systems [16]. The considered case study consists of a track based material flow system with autonomous shuttles. These shuttles supply the robots, which perform different productions tasks, autonomously with material.

Within the project we have integrated parts of the Specifications and Description Language (SDL) [14] and the UML resulting in an executable specification language. The integration mainly refers to the SDL-UML mapping defined in [13]. Of course, the SDL model and the corresponding UML model have to be consistent, according to these mapping rules. These consistency checks are performed by our consistency management system.

Another aspect of the case study is that we have to generate code for different target platforms. We use Programmable Logic Controllers (PLCs) [22, 23] to control the stations of the production line. Therefore, we have to take some restrictions into account. For example, when specifying the behavior of a PLC, we do not allow the instantiation of objects at runtime, because dynamic memory allocation is not possible for PLCs. Moreover, the statechart, which defines the reactive behavior of the PLC must not terminate. These two requirements are proved statically by applying consistency rules as described in section 3.

A second application of the presented consistency management of the FUJABA TOOL SUITE is currently developed within the Special Research Initiative 614 -- Self-optimizing Concepts and Structures in Mechanical Engineering -- University of Paderborn. A compositional model checking approach for real-time UML models of mechatronic systems [11] is supported by visualizing the validity of model checking results for given temporal logic constraints in the UML diagrams in a consistent manner.

At first, the consistency management is used to detect whether a model change might invalidate a temporal logic constraint that has been checked beforehand. If a model change might have affected the constraint, the status of the constraint is changed to „*unknown*“. Then, the model checking is initiated and

is processed in background mode to check the constraints for the relevant fraction of the modified model. While in the usual case the scalability problems of model checking renders such an approach impractical, the compositional nature of the employed approach ensures, that only small models have to be checked when changing the model. When the background model checking process terminates, the analysis result is integrated into the model by setting the constraint as either „*true*“ or „*false*“ depending on the outcome of the process.

If in the meantime before the model checking process has terminated a related model element has been changed again, the model checking process is aborted. To avoid the overhead of frequent starts and aboard of model checking processes for a specific constraint, the initiation of background processes is delayed until no change activities which could effect the constraint has occurred in a reasonable time frame.

Another problem arises when integrating the model checking results into the current model. As the model checking process runs in parallel in the background, directly modifying the model from within this process or a related Java thread could result in inconsistent model updates due to concurrent access to the data. Thus, the consistency management offers also help to schedule the required model updates in such a manner that problems due to the concurrent access can be excluded.

5. Conclusion and Future Work

In this paper we described a flexible and incremental consistency management, which defines a three-phase consistency lifecycle. The phases are decoupled to allow a detection and resolution of inconsistencies at different points of time. Concerning these points of time, each rule can be configured, depending on its complexity and the needs of the developer. Such rules can be combined to rule catalogues to support the software for different application domains. Moreover, we described a graphical and operational specification language, based on graph grammars, which enables us to define consistency rules and repair actions.

Our future work focuses on a more convenient way to monitor the inconsistencies for the user. A major task is to annotate the inconsistencies directly in the diagrams to provide a visual feedback. Further on, our approach allows us to incorporate other tools to

perform more complex consistency checks. We hope to gather more experience concerning the usage of external tools within our system, which also affects a more intelligent monitoring of checking results.

6. References

- [1] R. Balzer. Tolerating inconsistency. In *Proc. of the 13th International Conference on Software Engineering, Austin, Texas, USA*, pages 158–165. IEEE Computer Society Press, 1991.
- [2] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taenzer. Consistency checking and visualization of OCL constraints. In *UML 2000*, LNCS 1936. Springer Verlag, 2000.
- [3] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool integration at the meta-model level within the fujaba tool suite. In *Proc. of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3)*, September 2003.
- [4] B. Demuth, H. Hussmann, and S. Loecher. OCL as a specification language for business rules in data base applications. In M. Gogolla and C. Kobryn, editors, *6th International Conference on the Unified Modeling Language, Toronto, Canada*, LNCS 2185. Springer Verlag, October 2001.
- [5] A. Egyed. Automatically validating model consistency during refinement. In *Technical Report, Center for Software Engineering, University of Southern California, Los Angeles*, October 2000.
- [6] H. Ehrig and A. Tsiolakis. Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In H. Ehrig and G. Taentzer, editors, *ETAPS 2000 Workshop on Graph Transformation Systems, Berlin, Germany*, 2000.
- [7] G. Engels, J. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. Gruhn, editor, *Proc. of the 8th European Software Engineering Conference (ESEC)*, pages 186–195. ACM Press, September 2001.
- [8] A. Finkelstein. A foolish consistency: Technical challenges in consistency management. In M. T. Ibrahim, J. Küng, and N. Revell, editors, *Proc. of the 11th International Conference on Database and Expert Systems Applications (DEXA'00), London, UK*, LNCS 1873, pages 1–5. Springer Verlag, September 2000.
- [9] C. Ghezzi and B. Nuseibeh. Special issue on managing inconsistency in software development (1). *IEEE Transactions on Software Engineering*, 24(11):906–1001, November 1998.
- [10] C. Ghezzi and B. Nuseibeh. Special issue on managing inconsistency in software development (2). *IEEE Transactions on Software Engineering*, 25(11):782–869, November 1999.
- [11] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, September 2003.
- [12] A. Habel, R. Heckel, and G. Taenzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287–313, 1996.
- [13] International Telecommunication Union (ITU), Geneva. *ITU-T Recommendation Z.109: SDL Combined with UML*, November 1999.
- [14] International Telecommunication Union (ITU), Geneva. *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*, 2000.
- [15] C. Nentwich, W. Emmerich, and A. Finkelstein. Flexible consistency checking. In *Research note, University College London, Dept. of Computer Science*, 2001.
- [16] U. Nickel, W. Schäfer, and A. Zündorf. Integrative specification of distributed production control systems for flexible automated manufacturing. In M. Nagl and B. Westfechtel, editors, *DFG Workshop: Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen*, pages 179–195. Wiley-VCH Verlag GmbH and Co. KGaA, 2003.
- [17] B. Nuseibeh, S. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33(4):24–29, April 2000.
- [18] Object International. *Together Control Center, the Together case tool*. Online at <http://www.together-soft.com>.
- [19] OMG. *Unified Modeling Language Specification Version 1.5*. Object Management Group, 250 First Avenue, Needham, MA 02494, USA, September 2002.
- [20] Rational. *Rose, the Rational Rose case tool*. Online at <http://www.rational.com>.
- [21] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, Singapore, 1999.
- [22] SIEMENS AG, Nürnberg. *SIMATIC S7-300 - The universal PLC (Product Brief)*, September 2001.
- [23] SIEMENS AG, Nürnberg. *S7-300 Automation System, Hardware and Installation: CPU 31xC and CPU 31x*, June 2003.
- [24] R. Wagner. Realisierung eines diagrammübergreifenden Konsistenzmanagement-Systems für UML-Spezifikationen. Master's thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, November 2001.



Workshop on

"Consistency Problems in UML-based Software Development II"

Workshop Materials



ISSN 1103-1581
ISRN BTH-RES—03/06--SE