T U M

I N S T I T U T F Ü R I N F O R M A T I K

3rd International Workshop on Critical Systems Development with UML

Jan Jürjens, Eduardo B. Fernandez,
Robert France, Bernhard Rumpe

TUM-I0415
September 04

T E C H N I S C H E U N I V E R S I T Ä T M Ü N C H E N

[JFF+04] J. Jürjens, E. Fernandez, R. France, B. Rumpe (ed.).
3rd International Workshop on Critical Systems Development with UML.,
Informatik TUM-I0415
Technische Universität München. 2004
www.se-rwth.de/publications

TUM-INFO-09-I0415-100/1.-FI

Druck: Institut für Informatik der
Technischen Universität München

3rd International Workshop on Critical Systems Development with UML

Jan Jürjens, Eduardo B. Fernandez,
Robert France, Bernhard Rumpe

Preface

The high quality development of critical systems (be it real-time, security-critical, dependable/safety-critical, performance-critical, or hybrid systems) is difficult. Many critical systems are developed, deployed, and used that do not satisfy their criticality requirements, sometimes with spectacular failures.

Part of the difficulty of critical systems development is that correctness is often in conflict with cost. Where thorough methods of system design pose high cost through personnel training and use, they are all too often avoided. UML offers an unprecedented opportunity for high-quality critical systems development that is feasible in an industrial context.

The workshop series on "Critical Systems Development with UML (CSDUML)" aimes to gather practitioners and researchers to contribute to overcoming the challenges one faces when trying to exploit this opportunity.

The previous, editions of the series were the CSDUML'02 satellite workshop of the UML'02 conference in Dresden (Germany) and the CSDUML'03 satellite workshop of the UML'03 conference in San Francisco. Both had been very successful, and were among the largest satellite workshops of the UML conferences.

The proceedings at hand now present the accepted contributions for the CSDUML'04 workshop, which takes place on October 12, 2004, as part of the UML'04 conference (October 10 - 15, 2004, in Lisbon, Portugal). It is again organized in cooperation with the pUML (precise UML) group and the working group on Formal Methods and Software Engineering for Safety and Security (FoMSESS) of the German Computer Society (GI).

Out of a number of high quality papers submitted to the workshop, seven were selected to be presented in talks at the workshop and included as full papers in the proceedings. Three were selected to be presented as short talks and included as short papers. Furthermore, there are six posters presented at the workshop, which are included in the proceedings as abstracts. The highly selective acceptance rate again keeps the workshop focussed and on a high level of quality, and provides sufficient time for discussion.

In addition, the workshop features an invited talk by Connie Heitmeyer (Naval Research Laboratory) with title "On the Role of Tools in Specifying the Requirements of Critical Systems". Also, there will be a panel with the title "Providing tool-support for critical systems development with UML:

Problems and Challenges" including distinguished experts, which is hoped to create some lively discussions on the subject.

As with the CSDUML'02 and CSDUML'03 workshops, it is planned to edit a special section of the Journal of Software and Systems Modeling (Springer-Verlag) with selected contributions of the workshop. Up-to-date information on this and the workshop can be found at the workshop homepage.[1]

We would like to express our deepest appreciation to the authors of submitted papers, and to the program committee members and the additional referees. We would also like to thank the UML'04 conference chair Ana Moreira (New University of Lisbon), the workshop chair Ambrosio Toval (University of Murcia), the local arrangements chair Isabel Sofia Brito (Politécnico de Beja), and the students involved in the organization at TU Munich for their help. In addition, some of the organizers thank their various projects (including Verisoft) for their personal funding.

Jan Jürjens
Eduardo B. Fernandez
Robert France
Bernhard Rumpe
(Organization team for CSDUML'04)

September 2004

[1] http://www4.in.tum.de/~csduml04

Program committee

Doo-Hwan Bae, Korea Advanced Institute of Science and Technology (KAIST), South Korea
Marko Boger, Gentleware
Eerke Boiten, Univ of Kent
Ruth Breu, University of Innsbruck
Manfred Broy, TU München
Alessandra Cavarra, University of Oxford
Betty H. C. Cheng, Michigan State University
Gregor Engels, University of Paderborn
Martin Gogolla, University of Bremen
Radu Grosu, State University of New York at Stony Brook
Holger Hermanns, University of Saarbrücken
Heinrich Humann, LMU München
Johan Lilius, Åbo Akademi University
Ileana Ober, VERIMAG
Francesco Parisi-Presicce, Universita di Roma and George Mason University
András Pataricza, Budapest University of Technology and Economics
Gianna Reggio, University of Genova
Peter Schmitt, University of Karlsruhe
Bran Selic, IBM Rational Software
RK Shyamasundar, Tata Institute of Fundamental Research, Bombay
Ketil Stølen, SINTEF ICT, Oslo
Jon Whittle, NASA Ames Research Center
. . . and the organizers.

Additional referees

Folker den Braber
Zaid Dwaikat
Jan Hendrik Hausmann
Frank Innerhofer-Oberperfler
David N. Jansen
Mass Soldal Lund
Stefan Sauer
Fredrik Seehusen
Fredrik Vraalsen
Barbara Weber

Technical Organization

Michele Gatabazi, TU München
Dimitri Kopjev, TU München
Britta Liebscher, TU München
Mathias Riedl, TU München
Ines Rieger, TU München
Lucas Schib, TU München
Darina Velcheva, TU München

Contents

On the Role of Tools in Specifying the Requirements of Critical Systems

Constance Heitmeyer
Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, DC

ABSTRACT

In 1978, a group of researchers led by Dave Parnas developed a tabular notation for specifying software requirements called SCR (Software Cost Reduction) and used the notation to specify the requirements of a mission-critical program, the Operational Flight Program for the A-7 aircraft. Since then, the requirements of many critical programs, including control software for nuclear power plants and other flight programs, have been specified in SCR. To support formal representation and analysis of software requirements, NRL has developed a state machine model to define the SCR semantics and built a suite of tools based on this semantics for checking requirements specifications for properties of interest. Such tools are especially valuable for specifying and analyzing the requirements of software systems where compelling evidence is required that the system satisfies critical properties, such as safety and security properties. This talk describes the many different roles that formally based software tools can play in debugging, verifying, and validating the requirements of critical software systems. The author's recent experience and lessons learned in specifying the requirements of a security-critical cryptographic system and two software components of NASA's International Space System are also described.

Connie Heitmeyer, the chief designer of the SCR toolset, heads the Software Engineering Section of the Naval Research Laboratory's Center for High Assurance Computer Systems. Recently, she served as co-program chair for MEMOCODE 2004, the 2nd International Conference on Formal Methods in Hardware/Software Co-Design. She is currently serving as co-chair of the Experience Reports Track at ICSE 2005. She is a member of the editorial boards of the ACM Transactions on Software Engineering and Methodology, the Requirements Engineering Journal, and the Journal on Software and System Modeling. Her research interests are in formal specification and formal analysis of software and system requirements and of high assurance software systems. She is also very interested in transferring formal methods technology and tools to software practitioners.

Using UML, OCL and MDA to support development of Modular Avionics Systems

Philippa Conmy and Richard Paige

Department of Computer Science, University of York, Heslington,
YO10 5DD, U.K.
{philippa.conmy, richard.paige}@cs.york.ac.uk

Abstract. This paper presents a Model Driven Architecture approach to development of Integrated Modular Avionics Systems. Required system behaviour has been captured in a Platform Independent Model using UML annotated with the Object Constraint Language. The model and its constraints have then been transformed to form a Platform Specific Model. The paper describes the benefits this approach could have to the development, certification and incremental change of IMA systems. The paper also describes some of the practical difficulties with the model transformation process.

1 Introduction

The aviation industry (both civil and military) are moving towards a distributed computer system concept known as Integrated Modular Avionics (IMA) [1]. Core to this concept is the separation between dedicated software applications (such as flight control, fuel management) and computer resource management (e.g. operation systems). In traditional avionics systems software was monolithic and tightly coupled to the hardware in order to get the required real-time performance. Now that processor speeds have increased a modular approach can be used whilst still meeting performance requirements. This should allow more agile maintenance of systems for the upgrade of both applications and hardware. This separation of concerns matches the approach taken with Model Driven Architecture (MDA) [4, 9] where a Platform Independent Model (PIM) (which contains details of required functionality) is mapped to a Platform Specific Model (PSM) (which contains the functionality mapped to a specific execution environment). Therefore MDA is being explored as a way to bring the benefits of modelling to IMA development and also to support incremental change.

A substantial barrier preventing the full benefits of IMA being realised is the certification process. Avionics systems must go through a rigorous (and expensive) certification procedure which includes demonstrating that safety requirements have been met. Current practice is based on the traditional design approach and tends to examine software as a whole, making alteration of the evidence extremely difficult and a serious impediment to maintainability. Clearly to achieve the benefits of IMA a more modular approach is required. Most safety

requirements are placed on the system by safety analysis of the core functionality and design (i.e. of the PIM). This paper uses the Object Constraint Language (OCL) to capture these safety requirements as part of a PIM. These OCL constraints are then translated during a transformation process onto the PSM. Using a certification process which separates the verification that the safety requirements are correct from the verification that they are realised in the proposed platform will better help support change. This approach is complementary to the WG-60/SC200 initiative [5] which is producing certification guidance for IMA.

The contributions of this paper are to present an approach to using OCL, UML and MDA for IMA development. We discuss the complexity and required semantics for transforming OCL constraints when using this approach. The paper is laid out as follows; first we discuss in more detail the IMA concept and the requirements for safety and certification. We then present a brief overview to our MDA approach using UML and OCL. A small case study is then given demonstrating how these constraints may be derived and inserted into a PIM. We show how these constraints are converted during the transformation process. Finally, we discuss the potential benefits and limitations of our approach.

The paper authors are part of the High Integrity Real Time Systems (HIRTS) Defence and Aerospace Research Partnership (DARP) program in the U.K. and are looking at using different software modelling techniques to support HIRTS system design and certification. This research is jointly funded by the EPSRC, UK DTI, BAE SYSTEMS, QinetiQ and Rolls Royce.

2 Safety and Integrated Modular Avionics

2.1 Safety Critical Systems and Incremental Change

Safety is an emergent system property [10]. This means the combined behaviour of individual components contribute to the safety of the system they are in. There are numerous different techniques for system safety analysis however, for the purposes of this paper we have summarised these and produced a set of key definitions:

- **Hazard Analysis** - in general, some form of system hazard analysis is undertaken. This identifies system states which could lead to an accident, e.g. brakes are non-operational in a car.
- **Failure Analysis** - failure analyses look for potential failures or behaviours in system components which could lead to one of the hazardous states, e.g. brake cable breaks. They also look for combinations of failures among components.
- **Derived Safety Requirements (DSRs)** - one of the results of the analyses are a set of derived requirements on the components which mitigate against the relevant failures, e.g. thicker brake cable.
- **Safety Case** - a safety case presents a systematic argument that the hazards have been sufficiently addressed (depending on the required integrity) backed up with evidence from analysis and testing [7].

2.2 Integrated Modular Avionics (IMA)

IMA is the term used to describe a distributed computer platform on an aircraft. This paper concentrates on the civil IMA standard ARINC 653 [1]. The IMA concept has been introduced to help improve maintainability of systems, and combat long term problems such as hardware obsolescence.

An IMA network consists of a number of computer processor modules connected by a common communications network which links to a number of dedicated sensor and actuator devices. Each computer module has a three layer architecture of applications, Operating System (OS) and Hardware Interface Layer (HIL). Software in the application layer is divided into partitions. Each partition is separated from other partitions logically (they do not have any shared data areas) and temporally (each is executed sequentially in a round-robin manner). Access to the communications network is also strictly partitioned by the OS. The applications can only request access to hardware devices and other applications via an Application Programming Interface (API). By using an API the application code is not tied to any particular hardware platform. The OS in turn uses the HIL via an API. Again this is to support portability, this time for the OS.

Using this architecture has a number of potential benefits including the ability to reconfigure following a failure, to restrict the impact of a change and to upgrade the hardware with minimal impact on the applications. However, current certification practice, which examines software as a monolith, is not flexible enough to easily support these benefits. A number of initiatives are in place looking at producing new guidance to better support incremental change, such as the WG-60 working group [5]. This working group is producing a draft which recommends using a staged process for integrating applications with the IMA platform. The authors believe an MDA approach will complement this staged integration and that the use of modelling can provide significant benefits to help the change process.

3 Model Driven Architecture

The Model Driven Architecture (MDA) [4, 9] approach put forward by the Object Management Group (OMG) proposes the use of two system models (usually captured with UML), one Platform Independent Model (PIM) and one Platform Specific Model (PSM). The PIM contains details of the required core functionality of the system. This is then mapped or merged with a model of the platform to produce the PSM. The term platform refers to the supporting environment e.g. CORBA or OS and hardware. This separation matches that used in IMA. In IMA terms the PIM will contain models of dedicated avionics applications and their related hardware devices such as sensors and actuators and the PSM will model how these interact with the IMA bare platform, whilst still retaining the functional properties of the PIM. A model of the IMA platform will be required as part of the merging process which contains details such as performance and available resources.

The safety case for an IMA system developed with MDA would be constructed as follows. First, by performing hazard analysis and failure analysis on the PIM, derived requirements can be generated. These derived requirements are captured within the PIM. This helps demonstrate that the identified hazards have been addressed. Then the PIM is merged with the PSM. A verification process can be used to demonstrate that the safety requirements are still met once the applications are embedded onto the IMA platform. In addition to this, evidence would be required that the final implementation fulfils the requirements laid out in the model.

If a change is made to one of the dedicated applications then the PIM will be altered to reflect that change. Additional safety analysis may be required and new safety requirements captured in the model. The PIM will then be merged to form a PSM and re-verification undertaken. This represents a significant saving in time and effort to current practice as only the PIM needs to be re-analysed.

If the IMA platform is altered (e.g. to upgrade the processor) then only its model needs to be updated and the PIM remains the same. Then, re-merging and re-verification will take place to demonstrate that the required safety properties are still met. Again, this would mean a significant improvement on current practice.

To support this process the authors propose using the UML to represent the various system components and their interactions. The UML models are extended with safety requirements captured using the Object Constraint Language (OCL) [14]. In our experience safety is a inherent part of a safety critical system and cannot be separated from the core functionality, hence we capture the safety requirements within the main system model.

3.1 Using UML and OCL

This section presents an MDA approach for the development of a safety critical application which is to be embedded on an IMA network. The approach uses both platform specific and platform independent models of the system. The PIMs are independent of the processing platform and OS, but may be tied to a specific airframe. Each system model will consist of a number of different UML views of the system which show different system characteristics. Three levels of modelling are proposed, one level of PIM, and two PSM, to support staged development and integration of the IMA system.

The first model level has a PIM (or conceptual level model) for the safety critical system and will contain components and their interactions, demonstrating the core functional properties of the system. This model will include the specialised components which control and manipulate system data (such as application software) and the specialised components which they control (such as sensors and actuators). The PIM can be used to analyse the core safety properties of the system, and validate that the proposed design will meet the required behaviour and functionality. To capture this behaviour (and other non-functional properties such as interoperability) the model will use well-defined constraints captured in OCL. The constraints are generated by performing traditional safety

analyses of the proposed design and converting DSR's into OCL. Each component is represented as a single class within a UML model. The components each have a stereotype of either $\ll HWDevice \gg$ (for sensors and actuators) or $\ll ApplicationSW \gg$ for software. These stereotypes are used during the PIM to PSM translation process.

The PSM will include the components from the PIM integrated with generalised computer support components such as operating systems, processors, networking and so on. The PSM will demonstrate how these components will interact with one another. This document proposes the use of two levels of PSM; a division based on the types of properties which can be analysed at each level. The first level (behavioural) examines how the PIM interacts with a generic model of the supporting environment, whereas the second level (deployment) examines a specific instantiation of that environment. The reasons for this separation are threefold. Firstly, during the development process the OS and supporting environment may be simulated whilst the actual purchase of hardware is delayed to avoid obsolescence as far as possible. This means that actual deployment information may not be available for modelling. Secondly, using two levels of PSM allows a separation of concerns for constraint modelling. At the first level behavioural properties can be examined, whereas at the deployment level constraints for non-functional properties such as processor layout and execution times can be verified. This separation will assist incremental change as, for example, if a processor is upgraded then the deployment model should only need to be re-examined. Finally, the deployment level may include multiple applications, whereas the behavioural level should only include one application at a time.

Two aspects of transformation from the PIM onto a behavioural PSM have been examined so far, these are static structure diagrams and sequence diagrams. For this a Platform Description Model (PDM) has been produced. In this PDM, the IMA platform is modelled by two connected components, $Network$ and $OSAndHW$ (see figure 1), the latter representing the entire supporting elements in a single processing module. UML diagrams with template placeholders are used to represent the components in the PSM. This simple representation is used as this is what is visible to an Application. This is an abstract representation only, and has two functions only at this time (clearly a more complete model would have additional functions). The $Network$ component represents the communications bus and switches. Again, this is an abstract representation. These two components are inserted in between the PIM components, with the $Network$ component being attached to all items marked with the stereotype $\ll HWDevice \gg$ and the $OSAndHW$ being attached to all items marked with the stereotype $\ll ApplicationSW \gg$.

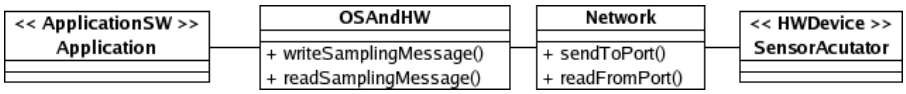


Fig. 1. Platform Description Model (PDM)

The second aspect examined is sequence diagrams. Connections between items in the PIM are similarly altered to include calls to the Operating System which are then relayed to the network and on to dedicated devices. At this level of PSM only generic classes for the $Network$ and $OSAndHW$ are used. At the final deployment level of PSM the $OSAndHW$ is converted into several separate instances representing each processor within the network. The mapping of individual applications onto processors is determined using the non-functional constraints set at the PIM. These translation processes are demonstrated in section 4.

3.2 OCL Constraints for Safety Critical Systems

As discussed previously, a safety critical system must be built so that potential hazards are avoided as far as possible. In order to do this the system must fulfil a number of safety requirements derived by analysis. By using OCL annotations on the UML model many of these requirements can be captured as part of the PIM, allowing automated checking and also separating the safety concerns from the platform specific concerns as far as possible. OCL allows a UML model to be annotated with constraints which describe pre and post conditions on class methods and also class invariants which apply to all methods in the class. It allows constraints to be specified for variables (such as valid input ranges) and also for actions (to show what must occur during function execution) [8, 14]. Timing constraints can also be specified using OCL extensions such as [2] and these are vital for any safety critical system with real-time requirements. These constraints apply when functions directly interact with one another and are well supported. These have been used to develop an OCL and UML based approach for modelling safety critical systems [6].

This paper extends these constraints to include a number of other domain specific concerns which should also be captured in order to more fully describe required behaviour and better support certification. These apply to "indirect" interactions. By this we mean one of two things:

- components acting independently but meeting a common goal, for example, the engine of a car provides power, but the steering provides control of that power
- assumptions about the operation of another component, for example, the frequency with which data is acquired by a measurement sensor, or the units it is measured in

These dependencies often appear in parts of the system safety case but are not explicit in system models. If they are lost then it is more difficult to accurately assess the impact of a system change. In addition, if a modular development process is used then guarantees are required that components are interoperable. This paper proposes using OCL in a way which helps to capture these. The following is a list of properties which the constraints will apply to:

- **Location** - sometimes two or more systems must be placed in different physical locations in order to prevent a common failure causing them all to fail.
- **Conceptual dependencies** - sometimes a common failure is avoided by using conceptually different methods to derive data
- **Interoperability Properties** - these are used to ensure that components have compatible data formats such as measurement units, refresh rates etc.
- **Ordering** - sometimes there is a dependency on the order of execution of otherwise independent applications, e.g. when one application must monitor the output of another.

The list has been derived as part of a wider study into modelling safety critical systems [12]. The first three sets on this list can be simply described by comparing static properties of the components and their methods and we have captured these in the following way. Firstly, a global method is added to the OclAny class which returns data which allows checking. For example, the method $units(a : OclAny) : String$ can be added which takes an argument and returns a string encoding the argument's units, e.g., kilos, miles, m/sec. This method is global in the sense that it needs to be available in every class, thus adding it to a class like OclAny is appropriate. Secondly, each relevant class has an invariant added which uses this method to set the value of an attribute. Then, finally, an integrity constraint is added which links two classes. E.g.

context ClassA inv:
units(self.temp) = "centigrade"

context ClassB inv:
units(self.temp) = "centigrade"
units(self.ClassA.temp) = units(self.temp)

There are other potential methods for solving this problem. For example, one could define a global singleton class that contains all static properties such as units; classes that need these properties use this class as desired. This is similar to the solution with OclAny but is not as flexible, as all static properties must be determined a priori, when the singleton is instantiated. Another solution is to provide interfaces that declare static properties, and have different classes implement the desired interfaces when the properties are needed. This solution works but does not appear to be as general as extending OclAny, the latter of which allows properties to be added to legacy models.

The more complex ordering constraints have not yet been fully tackled, as whilst it is possible to implicitly capture required behaviour by using separate timing constraints for each class. It is more difficult to explicitly describe that there is a relationship between these timings. The latter is required to assess the effect of an incremental change on the safe behaviour of a system.

4 Example

This section uses a simple example of a fictitious engine cooling system on an aircraft to demonstrate how the MDA approach for IMA works. In this system, application software takes a heat measurement from a sensor and calculates whether a cooling element actuator requires adjustment. The system has an independent health monitor to watch the input and output of the main application. This activates a warning light in the presence of an error. The pilot can take action if the warning light is activated.

4.1 System PIM

A static structure diagram for the system is shown in Figure 2. This diagram shows the 5 core components in the cooling control system. Firstly, a heat sensor measures the other systems temperature. The sensors value is used both by the cooling control system and also by the health monitor. The cooling control software sends the required command to the cooling system itself, and its output value is also input to the health monitor. The health monitor is attached to a warning light.

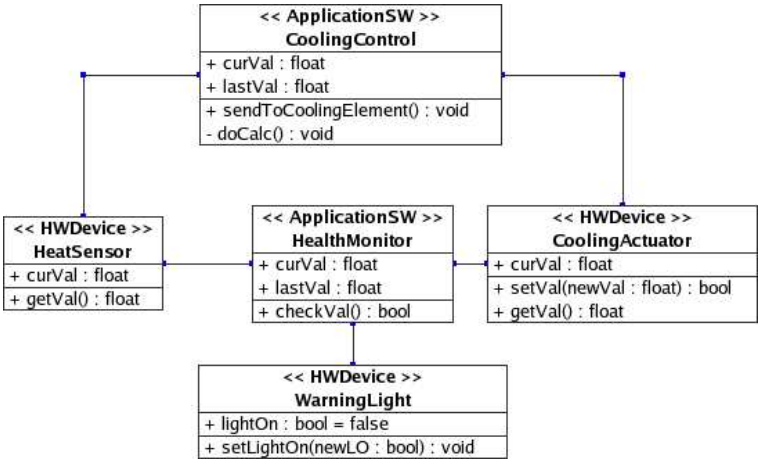


Fig. 2. Example cooling system PIM

Basic safety analysis of the system has been performed using a Failure Modes and Effects Analysis technique [11]. In this technique guidewords are used to suggest failures during component interactions. Extensive analysis is required in order to demonstrate that as many failures as possible have been identified, however for reasons of space only a small extract is shown in table 1.

There are a couple of comments on the analysis results. Firstly, the behaviour of the monitor in detecting if the system is overheating is crucial to preventing

Table 1. Extract from System FMEA

Failure Mode	Cause	Effect	Derived Requirement
Omission - Data not sent from Application to Cooling Actuator	Application Processor stops	System may overheat	Monitor detects and activates warning light
Omission - Monitor does not activiate warning light	Monitor is on same processor as stopped main application	Warning light not shown to pilot and system may overheat	Monitor is independently located from main application
Late - Data is late arriving at Cooling Actuator	Data is held up on network	System may overheat	Data being sent from main application to cooling actuator must arrive within 10 milliseconds of sending, and Monitor detects and activates warning light

a hazard occurring. This requirement has not been developed further in the paper as it converts to a larger set of constraints than can be demonstrated. Secondly, whilst the monitor will detect overheating if the data sent to the *CoolingActuator* is delayed, it is important to prevent this as far as possible in order for the system to be reliable. Therefore the timing requirement must be upheld.

The two requirements on location and timing have been converted directly into OCL constraints. Using the method described in section 3.2 the following OCL statements have been added to describe the location requirements:

```
context CoolingControl inv:
locate(self.location) = "undefined"

context HealthMonitor inv:
locate(self.CoolingControl.location) <> locate(self.location)
```

These constraints are used to state the location of each component and then state that the locations should not be the same. As the final location is at present undefined, this constraint cannot be verified to be true or false and to avoid false negatives the locate has only been called for the *CoolingControl*. However, it is an important requirement as it ensures the pilot will be alerted to the system overheating. In addition, this constraint will actually help guide the translation process for both levels of PSM.

The second timing requirement has been expressed using the methods described in [2, 6]:

context CoolingControl::SendToCoolingElement():
post: Time.now <= Time.now@pre + 10
and CoolingActuator ∧ setVal(curVal)

This constraint basically states that at the end of the operation to send data to the cooling element, no more that 10 milliseconds should have passed and the the $CoolingActuator :: setVal()$ method should have been called.

4.2 Behavioural PSM

This section describes the first level of PSM as described in section 3.1. In order to build a PSM the IMA PDM shown in figure 1 is merged with the PIM. The resulting static view is shown figure 3.

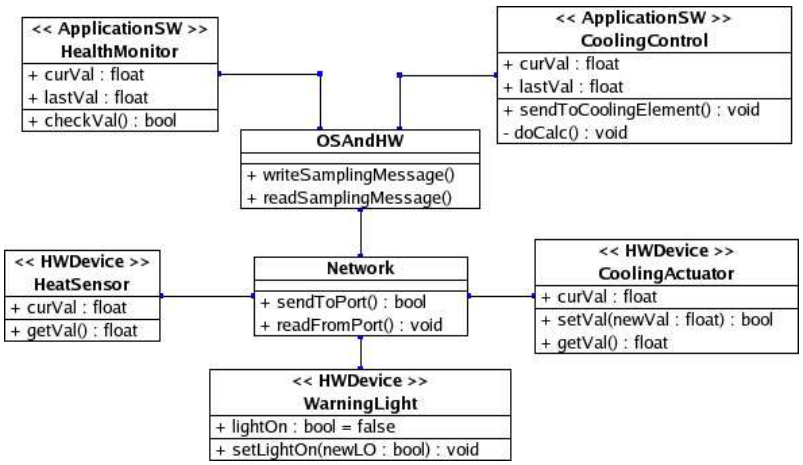


Fig. 3. Behavioural PSM

The next part of the transformation to be considered is the constraints. To assist this transformation a set of guidelines are being created, based on the type of constraint being transformed and its dependencies on other constraints. By formalising these it is hoped the transformation proceedure can be automated (see section 5.2). The thought processes behind some of these guidelines are now described.

Firstly, the independence constraint will need to be examined. The constraints which apply to the internal component property of "location" will remain unchanged and remain as "undefined" as no information on deployment is yet known. The constraint which prescribes that the locations should differ must be re-examined as the two components are no longer directly linked. This constraint needs to be updated to include the new connections.

Two possible connections are possible. Either the connection is via *OSAndHW* only (as all applications must communicate via the API in the IMA concept) or it is via *OSAndHW* and *Network*. From the perspective of the *HealthMonitor* and the *CoolingControl* the only component they see is the *OSAndHW*, however given that the final locations are to be different it seems logical to include the network in the path. This means that *OSAndHW* will appear twice as any sequence of execution will involve data being passed to *OSAndHW*, out to *Network*, and back in to *OSAndHW*. In this way the constraint itself has guided the transformation process.

context CoolingControl inv:
locate(self.location) = "undefined"

context HealthMonitor inv:
locate(self.OSAndHW.Network.OSAndHW.CoolingControl.location)
<> locate(self.location)

The effect on the timing constraint is more complex. *CoolingControl* will no longer be able to directly access the *CoolingActuator*, but instead will use the *OSAndHW* API call *writeSamplingMessage*() to request data to be sent to a defined communications port. The *OSAndHW* will send the data to the network using the *sendToPort*() function. It is then the job of the network component to send data to the correct location based on the portNo (in this case sending to the *CoolingActuator*). A *portNo* attribute has been added to *CoolingActuator* for the *Network* element to check. This sequence of events is shown below:

context CoolingControl::SendToCoolingElement():
post: Time.now <= Time.now@pre + 10
and OSAndHW ∧ writeSamplingMessage(portNo,curVal)

OSAndHW::writeSamplingMessage(portNo,curVal):
post: Network ∧ sendToPort(portNo, curVal)

Network::sendToPort(portNo, curVal):
post: if portNo = self.CoolingActuator.portNo then
CoolingActuator ∧ setVal(curVal)

It is important that the timing constraint for the *CoolingControl* is still preserved as this is a safety requirement. It now applies to the whole sequence of events though, and in order to determine if it can be met the execution times of all the methods in the sequence will need to be measured.

4.3 Deployment PSM

The final level of modelling includes all the hardware specific details such as performance times, reliability and architectural layouts. Generating this model will involve converting abstract components into specific instances. In addition

the constraints contained in the other models which relate to architectural layouts will be used to guide the conversion process. Figure 4 shows a UML deployment diagram which has been derived from the architectural model. The *CoolingControl* and *HealthMonitor* have been located on differing processors, as defined in the location constraint set. In fact the translation process has again been driven by this constraint. Two specific instances of *OSAndHW* have been created to support the functional components. The network and the associated hardware devices are now represented as physical devices. The physical devices are located at the end of the network.

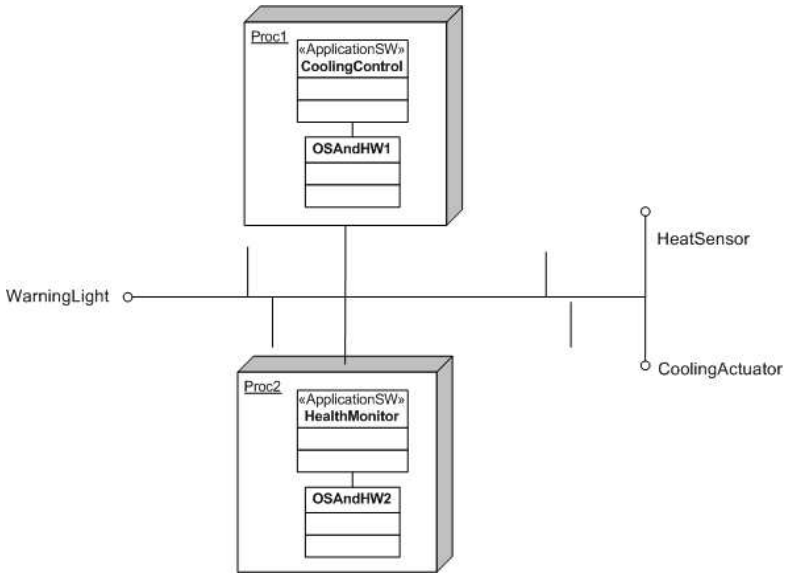


Fig. 4. Deployment PSM

The individual location constraints can now be updated with the actual value for their location, meaning they can now be checked. The comparison constraint also needs to be updated, replacing the abstract representation of *OSAndHW* with the specific instantiations. This resulting constraint set is:

context CoolingControl inv:
locate(self.location) = "proc1"

context HealthMonitor inv:
locate(self.location) = "proc2"
locate(self.OSAndHW2.Network.OSAndHW1.CoolingControl.location)
<> locate(self.location)

The timing constraint is simply updated to reflect the specific instance of *OSAndHW* which the *CoolingControl* will call. It is possible that this timing

constraint could also affect the transformation if the required total execution time could not be met for some configurations (e.g. if the *CoolingControl* was located at the far end of a network or if the schedule on some processors meant that the deadline could not be met).

context CoolingControl::SendToCoolingElement():
post: Time.now <= Time.now@pre + 10
and OSAndHW1 ∧ writeSamplingMessage(portNo,curVal)

OSAndHW1::writeSamplingMessage(portNo,curVal):
post: Network ∧ sendToPort(portNo, curVal)

Network::sendToPort(portNo, curVal):
post: if portNo = CoolingActuator.portNo then
CoolingActuator ∧ setVal(curVal)

5 Discussion

5.1 Using and mapping OCL

Capturing safety requirements within the model is essential if UML is to have full value for developing safety critical systems. Without this information the model cannot be used to fully assess the impact of a change, meaning the code will have to be altered instead which is more expensive and time consuming. Also, analysis tools can be used to assess whether the safety requirements will be met by the proposed solution. Again, this has many potential cost and time savings.

The OCL was used to capture safety requirements as it is the de facto standard for UML. There were a number of difficulties we encountered when using OCL though. Firstly, the types of dependencies discussed in 3.2 are not always easy to describe using OCL, for example OCL requires that there is an interface connection between components in order for a constraint to be set on their properties - in the case of emergent dependencies between independent systems there are no such connections. In this case we were able to use the common IMA platform as a basis to compare properties. Secondly, OCL is based on the use of traversals of object structures. Dependencies between components, such as location, do not have this relationship therefore an arbitrary decision needed to be made as to which component had the constraint attached. Thirdly, the relatively simple OCL constraints at the PIM level became rapidly complex at the PSM level when new components were introduced. In addition, it was not always obvious as to how to map the OCL from one layer to the next.

5.2 Automation

The transformations for the simple example given in this paper were undertaken by hand, but, even with this simple system the OCL annotations rapidly increased in number when translating from the PIM to the PSM. If the approach

is to scale to real systems then the transformation must be automated, perhaps using a technique such as Genetic Algorithms to help generate a solution. However, constraint transformation is difficult to automate. Firstly, with a hand written approach potential conflicts between constraints were quickly recognised and dealt with. Secondly, when converting constraints on a line by line basis, a set of complex constraints is built, but a much simpler generic solution may exist which would work for each of the lines. For example, the *Network* constraint looking at the *CoolingActuator.portNo* would become a large set of if/then/else statements if it was converted for each individual call made to a portNo. A better solution would be to use a generic portNo collection on the *Network* component, fill the collection based on the specific IMA configuration during transformation, and then select the appropriate post condition action for *sendToPort*() based on the input parameter. However, this approach was not used as it involves adding constraints to the PDM and merging these with the PIM constraints, which would be a more complex process to undertake automatically.

Another issue with automation is that of trust. The guidance documents for writing safety critical software (such as DO-178B [13]) stipulate that any tools used to automate development be qualified. This means that it must undergo some of the same type of testing and analysis as the software which is used on the system itself. The more complex the transformation process, the more difficult it is to certify. Certification can be an extremely expensive process, but certifying the tool is potentially a one off cost.

5.3 Implementation

One a complete model has been created for an IMA system it must still be implemented, and the certification process must ensure that the implementation meets the requirements in the model. The PDM is based on ARINC 653, and there are a number of ARINC 653 compliant OS's already available or being developed. This means that part of the certification process can be performed by ensuring that the new requirements placed on an existing product are met, probably by human review. However, the applications will still need to be developed and reviewd. The most acceptable solution is to hand code them but this has the obvious disadvantage that both the code and the model must be kept synchronised. An alternative approach would be to auto-generate the application code using the models, but, in the authors opinion, the acceptance of auto-generated code for highly critical systems is long way from reality.

6 Conclusions

This paper has presented an approach to using MDA for IMA systems. The models have been described using UML and OCL. Some extensions to the types of constraints which are included in the models were introduced. A transformation process was also introduced which converts both the UML and the OCL

annotations with a model. This has been applied successfully to a simple IMA system.

Whilst there are many potential benefits to using an MDA approach there are a number of issues which need to be resolved. The exact way MDA could be used given current certification practice needs to be examined further, in particular if tools are used for automation.

On a more technical level, whilst the OCL transformation from PIM to PSM must be automated for the approach to scale, the optimal approach has yet to be identified. Further to this, the suitability of OCL for capturing some of the required dependencies (such as independence properties) is being assessed.

Finally, this paper has not discussed the safety impact of the IMA platform. Since the IMA software can only cause hazardous behaviour once it is part of an avionics system, further analysis of the behavioural PSM may be required to ensure that no new safety concerns are introduced. Methods for examining OS usage and behaviour in safety critical systems have been produced [3] which could be tailored for use with the MDA approach.

References

1. ARINC. *Avionics Application Software Standard Interface ARINC 653.* Aeronautical Radio Inc., 1997.
2. M V Cengarle and A Knapp. Towards OCL/RT. *Lecture Notes in Computer Science*, 2391:390–409, 2002.
3. P. Conmy and S. Crook-Dawkins. A Systematic Framework for the Assessment of Operating Systems. In *Safety Critical Systems Symposium*, Warwick, UK, February 2004.
4. Object Management Group. MDA Guide Version 1.0. Technical report, May 2003.
5. WG-60/SC200 Working Group. Modular Avionics. http://www.rtca.org/comm/sc200.asp, 2004.
6. R Hawkins, I Toyn, and I Bate. An Approach to Designing Safety Critical Systems using the Unified Modelling Language. In *Critical Systems Development with UML*, 2003.
7. T.P. Kelly. Arguing Safety - A Systematic Approach to Managing Safety Cases Ph.D. thesis. Technical report, Department of Computer Science, University of York., 1999.
8. A Kleppe and J Warmer. Extending OCL to Include Actions. *Lecture Notes in Computer Science*, (1939):440–450, 2000.
9. A Kleppe, J Warmer, and W Bast. *MDA Explained, The Model Driven Architecture: Practice and Promise.* Addison-Wesley, first edition, 2003.
10. N G Leveson. *Safeware.* Addison-Wesley, 1995.
11. D.J. Pumfrey. *The Principled Design of Computer System Safety Analyses.* PhD thesis, Department of Computer Science, University of York., 2000.
12. A. Radjenovic, R.F. Paige, P. Conmy, M. Wallace, and J. McDermid. An Information Model for High-Integrity Real-Time Systems. In *Second RTAS Workshop on Model-Driven Embedded Systems*, 2004.
13. RTCA-EUROCAE. Software Considerations In Airborne Systems and Equipment Certification DO-178B/ED-12B. Technical report, RTCA and EUROCAE, 1992.
14. J Warmer and A Kleppe. *The Object Constraint Language, Precise Modeling with UML.* Addison-Wesley, first edition, 1999.

A Lightweight Approach to Critical Embedded Systems Design Using UML

Michaela Huhn, Martin Mutz*, and Bastian Florentz*

TU Braunschweig
Institute for Programming and Reactive Systems
{huhn,mutz,florentz}@ips.cs.tu-bs.de
http://www.cs.tu-bs.de/ips/

Abstract. We present a pragmatic approach to the UML-based design of critical systems that we are applying successfully in the automotive domain. We concentrate on so-called lightweight formal methods [14] like automated static analysis and validation of dynamic behaviour by simulation, but the approach can be strengthened to fully formalised analysis of models. Our method is supported by a tool that interoperates with a number of commercial CASE tools used in industry.

1 Introduction

Model-based CASE tools and in particular UML tools [15] are widely applied in the development of critical systems. However, in critical systems design, deeper analysis of the static and dynamic properties of the models is needed. To ensure the correct operation of a critical system, quality assurance has to provide adequate evidence that the system under design satisfies safety, timing, and other kinds of criticality requirements.

A lot of work has been done to improve the development of critical systems by the rigorous use of formal methods. This work has shown splendid results in pioneer projects. But already in 1996, J. Wing stated that the applicability of formal methods in industrial projects is strictly limited by the enormous costs [14]. Costs result from personnel training, the formal specification and verification process itself, and tools that have to be tuned for nearly each particular setting.

Some obstacles against formal methods could be overcome due to the fact that UML has become a well accepted de facto standard in industry:

- UML is familiar to a huge community of software developers.
- Compared to most other notations previously accepted in industry, UML has to be classified as a semi-formal notation. Moreover, the graphical models can be underpinned by textuell expressions (e.g. OCL, Object Constraint Language) to achieve a more formal description.
- UML-based tools for the specification, design, and analysis are available.

* The work of these authors was partially funded by the Volkswagen AG, Wolfsburg.

Consequently, many research groups work on formalising UML or parts of the notation (see [16, 24, 7]) to adopt it to critical system design. But as long as there is no agreement on a formal semantics of UML[1], industry hesitates to introduce formal methods for similar reasons as before (see e.g. [4]).

- Many approaches are based on semantics that are tuned for the underlying formal method. Even if the semantics conforms to the UML standard, only in rare cases the semantics is compatible to one of the leading UML tools preferred in industry.
- Formal methods capable of solving industrial sized problems are usually optimized for a specific analysis domain like timing behavior, safety, or security. Since large scale applications often incorporate different kinds of criticality requirements, the user has to work with several formal approaches (and relate the results appropriately) to achieve a sufficient coverage in validation.

Thus, the knowledge of the software experts has grown in the last years. But the efforts of specification and verification and the tool support, which is still poor from an industrial point of view, make formal methods still prohibitive in many industrial projects.

This review of the situation is disillusioning but coherent with the observations made by many others [8, 4, 20].

To improve product quality in practice and to increase the acceptance of formal methods, we took a *partial* approach as suggested in [14]:

- Many severe problems in model-based development can already be identified by partial analysis techniques like static analysis of the models or systematic simulation.
- To make analysis economically feasible, the specification and analysis process must be automated as much as possible.

In this paper we present a pragmatic approach for the analysis of UML state diagrams. State diagrams are heavily used for behavioural modelling in the design of embedded systems, and modelling errors in state diagrams cause a significant portion of faults. One reason is that the semantics of state diagrams is not really simple, in particular, if a set of objects is interacting in large systems. To make things worse, large scale embedded applications like in automotive or railway industries are developed with a number of CASE tools. Software engineers work simultaneously with models from different development environments. Even in case only UML is used, the tools show subtle differences in syntax and semantics. Thus, to guarantee understandability and consistency of models is a high priority requirement to improve model-based design. To address these problems companies have established style guidelines and catalogues of modelling rules and patterns [1] and also many UML tools offer some rudimentary consistency checks specified by the OMG [21]. However, so far these style guides are at most an advisory aid.

[1] or a set of formal semantics, each adopted for a particular application domain

We go one step further in our systematic, automated, and tool independent approach to build formalised rule catalogues and to check whether a state-based design adheres to a specified catalogue of rules. Moreover, to deal with the subtle behavioural differences of state diagrams in the various UML versions, we use a statechart simulator *SC-Simulator* that offers different semantics. Thus, the user can choose the handling of events, non-determinism, etc. Thereby, the same statechart can be simulated according to a standard UML semantics, the semantics of a particular CASE tool, or the user's preferred variant.

The rule-based static analysis of state diagrams is described in Section 2. Section 3 deals with the simulation of state diagrams with different semantics. Section 4 concludes.

2 Static Analysis of State Diagrams

The first part of our approach is concerned with the automated static analysis of UML state diagrams [21][2] with respect to user-defined modelling rules. The approach is supported by a tool called *Rule Checker* which does not depend on a specific development environment and can handle a fairly good selection of statechart variants commonly used in automotive industries. We have realised an amount of modelling rules taken from existing rule catalogues [1] to analyse static properties. Design rules expressing widely accepted "best practice" as well as individual preferences can be checked automatically on different statechart notations. Models can be analysed with respect to their conformance to a standard and their compatability to other CASE tools.

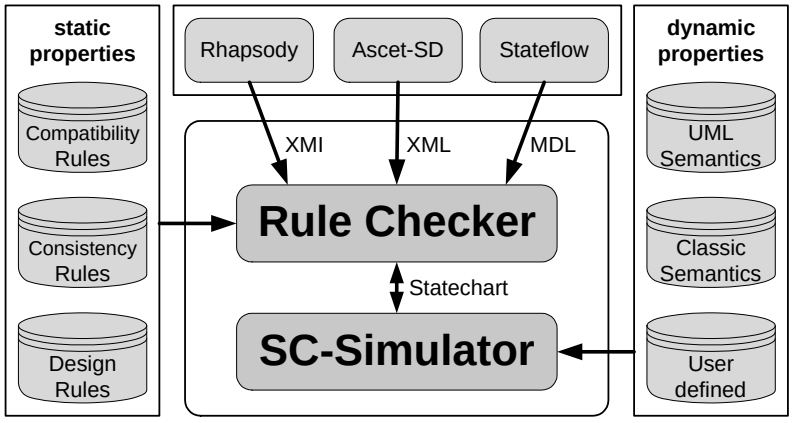


Fig. 1. The context of automated analysis

[2] but also other state based formalisms like Harel's statecharts [11], MATLAB/Simulink/Stateflow charts [17] or Ascet-SD state machines [6]

2.1 Checking Static Properties of Statecharts

We defined and implemented more than 100 modelling rules in student theses [23, 3] that can be checked using static analysis techniques on a structural representation of a state diagram (see Section 2.2). To keep track of the rules, they are grouped into three categories:

Compatibility Rules: Unfortunately, model exchange between different CASE tools is not possible so far, because many tool providers developed proprietary variants. To overcome this situation, we have rules to analyse a statechart with respect to its compatibility with different state-based tools. On the one hand, we analyse whether a model contains only elements specified in the UML standard [21]. On the other hand, we define rules to identify model elements that are not compatible with other CASE tools. Examples for this rule type are:

- Terminator-connectors from Rhapsody [13] are not supported by UML. Therefore, they should be avoided.
- The horizontal ordering of concurrent regions determines the execution ordering in some tools (e.g. in ARTiSAN Realtime Studio [2]). That should be avoided because other UML tools do not support this feature.

Consistency Rules: Consistency rules check the syntactically correct usage of modelling elements in a statechart. A violation of these rules causes serious trouble in the execution of the statechart. Most commercial UML tools check at most a subset of the well-formedness rules which are recommended by the OMG. Examples for consistency rules are:

- The content of a guard must be syntactically correct. To check this, a parser for guard expression is included in the *Rule Checker*.
- A join state must have at least two incoming transitions and exactly one outgoing transition. Whereas in a fork state it is vice versa.

```
public void check(StateChart sc) {
  boolean checkFinals=getParameterValue(1).equalsIgnoreCase("true");

  Iterator it=sc.getStates().iterator();
  State state=null;

  while (it.hasNext()){
    state=(State)it.next();
    if ((checkFinals) && (state.getType()==State.FINAL_STATE))
      if (state.getOutgoingTransitionCount()!=0)
        ....
}}
```

```
context FinalState
inv: self.outgoing->size = 0
```

Fig. 2. Well-formedness rule for final states: Java vs. OCL

Design Rules: In difference to consistency rules, design rules are concerned with less critical aspects of a model. The positioning of states and labels, the direction of data flow, and other aspects to improve homogeneity and readability are addressed in design rules. Examples for design rules are:

– The transition labels should be positioned left hand from the flow direction.
– The initial state should be located in the upper left corner of a composite state and the init-transition should be attached on the left side of the state.

Additionally, we search for hints on behavioural problems that can be recognized on structural properties of the statechart like miracle states (states without incoming transitions, but outgoing ones), useless regions, or mistaken entity names.

Most rules are implemented in Java. But we also implemented a module to support the declaration of rules in the standard format OCL (Object Constraint Language) [21]. In OCL rules can be expressed more directly in terms of a high level logical language and the user does not need to have detailed knowledge on Java programming and the internal data structure of the *Rule Checker*. In Figure 2 the declaration of a rule in Java and OCL is compared. The example refers to the OMG well-formedness rule on final states, namely a final state cannot have any outgoing transitions.

2.2 Using the *Rule Checker*

To analyse a statechart as depicted in Figure 3, the diagram is imported in the *Rule Checker*. The statechart models part of a power window. It contains typical modelling errors like a missing initial state in a compound state and a miracle state.

Figure 4 shows a screenshot of the *Rule Checker*'s main window. In its upper right part the window provides a hierarchical tree representation of the statechart, which is deduced from its internal data structure via so-called higraphs [12]. It clearly shows the hierarchy of states and pseudo-states and the resulting dependencies. This representation is independent from the positioning or scaling of model elements in a CASE tool. Additionally, it allows for a comparison of structural similarities between statechart models. In our tool the tree representation is used to present the statechart to the user during the analysis process and to mark states causing a rule violation.

On the left hand side of the main window the result of the analysis is presented to the user as a list of violations found in the model. The list is sorted due to the relevance of violation (error, warning, proposal). If the user selects a rule violation in the list, the node in the tree representation will be highlighted by an arrow. Additional analysis results on the faulty node will be offered in a pop-up window if the user double clicks on the rule violation.

In the bottom part of the *Rule Checker*'s main window, user and system calls are logged. During a run of the *Rule Checker*, the rules are listed in the order they are checked on the model.

Configuration of statechart analysis is one of the main capabilities of the *Rule Checker*. How to select or deselect rules manually and how to decide, which rules should be considered, is illustrated in Figure 5 and Figure 6.

To improve applicability, the *Rule Checker* can be easily configured and extended in several directions:

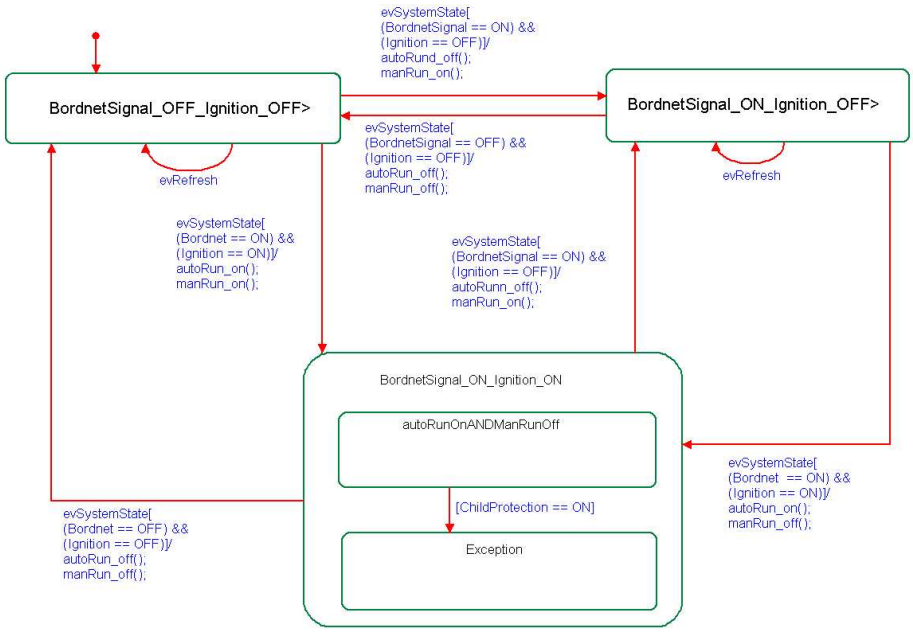


Fig. 3. Statechart with modelling errors

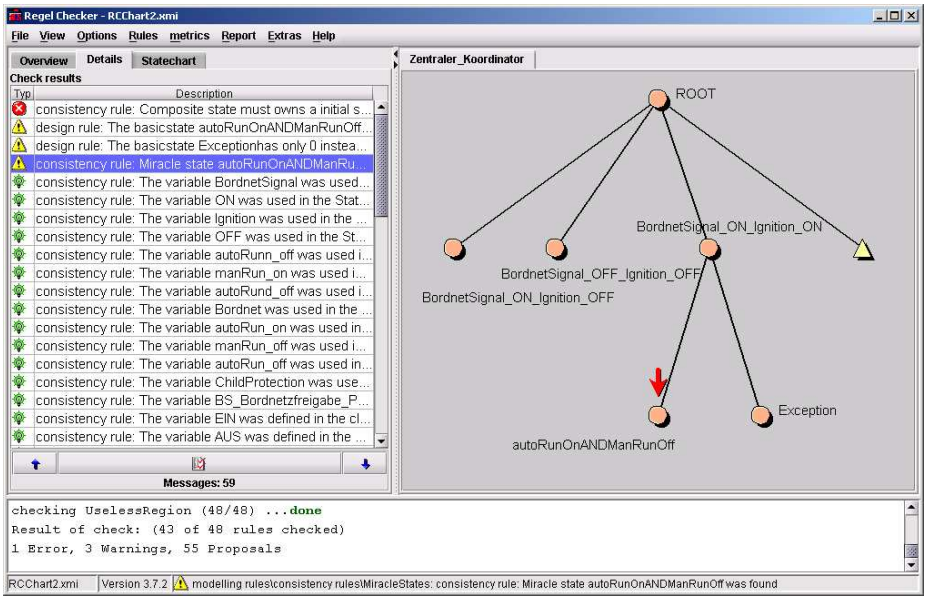


Fig. 4. Main window of the *Rule Checker*

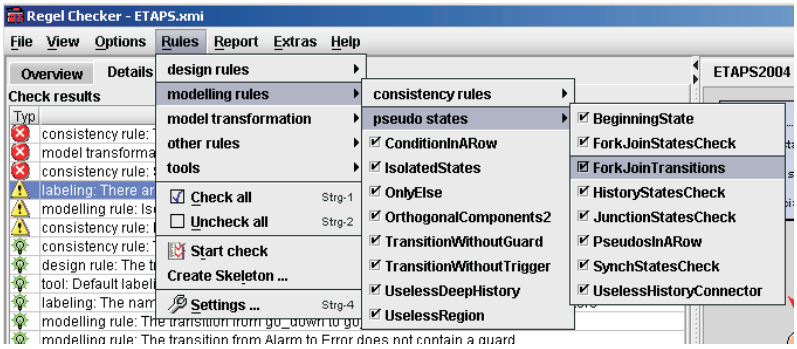


Fig. 5. Selecting modelling rules

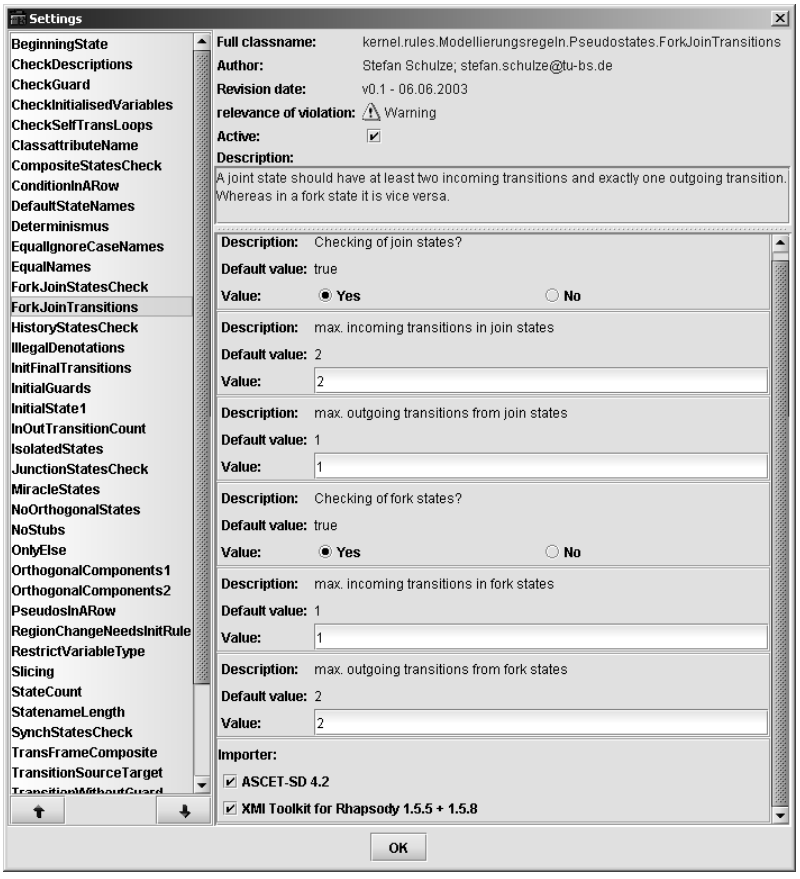


Fig. 6. Parameterising modelling rules

– The rules are grouped in catalogues so that the user can activate and deactivate rules according to his/her needs (see Figure 5). Moreover, predefined rule catalogues like checking conformance to the UML standard are provided.
– Many rules can be instantiated by parameters. Working with generic rules keeps the catalogues concise and makes analysis more efficient because a number of checks can be executed in one step (see Figure 6).
– By realising another interface, the *Rule Checker* can be connected to other CASE tools with their own proprietary statechart variant.
– To extend the set of rules, the user may implement a new modelling rule in Java or OCL.

With respect to UML, the *Rule Checker* only supports the import from Rhapsody [13] and our own proprietary editor at the moment, which is due to the fact that many UML tools provide XMI export but the interfaces are not at all compatible. As soon as the problems on the tool suppliers side are resolved, interfaces from other tools can be implemented easily. However, the rule catalogue implemented so far contains already a number of rules that refer to other tool environments like ARTiSAN Realtime Studio [2].

In difference to our *Rule Checker*, other tools like the prototype *GDC* [18] or the commercial tool *mint* [22] only cover limited spots of the field:

– By the script language both tools are restricted to a particular system platform.
– Function queries and access to the internal representation are mixed up.
– Rules are static, i.e. they cannot be modified at runtime or/and have no parameters for a flexible handling.
– Rule checking is realised only for models of the MATLAB/Simulink/Stateflow family.

The USE tool [25] deals with UML class diagrams and OCL to express system specifications. UML Object diagrams can be tested against this specifications. The *Rule Checker* uses OCL to specify rules for state-based models. Therefore, the USE tool and the *Rule Checker* cannot be compared directly.

We aim for a more general and flexible approach which provides interfaces to various UML tools and also supports other state-based models relevant in automotive like MATLAB/Simulink/Stateflow charts and Ascet-SD state machines. Consequently, our analysis is based on a generalised structural statechart representation with the expressive power to capture the features of various statechart formalisms.

3 Validating the Dynamic Behaviour

Since our approach aims for the analysis of state-based models originating from different development environments, a validation of the behaviour should be based on that semantics that is implemented in the original CASE tool. In the case of UML tools, nearly each tool exposes its particular interpretation of the

UML standard with subtle differences in syntax and semantics. For instance, ARTiSAN Realtime Studio and Rhapsody differ with respect to the execution order of concurrent regions and the implicit priorities assigned to transitions. Thus, rebuilding the semantics of some tool is a tedious business. But recently several authors started to work on formalising the semantics of tools most relevant in practice (see [10] for Rhapsody and [9] for MATLAB), and also many tool suppliers provide good support.

By simulation, the developer can detect errors, analyse the behaviour of models partially, and investigate whether the models behave as expected. But in contrast to rigorous formal approaches, error-freeness cannot be proven. However, using the *Rule Checker* for static analysis in a first step, the developer is directed to conspicuous parts of the model which should be analysed in more detail.

Another application of a configurable simulator is to compare the model's behaviour under different semantics. Thereby, the designer gets valuable insights how the same model will behave in different development environments.

3.1 Building a Modular Semantics

We aim for a simulator that can be configured to different statechart semantics. We have chosen a modular semantics to realise this goal. Each module concerns an orthogonal semantic aspect like the handling of events, in particular their ordering, or the priority of transitions with respect to their hierarchical position in the statechart, which is a substantial difference between the original statechart semantics by Harel [11] and the UML semantics. The main modules are the `EventSemantics`, the `ActionSemantics`, the `StateSemantics`, the `TransistionSemantics`, and the `StepSemantics`. Each main module is composed of a number of submodules to handle particular aspects of the semantic entity, for instance the module `EventSemantics` contains among other the modules `CompletionEventSemantics`, and `DeferEventSemantics`. As long as a statechart formalism can be modularised according to this scheme, it can be rebuilt by implementing the specific semantic choices in the corresponding modules.

To configure a statechart semantics, a collection of semantic modules has to be selected that covers all entities ocurring in the model to be simulated. Exchanging a module will result in changes in the semantics and therefore in a modified behaviour of the statechart.

3.2 Using the *SC-Simulator*

The simulation can be run independently from the *Rule Checker* with its own GUI (graphical user interface), which is shown in Figure 7. Statechart models can be imported using the *Rule Checker*'s importer or a proprietary editor.

The simulated model is represented graphically with the usual features of a simulator like highlighting the current global configuration. Detailed information on the current configuration is provided in the four lists on the right side of the simulation main window. The upper left list contains the events invoked by the

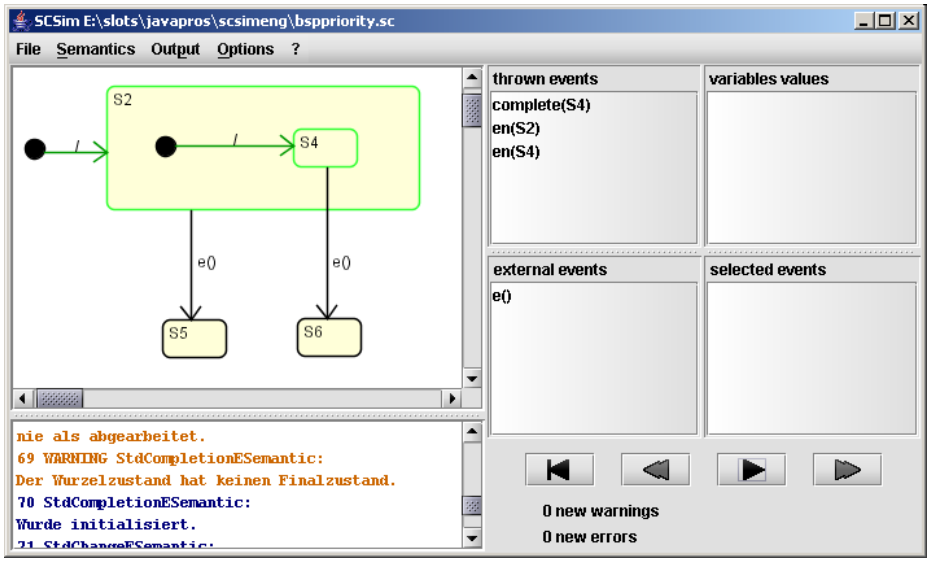


Fig. 7. The simulation main window

statechart or its environment and not yet consumed or dispatched. The upper right one shows the current values of the variables in the model. These values are part of the current configuration of the model. The lower left list contains all events invocable by the environment (i.e. external events) and not yet selected. The right list contains the events for invocation in order of their selection. To get a deeper understanding on the execution of steps, the user may analyse the log file that contains a detailed textual description of each micro-step that has occurred (see the bottom area in Figure 7). This feature makes implicit semantic variations explicit to the user.

Such implicit semantic variations can lead to unpredictable behaviour as we observed when we compared the semantics of different UML tools in detail: For instance, according to the UML standard, completion events are generated just after completing a state. However, completion of a (compound) state is interpreted differently in different tools. For instance, we observed that in case of a complicated situation with interlevel transitions and concurrent regions even developers who were quite experienced with UML were not able to predict the behaviour, in particular regarding a specific tool. When such a difficult situation is observed, one should think about a structural characterisation of the problem. In the example, this would be the combination of concurrent subregions, completions events, and interlevel transitions in a subchart. In a further step, the structural characterisation could be the basis for a new modelling rule because modelling a situation in which a clear understanding of the behaviour is missing

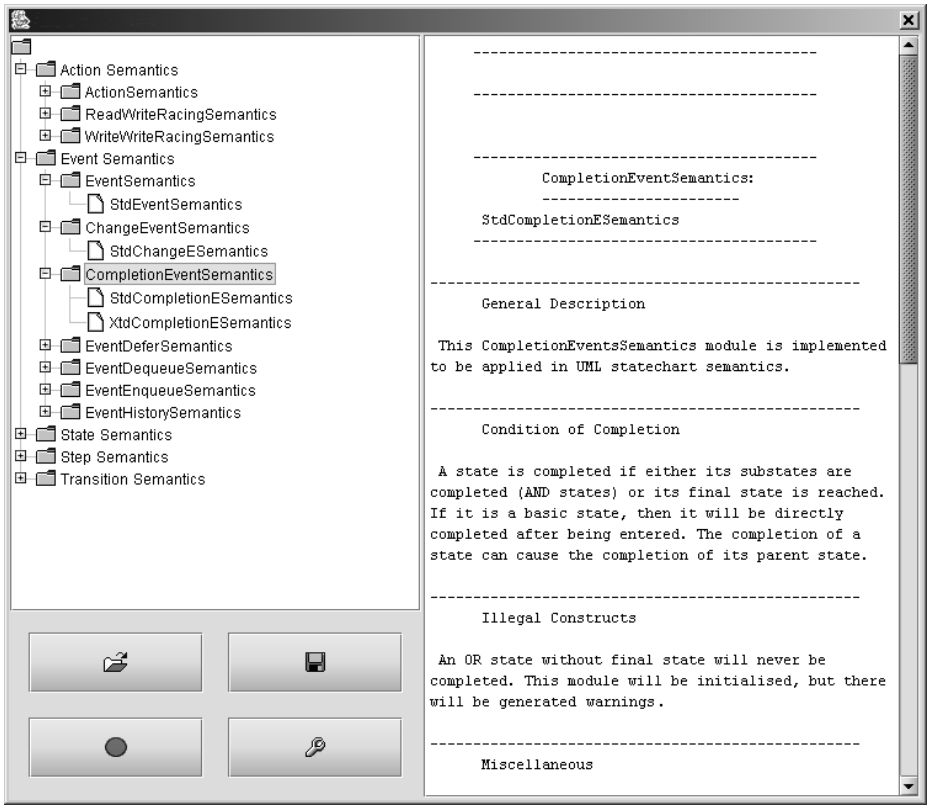


Fig. 8. The *Semantics Selector* to configure the semantics

should be avoided. Such a feedback from the dynamic analysis will improve the static analysis of models in future.

The feature of configuring the semantics is illustrated in Figure 8. The selection window contains a tree representation of all available semantic modules. A description of all these modules is provided. The selection of modules in the actual semantics can be displayed and modified as well. Furthermore, sets of modules forming a complete statechart semantics, as e.g. the Rhapsody semantics, can be stored.

4 Conclusion

We presented a lightweight approach to the correct design of embedded systems for which the state-based behavioural modelling is a predominant task. Our work aims for the automated detection of design errors in a model-based development process by means of static analysis and simulation. Thereby, we

are able to verify a broad spectrum of properties like consistency, conformance to the UML standard, and compatibility to domain specific modelling styles but also properties relevant to the dynamic behaviour like miracle states, dead code, etc. The simulation is used to validate the behaviour, in particular in those parts of a model that have been identified as conspicuous by the static analysis. As experience shows, errors are more likely in such areas.

Our approach is tailored for an immediate improvement of state-based modelling in present industrial settings. Therefore, our report can also be considered as an experience report on formal methods at work.

At the present status, our tools do not provide rigorous formal verification functionality like model checking, but it helps the designer to *efficiently* overcome at least two common problems in model-based design of large systems [20]: Understandability and consistency of models are improved, which is crucial in large development teams. The attention of the developer is drawn to conspicuous modelling situations that need further investigation.

Our approach was developed in close cooperation with the electronic design and vehicle system integration of the Volkswagen AG. It was validated in a medium sized industrial case study (more than 1000 states) [5], namely a power window for the Volkswagen AG. The Volkswagen AG applied for a patent for the kernel of the *Rule Checker*. However, some parts are still under development. But the *Rule Checker* and *SC-Simulator* are runnable tools, which are already helpful increasing the quality of state-based models.

In future we plan to extend our approach in several directions:

- We will consider also structural UML diagrams.
- Based on the simulator, we plan to implement state space exploration which will establish formal verification at least for classical safety properties and small models.
- The static analysis of models is executed on a structural representation which cannot only be used for static analysis but also to measure a design with respect to model metrics. Metrics also give valuable hints about which parts of a model should be subject of a more rigorous analysis. In [19] we investigated metrics concerning the number and structuring of states, the complexity of transitions, and the layout of a statechart to measure readabiblity and complexity of state diagrams.

References

1. S.W. Ambler. *The Elements of UML Style*. Cambridge University Press, 2003.
2. Artisan. http://www.artisansw.com/products/professional_overview.asp, 2004.
3. K. Chkhihvadze. Erstellung von Modellierungsrichtlinien für das UML-Tool Artisan. Studienarbeit, TU Braunschweig, Institut für Software, May 2004.
4. S. Cigoli, P. Leblanc, S. Malaponti, D. Mandrioli, M. Mazzucchelli, A. Morzenti, and P. Spoletini. An experiment in applying UML 2.0 to the development of an industrial critical application. In *[16]*, pages 19–34, 2003.

5. K. Diethers and M. Mutz. Improving software development in the automotive area through tool supported modelling and formal analysis. In *System Design Automation (SDA)*, pages 81–88, 2002.
6. ETAS. http://de.etasgroup.com/products/ascet_sd/, 2004.
7. S. Graf and J. Hooman. Correct development of embedded systems. In *European Workshop on Software Architecture: Languages, Styles, Models, Tools, and Applications (EWSA)*, Lecture Notes in Computer Science, 2004.
8. K. Grimm. Software technology in an automotive company - major challenges. In *Proceedings of the 25th International Conference on Software Engineering*. IEEE, 2003.
9. G. Hamon and J. Rushby. An operational semantics for Stateflow. In *Fundamental Approaches to Software Engineering (FASE)*, volume 2984 of *Lecture Notes in Computer Science*, pages 229–243, 2004.
10. D. Harel and H. Kugler. The Rhapsody semantics of statecharts (or, on the executable core of the UML. In *Integration of Software Specification Techniques for Application in Engineering (CHARTS)*, Lecture Notes in Computer Science, 2004.
11. D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Transactions on Software Engineering Methods*, 5(4), October 1996.
12. D. Harel and G. Yashchin. An algorithm for blob hierarchy layout. In *Advanced Visual Interfaces*, pages 29–40, 2000.
13. I-Logix. www.ilogix.com/products/rhapsody, 2004.
14. D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 29(4), 1996.
15. M. Jeckle. http://www.jeckle.de/umltools.htm, 2004.
16. J. Jürjens, B. Rumpe, R. France, and E. B. Fernandez, editors. *Critical Systems Development with UML*. TU Munich Technical Report TUM-I0323, 2003.
17. Mathworks. http://www.mathworks.com/products/prodoverview.shtml, 2004.
18. M. Moutos, A. Korn, and C. Fisel. Guideline-Checker. Master thesis, TU Esslingen, 2000.
19. M. Mutz. Metriken und Regeln für eine durchgängige und modellbasierte SW-Entwicklung im Automobilbereich. In *2. Workshop on Automotive Software Engineering*, 2004. to appear.
20. M. Mutz, M. Huhn, U. Goltz, and C. Krömke. Model based system development in automotive. In *SAE World Congress 2003*. Society of Automotive Engineers, 2003.
21. OMG. OMG Unified Modeling Language Specification, 2003. Version 1.5.
22. P. Gilhead and R. Tarragon. Style matters, 2004. http://www.ricardo.com/mint.
23. S. Pietsch. Automatische Überprüfung von UML Statecharts anhand definierbarer Design-Regeln. Master thesis, TU Braunschweig, Institut für Software, March 2003.
24. The precise UML group. http://www.cs.york.ac.uk/puml/publications.html, 2004.
25. P. Ziemann and M. Gogolla. Validating ocl specifications with the use tool—an example based on the bart case study. In Thomas Arts and Wan Fokkink, editors, *Electronic Notes in Theoretical Computer Science*, volume 80. Elsevier, 2003.

Exploration games for safety-critical system design with UML 2.0

Jennifer Tenzer*

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh

Abstract. UML has its origin in mainstream software engineering and is often used informally by software designers. Critical systems on the other hand are usually specified very precisely and frequently require formal verification. In this paper we introduce the idea of using games to traverse smoothly from an informally defined design in UML to one that is verifiable. The modeller repeatedly plays an exploration game to detect flaws and to find out where he has to be more precise. The incrementation of the design in response to these discoveries is part of the game. We discuss exploration games with UML 2.0 activity diagrams and state machines on the example of a small critical system. Moreover we give a brief summary of the planned tool support for this approach.

1 Introduction

Safety-critical systems may cause injury or death to human beings when they fail. Therefore their design needs particular attention with respect to the fulfilment of critical safety requirements. Formal methods for the verification of a system against its requirements are common tools for improving safety. However, the application of formal verification techniques becomes difficult if the system is modelled in UML, because UML is often used informally.

In this paper we introduce exploration games as technique for making a UML design model more precise with the help of the human designer. The driver for the incrementation of the model is the requirements specification, which is usually very detailed in the case of critical systems. The designer plays a game to look at the design from different perspectives and to add more detail where necessary. At any point he can decide to play the game in "strict mode" to verify parts of the model. During the verification the designer may discover flaws in the model or realise that the model is too incomplete to be verified, which leads to further exploration of the design. The exploration by playing games continues until the designer believes that the model is precise enough for his purpose.

At the moment none of the existing UML design tools provides much support for design exploration as suggested here. Advanced tools like Rhapsody [11] or Real Time Studio [10] allow the interactive animation of behavioural diagrams,

* Email: J.N.Tenzer@sms.ed.ac.uk Fax: +44 131 667 7209

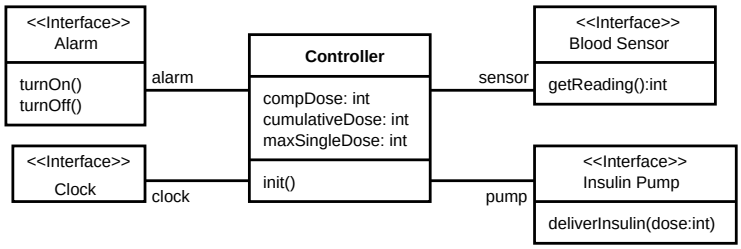


Fig. 1. Class diagram for the insulin pump system

but they cannot point the user to flaws in the design and do not permit modifications of the model while the animation is running. These tools do not help the user to decide whether a design is suitable, to improve it, or compare it to other solutions. The motivation for applying games to UML is to develop tool concepts which help to fill this gap. This idea has first been introduced in [14] on a general level.

Here we focus on the definition of the games with UML 2.0 activity diagrams and state machines and consider this approach with respect to safety-critical systems. We assume that the reader is familiar with UML, in particular with UML state machines and the object constraint language OCL. In this paper we use the terminology and notation of UML 2.0 [17] and OCL2.0 [7].

The following section introduces the example of a critical system design which we will use throughout this paper. We point out problems in the design which can be discovered and solved by an exploration game. In section 3 we give a short summary of the exploration game framework. Section 4 explains how exploration games are applied to design with UML. Two game variants based on activity diagrams and state machines are considered in detail. Section 5 briefly describes how exploration games with UML can be supported by a tool. Section 6 discusses related work and in section 7 we conclude and point out possibilities for future work.

2 Motivation

As motivating example we consider parts of the control software for an insulin pump. This example is based on a case study used in [13]. The insulin pump system is a safety-critical system which delivers regular doses of insulin to diabetics to reduce the patient's sugar level. Figure 1 shows its components in the form of a UML class diagram. Here we assume that the interfaces of the hardware components are fixed. The structure of the **Controller** is variable and has to be defined by the designer. The class diagram contains a first version of the **Controller** class.

We assume that the designer has access to a detailed requirements specification for the system. As starting point for designing the system he chooses

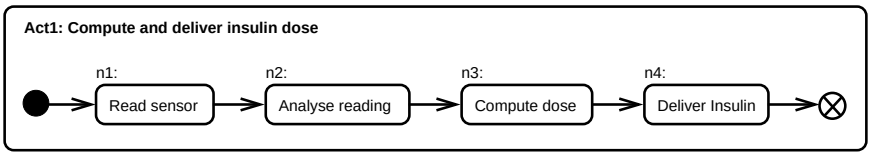


Fig. 2. Activity diagram modelling the normal operation of **Controller**

to model the basic functionality of **Controller** by a UML activity diagram as shown in figure 2. The diagram is defined informally and on a high level of abstraction. The nodes are labelled by n1, n2, ... in order to allow easier reference later in this paper. The activity diagram itself is labelled by Act1. The modeller's task is now to make this model more precise and add information according to the requirements specification.

In this example we concentrate on the safety requirement that the computed dose should never exceed the maximum single dose. Using the information in the class diagram this can be specified more formally by an OCL constraint:

```
context Controller
   inv: self.compDose <= self.maxSingleDose
```

We assume that the modeller identifies that action Compute dose is constrained by this requirement and refines it by the activity diagram in figure 3. This preliminary version does not cover all cases that are relevant for the system adequately. For example, the diagram does not model what should happen if the sugar is sugar level is low. What the designer has in mind is to model the choice between two different algorithms depending on the analysis of the last sensor reading. The result of the selected algorithm is stored in attribute compDose of **Controller**. If we choose to interpret the invariant for **Controller** strictly[1], the current design is not correct with respect to the requirements. If one of the algorithms yields d>maxSingleDose as result, the constraint does not hold. A simple solution to this problem is to insert an action node before n8 which adjusts the value to which compDose is set such that it fulfils the safety condition.

The design also leaves open many possibly important details. Some of the open design questions that the modeller might ask himself are the following:

- What should happen if none of the conditions at the edges emerging from the decision node hold?
- Should there be postconditions for the algorithms which guarantee that the return value d is in a specific range?

[1] UML does not specify whether an invariant must hold during the execution of an activity. For activities that represent methods it is not sensible to request this because the object is in an unstable state during the execution. However, for activity diagrams on a high abstraction level involving actions performed by different objects a strict interpretation may be appropriate.

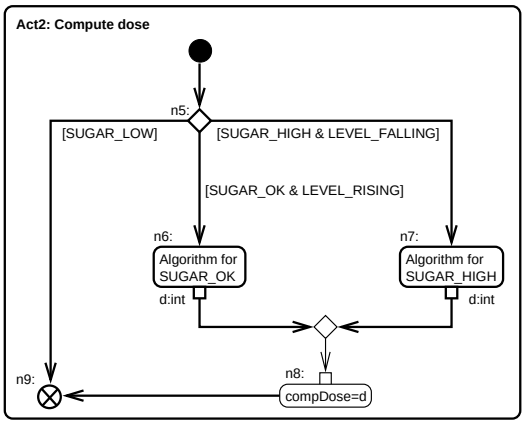


Fig. 3. Activity diagram Compute Dose

– Can an algorithm fail? If so, how do we recognise failure and what should happen in this case?
– In the description of the system it is mentioned that a low sugar level is dangerous for the patient. The sugar level can be increased by eating something. Should the system suggest this to the user? Does the requirements specification consider this case or is it possibly incomplete?

Some of these questions may be answered by the requirements specification. Where this is not the case the designer has to come up with a solution. Playing an exploration game with the design encourages the designer to ask himself some of these questions, and also to experiment with possible answers.

3 Summary of the exploration game framework

In this section we give a brief informal explanation of the exploration game framework. Exploration games are extensions of verification games (for an overview on verification games see, for instance, [16] and [3]) where the moves have parameters and possibly informally defined preconditions. Exploration games are played between two players called Verifier and Refuter. Whenever we refer to one of the players specifically we use the female form for Verifier and the male form for Refuter. The aim of Verifier is to show that the system on which the game is based fulfils a certain property, Refuter's aim is to show that this is not the case.

An exploration game is set in an arena, which is a directed graph that consists of positions as vertexes and moves as edges. A position usually represents the current state of the system and an edge a state transformation. Each position is owned by one of the players. A move has a parameter signature and is labelled by

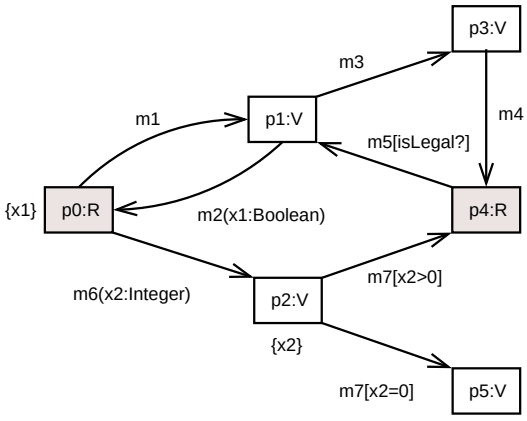


Fig. 4. Arena of an exploration game

a name, precondition, and parameter assignment. An example arena is shown in figure 4. For reasons of presentation we abstract from the inner structure of the positions and from the concrete parameter assignment in this summary. The sets at the positions indicate which parameters are "in scope" and part of the position's inner structure. For all positions where no set of parameters is attached there are no parameters in scope. The positions of Refuter are shown as grey-shaded rectangles, which are labelled by the position name and R. Those of Verifier are represented by unfilled rectangles and contain V in the label. The preconditions for moves are shown within square brackets. If there is no precondition specified for a move we assume that it is *true*.

An exploration game is defined by an arena, an initial position, a set of winning conditions and a set of rules. As an example we consider a game with the arena shown in figure 4. We choose position p0 as initial position and assume that x1 has the value $false$. We define that Refuter wins if position p3 is reached. All other plays, including infinite plays, are won by Verifier. The rules of the game are that a player makes all moves which emerge from positions that are owned by him and provides the parameters for these moves. A player wins a play if one of his winning conditions is fulfilled or his opponent cannot make another move. Moreover an independent Referee decides about preconditions which are informally defined, such as, for instance, isLegal?.

An exploration game can be played in strict mode without any exploration. In this case the players move exclusively as shown in the arena, starting from the initial position. At each position the preconditions at the moves that emerge from it are evaluated. If a precondition is informally defined the Referee decides whether it is *true* or $false$. The moves whose preconditions are evaluated to *true* are the legal moves from this position. The player who owns the current position chooses a legal move and provides parameter values for it.

Now let us consider some example plays. If Refuter chooses m6 from the initial position p0 and provides 0 as value for x2 Verifier has to respond by move m7 to position p5 because this is the only legal move. Since Verifier cannot move anymore from p5 Refuter wins this play. If Refuter always selects m1 from position p0 and Verifier always responds by choosing m2 we have an infinite play which is won by Verifier. Finally, assume Refuter chooses m6 and provides 3 as value for x2 such that Verifier is forced to move to p4. The Referee declares that isLegal? is evaluated to *true* and Refuter can move to position p1. If Verifier selects m3, Refuter wins the play because position p3 is reached. If Refuter is clever, he plays as described in the first example, because he then wins the play under all circumstances. His choice of moves describes a winning strategy in this case.

If the game is played in exploration mode, one of the players is allowed to "cheat" in various ways in order to improve his chances of winning the game. We call the player who is allowed to cheat the Explorer, the other one the Defender. In addition to the moves in the arena the Explorer can either backtrack to an earlier position, change the current position, or make a metamove which changes parts of the game definition. A metamove modifies the arena, including the guard conditions at moves, the winning conditions or the set of rules. Since the game definition is modified while the game is played, a play is now a sequence of positions and game definitions. A play is always continued according to the current game definition. If the Explorer backtracks to a position p he travels back in the play history. Thus the game will be continued under the game definition that was the current one at p. Backtracking is extremely useful if the Explorer has made a mistake earlier in the play and wants to correct it. It is often necessary for the Explorer to combine backtracking and metamoves if he wants to be successful in his exploration. Changing the current position in a play permits "what-if"-explorations where the Explorer can pretend that the last move resulted in a different position. We will see later that a move like this is particularly useful for refining the inner structure of a position if parts of it are undefined.

Again we consider some examples. If Refuter applies his winning strategy, Verifier loses at position p5. If Verifier is the Explorer she can spoil Refuter's winning strategy at this point by adding a new move from position p5 to p4 to the arena. Instead of being stuck, Verifier can now escape from position p4. If this new variant of the game is played in strict mode, Verifier wins all plays. Either Refuter is stuck at position p4 if the Referee evaluates isLegal? to $false$, or the play is infinite.

Now let us assume that Refuter acts as Explorer in the new game which was the result of Verifiers exploration. He may begin the play by a metamove that changes the winning conditions. Refuter defines that a play during which position p3 is reached is no longer won by him. Instead he declares that he wins all plays in which move m2 has been made. Refuter will now try to force Verifier to make this move and selects m1 which leads to position p1. Unfortunately Verifier selects m3 and m4 as next moves. Refuter realises that it depends on the decision of the Referee whether he will be able to get back to position p1,

which is the source of m2. Hence he makes another metamove and changes the precondition at m5 to *true*. Refuter can now safely move to p1. However, Verifier still is allowed to choose between m2 and m3 at position p1. Assume Verifier chooses m3 and m4 again. Refuter realises that he has to prevent this move sequence because Verifier can enforce an infinite loop by it. He changes the rules of the game such that he is now allowed to select the next move at position p1. At position p1 he chooses m2 and wins the play. In fact, the last metamove which gave him the power to decide about the next move at p1 would have been enough to reach this aim. If Refuter prefers a simpler model, he could backtrack to the initial position and game definition, and then make this metamove again. If this variant of the game is played in strict mode, Refuter wins all plays if he plays rationally, i.e. there exists a winning strategy for him.

By repeated exploration the players add more detail to the game definition. If they focus on making the preconditions of moves more precise, the Referee may not be needed anymore at some point. In this case the exploration game amounts to a verification game if it is played in strict mode. If Verifier has a winning strategy for the game, the property which is expressed by the winning conditions holds. The winning strategy can be regarded as a formal proof of this fact.

4 Exploration games with UML

There are several possibilities for applying the exploration game framework to UML. The basic idea is to use the UML diagrams for the definition of the arena and OCL constraints as winning conditions. Our motivation for using these games for UML is to extend tool support for design with UML, which will be discussed further in section 5. If the game is played with a tool the Explorer has to be controlled by the human designer, because the exploration requires knowledge about the system and some "intelligence". Exploring in the roles of Refuter and Verifier provides different views on the design.

Here we focus on exploration games with UML which are played over a fixed set of objects[2]. For the examples in this section we use an object a of **Alarm**, and c of **Controller**. A position always contains a system snapshot which specifies the current configuration of each object. An object configuration does not have to be complete. It does not have to contain a value for all structural features and links, but may leave some parts undefined. Furthermore it is not required that each class is represented by an object. An example system snapshot S1 with configurations for a and c is shown as UML instance diagram in figure 5.

The changes of object states over time are modelled by behavioural diagrams and form the moves of the game. How exactly moves and positions are defined depends on the diagram types that are considered. In the following sections we will discuss activity diagrams and state machines. The definition of metamoves

[2] If we allow object creation and destruction, we have to ensure that the state space of the arena does not become infinite by introducing boundaries on how many objects may exist at the same time for each class.

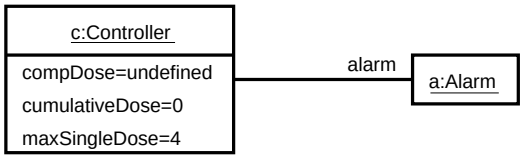


Fig. 5. System snapshot S1

also differs with respect to the diagrams that are used. A common property of games with UML is that a metamove never affects the arena directly. Instead the UML diagrams are modified which induces changes in the arena. Changing the OCL invariants, which represent the winning conditions, or the rules of the game are metamoves which are not diagram-specific. They can always be made by the Explorer and we refer to them as *general metamoves*.

4.1 Activity diagrams

In an exploration game which is based on activity diagrams each position consists of the following parts.

- A system snapshot.
- The set of activities which are running with their markings.
- The set of parameters which are in scope with their assignments.

The marking specifies on which nodes of the activity diagram data and control tokens are placed. Notice that we only consider single execution of activities here, which means that for each invocation of the activity a separate marking is recorded. All positions where no activity is executing belong to Refuter, all other positions to Verifier. The parameters which are in scope at a position are the data tokens that exist at this time. Figure 6 shows an excerpt of the arena for a game based on Act1 and Act2 which we will discuss further. Refuter owns position p0 and all other positions belong to Verifier.

Refuter can only move by invoking an activity which is modelled by an activity diagram. Verifier can move by moving tokens or executing an action in one of the running activities if the action has all necessary control and data tokens. Thereby an action may cause the invocation of other activities like Compute dose in our example[3]. Verifier may also complete an activity if all tokens are consumed or an activity final node is reached. The execution of an action changes the marking of the activity according to the UML semantics and may also affect the object configurations within the positions. If an action is informally specified, like Read sensor in our example, the snapshot after its execution is undefined.

[3] Here we assume that Compute dose is a synchronous call action. That means the execution of Act1 is only continued after Act2 is completed. UML also permits asynchronous call actions.

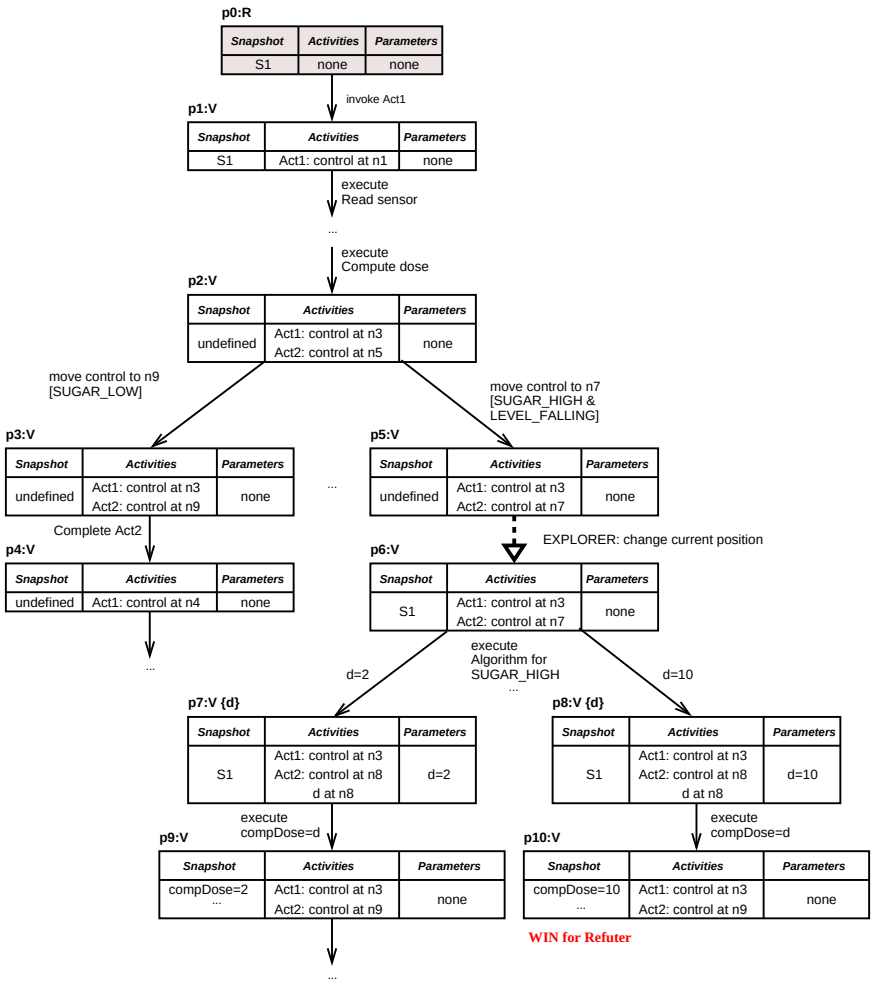


Fig. 6. Excerpt of an arena based on Act1 and Act2

For each move by which the marking of the diagrams is changed, the guards at activity edges define the precondition of the move. The parameters are specified by the output pins of actions because values of these types are present as data tokens after the action has been performed. We assume that each parameter is available at the succeeding positions until the corresponding data token is consumed. For example, parameter d is in scope at position p7, but not anymore at p9, because the corresponding token was used during the move.

Refuter wins a play of the game if Verifier cannot make a move, fails to provide parameter values for a move, or if an OCL invariant is violated. All other plays are won by Verifier. For our example we use the OCL invariant expressing a safety condition for compDose which was specified in section 2. As initial position of the game we choose p0. The excerpt in figure 6 shows some of the positions which are reachable from this position.

The Explorer in an exploration game for activity diagrams may make diagram-specific metamoves as follows.

- Add or delete activity edges.
- Add or delete activity nodes. If a node is deleted, all edges of which this node is a source or target are deleted as well.
- Modify guard conditions at activity edges.
- Change an action.
- Add or delete an activity diagram.

In addition to that he can make general metamoves, change the current position or backtrack at any time as for every exploration game.

We consider some plays of our example game. First assume that Refuter is the Explorer and changes the rules of the game such that he decides about the legality of informal preconditions at all moves. With this setting it is easy for Refuter to win. The preconditions of all moves emerging from p2 are informally defined, and Refuter can declare that none of them holds. In that case Verifier is stuck and Refuter wins the play. Verifier may counter with an exploration during which the adds an else-branch to the decision node n5. Alternatively she might think about the other cases which are not yet covered and introduce new branches and appropriate actions for them. Another possibility is to make the guard conditions more precise, such that they can be evaluated automatically and it is not Refuter's responsibility to decide whether they hold.

In another exploration Refuter changes the current position at position p5. He specifies that S1 should be the snapshot within the current position. The change is indicated in figure 6, where p6 is is the new current position selected by Refuter. There is now a value defined for singleMaxDose which is part of the winning condition for Refuter. Refuter changes the rules such that he provides a parameter value for d if one of the two algorithms is executed. If he sets d>4 he wins the game because the OCL invariant is violated when compDose is set to d. An example of a winning position for Refuter is p10 in figure 6. The removal of this flaw could be attempted in another exploration by Verifier.

4.2 State machines

UML provides two different kinds of state machines called behavioural state machines and protocol state machines. In the context of exploration games behavioural state machines are used in a similar way as activity diagrams. They serve to build up the arena and model how the configuration of objects changes over time. Protocol state machines on the other hand can be used to restrict

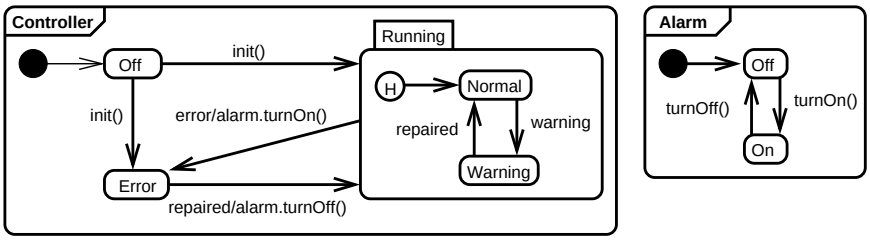


Fig. 7. State machines for **Controller** and **Alarm**

Refuter's choice of moves. We first consider the two behavioural state machines in figure 7 as example. Notice that the state machine for Controller is nondeterministic and that the event repair can have different effects depending on which state the object is in.

A position always consists of the following parts.

- A system snapshot.
- For each object
  - a "copy" of the state machine of its class and the current state configuration, and
  - an event queue.

All positions where the event queue is empty belong to Refuter, all others to Verifier. For our example we choose a position where a and c are in their default states as initial position. Furthermore we assume that c can access a via role name alarm, i.e. the two objects are linked, and that the event queue is empty.

Refuter can move by generating an event that is put into the event queue of the object(s) at which it is targeted. The parameters of an event are used as parameters for the corresponding move. Verifier has to fire a state machine transition which is triggered by the first event in the queue for each object. The state configurations of the objects change according to state machines. The guard conditions at the transitions constitute the preconditions of Verifier's moves. Firing a transition may cause other events if an effect is specified. Firing the transition from Running to Error, for example, generates a new event turnOn() which is put into the event queue of alarm. According to the UML run-to-completion semantics the **Controller** object only completes the transition to state Error when the processing of this event is completed. Until that point the object is in an undefined state.

The winning conditions of the game are induced by requirements as before. For our example we use the following requirement: if the controller detects an error the alarm must be turned on. This can be expressed by an OCL invariant:

```
context Controller
   inv: self.oclIsInState(Error) implies alarm.oclIsInState(On)
```

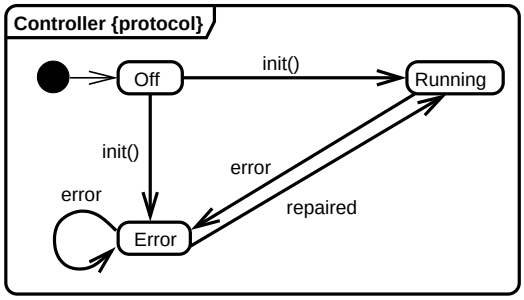


Fig. 8. Protocol state machine for **Controller**

Verifier also loses a play if she cannot fire a transition in response to an event. This can happen because there is no transition triggered by that event, the target object is undefined, or none of the guard conditions at the triggered transitions are fulfilled. In UML events which do not trigger a transition are simply discarded. However, since we consider critical systems here, we use a stricter definition[4] and make the game more difficult for Verifier. If we consider our example game Refuter can win very easily at the moment. He just has to generate an event which does not trigger a transition. For example, he can win by generating error when c is in state Off or Error. In order to restrict the choices of Refuter a protocol state machine can be defined. For our example we use the protocol state machine shown in figure 8. If Refuter violates the protocol he loses the game.

The Explorer in an exploration game based on state machines can make the following metamoves:

- Add or delete a transition between two existing states.
- Add or delete a state. All transitions which point to or emerge from a state that is deleted are also deleted.
- Change a guard or effect at a transition.
- Add or delete a state machine (either a protocol or behavioural state machine).

Let us first consider an exploration where Refuter is the Explorer. He changes the rules such that he can resolve the nondeterminism for the transitions triggered by init. He challenges by generating init and declares that the transition to Error has to be taken. After this move the invariant is violated because a is still in state Off, and Refuter wins the game. During the next play Verifier might act as Explorer. She can then add an effect alarm.turnOn() to the transition. Refuter can still win by generating init() and error. This sequence does not violate the

[4] This definition is problematic with respect to recursive calls. An object may be in an undefined state when an event arrives and cannot react to it. A detailed discussion of this problem can be found in [15]. Here we do not consider recursive calls.

protocol and Verifier cannot fire a transition for error if c is in state Error. Verifier might explore again and add a loop transition triggered by error to solve this problem.

5 Tool support

A UML design tool which is based on exploration games should help the human designer to set up a game, manage different versions of the design model, and store important information such as the play history. During a play, the tool fulfils different functions. First of all it should ensure that the game is played according to the current rules and evaluate preconditions of moves which are formally defined. The human modeller should be able to decide in which roles he wants to perform during a play. For example, he could act both as Verifier and Refuter, and the tool then merely helps him to set up and administrate the game.

Alternatively the tool can try to act as one of the players. How good the tool can play depends on how precise the model is defined and whether the tool can compute a winning strategy. However, since decisions about informally defined parts and exploration require knowledge about the system and design skills, we do not expect that the tool can act as Referee or Explorer. These decisions will probably always be made by the human designer.

In practice the design model will usually lead to an infinite arena. The tool can try to build up the part of the arena that is currently needed from the initial position and change it when necessary (e.g. after a metamove or change of the current position). In order to avoid infinite sets of parameter assignments, the modeller can be asked to provide a small set of test samples for each parameterised move.

We are currently working on tool support for exploration games with state machines as introduced in section 4.2. Instead of defining formal OCL constraints we expect the user to fill in a pattern for the kind of winning condition he wants to specify. For example, he has to define which state combinations lead to a win of Refuter in case of a reachability game like the one considered in our example. This simplification allows people who are not familiar with OCL to use the tool. We would like to read in preliminary design models as XMI files such as created by the Poseidon UML tool [8]. The model is then manipulated by the game tool and can be exported back to XMI at any point.

6 Related work

The tools HUGO [4] and vUML [6] help the designer check general properties of UML state machines, such as the possibility for deadlocks. HUGO additionally verifies whether desired (or undesired) behaviour specified by UML interaction diagrams can be realised by state machines. Both tools translate the UML model into a formal model which is then used as input for the model checking tool SPIN. A general overview on how model checking can be used for debugging UML

designs is given in [1]. An example of a tool which validates OCL constraints over system states specified by a collection of objects and links is USE (UML-based Specification Environment) [12].

The approaches mentioned here differ from ours in two respects. First, they do all require a UML model as input which is precisely defined. Informal guard conditions, undefined object attributes and non-deterministic state machines are usually not permitted. Second, they concentrate on the evaluation of a UML model, while our exploration games are focused on interactive modification of the design.

Since the UML semantics is not formally defined, there is a certain degree of freedom in the interpretation of a UML model. There is a large amount of work on formalising UML to make its semantics more precise. Usually these approaches focus on a particular part of UML. For example [2] concentrates on activity diagrams and [5] on state machines. A good overview on publications in this area can be found on the webpage of the Precise UML Group [9].

7 Conclusion and future work

In this paper we have presented how exploration games can help to make the design of a critical system in UML 2.0 more precise. The modeller repeatedly plays exploration games to find out if the model fulfils the requirements and his expectations. He has to act as Referee whenever the model does not provide enough information to continue the play. If he wants to change the game he has to act as the Explorer who is allowed to make metamoves. Most of the metamoves are made indirectly by changing the UML design model. The designer can experiment with the design and verifies parts of it while he is playing the game. He can focus on the most critical parts of the system in his explorations and check them against safety conditions given in the requirements specification.

On the game-theoretical side we have to investigate in future work to which extent a tool can play the role of one of the players. Since we would like the tool to play as best as possible we have to develop algorithms for the computation of winning strategies. Another interesting question is how far the interaction between the modeller and the tool can go. For instance, there might be situations where the tool can suggest metamoves or improve its strategy by asking the user for more information.

Concerning the connection between games and UML, further possibilities for using the different diagram types could be examined. For example, we have not considered combinations of state machines and activity diagrams in this paper. In fact, state machines can invoke activities in various ways, and activities can generate events which trigger state machine transitions. Another possibility is to take interaction diagrams into account. These diagrams could be used to specify both legal and illegal sequences of events which restrict the moves of Refuter. In contrast to protocol state machines interaction diagrams allow the formulation of event sequences for a set of objects, not only for a single object.

Acknowledgements

I would like to thank the DEGAS (Design Environments for Global ApplicationS – IST-2001-32072) project funded by the FET Project Initiative on Global Computing, the DIRC (Interdisciplinary Research Collaboration in Dependability – GR/N13999/01) project funded by the UK Engineering and Physical Sciences Research Council, and the Informatics Graduate School for their support.

References

[1] María del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. Debugging UML designs with model checking. *Journal of Object Technology*, 1(2):101–117, July-August 2002.

[2] R. Eshuis and R. Wieringa. A real-time execution semantics for UML activity diagrams. In *Fundamental Approaches to Software Engineering, FASE'01*, volume 2029 of *LNCS*, pages 76–90. Springer, 2001.

[3] Erich Grädel, W. Thomas, and T. Wilke, editors. *Automata, Logics, and Infinite Games: A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.

[4] Alexander Knapp and Stephan Merz. Model checking and code generation for UML state machines and collaborations. In *5th Workshop on Tools for System Design and Verification, FM-TOOLS'02*, Report 2002-11. Institut für Informatik, Universität Augsburg, 2002.

[5] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proceedings of Formal Methods for Open Object-Based Distributed Systems, FMOODS'99*, volume 139 of *IFIP*. Kluwer, 1999.

[6] J. Lilius and I. Paltor. vUML: A tool for verifying UML models. In *Proceedings of Automated Software Engineering, ASE'99*. IEEE, 1999.

[7] *UML 2.0 OCL Final Adopted specification*, October 2003. Available from the OMG at `http://www.omg.org/uml`.

[8] Poseidon for UML, version 2.3.1. Available from Gentleware at `http://www.gentleware.com`.

[9] The precise UML group. Website at `http://www.cs.york.ac.uk/puml/`.

[10] Real Time Studio professional, version 4.3. Available from Artisan Software at `http://www.artisansw.com`.

[11] Rhapsody, version 5.0. Available from I-Logix at `http://www.ilogix.com`.

[12] Mark Richters and Martin Gogolla. Validating UML models and OCL constraints. In *Proceedings of the 3rd International Conference on the Unified Modeling Language, UML'00*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000.

[13] I. Sommerville. *Software Engineering*. Addison Wesley, Seventh edition, 2004. The insulin pump case study is also described in various documents at the book's webpage at `http://www.software-engin.com/`.

[14] P. Stevens and J.Tenzer. Games for UML software design. In *Formal Methods for Components and Objects, FMCO'02*, volume 2852 of *LNCS*. Springer, 2003.

[15] J. Tenzer and P. Stevens. Modelling recursive calls with UML state diagrams. In *Proceedings of Fundamental Approaches to Software Engineering, FASE '03*, volume 2621 of *LNCS*, pages 135–149. Springer, April 2003.

[16] W. Thomas. Infinite games and verification. In *Computer Aided Verification, CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[17] *UML 2.0 Superstructure Final Adopted specification*, August 2003. Available from the OMG at `http://www.uml.org`.

Supporting Confidentiality in UML: A Profile for the Decentralized Label Model

Rogardt Heldal[1], Steffen Schlager[2], and Jakob Bende[1]

[1] Chalmers University of Technology
SE-412 96 Göteborg, Sweden
heldal@cs.chalmers.se
[2] University of Karlsruhe
Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
schlager@ira.uka.de

Abstract We present a way of incorporating a decentralized label model into the UML by defining a profile which is the built-in extension mechanism of the UML. Our profile permits specifying confidentiality of data in UML by offering annotations for classes, attributes, operations, values of objects, and parameters of operations. The profile also supports generation of Jif (Java information flow) code and the Jif compiler guarantees that the specified confidentiality constraints are not violated. Our approach is appealing in the sense that it offers the possibility to consider confidentiality in UML and that the obtained code is guaranteed to behave correctly.

1 Introduction

Our philosophy is that it should be convenient to consider security issues during the system development process, and that security should be automatically verifiable at code level. Addressing both of these aspects is important in order to build a secure software system.

The UML [25] has become the *de facto* standard for the development of object-oriented software systems in industry. There are several reasons for this: it is relatively easy to understand and to learn and it offers several views on a system giving a good overview on its architecture. We aim to make it possible to consider security issues[1] during the industrial development process—so UML is a suited starting point.

One of the main problems with UML is that the focus has been more on functionality than on constraints such as security. In this paper we adapt the UML by defining a profile offering security annotations for a seamless integration of security aspects into a UML-based development process. But how can the

[1] In this work we concentrate on *confidentiality*. "Security" has a lot of other aspects, like e.g. authenticity, data integrity, non-repudiation, access control, or availability which are not considered here.

required rigor for handling security be obtained? Here, language-based checkers play an important role where security information is derived from a program written in a high-level language during the compilation process and is included in the compiled object. This extra security information can take several forms, e.g. a formal proof or type annotations. There have been several overview papers in this area, e.g. [17,27,26].

Our UML profile (addressing the specification of secure systems) is intended to be used together with a language-based checker to validate that the code really satisfies the security constraints. We decided to use the Java information flow (Jif) system [21,22]. Jif handles a large subset of the object-oriented[2] language Java [11] but also contains some additional language constructs, e.g. to control the propagation of confidential data. The Jif type checker guarantees that confidential data can only leak in a controlled manner. What kind of data is allowed to leak should already be part of the specification[3]. A prerequisite for that is, of course, that the specification language supports such constraints. It is important to notice that UML diagrams cannot be validated on the same level as code. The code is needed to consider for example indirect information flow [8] and covert channels [18].

In standard security models, like the Bell-LaPadual model [2] and the Biba model [4], the security policy is separated from the code. In this respect Jif differs since the policy is incorporated into the code in form of *labels*, implementing a *decentralized label model* [23,21]. Data is annotated with labels that specify the ownership and read permissions. The Jif type checker guarantees that the confidentiality policies declared in the labels are not violated.

The decentralized label model gives fine-grained control of data based on decentralized labels. So, it can be guaranteed that a program working with confidential data propagates information only in a controlled manner. Other approaches, like e.g. access control [3], give you all or nothing: they help to prevent information release but do not control information propagation, i.e. do not control how a program distributes confidential data that it is allowed to read. Also not suited for many applications is the sandbox model which e.g. is used for the execution of Java applets that can be downloaded from the internet. It prevents access to data outside the sandbox which is often too restrictive.

Java is not adequate for developing programs which require tight control of confidentiality. That is why we use Jif instead. Similarly, UML is not suited for specifying such programs. That is why we have previously created an extended version of UML called UMLS [12] (UML for Security). UMLS is also based on the decentralized label model. The purpose of that work was to demonstrate that the combination of model-based and language-based security is compelling. However, we did not extend UML in the standard way by defining a profile. This has serious drawbacks: UMLS is not UML-compliant, general UML tools cannot

[2] Object-orientation is important because the UML is tailored to the development of object-oriented systems.

[3] In fact, this information should already be identified during the analysis phase since the customer usually knows exactly which information has to be kept confidential.

be used and the interchangeability of models is harmed. The aim of this paper is to offer support for confidentiality in UML by casting UMLS in a UML profile.

In order to obtain an implementation from the model that satisfies the security constraints, Jif code skeletons can be generated automatically. A type checker then can automatically verify that the (manually) added implementation does not violate the specified confidentiality constraints. In this paper, we concentrate on class diagrams but the profile can be extended to the other diagram types considered in UMLS (interaction diagrams, use cases, and activity diagrams).

Structure of the paper. In Sect. 2 we shortly present the decentralized label model. The built-in extension mechanism of UML is introduced in Sect. 3. Our security profile UMLsProfile is defined in Sect. 4 where we also show some examples and discuss the profile. In Sect. 5 we mention related research before we draw conclusions and point out future work in Sect. 6.

Due to space restrictions we do not give an introduction to Jif. Rather, we introduce Jif bit by bit when needed. For more information on Jif, the reader is referred to [21,22].

2 Decentralized Label Model

The decentralized label model [23] is a security model that improves existing models by allowing users to declassify information in a decentralized way and by supporting fine-grained data sharing. Its main elements and ideas—labels, constraints, and declassification—are shortly explained in the following.

2.1 Labels

The central element of the decentralized label model is the *label*. Labels are used to annotate data in order to guarantee confidentiality—they specify ownership and read permission of data helping to control the propagation of (confidential) data.

Jif is an extension of the Java language [11] implementing the decentralized label model. In this paper we will incorporate the decentralized label model into UML, so UML can be used for deriving Jif code skeletons.

A label consists of a possibly empty set of *policies* where a policy consists of a list of *principals* (e.g. users, groups, or roles). Each policy has a dedicated principal as its *owner*. Each owner controls a set of *readers* that are allowed to read the data. By definition, an owner is implicitly contained in its reader set. A principal is allowed to read data if and only if it is contained in the reader set of all policies of the label attached to the data.

Example 1. The label $\{Bob : Lise\}$ consists of only one policy where the owner is Bob and the readers are Bob (the owner is always a reader) and Lise. The label $\{Bob : Lise, Lars; Lise :\}$ consists of two policies. Only Lise is allowed to read the data. She is the only one contained in the reader set of both policies.

$$Label = \{Components \mid \epsilon\}$$

$$Components = Component \mid Components\textbf{;}\ Component$$

$$Component = \textbf{\textit{principal:}}\ Principals \mid \textbf{\textit{identifier}} \mid \textbf{\textit{*identifier}} \mid \textbf{\textit{this}}$$

$$Principals = \textbf{\textit{principal}} \mid Principals\textbf{,}\ \textbf{\textit{principal}} \mid \epsilon$$

Figure 1. Syntax of Labels in BNF.

In Example 1 labels and principals are *static*. The advantage of static labels is that they can be checked at compile-time. Working only with static labels is however sometimes too restrictive. Thus, there is a need for two new primitive types *label* and *principal* and first-class values of these types represent labels and principals, respectively. We will see examples of how to use these types later.

Fig. 1 shows the syntax of label expressions where ϵ denotes the empty word and symbols in bold represent literals. As can be seen, a label may be empty (meaning that the data is not confidential) or consist of different components which we explain in turn.

A component can be a variable denoted by an identifier. Let us consider the following Jif code:

$$int\{Bob:\}\ x;$$
$$int\{x\}\ y;$$

Here the variable x is owned by Bob. In the label for y we have variable x, meaning that y has the same label as x—in this case $\{Bob:\}$. A component can also be a reference to a label. Let us consider the Jif code:

$$label\{Bob:\}\ lb;$$
$$int\{*lb\}\ y;$$

The label $\{*lb\}$ denotes the label stored in lb rather than the label of lb (which is denoted by $\{lb\}$ and would be $\{Bob:\}$ here). Finally, the reserved label $\{this\}$ represents the label of an object of the class.

Principals can be arranged in hierarchies where a principal can act for another principal ("A can act for B" means that A can do anything that B can do assuming his power). Jif contains an *actsFor* clause which executes a statement only if a certain constraint on the principal hierarchy is satisfied. There is also an *actsFor* constraint on methods which guarantees that certain defined hierarchies hold in the method body. The *actsFor* constraint will be contained in our profile. For more information on principal hierarchies see [21,22].

In Sect. 4 we will need the *join* of two labels, which is the least restrictive label that maintains all restrictions expressed by the two labels. Due to lack of space we omit a formal definition here (it can be found in [23]), we just give the following example.[4]

[4] Intuitively, the joined label is built from the union of the owners and the intersection of their reader sets.

Example 2. The join of labels $\{Bob : Lise\}$ and $\{Bob : Lise, Lars; Lise :\}$ from Example 1 is $\{Bob : Lise; Lise :\}$.

2.2 Declassification

Labeling of data guarantees that information does not leak to users without appropriate authority. Having only labels at hand is however often not sufficient. Sometimes it is necessary to consciously weaken the confidentiality of data, e.g. when an operation processes confidential data but the result should be made less confidential to permit the caller to use it. The problem is solved by giving *authority* (which consists of a list of principals) to classes and operations using the Jif keyword *where* (see example in Fig. 2). An operation must not be given more authority than its owning class[5]. Giving the authority $(p_1, \ldots, p_n)$ to a method means that the method can act on behalf of the principals p_i (even if the caller of the method has lower authority than p_i). This can be used for the *declassification* of data if the owner of the data is one of the principals $p_1, \ldots, p_n$. So, any principal $p_1, \ldots, p_n$ is allowed to relax its own policy (e.g. add readers) without weakening policies of other principals. E.g. the Jif statement declassify(e,L) relabels the result of expression e with label L.

```
class PasswordFile {
  private String[] nameList;
  private String{root:}[] passwordList;
  public boolean check(String user, String password)
    where authority(root){
    boolean match=false;
    try {
      for (int i=0; i<nameList.length; i++) {
        if (nameList[i].equals(user) &&
            passwordList[i].equals(password)) {
            match=true; break;
        }
      }
    } catch (NullPointerException e) {}
      catch (IndexOutOfBoundsException e) {}
    return declassify(match, {user; password});
  }
}
```

Figure 2. Jif Implementation of Class PasswordFile.

Now, we will consider an example (taken from [22]) where declassification is needed. Fig. 2 shows a class *PasswordFile* having an operation *check* which

[5] The Jif checker verifies that this property of the *least privilege* is obeyed.

takes a login name (*user:String*) and a password (*password:String*) and returns a boolean depending on whether *user* and *password* is contained in the arrays *nameList* and *passwordList*, respectively. On the one hand the operation should return a boolean value (i.e. leaking the information whether the password was correct), but on the other hand one does not want to leak the whole content of the array *passwordList*. To ensure this the elements of the array are labeled with {*root:*}. Certainly, one does not want to give the authority *root* to a normal user. So, the return value of operation *check* has to be *declassified* not to contain root. This means that principal *root* is removed from the owners of the return value (for more information on declassification see [22]). Thus, we have permitted to leak some information about the array *passwordList*. Declassification should be used with care. E.g. the above method can in fact leak all information in *passwordList* if the user is given the opportunity to call it repeatedly. We will come back to this issue later.

3 UML Extension Mechanism

The UML is a general purpose specification language. It can be adapted to particular domains by defining a *profile*. A profile is a conservative extension in the sense that it is not allowed to modify the metamodel. The application of a profile always results in a model that is still compliant with the metamodel. Thus, problems concerning semantics and interchangeability between tools are avoided.

The means for defining a profile are *stereotypes*, *tag definitions*, and *constraints*. A stereotype is used for extending metaclasses (defined in the metamodel) or other stereotypes. Like classes, a stereotype may have properties (in that context called *tag*). When a stereotype is applied to a model element, the values of its defined tag may be referred to as tagged values.

Finally, constraints can be used to define or refine the semantics of model elements. Constraints can be stated informally (e.g. using natural language) or formally using an adequate language (e.g. using the Object Constraint Language [24] which is an integral part of the UML).

4 Profile for Decentralized Label Model

In this section we define our profile which we call UMLsProfile (profile for security in UML). Like our previous UML extension UMLS [12], it is built on the decentralized label model. It permits confidentiality aspects to be considered in class diagrams.

4.1 Stereotypes

The aim of our profile is to provide stereotypes for annotating classes, attributes, operations, parameters, and return types of operations with confidentiality labels

and constraints. Tab. 1 shows the metaclasses that are extended (first column) by stereotypes (second column). The tags of the stereotypes are defined in the third column. The meaning of the stereotypes is explained in the following.

Metaclass, **Stereotype**	**Stereotypes**	**Tags**
TypedElement	*confidential*	*l:label*
Class	*authorityConstraint*	*authority:principal[*]*
Operation	*authorityConstraint*	*authority:principal[*]*
	actsForConstraint	*actsFor:(principal,principal)[*]*
	callerConstraint	*caller:principal[*]*
	beginLabel	*l:label*
	endLabel	*l:label*
<< send >>	*sendConfidential*	*l:label*

Table 1. Metaclasses extended by Stereotypes.

By using stereotype *confidential*, labels can be attached to instances of *TypedElement* which are attributes, formal parameters and return type of operations, and the values of objects.

As described in Sect. 2.2 classes might be given authority to permit declassification. In Tab. 1 we can see that this is achieved by extending the metaclass *Class* with stereotype *authorityConstraint*.

In addition to *authorityConstraint*, the profile (and Jif) offer the *callerConstraint* and *actsForConstraint* which extend metaclass *Operation*. The *callerConstraint* allows a caller to dynamically grant authority to the invoked operation. An operation with a *callerConstraint* may be called only if the caller possesses the required static authority.

In a hierarchy of principals, some principals can act for some other principals. This can be specified using the stereotype *actsForConstraint* which can be attached to operations. Then the operation can only be invoked if the specified constraint holds at the call site.

To prevent information leaks through implicit flows, the compiler associates a program-counter label with every statement and expression, representing the information that might be learned by their evaluation. A *beginLabel* can be specified to restrict the program-counter label at the point of invocation of a method—preventing a method from causing side-effects that have lower security than the value of *beginLabel*. Stereotype *endLabel* specifies what information can be learned from the fact that the method terminates normally. We will see an example on how to use them later.

It is also possible to give labels to individual exceptions which an operation might throw. In UML there already exists a stereotype *send* which can be applied to dependencies whose source is an operation and whose target is a signal,

specifying that the source sends the target signal. In Tab. 1, stereotype *sendConfidential* extends stereotype *send*. This permits us to attach labels to exceptions that are potentially sent by an operation.

The whole profile UMLsProfile is depicted in Fig. 3. The figure also shows the tags (and their types) of the stereotypes.

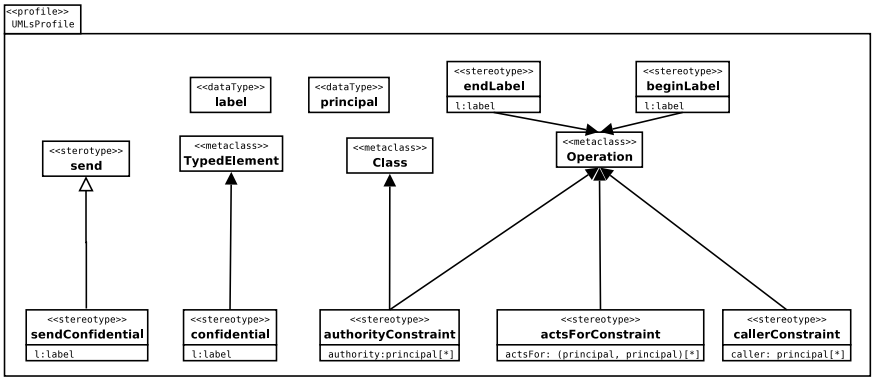


Figure 3. The Profile UMLsProfile.

The tags defined for the stereotypes above are of type *label*, *principal*, or are arrays of the respective type. The syntax of values of these types is defined in Fig. 1.

4.2 Examples and Default Values

In this section we first consider an example which involves labels and explain which default labels apply if no label is given. Thereafter, we consider declassification and shortly discuss the profile.

Labels. Fig. 4 shows an example of UMLsProfile applied to a class *Account*. In a UML diagram, the tagged values of a stereotype are denoted by UML notes attached to the element that is adorned with the stereotype. Consider for example attribute x annotated with stereotype *confidential*. A UML note is attached to attribute x defining its label $\{Bob:\}$. Attribute y is annotated with a label containing variable x. This means that the value of the label will be the same as for attribute x—in this case $\{Bob:\}$. Attribute z has no label. For attributes the default label is the empty label meaning that the attribute does not contain confidential data.

Next, let us consider the annotation of operation parameters. The formal parameter of operation *set* has the static label $\{Bob: Lise\}$. According to Jif,

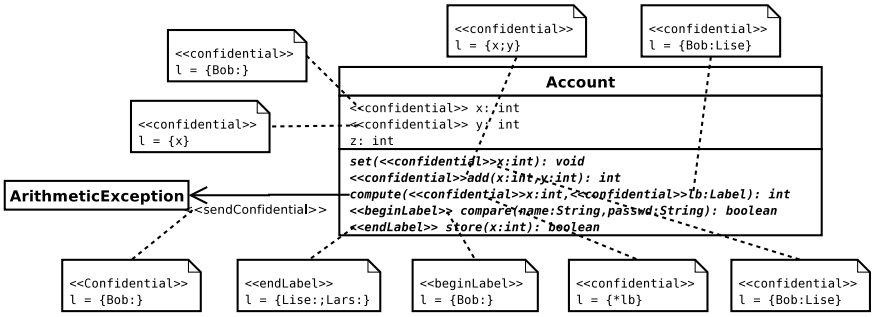


Figure 4. Example of Applying the Profile to Attributes and Operations.

this means that the *set* operation can only be invoked with arguments having this label. It would be quite tedious if one had to define a specific operation for every static label. The solution to that problem is *label polymorphism*. For example, in Fig. 4 the parameters x and y of operation *add* have no labels. This means that *add* can be called with any labels for its arguments—x and y will get the same labels as the arguments.

The first parameter of method *compute* shows how one can access the label contained *in* a variable of type *label*. The syntax is similar to the dereference operator * in the programming language C. Thus, the label $\{*lb\}$ denotes the label contained in *lb* rather than the label of *lb* (which is denoted by $\{lb\}$). When *compute* is invoked with the label value $\{Bob:\}$ for parameter *lb*, parameter x will also have the label $\{Bob:\}$. The second parameter *lb* of method *compute* is of type *label*. *lb* is a dynamic label and it would be legal to annotate it with a static label (e.g. $\{Bob : Lise\}$) since *lb* is a normal operation parameter.

Operation *compare* is an example for the use of a *begin-label*. A begin-label attached to an operation prevents the operation to be be called from a context with lower security. The default value for this kind of label is the program-counter label of the caller. In the example, the begin-label is $\{Bob:\}$. It forbids any assignment to attribute z in the body of *compare* since the empty label of z has lower security then the begin-label $\{Bob:\}$.

In the operation *add* we have attached the label $\{x; y\}$ to the return type. It was not necessary to explicitly state the return label in this case since the default label of the return value is the join of the parameter labels and the end-label, which in our case is $\{x; y\}$.

If an operation throws an exception, by default the exception has the same label as the end-label. However, it can also be given a different label as operation *compute* shows. Here, the label $\{Bob:\}$ is attached to the exception.

Declassification. In Sect. 2.2 we considered the class *PasswordFile* with the operation *check* that takes a login name (*user:String*) and a password (*pass-*

word:String) and returns a boolean. We argued that the class *PasswordFile* and the operation *check* needed the authority *root* to be able to declassify the boolean return value. Fig. 5 shows the class annotated with the authority constraint. Given this specification the code skeleton in Appendix A is generated automatically, where the type $Array < String, \{root:\} >$ of *passwordList* is translated into an array of type *String* whose elements are labeled with {*root:*}.

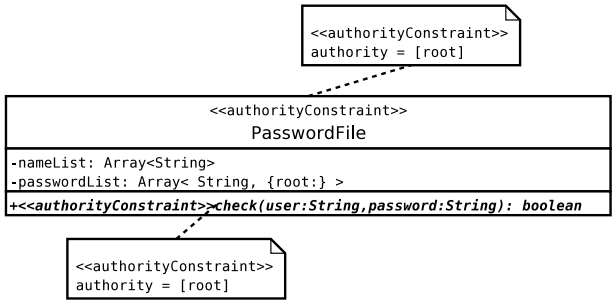


Figure 5. Example for a Class containing an *authorityConstraint*.

4.3 Notation

In a UML model the tagged values of a stereotype are denoted by UML notes attached to the element that is adorned with the stereotype. The example in Fig. 4 clearly shows that the default notation for stereotypes and tagged values clutters up the UML model and makes it hard to read. Fortunately, UML allows a profile to define its own notation that can be used instead of the standard notation of the model element which the stereotype is applied to [25, Sect. 18.3.7][6].

Our notation for the extended metaclasses follows the one in UMLS [12] and is depicted in Tab. 2. For the notation of values of *labels* and *principals* we use the syntax defined in Fig. 1.

Fig. 6 shows the example from Fig. 4 using the new notation. The advantages of our notation compared to the default notation are obvious.

Since we follow the UML standard, it is possible to use our profile with any UML-compliant tool. If a tool allows for adapting the concrete syntax, it is possible to use our more convenient notation. To guarantee interchangeability, it is however important that the XMI representation of the diagrams is independent of the chosen notation.

[6] Note, that we follow the "UML 2.0 Final Adopted Specification" which has not been finalized yet.

Metaclass	Notation
TypedElement	*element* ***label***
Class	Additional compartment for *authority* (see Fig. 5)
Operation	*visibility name* ***begin-label*** (*parameter-list*) ***end-label***
	:return-type-expression ***return-label*** ***constraints***

Table 2. Notation for Metaclasses.

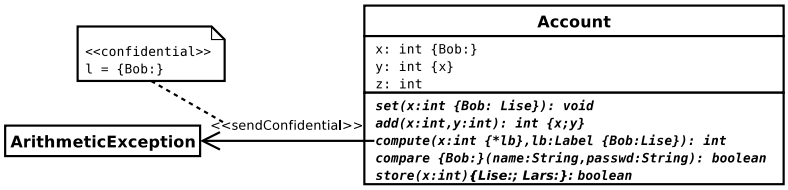


Figure 6. Application of UMLsProfile Notation to the Example shown in Fig. 4.

Parameterized Classes. The language Jif offers parameterized classes with respect to labels and principals. The UML 2.0 (final adopted specification) supports parameterized classes via the package *Template* as well. We give a little example here since parameterized classes play an important role in making reusable data structures with respect to labels and principals.

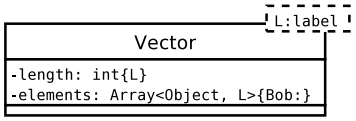


Figure 7. Vector parameterized on Label L.

Figure 7 shows a class *Vector* parameterized on a label L (in the dashed box). This label can be used to annotate attributes and operations of the class and makes it possible to instantiate *Vector* with different labels. In the example, attribute *length* is annotated with the parameter label L. Attribute *element* is of an array type which can have two labels: one for the array elements of type *Object* (here $\{L\}$) and one for the array reference (here $\{Bob:\}$).

4.4 Discussion

The decision on what data is confidential is usually not (and should not be) a programmer's job. Rather the customer/domain expert has to know which data

is confidential. Our profile permits this decision to be moved from code to the analysis/design level. We believe that this is crucial for building systems working with confidential data.

In some cases declassification is needed for being able to intentionally leak information (as the example from Fig. 2 and Fig. 5 showed). Note, that declassification is extremely powerful but therefore also very dangerous. Again, the question when to use declassification is an issue to be addressed on the analysis/design level. Also giving authority to classes or methods should be used with care. The places with authorities are spots where extra care has to be taken.

Finally, a short note on current UML tools. The idea behind producing a profile is that it can be applied by any UML-compliant tool. However, we encountered that most tools are not fully UML-compliant. For example, they do not allow to attach notes to arbitrary model elements (which is permitted according to the UML specification).

5 Related Work

Considering security in UML is a relatively new idea. Blobel, Pharow, and Roger-France [5] used use cases to consider security in a very informal way in a medical setting. We find it difficult to say anything about use cases since their semantics is not very well understood [10]. Furthermore, there has been work on developing a framework for model-based risk assessment of security-critical system using UML [13].

In our previous work [12] which was mainly focused on a case study we have already considered the treatment of confidential data in UML. However, we extended UML in a non-standard way. Furthermore, some features were left out, for example the treatment of *caller* and *actsFor* constraints. In this paper we give a more complete treatment of the decentralized label model in a UML-compliant manner.

The connection between language-based security and security on the specification level has been previously established by Mantel and Sabelfeld [20]. Their approach is more theoretical than ours. We hope that by choosing a more practical approach we will be able to reach more designers.

Closely related to our research is Jürjens' work on modeling confidentiality in UML [15,14,16]. Jürjens also uses the built-in extension of UML to define a profile called UMLsec. For checking constraints associated with the stereotypes of his profile, Jürjens defines a precise semantics for a restricted and simplified fragment of the UML building on a state chart semantics based on abstract state machines [6].

This approach has some limitations. The developer has to convince himself that the system is correct by examining the—possibly quite complex—UML diagrams. Furthermore, it is uncertain that the code created from these diagrams is correct. However, both problems can be eased by providing tool support.

A more serious problem are covert channels which arise from the concrete implementation of a program. For example, control flow or information about

termination of a program may reveal confidential information. So, even if confidentiality properties are proven on the UML level, which might be quite difficult in itself, there might be covert channels in the code. Our approach combining UML and Jif addresses this problem of indirect information flow [8]. The main difference between the two approaches is that Jürjens verifies security properties on specification level while we are working on the code level using the Jif type checker. So, both techniques should be complementary to each other. Furthermore, the decentralized label model permits more fine-grained control of confidential data than Jürjens' approach.

There has been some work that considers role-based access control in a UML setting [9,19]. Even though we have focused on information flow, there are some interesting parallels to this research. UMLsProfile/Jif permits declassification of data which can perhaps be considered as a form of access control.

6 Conclusion and Future Work

In this paper we have presented a profile UMLsProfile incorporating the decentralized label model into the UML. The profile allows fine-grained control of confidential data.

The decisions on which data must be kept confidential should be made at an early stage. This work permits confidentiality to be considered in the design phase of the development process. Using our profile in combination with Jif therefore contributes to building secure software systems. Jif code skeletons can be automatically generated from a class diagram making use of UMLsProfile.

In this paper we focused on class diagrams, probably the most important diagram type with a clear semantics that allows for code generation in a straightforward way. The next step in this line of work is to extend the profile UMLsProfile the other diagram types considered in UMLS. How Jif code can be generated from other diagram types needs further investigation.

There is one area we have not addressed in this paper, but which is important for our work: secure environments. Here, the deployment diagram in UML might be very useful when specifying secure environments for Jif programs. Furthermore, it would be interesting to look at state charts as well because they can be used to generate additional code which considers confidentiality. In particular, Jürjens' work [15] might be useful to take into account here.

Finally, Jif supports dynamic labels, but only in a restricted way to make sure that static checking of confidentiality is still possible. The restriction is that variables of type *label* may only be used to construct labels if they are *immutable*. We believe that this restriction could be dropped if in addition to type checking a theorem prover is used. The KeY tool [1] seems to be suited for that task for several reasons. KeY is a tool that supports the specification and verification of object-oriented software. It supports the specification languages UML/OCL and the implementation language Java. Thus, KeY seems to fit very well to UMLsProfile and Jif. Last but not least, KeY has already been successfully used to analyze secure information flow [7].

Acknowledgment. We thank R. Bubel, A. Roth, A. Sabelfeld, and the anonymous referees for important feedback on drafts of the paper.

References

1. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. *Software and System Modeling*, pages 1–42, 2004. To appear.
2. D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report MTR 2547 v2, The MITRE Corporation, Nov 1973.
3. E. Bertino, S. Jajodia, and P. Samarati. Access Controls in Object-Oriented Database Systems: Some Approaches and Issues. In N. Adam and B. Bhargava, editors, *Advanced Database Concepts and Research Issues*, LNCS 759, pages 17–44. Springer, 1993.
4. K. J. Biba. Integrity Consideration for Secure Computer System. Technical Report ESDTR-76-372,MTR-3153, The MITRE Corporation, Bedford,MA, April 1977.
5. B. Blobel, P. Pharow, and F. Roger-France. Security Analysis and Design Based on a General Conceptual Security Model and UML. In P. M. A. Sloot, M. Bubak, A. G. Hoekstra, and B. Hertzberger, editors, *High-Performance Computing and Networking, 7th International Conference, HPCN Europe 1999, Amsterdam*, volume 1593 of *LNCS*, pages 918–930. Springer, April 12-14 1999.
6. E. Börger, A. Cavarra, and E. Riccobene. Modeling the Meaning of Transitions from and to Concurrent States in UML State Machines. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1086–1091. ACM Press, 2003.
7. A. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In R. Gorrieri, editor, *Workshop on Issues in the Theory of Security (WITS)*. IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS, 2003.
8. D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Communications of the ACM*, 20(7):504–513, July 1977.
9. P. Epstein and R. Sandhu. Towards A UML Based Approach to Role Engineering. In *RBAC '99, Proceedings of the Fourth ACM Workshop on Role-Based Access Control*, pages 135–143, October 28-29 1999.
10. G. Génova, J. Llorens, and V. Quintana. Digging into Use Case Relationships. In J. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*, pages 115–127. Springer, September/October 2002.
11. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
12. R. Heldal and F. Hultin. UMLS Bridging Model-based and Language-based Security. In E. Snekkenes and D. Gollmann, editors, *Computer Security - ESORICS 2003*, volume 2808 of *LNCS*, pages 235–252. Springer, 2003.
13. S. H. Houmb, F. Braber, M. S. Lund, and K. Stolen. Towards a UML Profile for Model-Based Risk Assessment. In *UML 2002 Satellite Workshop on Critical Systems Development with UML*, pages 79–91, September 2002.
14. J. Jürjens. Secure Java Development with UMLsec. In B. D. Decker, F. Piessens, J. Smits, and E. V. Herrenweghen, editors, *Advances in Network and Distributed Systems Security*, pages 107–124, Leuven, November 26-27 2001. International Federation for Information Processing (IFIP) TC-11 WG 11.4. Proceedings of the First Annual Working Conference on Network Security (I-NetSec '01).

15. J. Jürjens. Towards Development of Secure Systems using UMLsec. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering (FASE, 4th International Conference, Part of ETAPS)*, volume 2029, pages 187–200, 2001.
16. J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002*, volume 2460 of *LNCS*, pages 412–425, Dresden, Sept. 30 – Oct. 4 2002. sv. 5th International Conference.
17. D. Kozen. Language-Based Security. In *Mathematical Foundations of Computer Science*, pages 284–298, 1999.
18. B. W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.
19. T. Lodderstedt, D. Basin, and J. Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In J.-M. Jezequel, H. Hussmann, and S. Cook, editors, *The unified modeling language: model engineering, concepts, and tools; 5th international*, volume 2460, pages 426–441. Springer, 2002.
20. H. Mantel and A. Sabelfeld. A Generic Approach to the Security of Multi-Threaded Programs. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, pages 126–142, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society Press.
21. A. Myers. Mostly-Static Decentralized Information Flow Control. Technical Report MIT/LCS/TR-783, MIT, 1999.
22. A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Symposium on Principles of Programming Languages*, pages 228–241, 1999.
23. A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997.
24. OMG. UML 2.0 OCL Specification. OMG Document, October 2003.
25. OMG. Unified Modeling Language: Superstructure, version 2.0, Final Adopted Specification. OMG Document, August 2003.
26. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.
27. F. B. Schneider, G. Morrisett, and R. Harper. A Language-Based Approach to Security. In R. Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead. Conference on the Occasion of Dagstuhl's 10th Anniversary*, volume 2000 of *LNCS*, pages 86–101, Saarbrücken, Germany, August 2000. Springer.

A Jif Code Skeleton for PasswordFile

The following Jif code skeleton can be automatically generated from the UML class diagram depicted in Fig. 5.

```
class PasswordFile {
  private String[] names;
  private String{root:}[] passwords;

  public boolean check(String user, String password) {}
}
```

Using Aspects to Manage Security Risks in Risk-Driven Development

Siv Hilde Houmb, Geri Georg, Robert France, and Dan Matheson
Department of Computer Science
Colorado State University
Fort Collins, CO 80523, Colorado, US
sivhoumb@idi.ntnu.no, georg@CS.ColoState.EDU, france@CS.ColoState.EDU,
and dan.matheson@comcast.net

Abstract. The EU IST-project CORAS has developed an integrated risk management and system development process for security-critical systems based on AS/NZS 4360, RUP, and RM–ODP. The approach presented in this paper is based on the concepts of risk-driven development and extends the CORAS framework by using aspects to specify security risk treatment options. This enhances the evaluation of the treatment options since aspects models are decoupled from the primary model. The result is an aspect-oriented risk-driven development approach, in which security requirements or security risks may be identified in each phase of the development. The treatments that addresses these requirements or security risks are specified and implemented as aspects. Using aspects makes it easier to develop and evaluate security treatments options and to evolve the treatments.

Keywords: Risk-driven development (RDD), Aspect-oriented modeling (AOM), Security risk assessment, and Trade-off analysis

1 Introduction

Developers of security-critical systems must cost-effectively identify and address security risks throughout the development. Along with implementing functional requirements the development of such systems involves specifying and implementing security solutions or treatments that address the identified risks. The pervasive nature of security concerns often results in treatments that are spread across and tangled with elements in design modules. The crosscutting nature of these security risk treatments poses a significant challenge when considering alternative treatment options or whenever one have to modify or evolve the treatments.

Aspect-oriented modeling (AOM) techniques allow system developers to describe solutions that crosscut a design in separate design views called aspects [7]. An aspect is a pattern that characterizes a family of concern realizations and an aspect model consist of a set of UML diagrams, both structural and behavioral, specifying the internal structure and the behavior of the aspect. An AOM design

model consist of one or more aspects models and a primary model, which is specified using a set of UML diagrams. The aspect models describe solutions that crosscut the modular structure of the primary model. Using the AOM approach, developers of security-critical systems can separate the treatment of security risks from the treatment of other concerns: security treatments are described by aspects, while other design concerns are described by the primary model. This separation eases the task of evolving the security treatments and makes it easier to swap in and out and evaluate alternative treatment options.

This paper described an risk-driven AOM-based approach to manage security risks called the aspect-oriented risk-driven development (AORDD) approach. The approach is a result of integrating AOM techniques into the iterative system development and risk management process of CORAS [1]. Security risk assessment is performed as part of each iteration in the AORDD approach and involves identifying potential mis-uses that pose security risks, and evaluating the mis-uses using security risk acceptance criteria to determine which mis-uses need to be treated. The result of a security risk assessment are aspects that describe security risk treatments.

The reminder of this paper is structured as following. Section 2 gives a brief overview of related work. In Section 3 we describe the AORDD approach. Section 4 outlines how the approach, and in particular the use of aspects, can be used to support cost-benefit trade-off analysis evaluating alternative treatment options, while Section 5 gives an small example to illustrate how to use the AORDD approach. Section 6 concludes the paper and outlines further work.

2 Related Work

Risk-driven development combines development and risk management to effectively handle security issues throughout the development. The main goal is to cost-effectively achieve a correct level of security. There are currently few approaches targeting risk-driven development. However, there are a number of specialized risk assessment methodologies targeting the security domain. Within the domain of health care information systems the Central Computer and Telecommunication Agency (CCTA) in Britain has developed CRAMM, CCTA Risk Analysis and Management Methodology[3]. CRAMM offers a structured approach to manage computer-based systems. CRAMM is asset-driven, meaning that the focus is on identifying the main assets connected to the system as well as identify and assess risks to those assets.

The CORAS framework [17] is inspired by CRAMM and has adapted the asset-driven strategy. CORAS uses UML as their modeling language in the context of model-based risk assessment (MBRA). The objective of the approach is to provide methods and tools for precise and efficient risk assessment of security-critical systems using semi-formal models. It is a risk-driven approach where models are used to describe the target of assessment, as a medium for communication between different groups of stakeholders involved in a risk assessment, and to document risk assessment results and the assumptions on which these

results depend. CORAS also supports the evaluation of alternative treatment options, but does not provide any guideline as such. The risk-driven approach described in this paper makes use of aspects and the concept of aspect-oriented modeling (AOM) to address this issue.

A growing number of researchers are developing approaches that supports multidimensional separation of concerns throughout development. Clarke et al. [4] describe an approach where requirements are addressed as themes before composed to obtain a comprehensive design. Rashid et al. [21] build on this work to produce an aspect-oriented approach to requirement analysis. Their work supports modularization of crosscutting properties at the requirements level to support early trade-off analysis. In the AOM approach developed by the AOM team at Colorado State University (CSU) [7, 10, 9, 8] aspect models are used to describe crosscutting solutions that address dependability concerns such as security.

The work described in this paper makes use of the asset-oriented approach of CRAMM and is based on the integrated system development and risk management process of CORAS [26]. The CORAS approach is integrated with the AOM approach developed at CSU to provide support for specifying and implementing security risk treatments as aspects and thereby support the process of evaluating alternative treatments as input to a cost-benefit trade-off analysis.

3 Aspect-Oriented Risk-Driven Development (AORDD)

The iterative system development and risk management process developed by CORAS [6] is structured according to the phases of the RUP (Rational Unified Process) [25]. In each iteration, one can design, analyze, and compose a part of the system or the system as a whole according to a particular RM-ODP (Reference Model for Open Distributed Processing) viewpoint [11]. Risk assessment is an integrated activity of each iteration and security are thereby addressed as early as possible.

Table 1 depicts the activities of the CORAS risk management process. Sub-process 2 addresses the security risk identification, while sub-process 5 addresses the allocation and evaluation of security risk treatments. Figure 1 illustrate the relationship between the concepts involved in security risk identification, evaluation, and treatment. This represent the security risk assessment ontology and are used when assessing the effect of alternative treatment options.

The AORDD process extends the CORAS process by providing techniques and notations for describing security risk treatments as aspects, and for composing the aspects with a primary design model. Modeling the security treatments as aspects eases the task of developing and evaluating alternative treatment options in sub-process 5, and enhance software evolution and reusability.

Figure 2 illustrates the iterative nature of the AORDD process, consisting of a requirement phase, a design phase, implementation phase, deployment phase, and a maintenance phase. Development spirals through requirements to maintenance, and in each phase development activities are structured into iterations.

Sub-process 1: Identify Context – Activity 1.1: Identify areas of relevance – Activity 1.2: Identify and value assets – Activity 1.3: Identify policies and evaluation criteria – Activity 1.4: Approval **Sub-process 2: Identify Risks** – Activity 2.1: Identify threats to assets – Activity 2.2: Identify vulnerabilities of assets – Activity 2.3: Document unwanted incidents	**Sub-process 3: Analyze Risks** – Activity 3.1: Consequence evaluation – Activity 3.2: Frequency evaluation **Sub-process 4: Risk Evaluation** – Activity 4.1: Determine level of risk – Activity 4.2: Prioritize risks – Activity 4.3: Categorize risks – Activity 4.4: Determine interrelationships among risk themes – Activity 4.5: Prioritize the resulting risk themes and risks **Sub-process 5: Risk Treatment** – Activity 5.1: Identify treatment options – Activity 5.2: Assess alternative treatment approaches

Table 1. Sub-processes and activities of the CORAS risk management process [23]

Work moves from one phase to the next after first iterating through various sub-phases that end with acceptable analysis results. In the following we will only focus on the requirement and design phase of the AORDD process. Details on the main steps in these two phases is provided in Figure 3. The activities of the security risk management step of each phase is as follows.

- Sub-process 1: Context identification
  - Activity 1.1: Identify purpose, scope, the target of assessment, business perspectives, and the system environment
  - Activity 1.2: Identify and value assets, identify stakeholders, and create the asset-stakeholder graph
  - Activity 1.3: Identify policies and security risk acceptance criteria
- Sub-process 2: Risk identification
  - Activity 2.1: Identify security threats to assets
  - Activity 2.2: Identify vulnerabilities in ToA, business processes, and the system environment
  - Activity 2.3: Document mis-use scenarios
- Sub-process 3: Risk analysis
  - Activity 3.1: Impact estimation
  - Activity 3.2: Frequency estimation
- Sub-process 4: Risk evaluation
  - Activity 4.1: Determine level of risk
  - Activity 4.2: Categorize risk in risk themes
  - Activity 4.3: Determine interrelationships among risk themes

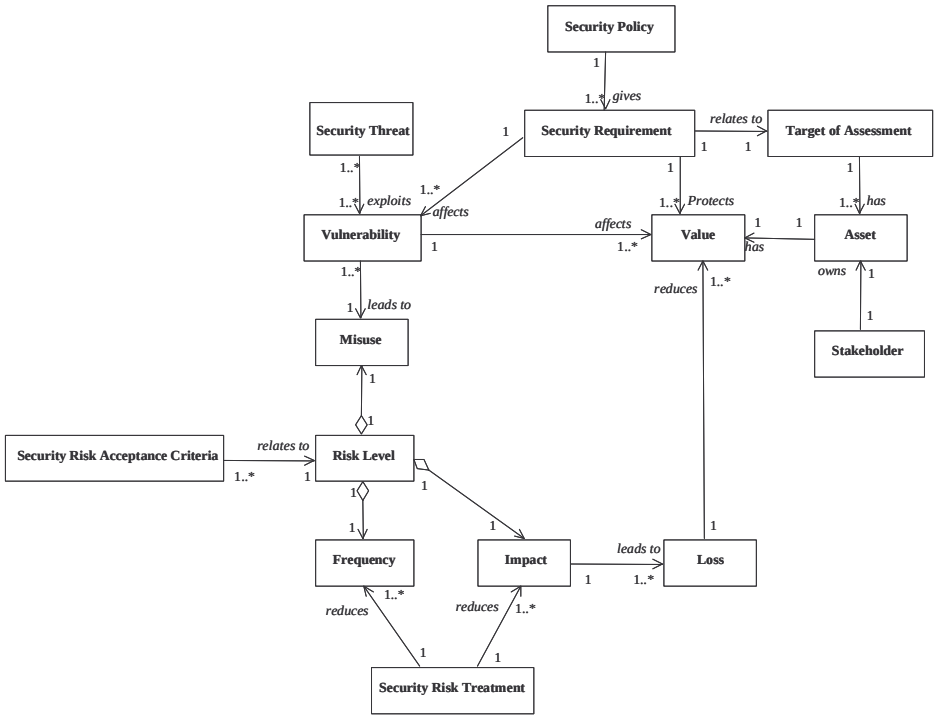


Fig. 1. Concepts and their relationship in security risk assessment

- Activity 4.4: Identify conflicts
- Activity 4.5: Prioritize risk themes and risks
- Activity 4.6: Solve conflicts

– Sub-process 5: Risk treatment
- Activity 5.1: Identify risk treatment option
- Activity 5.2: Identify effect and cost of treatment options
- Activity 5.3: Create aspect models for each treatment option
- Activity 5.4: Assess alternative treatment options using the aspect models as input to the cost-benefit trade-off analysis

The main difference from the activities of the CORAS risk management process is the refinement of the activities in sub-process 1; the context description, the use of the concept mis-use rather than unwanted incident, and the activities of sub-process 5; risk treatment, where we use aspects and cost-benefit trade-off analysis. Each iteration in a phase starts with specifying the system according to a RM-ODP viewpoint at different levels of abstraction. The process moves to security risk assessment as soon as sufficient amount of information of the system is gathered (depending on the phase, viewpoint, and the level of abstraction).

In security risk assessment one perform two types of security risk identification; vulnerability and security threat identification (for more information see AS/NZS 4360 [2] and the Common Criteria [5]). A mis-use scenario is defined as the composition of a vulnerability and a security threat, meaning that we have a mis-use if there exist a vulnerability and a security threat such that the security threat might exploit the vulnerability as illustrated in Figure 4. The result from a security risk assessment are a set of security treatment aspects. Security risk treatment aspects are solutions to the mis-use scenarios. As part of the security risk assessment we perform a cost-benefit analysis (sub process 5; risk treatment) where we evaluate alternative treatment options. This is done by assessing the effect and cost of each treatment option. The use of aspects in the trade-off analysis makes it easy to swap in and out treatment options to evaluate their effect. More information on the trade-off analysis is given in the next section.

In order to identify design stress points and design flaws we need to compose the aspect and primary models. A design flaw is undesirable emergent behavior arising as a result of integrating aspects with primary model, while a design stress point is a non-robust part of a design that can be exploited to produce unauthorized behavior (which gives input to the vulnerability identification in security risk assessment). The main difference between the analyze and security risk assessment activity is that in risk assessment one looks at how the system environment may influence the system, while the analyze activity looks at the system and flaws within the system. Composed aspect and primary models are analyzed to determine their validity. The methods used in the analyze activity depends on the size and the criticality of the system and might involve model checking, different types of testing, etc. utilizing formally defined operational semantics for UML models that supports rigorous static and dynamic analysis. The main purpose of the analyze activity is to ensure that the security risk treatments are consistent with the system quality objectives.

In the requirements specification phase, security risk assessment is done using the functional and security requirements as the target of assessment. The result of security risk assessment is a set of unresolved security issues. These issues are either treated in sub-process 5, or transformed into security requirements for the next iteration. In the design phase the security risk assessment is done using the available design specification as the target of assessment. The result is either treated in sub-process 5 of the assessment by a design decision, or transformed into security requirements. This means that whenever new security requirements are introduced, the relevant parts of the system go through a new requirement security risk assessment and analysis (refers to the analyze activity in Figure 2) before proceeding.

Security risk assessment in all phases is done by identifying critical assets and assigning asset values before doing a security risk identification (sub-process 2). In security risk identification threats, vulnerabilities, and mis-use scenarios are identified as described earlier. The result from the risk identification is then evaluated in the risk analysis sub-process, where impact and frequency values are

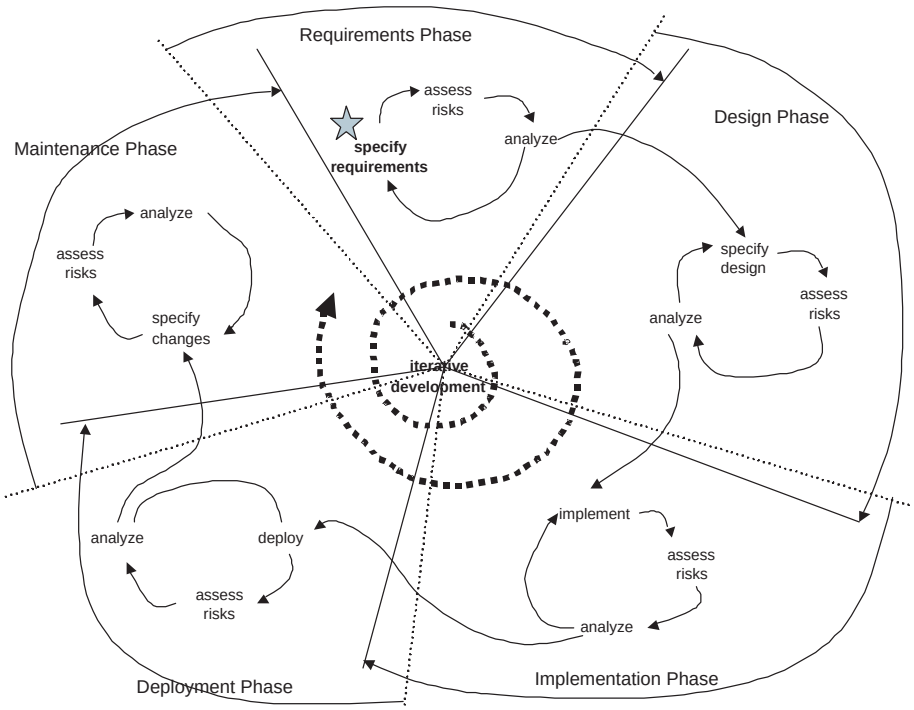


Fig. 2. Outline of the aspect-oriented risk-driven development process

estimated. In sub-process 4; risk estimation, the security risk level is evaluated against a set of security risk acceptance criteria. Describing the security risk acceptance criteria is part of activity 1.3, but are often also provided at later stages in the assessment. The security risk level is evaluated against these criteria, where security risk level is composed by one mis-use, one frequency, and one impact. An impact describes loss or gain (opportunities) of value for one or more assets. A security risk treatment addresses the security risk level and can either reduce frequency, impact or both, or transfer the impact to a third party.

4 Trade-off analysis

Security risk treatment is a trade-off between minimizing risk and optimizing benefit [20]. This is supported through the cost-benefit trade-off analysis in activity 5.4; assess alternative treatment options using the aspect models as input to the cost-benefit trade-off analysis. Figure 5 illustrate the relationship between impact loss and impact gain. The up-arrows represent opportunities and down-arrows represent security risks. The main idea is to balance the security risks

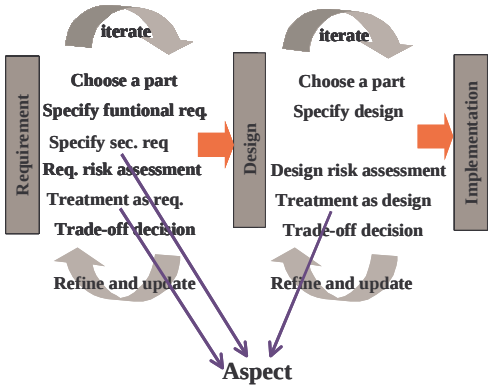


Fig. 3. Details for the requirement specification and design phase of the aspect-oriented risk-driven development process

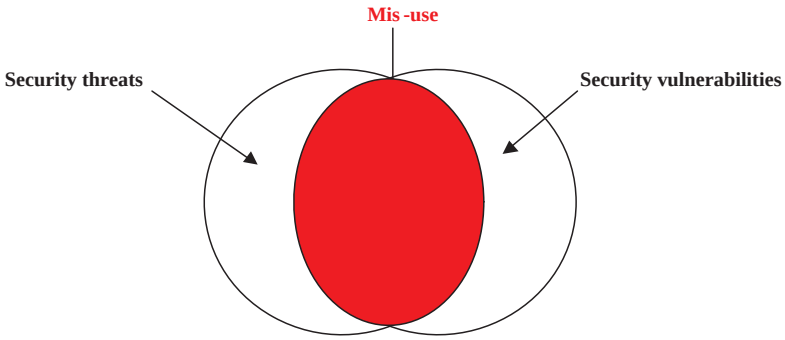


Fig. 4. Relation between threat, vulnerability, and mis-use

and the opportunities. The trade-off analysis consist of two main phases; (1) evaluate security risks, and (2) trade-off according to prioritizes. The security acceptance criteria must be evaluated prior to the trade-off analysis, since the security acceptance criteria has the highest priority and must be meet. The result from step 1 is a list of security risks in need of treatment. The prioritizing of these security risks is done during the security risk assessment and is also used as input to the evaluate security risk step.

Loss of asset value is the main measure used in security risk assessment. However, when one assesses the trade-off between different properties of a system it is also necessary to look into potential gains. When evaluating which risks to treat and which security treatments to implement, one needs to assess the security risks against the prioritizing algorithm before doing the trade-off analysis

and looking into maximizing the effect of the resources spent. In addition to making sure that the security risk acceptance criteria is meet, one also need to consider relevant and mandatory standards, policies, and laws before performing economic maximization. We will not describe or illustrate the trade-off analysis in this paper, but rather focus on giving a small example applying the AORDD approach.

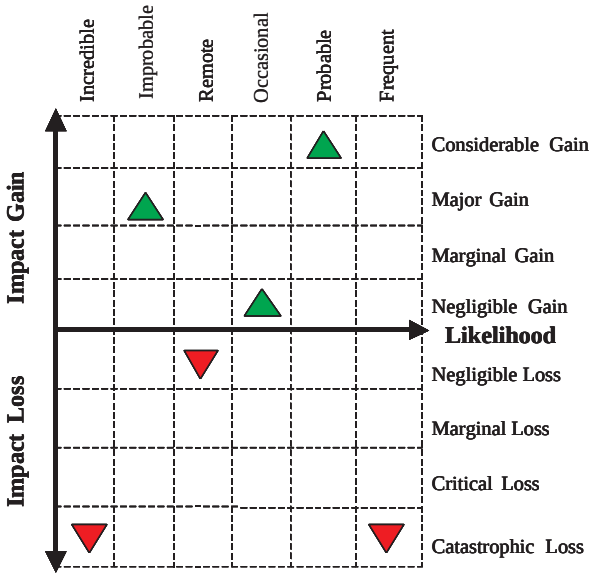


Fig. 5. The relationship between frequency and impact loss and impact gain

5 Developing e-Commerce system using AORDD

An aspect-oriented design model consists of a set of aspects and a primary model. An aspect model describes how a single objective is addressed in a design, and the primary model addresses core functional concerns. The aspects and the primary model are composed before security risk assessment and analysis is performed, as well as before implementation or code generation. In the following we give a small example illustrating the use of aspects to specify security risk treatments in the AORDD approach.

Consider an e-commerce system for online purchase of travel tickets running over the Internet with a front-end web server and a back-end database server located on the LAN (local area network) inside the Internet router. The router

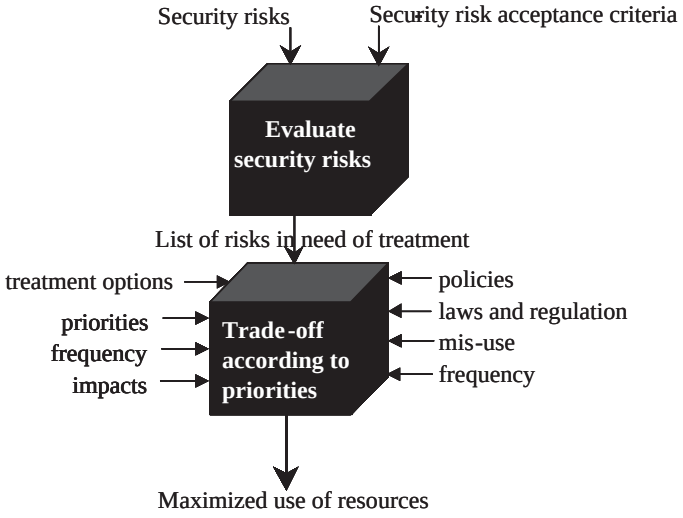


Fig. 6. Overview of the trade-off procedure

is configured to accept all incoming and outgoing requests. Figure 7 provide an overview of the network.

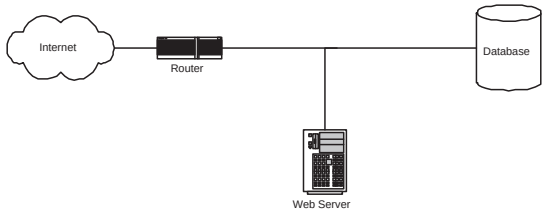


Fig. 7. Overview of the network

The system is defined as the software running the e-Commerce system as well as the network the e-Commerce application is running on, meaning the LAN including the router. The system environment is The Internet. The system currently has no security mechanism implemented and there is no enforcement of security policies, laws, and regulations. The focus of the example is on the security risk assessment part of the AORDD approach, and how treatment options can be represented as aspects.

In the first iteration of the development process we specify the functional requirements and perform the requirement security risk assessment based on these requirements (sub-process 1). In this example we consider the functional requirement "Consumers pays for services online" [18]. This clearly represent a problem since the communication is not encrypted or in any other way secured. The security risk identification and analysis (sub-process 2 and 3) is performed using appropriate security risk analysis methods. In this example we used security-HazOp [27], where one does security risk identification using the negation of the security attributes combined with traditional HazOp guidewords such as to late, never, altered, to early etc. [19].

The result from the requirement security risk assessment gives statement regarding potential treatment solutions. These solutions is evaluated in sub-process 5 using the cost-benefit trade-off analysis. Security risks identified are evaluated against the security risk acceptance criteria. Security risks are measured in terms of security risk levels, which is a combination of the impact and the frequency value. In this context the security risk acceptance criteria is defined such that all security risks with risk level equal to or higher than "High" must be treated (scale Low, Medium, High, and Extreme [19]). Table 2 gives an overview of the main result of the assessment.

Threat	Risk Level	Security risk treatment
Payment information is disclosed to unauthorized party	High	Encrypt link (using VPN is sufficient) OR Encrypt data (encrypted data sent on unsecure channel) OR use TSL
Payment is prohibited by unauthorized party (DOS or similar)	High	Use time outs and session ID

Table 2. Main result of the security risk assessment

The treatment option suggested for the two security risks in need of treatment is encryption (see Table 2). Encryption is crosscutting and may be used in various modules in a system. Another important issue with encryption is to be able to easily change the encryption algorithm. In AORDD treatment options is modeled as aspects in order to swamp treatment solutions in and out during trade-off analysis and whenever the treatment solutions evolve or need to be replaced.

Results from security risk assessment in one iteration of the development is either treated in that iteration or transferred into security requirements in the subsequent iteration. To specify security requirements one may use UMLsec [16], [15], the UML extension for secure systems development. For further explanation on how to use UMLsec in a risk-driven development context the reader is referred to [24]. Figure 8 gives one example of security risk treatment solution using a UML behavioral diagram. The diagram shows encryption of payment information using a simple secrete key encryption protocol. In this example we

only focus on the behavioral aspects. A complete aspect design covers all design views and consist of a set of structural and behavioral diagrams. The set of structural and behavioral diagrams is then weaved with the primary model before coding.

- Security requirement: Ensure confidentiality of payment information.
- Specified using UMLsec: To ensure confidentiality we use the UMLsec stereotype *secrecy* for the all payment information.

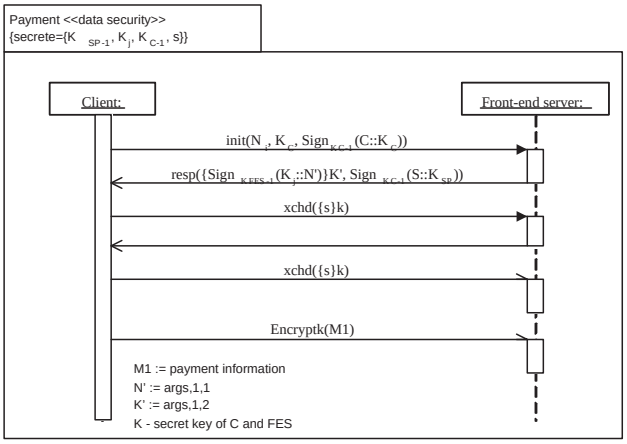


Fig. 8. Security risk treatment aspects for encryption of payment information

6 Conclusion and Further Work

In this paper we have presented an AORDD approach based on the CORAS framework for risk assessment of security critical systems and AOM. Aspects are used to describe security risk treatments, while core functionality is described in the primary model. The process handles security issues in all phases of the development. Security is addressed using security risk assessment and static and dynamic system analysis. Security risks identified in one phase is treated in that iteration and analyzed in the following iteration. Before system analysis, aspects, and the primary model is composed and the analysis is done on the composed model. This makes sure that consistency of the system is maintained and that the system fulfills its requirements, both functional and security requirements, as well as reveal potential stress points and errors throughout the development.

By treating security risk treatment as aspects one gain a great degree of reusability and a low degree of coupling. This opens for the ability to assess

different security treatment options. A aspect-oriented design does also enhance the ability to do cost-benefit trade-off analysis as part of the security risk assessment. The goal of the trade-off analysis is to minimize the resource spent on managing security risks or at least to make effective use of the resources spent.

The current version of the AORDD approach is preliminary and we are currently working on extending and refining all phases of the approach. Further work includes extending and refining the trade-off approach to handle both fault tolerance and security issues. We will also extend the risk assessment phase in the AORDD approach to include assessment of fault tolerance. An important issue to consider in this work is to look into standards targeting these two attributes and incorporate guidelines and recommendation from domain specific standards. Relevant standards targeting security is IEC13355 [14], ISO17799 [13], and Common Criteria [5]. Important standards targeting fault tolerance is IEC61508 [12], DO-178B [22], MoD 00-56 [19], and similar safety relevant standards.

References

1. CORAS (2000-2003). A platform for risk analysis of security critical systems. IST-2000-25031, http://www.nr.no/coras/, 9. March 2003.
2. AS/NZS. AS/NZS 4360:1999, Risk Management. Standards Australia, Strathfield, 1999.
3. B. Barber and J. Davey. The use of the ccta risk analysis and management methodology cramm in health information systems. In K.C. Lun, P. Degoulet, T.E. Piemme, and O. Rienhoff, editors, *MEDINFO 92*, pages 1589–1593, Amsterdam, 1992. North Holland Publishing Co.
4. S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Separating concerns throughout the development lifecycle. In C. Lopes, L. Bergmans, A. Black, and L. Kendall, editors, *Proceedings Of The Aspect-Oriented Programming Workshop at ECOOP'99*. Lisbon, Portugal, June 1999.
5. *Common Criteria for Information Technology Security Evaluation*, Version 2.1, CCIMB–99–031 edition, August 1999. Part 1: Introduction and general model.
6. CORAS, The CORAS Intregrated Platform. Poster at the CORAS public workshop during ICT-2002, 2002.
7. R. B. France, I. Ray, G. Georg, and S. Ghosh. An aspect-oriented approach to design modeling. *To be published in IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, to appear, 2004.
8. G. Georg, R. France, and I. Ray. An aspect-based approach to modeling security concerns. In *Workshop on Critical Systems Development with UML (CSDUML'02)*. Dresden, Germany, October 2002.
9. G. Georg, R. France, and I. Ray. Designing high integrity systems using aspects. In *The Fifth IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems (IICIS 2002)*. Bonn, Germany, November 2002.
10. R. Georg, G.and France. Uml aspect specification using role models. In *Advances in Object-Oriented Information Systems: OOIS2002*, September 2002.
11. ISO/IEC. ISO/IEC 10746: Reference Model for Open Distributed Processing, 1995.

12. ISO/IEC. *ISO/IEC 61508 Functional safety of electical/electronic/progammable electronic safety–related systems*, 1998.
13. ISO/IEC. *ISO/IEC 17799:Information technology – Code of Practice for information security management*, 2000.
14. ISO/IEC. *ISO/IEC 13335: Information technology – Guidelines for management of IT Security*, 2001.
15. J. Jürjens. *Principles for Secure Systems Design*. PhD thesis, Wolfson College, 2002.
16. J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In *UML 2002 – The Unified Modeling Language*, 2002.
17. K. Stølen, F. den Braber, R. Fredriksen, B. A. Gran, S. H. Houmb, Y. C. Stamatiou, J. Ø. Aagedal. *Model-based risk assessment in a component-based software engineering process - using the CORAS approach to identify security risks*, chapter Chapter 11 in Business Component-based Software Engineering. Kluwer Academic Publishers, 2002.
18. Ø. M. Lillevik. An model-based approach to handling risk in security critical systems. Master's thesis, Norwegian University of Science and Technology, http://www.stud.ntnu.no/ lillevik/CORAS/masterthesis.pdf, 2002.
19. Ministry of Defence. *Defence Standard 00–56 issue 2: Safety Management Requirements for Defence Systems*, 1996.
20. S. Northcutt. *Network Intrusion Detection – An Analyst's Handbook*. New Riders, 1999.
21. A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early aspects: A model for aspect-oriented requirements engineering. In *IEEE Joint International Conference on Requirements Engineering*, pages 199–202. Essen, Germany, IEEE Computer Society Press, September 2002.
22. RTCA. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, 1985.
23. S. H. Houmb, F. den Braber, M. Soldal Lund, K. Stølen. Towards a UML profile for Model–Based Risk Assessment. In *UML'2002, Satellite Workshop on Critical Systems Development with UML*, 2002.
24. S. H. Houmb, J. Jürjens. Developing Secure Networked Web–Based Systems Using Model–based Risk Assessment and UMLsec. In *Proceedings of Asia-Pacific Software Engineering Conference*, pages 488–498. IEEE Computer Society, 2003.
25. R. Software. The rational unified process, 1998.
26. K. Stølen, F. den Braber, T. Dimitrakos, R. Fredriksen, B.A. Gran, S.H. Houmb, Y.C. Stamatiou, and J.Ø. Aagedal. Model-based risk assessment in a component-based software engineering process: The CORAS approach to identify security risks. In Franck Barbier, editor, *Business Component-Based Software Engineering*, pages 189–207. Kluwer, 2002. ISBN: 1-4020-7207-4.
27. R. Winther, O. Johnsen, and B. A. Gran. Security Assessments of Safety Critical Systems Using HAZOPs. In *Computer Safety, Reliability and Security, 20th International Conference, SAFECOMP 2001, Budapest, Hungary, September 26-28, 2001, Proceedings*, volume 2187 of *Lecture Notes in Computer Science*. Springer, 2001.

UML 2.0 Interactions: Semantics and Refinement

María Victoria Cengarle[1] and Alexander Knapp[2]

[1] Technische Universität München
cengarle@in.tum.de
[2] Ludwig-Maximilians-Universität München
knapp@pst.ifi.lmu.de

Abstract. The UML 2.0 integrates a dialect of High-Level Message Sequence Charts (HMSCs) for interaction modelling. The most noteworthy addition of UML 2.0 interactions to HMSCs is the introduction of negated specifications which can be used to rule out behaviour from implementations. A trace-based semantics for UML 2.0 interactions is proposed which captures both the standard composition operators for UML 2.0 interactions from HMSCs, and the proprietary negation and assertion operators. The semantics lays the ground for discussing several alternatives for treating negation in interactions. In particular, the semantics decides whether a trace is positive or negative for a given interaction; all other traces are deemed to be inconclusive. Based on these verdicts, notions of implementation and refinement for interactions are defined.

1 Introduction

UML interactions describe possible message exchanges between system instances. In UML 2.0, a dialect of High-Level Messages Sequence Charts (HMSC [4]) replaced the quite inexpressive notion of UML 1.x interactions [5]. Besides integrating the standard HMSC primitives like sequential, parallel, and iterative composition of interactions, UML 2.0 provides means to specify negative behaviour, i.e., behaviour forbidden in system implementations. The ensuing increase in expressiveness makes UML 2.0 an acceptable choice for modelling safety-critical systems. However, in order to put UML 2.0 interactions on an equal footing with HMSCs or Live Sequence Charts (LSC [2]), a formal understanding of the semantics of its interaction language is indispensable. Moreover, the notion of implementation and refinement, based on the formal semantics, form a necessary prerequisite for using UML 2.0 interactions as a formal design language.

In fact, the UML 2.0 specification document [6] is rather vague on the innovative features of the interaction language, like negation. The semantics of what may be called the positive fragment of UML 2.0 interactions, i.e., the language part that does not contain negation, can be equipped straightforwardly with a formal semantics following the specification [9]. Not surprisingly, however, different interpretations of negative interactions have been proposed in the literature. According to the specification, a UML 2.0 interaction describes valid (or positive) and invalid (or negative) traces of event occurrences where invalid traces are induced by using the unary interaction operators $\mathsf{neg}(-)$ and $\mathsf{assert}(-)$. The set of positive and negative traces defined by an interaction need not cover all possible interactions, so the remaining traces may be called inconclusive

for the interaction. Störrle [8] discusses several alternatives for the negated interaction $\mathsf{neg}(S)$ ranging from "not the [valid] traces of S" over "anything but the [valid] traces of S" to exchanging the valid and the invalid traces of S; he finally adopts the last view in order ensure that double negation is the identity. Each of these interpretations shows a drawback: In general, the first and the last approach assign no positive traces to $\mathsf{neg}(S)$ and thus the combination of negation with non-negated interaction fragments leads to an empty set of positive traces. The second approach discards the possibility of inconclusive traces. In contrast, Haugen and Stølen [3] interpret the valid traces of $\mathsf{neg}(S)$ as consisting just of the empty trace; a formal definition of valid and invalid traces of an interaction, however, is not given.

We propose a trace-based, formal semantics for UML 2.0 interactions including part of the positive fragment but concentrating on the language constructs for specifying negative traces. For the definition of the semantics we employ Pratt's framework of partially ordered multisets or pomsets [7] for modelling concurrency. On the one hand, this framework simplifies the definition of the various composition operators for interactions; on the other hand, traces are subsumed by linear pomsets. The semantics decides if a trace is positive or if it is negative for an interaction. We only briefly summarise the semantics of the positive interaction fragment which coincides with Störrle's interpretation [9]. For negated interactions, we build on Haugen and Stølen's view [3] and define the negative traces of combined interaction fragments. We detail the consequences of this approach and contrast it with Störrle's interpretations. Moreover, we provide means for reducing the semantics to only calculating the positive traces of an interaction, albeit at the expense of a classical not-operator. The semantics is put to use by introducing a notion of an implementation of an interaction as a process that shows a least one positive trace of the interaction and no negative trace. In particular, our interaction semantics implies that a trace may be simultaneously positive and negative for the same interaction. We discern between such overspecified interactions and interactions that are contradictory in the sense that they do not admit an implementation. Based on interaction implementations, we introduce a model-theoretic notion of refinement of interactions.

The remainder of this paper is structured as follows: In Sect. 2 we briefly recall the notion of pomsets and traces. The fragment of the interaction language of UML 2.0 considered here is introduced in Sect. 3, together with its abstract syntax. In Sect. 4 the language of interactions is equipped with a trace-based formal semantics, which includes both valid and invalid traces. In Sect. 5 the reduction of the semantics of negation to the semantics of valid traces is studied. The semantics is used in Sect. 6 to define the concepts of implementation and refinement of interactions. In Sect. 7 we analyse implications of the introduced notions with respect to related work. We conclude in Sect. 8 with an outlook to future research.

2 Preliminaries

We briefly review the basic definitions on partially ordered, labelled multisets as introduced by Pratt [7] for modelling concurrency. In particular, we define sequential and parallel composition operators and the notion of traces and processes.

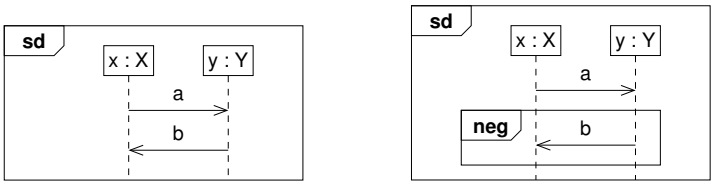
A partially ordered, labelled multiset, or *pomset*, is the isomorphism class $[(X, \leq_X, \lambda_X)]$ of a labelled partial order $(X, \leq_X, \lambda_X)$ w.r.t. monotone, label-preserving maps. A *trace* is a pomset whose ordering is total. We write $lin(p)$ for all possible linearisations of a pomset p, i.e., all traces that extend the ordering of p: $[(X', \leq_{X'}, \lambda_{X'})] \in lin([(X, \leq_X, \lambda_X)])$ if, and only if $X' = X$, $\lambda_{X'} = \lambda_X$, and $\leq_X \subseteq \leq_{X'}$ where $x_1 \leq_{X'} x_2$ or $x_2 \leq_{X'} x_1$ for all $x_1, x_2 \in X'$.

The *empty* pomset, represented by $(\emptyset, \emptyset, \emptyset)$, is denoted by ε. Let $p = [(X, \leq_X, \lambda_X)]$ and $q = [(Y, \leq_Y, \lambda_Y)]$ be pomsets such that $X \cap Y = \emptyset$. The *concurrence* of p and q, written as $p \| q$, is given by $[(X \cup Y, \leq_X \cup \leq_Y, \lambda_X \cup \lambda_Y)]$. The *concatenation* of p and q, written as $p ; q$, is given by $[(X \cup Y, (\leq_X \cup \leq_Y \cup (X \times Y))^*, \lambda_X \cup \lambda_Y)]$. Given a binary, symmetric relation $\asymp$ on labels, the *$\asymp$-concatenation* of p and q, written as $p ;_{\asymp} q$, is given by $[(X \cup Y, (\leq_X \cup \leq_Y \cup \{(x, y) \in X \times Y \mid \lambda_X(x) \asymp \lambda_Y(y)\})^*, \lambda_X \cup \lambda_Y)]$. Note that concatenation and $\asymp$-concatenation are associative, and concurrence is associative and commutative.

A *process* is a set of pomsets. An n-ary function f on pomsets is lifted to processes $P_1, \ldots, P_n$ by defining $f(P_1, \ldots, P_n) = \{f(p_1, \ldots, p_n) \mid p_1 \in P_1, \ldots, p_n \in P_n\}$.

3 UML 2.0 Interactions

UML 2.0 interactions describe message exchanges between instances. Consider the sample basic interaction in Fig. 1(a) which specifies two instances x and y which exchange the messages a and b. The dispatch of a message (depicted by the arrow tail) and the arrival of a message (arrow head) on the lifeline of an instance (dashed line) are called event occurrences. The pictorial representation of a basic interaction carries the intuitive meaning of a partial order of event occurrences: The dispatch of a message occurs before the arrival of the same message; and the event occurrences on the lifeline of an instance are ordered from top to bottom. Thus, the interaction in Fig. 1(a) defines a single valid trace in which the following event occurrences appear in this order: a is sent from x to y; a is received by y from x; b is sent from y to x; b is received by x from y. In particular, all other traces are inconclusive for this interaction. On the other hand, the interaction in Fig. 1(b) defines both negative and positive traces. The trace of first sending and receiving a and then sending and receiving b is negative, whereas the trace just consisting of sending and receiving a is positive. Again, all other traces are inconclusive, as the interaction provides no verdicts on these traces.



(a) Basic interaction diagram (b) Interaction diagram with negation

Fig. 1. Sample interactions

More generally, a UML 2.0 basic interaction consists of event occurrences and a general ordering relation which induces an arbitrary partial order on the set of event occurrences, subject to the following constraints: The dispatch of a message occurs before the arrival of the message; and all event occurrences for the same lifeline are totally ordered. Moreover, UML 2.0 puts a number of interaction-building operators at disposal. In sequential composition, the behaviour of the resulting interaction is the behaviour of the first given interaction followed by the behaviour of the second given interaction. There are two kinds of sequential composition which differ in the meaning of the word "followed". Strict composition requires the behaviour of the first interaction to be completely performed before starting with the behaviour of the second interaction. Weak composition only requires the behaviour specified for an instance in the first interaction to be completely performed before starting with the behaviour for that instance in the second interaction. Other operators are parallel composition, disjunction, loop, ignore, assert, and negation. Two parallel interactions are to be executed simultaneously. Disjunction means to execute any one of two given interactions. Loop repeatedly executes its interaction argument, as long as given by two additional natural numbers m and n passed as parameter: at least m and at most n times, where n can also be ∞ meaning an arbitrary number of times. Ignore allows additional messages to occur besides the ones specified in its interaction argument. Finally, assertion discards inconclusive traces, and negation prohibits the behaviour specified by its argument.

We define the abstract syntax of the fragment of the language of UML 2.0 interactions introduced above, by first characterising basic interactions as pomsets and then capturing the interaction operators by a context-free grammar. We assume two primitive domains for *instances* $\mathbb{I}$ and *messages* $\mathbb{M}$. An *event* e is either of the form $\mathsf{snd}(s, r, m)$ or of the form $\mathsf{rcv}(s, r, m)$, representing the dispatch and the arrival of message m from *sender* instance s to *receiver* instance r, respectively. The set of events is denoted by $\mathbb{E}$. We say that the instance s is *active* for $\mathsf{snd}(s, r, m)$ and, similarly, that the instance r is *active* for $\mathsf{rcv}(s, r, m)$. We define a binary, symmetric *conflict* relation $\asymp$ on events: If an instance is active for both events e and e' then $e \asymp e'$.

A *basic* interaction is given by an event-labelled pomset $[(E, \leq_E, \lambda_E)]$ such that conflicting events do not occur concurrently, i.e., if $e_1, e_2 \in E$ with $\lambda_E(e_1) \asymp \lambda_E(e_2)$, then $e_1 \leq_E e_2$ or $e_2 \leq_E e_1$.

$$
\begin{array}{rcl}
\textit{Interaction} & ::= & \textit{Basic} \\
& | & \textit{CombinedFragment} \\
\textit{CombinedFragment} & ::= & \mathsf{strict}(\textit{Interaction}, \textit{Interaction}) \\
& | & \mathsf{seq}(\textit{Interaction}, \textit{Interaction}) \\
& | & \mathsf{par}(\textit{Interaction}, \textit{Interaction}) \\
& | & \mathsf{loop}(\textit{Nat}, (\textit{Nat} \mid \infty), \textit{Interaction}) \\
& | & \mathsf{ignore}(\textit{Messages}, \textit{Interaction}) \\
& | & \mathsf{alt}(\textit{Interaction}, \textit{Interaction}) \\
& | & \mathsf{neg}(\textit{Interaction}) \\
& | & \mathsf{assert}(\textit{Interaction})
\end{array}
$$

Table 1. Abstract syntax of interactions (fragment)

The abstract syntax of interactions is given by the grammar in Tab. 1. Therein, *Basic* ranges over the basic interactions, *Nat* ranges over the natural numbers, and *Messages* over the subsets of $\mathbb{M}$.

From the notion of basic interactions and the interaction operators in Tab. 1 a number of auxiliary interaction operators can be derived. We use the name skip for the empty (basic) interaction, which is given by the pomset $[(\emptyset, \emptyset, \emptyset)]$. The operator $\mathsf{opt}(-)$ is defined by $\mathsf{opt}(S) = \mathsf{alt}(\mathsf{skip}, S)$, the operator $\mathsf{consider}(-,-)$ by $\mathsf{consider}(M, S) = \mathsf{ignore}(\mathbb{M} \setminus M, S)$. In fact, the UML 2.0 specification defines several other interaction operators, in particular break and $\mathsf{critical}$; these operators, as well as message parameters and conditions, are not considered in this work.

4 Semantics

We define a classical satisfaction relation between traces and interactions that do not contain occurrences of the operator for negation. We afterwards extend this definition for negation, and complement it with a negative satisfaction relation. After presenting some notorious examples, we show some properties of the notions introduced so far.

4.1 Semantic Domains

The domain $\mathbb{P}$ comprises all pomsets $[(E, \leq_E, \lambda_E)]$ labelled with events from $\mathbb{E}$ such that if $e_1, e_2 \in E$ with $\lambda_E(e_1) \asymp \lambda_E(e_2)$, then $e_1 \leq_E e_2$ or $e_2 \leq_E e_1$. The subdomain $\mathbb{T}$ of $\mathbb{P}$ comprises all pomsets in $\mathbb{P}$ that are traces. In particular, the empty pomset ε is in $\mathbb{T}$. When representing a finite pomset in $\mathbb{P}$ we will also use a more concrete, set-based notation like writing $\{\mathsf{snd}(s,r,m) \leq \mathsf{rcv}(s,r,m)\}$ instead of $[(\{e_1, e_2\}, \{(e_1,e_1),(e_1,e_2),(e_2,e_2)\}, \{e_1 \mapsto \mathsf{snd}(s,r,m), e_2 \mapsto \mathsf{rcv}(s,r,m)\})]$. Similarly, for the representation of finite traces in $\mathbb{T}$, as in the example above, we also employ the more succinct notation $\mathsf{snd}(s,r,m) \cdot \mathsf{rcv}(s,r,m)$.

On pomsets in $\mathbb{P}$, the filtering relation $\mathit{filter}(M) : \mathbb{P} \to \wp\mathbb{P}$ removes some elements of p whose labels show a message in M. More precisely, we first define $\mathit{filter}(M)$ on event-labelled sets: Let E be a set and $\lambda : E \to \mathbb{E}$ a labelling function; then $E' \in \mathit{filter}(M)(E, \lambda)$ if $E' \subseteq E$ and, if $e \in E \setminus E'$, then $(\lambda(e) = \mathsf{snd}(s,r,m) \vee \lambda(e) = \mathsf{rcv}(s,r,m)) \wedge m \in M$. For an event-labelled partial order $(E, \leq_E, \lambda_E)$ we set $(E', \leq_E \cap (E' \times E'), \lambda_E{\restriction}E') \in \mathit{filter}(M)(E, \leq_E, \lambda_E)$ if $E' \in \mathit{filter}(M)(E, \lambda_E)$. Finally, we extend these definitions to event-labelled pomsets by setting $[(E', \leq_{E'}, \lambda_{E'})] \in \mathit{filter}(M)([(E, \leq_E, \lambda_E)])$ if $(E', \leq_{E'}, \lambda_{E'}) \in \mathit{filter}(M)(E, \leq_E, \lambda_E)$, which is obviously well-defined. Note that $\mathit{filter}(M)$ restricted to traces delivers traces, i.e., $\mathit{filter}(M)$ is also a relation $\mathit{filter}(M) : \mathbb{T} \to \wp\mathbb{T}$.

4.2 The Positive Fragment

Let us begin considering interactions with no occurrence of negation or assertion, which we call the *positive fragment* of the language. The positive satisfaction relation between traces and interactions, denoted by $t \models_{\mathrm{p}} S$ and read *t positively satisfies S*, where t is a trace and S an interaction of the positive fragment, is inductively defined on the

$t \models_{\mathrm{p}} B \quad \text{if } t \in \mathit{lin}(B)$

$t \models_{\mathrm{p}} \mathsf{strict}(S_1, S_2) \quad \text{if } \exists t_1, t_2 \,.\, t = t_1 \,;\, t_2 \wedge t_1 \models_{\mathrm{p}} S_1 \wedge t_2 \models_{\mathrm{p}} S_2$

$t \models_{\mathrm{p}} \mathsf{seq}(S_1, S_2) \quad \text{if } \exists t_1, t_2 \,.\, t \in \mathit{lin}(t_1 \fatsemi t_2) \wedge t_1 \models_{\mathrm{p}} S_1 \wedge t_2 \models_{\mathrm{p}} S_2$

$t \models_{\mathrm{p}} \mathsf{par}(S_1, S_2) \quad \text{if } \exists t_1, t_2 \,.\, t \in \mathit{lin}(t_1 \parallel t_2) \wedge t_1 \models_{\mathrm{p}} S_1 \wedge t_2 \models_{\mathrm{p}} S_2$

$t \models_{\mathrm{p}} \mathsf{loop}(0, 0, S) \quad \text{if } t = \varepsilon$

$t \models_{\mathrm{p}} \mathsf{loop}(0, n+1, S) \quad \text{if } t = \varepsilon \vee t \models_{\mathrm{p}} \mathsf{seq}(S, \mathsf{loop}(0, n, S))$

$t \models_{\mathrm{p}} \mathsf{loop}(m+1, n+1, S) \quad \text{if } t \models_{\mathrm{p}} \mathsf{seq}(S, \mathsf{loop}(m, n, S))$

$t \models_{\mathrm{p}} \mathsf{loop}(m, \infty, S) \quad \text{if } \exists n \geq m \,.\, t \models_{\mathrm{p}} \mathsf{loop}(m, n, S)$

$t \models_{\mathrm{p}} \mathsf{ignore}(M, S) \quad \text{if } \exists t_1 \,.\, t_1 \in \mathit{filter}(M)(t) \wedge t_1 \models_{\mathrm{p}} S$

$t \models_{\mathrm{p}} \mathsf{alt}(S_1, S_2) \quad \text{if } t \models_{\mathrm{p}} S_1 \vee t \models_{\mathrm{p}} S_2$

(a) Semantics of the positive fragment

$t \models_{\mathrm{p}} \mathsf{neg}(S) \quad \text{if } t = \varepsilon$

$t \models_{\mathrm{p}} \mathsf{assert}(S) \quad \text{if } t \models_{\mathrm{p}} S$

$t \models_{\mathrm{n}} \mathsf{strict}(S_1, S_2) \quad \text{if } \exists t_1, t_2 \,.\, t = t_1 \,;\, t_2 \wedge (t_1 \models_{\mathrm{n}} S_1 \vee (t_1 \models_{\mathrm{p}} S_1 \wedge t_2 \models_{\mathrm{n}} S_2))$

$t \models_{\mathrm{n}} \mathsf{seq}(S_1, S_2) \quad \text{if } \exists t_1, t_2 \,.\, t \in \mathit{lin}(t_1 \fatsemi t_2) \wedge (t_1 \models_{\mathrm{n}} S_1 \vee (t_1 \models_{\mathrm{p}} S_1 \wedge t_2 \models_{\mathrm{n}} S_2))$

$t \models_{\mathrm{n}} \mathsf{par}(S_1, S_2) \quad \text{if } \exists t_1, t_2 \,.\, t \in \mathit{lin}(t_1 \parallel t_2) \wedge ((t_1 \models_{\mathrm{n}} S_1 \wedge t_2 \models_{\mathrm{n}} S_2) \vee (t_1 \models_{\mathrm{n}} S_1 \wedge t_2 \models_{\mathrm{p}} S_2) \vee (t_1 \models_{\mathrm{p}} S_1 \wedge t_2 \models_{\mathrm{n}} S_2))$

$t \models_{\mathrm{n}} \mathsf{loop}(0, n+1, S) \quad \text{if } t \models_{\mathrm{n}} \mathsf{seq}(S, \mathsf{loop}(0, n, S))$

$t \models_{\mathrm{n}} \mathsf{loop}(m+1, n+1, S) \quad \text{if } t \models_{\mathrm{n}} \mathsf{seq}(S, \mathsf{loop}(m, n, S))$

$t \models_{\mathrm{n}} \mathsf{loop}(m, \infty, S) \quad \text{if } \exists n \geq m \,.\, t \models_{\mathrm{n}} \mathsf{loop}(m, n, S)$

$t \models_{\mathrm{n}} \mathsf{ignore}(M, S) \quad \text{if } \exists t_1 \,.\, t_1 \in \mathit{filter}(M)(t) \wedge t_1 \models_{\mathrm{n}} S$

$t \models_{\mathrm{n}} \mathsf{alt}(S_1, S_2) \quad \text{if } t \models_{\mathrm{n}} S_1 \wedge t \models_{\mathrm{n}} S_2$

$t \models_{\mathrm{n}} \mathsf{neg}(S) \quad \text{if } t \neq \varepsilon \wedge t \models_{\mathrm{p}} S$

$t \models_{\mathrm{n}} \mathsf{assert}(S) \quad \text{if } t \not\models_{\mathrm{p}} S$

(b) Extended semantics for negation

Table 2. Semantics of interactions

structure of S as shown in Tab. 2(a). Therein, B ranges over basic interactions. This semantics is a reformulation of Störrle's definition [9] using pomsets.

In particular, the only trace positively satisfying the emtpy interaction skip is the empty pomset. A basic interaction $B = \{\mathsf{snd}(s, r, m) \leq \mathsf{rcv}(s, r, m)\}$ is positively satisfied solely by the trace $t_B = \mathsf{snd}(s, r, m) \cdot \mathsf{rcv}(s, r, m)$.

4.3 Negation

The semantics of a negated interaction $\mathsf{neg}(S)$ is classically defined by making positive for $\mathsf{neg}(S)$ all those traces that are not positive for S and making negative for $\mathsf{neg}(S)$ all those traces that are positive for S. Such a definition, however, rules out inconclusive

traces. In general, thus, we need to distinguish *positive*, *negative*, and *inconclusive* runs for an interaction. We write $t \models_{\mathrm{n}} S$ if t *negatively* satisfies S. The inductive definition of $\models_{\mathrm{p}}$ is extended and the relation $\models_{\mathrm{n}}$ is inductively defined on the structure of S as shown in Tab. 2(b).

In particular, we define $\models_{\mathrm{n}}$ for all combined interaction fragments and, in accordance with Haugen and Stølen [3], we regard the empty trace as being positive for $\mathsf{neg}(S)$. For the combined fragments $\mathsf{strict}(-,-)$ and $\mathsf{seq}(-,-)$ we adopt the view that only those traces are negative that either run through the first operand negatively or fulfil the first operand positively but the second operand negatively. A similar stance is taken towards $\mathsf{par}(-,-)$ where either both operands have to be run through negatively or one of the operands negatively the other one positively in order to make a run negative. In $\mathsf{alt}(-,-)$ both operands have to be run through negatively. Our semantics for assertion is the "assertion as affirmation" interpretation of Störrle [8].

Störrle [8] considers three different interpretations of $\mathsf{neg}(S)$. All of them coincide in declaring negative for $\mathsf{neg}(S)$ all those traces that are positive for S. For the positive traces of $\mathsf{neg}(S)$, interpretation (1), called "not the [valid] traces of S", assigns no positive traces to $\mathsf{neg}(S)$; interpretation (2), called "anything but the [valid] traces of S", makes all traces that are not positive for S the positive traces of $\mathsf{neg}(S)$; interpretation (3) declares the negative traces of S to be the positive traces for $\mathsf{neg}(S)$. Employing the interpretations (1) or (3), the usage of negation inside combined fragments leads to the undesirable consequence that the overall interaction shows no positive traces at all. Interpretation (2) excludes the possibility of inconclusive traces for $\mathsf{neg}(S)$.

4.4 Examples

Let B_i be the basic interactions $\{\mathsf{snd}(s_i, r_i, m_i) \leq \mathsf{rcv}(s_i, r_i, m_i)\}$ $(i = 1, 2, 3)$, where all m_i are different, and let t_i be the traces $\mathsf{snd}(s_i, r_i, m_i) \cdot \mathsf{rcv}(s_i, r_i, m_i)$ $(i = 1, 2, 3)$. We then have that

- $t_1 \models_{\mathrm{p}} \mathsf{strict}(B_1, \mathsf{neg}(B_2))$
- $t_1; t_2 \models_{\mathrm{n}} \mathsf{strict}(B_1, \mathsf{neg}(B_2))$
- $t_1; t_2 \models_{\mathrm{n}} \mathsf{strict}(B_1, \mathsf{strict}(\mathsf{neg}(B_2), B_3))$
- $t_1; t_3 \models_{\mathrm{p}} \mathsf{strict}(B_1, \mathsf{strict}(\mathsf{neg}(B_2), B_3))$
- $t_1; t_2; t_3 \models_{\mathrm{n}} \mathsf{strict}(B_1, \mathsf{strict}(\mathsf{neg}(B_2), B_3))$
- $t_2 \models_{\mathrm{p}} \mathsf{par}(\mathsf{neg}(B_2), B_2)$ and
- $t_2 \not\models_{\mathrm{n}} \mathsf{par}(\mathsf{neg}(B_2), B_2)$.

A more interesting case is given by the following two facts:

- $t_2 \models_{\mathrm{p}} \mathsf{strict}(\mathsf{neg}(B_2), B_2)$ and
- $t_2 \models_{\mathrm{n}} \mathsf{strict}(\mathsf{neg}(B_2), B_2)$.

Thus, t_2 is simultaneously positive and negative for $\mathsf{strict}(\mathsf{neg}(B_2), B_2)$. We therefore call $\mathsf{strict}(\mathsf{neg}(B_2), B_2)$ an overspecified interaction.

Definition 1. *An interaction S is* overspecified *if there exists a trace t with $t \models_{\mathrm{p}} S$ and $t \models_{\mathrm{n}} S$.*

For the same B_2, a further overspecified interaction is $\mathsf{par}(\mathsf{assert}(B_2), \mathsf{neg}(B_2))$. The trace t_2 satisfies this interaction both positively and negatively.

4.5 Properties

It is easy to check that both forms of sequential composition are associative, and that parallel and alternative composition are associative and commutative.

Lemma 1. *Let S_1, S_2, and S_3 be interactions, and t be a trace.*

1. $t \models_{\mathrm{p}} \mathsf{strict}(S_1, \mathsf{strict}(S_2, S_3))$ *iff* $t \models_{\mathrm{p}} \mathsf{strict}(\mathsf{strict}(S_1, S_2), S_3)$
2. $t \models_{\mathrm{p}} \mathsf{seq}(S_1, \mathsf{seq}(S_2, S_3))$ *iff* $t \models_{\mathrm{p}} \mathsf{seq}(\mathsf{seq}(S_1, S_2), S_3)$
3. $t \models_{\mathrm{p}} \mathsf{par}(S_1, \mathsf{par}(S_2, S_3))$ *iff* $t \models_{\mathrm{p}} \mathsf{par}(\mathsf{par}(S_1, S_2), S_3)$
4. $t \models_{\mathrm{p}} \mathsf{par}(S_1, S_2)$ *iff* $t \models_{\mathrm{p}} \mathsf{par}(S_2, S_1)$
5. $t \models_{\mathrm{p}} \mathsf{alt}(S_1, \mathsf{alt}(S_2, S_3))$ *iff* $t \models_{\mathrm{p}} \mathsf{alt}(\mathsf{alt}(S_1, S_2), S_3)$
6. $t \models_{\mathrm{p}} \mathsf{alt}(S_1, S_2)$ *iff* $t \models_{\mathrm{p}} \mathsf{alt}(S_2, S_1)$

Furthermore, all these propositions also hold when replacing $\models_{\mathrm{p}}$ by $\models_{\mathrm{n}}$.

By abuse of notation we thus abbreviate e.g. $\mathsf{strict}(S_1, \mathsf{strict}(S_2, \mathsf{strict}(\ldots, S_n)))$ to $\mathsf{strict}(S_1, S_2, \ldots, S_n)$ and $\mathsf{alt}(S_1, \mathsf{alt}(S_2, \mathsf{alt}(\ldots, S_n)))$ to $\mathsf{alt}(S_1, S_2, \ldots, S_n)$.

Basic interactions are not negatively satisfiable.

Lemma 2. *$t \not\models_{\mathrm{n}} B$ for any basic interaction B and any trace t.*

The satisfaction relations $\models_{\mathrm{p}}$ and $\models_{\mathrm{n}}$ as defined in Sects. 4.2 and 4.3 are not conclusive, that is, there exist inconclusive traces.

Lemma 3. *There exist a trace t and an interaction S with $t \not\models_{\mathrm{p}} S$ and $t \not\models_{\mathrm{n}} S$.*

Proof. Take for S the basic interaction $B = \{\mathsf{snd}(s, r, m) \leq \mathsf{rcv}(s, r, m)\}$ and for t the trace $\mathsf{rcv}(s, r, m) \cdot \mathsf{snd}(s, r, m)$.

The operator $\mathsf{assert}(-)$ discards inconclusive traces of its operand, that is, it establishes the link between the semantics of interactions and classical two-valued logic.

Lemma 4. *Let S be an interaction and t be a trace. Then $t \models_{\mathrm{p}} \mathsf{assert}(S)$ or $t \models_{\mathrm{n}} \mathsf{assert}(S)$.*

On the syntactic structure of interactions we define a well-founded ordering, which can be used to demonstrate further properties of interactions by induction.

Definition 2. *We define a partial order on interaction terms as the reflexive and transitive closure of the following binary relation:*

$$
\begin{aligned}
\mathsf{skip} &\leq S & S &\leq \mathsf{neg}(S) \\
S_1 &\leq \mathsf{strict}(S_1, S_2) & S_2 &\leq \mathsf{strict}(S_1, S_2) \\
S_1 &\leq \mathsf{seq}(S_1, S_2) & S_2 &\leq \mathsf{seq}(S_1, S_2) \\
S_1 &\leq \mathsf{par}(S_1, S_2) & S_2 &\leq \mathsf{par}(S_1, S_2) \\
S_1 &\leq \mathsf{alt}(S_1, S_2) & S_2 &\leq \mathsf{alt}(S_1, S_2) \\
S &\leq \mathsf{ignore}(M, S) & S &\leq \mathsf{assert}(S) \\
\mathsf{seq}(S, \mathsf{loop}(m, n, S)) &\leq \mathsf{loop}(m, n+1, S) & \mathsf{loop}(m, n, S) &\leq \mathsf{loop}(m, \infty, S)
\end{aligned}
$$

where S, S_1, and S_2 are arbitrary interactions, and m and n natural numbers.

Lemma 5. *The above defined ordering $\leq$ on interactions is well founded, i.e., there exists no infinite descending chain $S_1 \geq S_2 \geq \cdots \geq S_n \geq \cdots$.*

5 Negation Revisited

The non-classical interpretation of negation is difficult to deal with. Above all, the need for two satisfaction relations, one positive and one negative, makes it hard to decide what kind of trace is a given one for an interaction, i.e., if the trace is positive, inconclusive, or negative for the interaction. When reconsidering the semantics introduced in the previous section, we firstly discover that negation is unnecessary for testing positive satisfaction. Secondly, when handling negative satisfaction, negation is of course needed, but it can be replaced by a classical version. We introduce therefore the language of *interactions in the classical sense* as opposed to the language of UML 2.0 interactions considered so far.

We begin by reinvestigating the interplay between negation and positive satisfaction. Observe that a negative interaction can only be positively satisfied by the empty process, the same as skip. It therefore seems natural, when it comes to check positive satisfaction, to replace any negative subinteraction by skip.

Definition 3. *The function σ from interactions to interactions of the positive fragment is given by induction on the syntactic structure of its argument as follows:*

$$\begin{aligned}
\sigma(B) &= B \\
\sigma(\mathsf{strict}(S_1, S_2)) &= \mathsf{strict}(\sigma(S_1), \sigma(S_2)) \\
\sigma(\mathsf{seq}(S_1, S_2)) &= \mathsf{seq}(\sigma(S_1), \sigma(S_2)) \\
\sigma(\mathsf{par}(S_1, S_2)) &= \mathsf{par}(\sigma(S_1), \sigma(S_2)) \\
\sigma(\mathsf{loop}(m, \overline{n}, S)) &= \mathsf{loop}(m, \overline{n}, \sigma(S)) \\
\sigma(\mathsf{ignore}(M, S)) &= \mathsf{ignore}(M, \sigma(S)) \\
\sigma(\mathsf{alt}(S_1, S_2)) &= \mathsf{alt}(\sigma(S_1), \sigma(S_2)) \\
\sigma(\mathsf{neg}(S)) &= \mathsf{skip} \\
\sigma(\mathsf{assert}(S)) &= \sigma(S)
\end{aligned}$$

where B ranges over basic interactions, S, S_1, and S_2 over interactions, M over sets of messages, m over the natural numbers, and $\overline{n}$ over the natural numbers or ∞.

Lemma 6. *Let S be an interaction and t be a trace. Then, $t \models_{\mathrm{p}} S$ iff $t \models_{\mathrm{p}} \sigma(S)$.*

This means that the positive fragment of the language and the positive satisfaction relation defined for it as given in Sect. 4.2 and in Tab. 2(a) are sufficient for testing positive satisfaction of arbitrary interactions.

Now we turn our attention to negative satisfaction. The question is if something similar cannot be done for it as well. More precisely, it would be advantageous to get rid of the negative satisfaction relation by defining it in terms of the positive one. This is obviously true for a sublanguage, namely for sequences of interactions involving a negated subinteraction.

Lemma 7. *Let $S = \mathsf{strict}(S_1, \mathsf{neg}(S'), S_2)$ be an interaction with S_1, S', and S_2 from the positive fragment and t be a trace. Then, $t \models_{\mathrm{n}} S$ iff there exists a prefix t' of t such that $t' \models_{\mathrm{p}} \mathsf{strict}(S_1, S')$.*

This result, however, cannot be generalised to the full language of interactions: a binary logic without negation is not enough. A binary logic with classical negation, on the contrary, does suffice. We add an operator $\mathsf{not}(-)$ to the positive fragment of UML 2.0 interactions, which gives rise to the so-called interactions in classical sense. This new unary operator is provided with the classical semantics of negation. We define a transformation from UML 2.0 interactions to interactions in the classical sense, and show that the positive satisfaction of the resulting interaction is equivalent to negative satisfaction of the given one.

Definition 4. *The syntax of* interactions in the classical sense *is given by the syntax in Tab. 1 where* $\mathsf{neg}(-)$ *and* $\mathsf{assert}(-)$ *are removed and* $\mathsf{not}(-)$ *is added to CombinedFragment.*

The positive semantics of interactions in the classical sense is given by the semantics for the positive fragment of UML 2.0 interactions in Tab. 2(a) and

$$t \models_{\mathrm{p}} \mathsf{not}(S) \quad \textit{if } t \not\models_{\mathrm{p}} S$$

We furthermore use the following abbreviations:

$$\begin{aligned} \mathsf{Any} &= \mathsf{ignore}(\mathbb{M}, \mathsf{skip}) \\ \mathsf{None} &= \mathsf{not}(\mathsf{Any}) \\ \mathsf{and}(S_1, S_2) &= \mathsf{not}(\mathsf{alt}(\mathsf{not}(S_1), \mathsf{not}(S_2))) \end{aligned}$$

Definition 5. *The function ν from UML 2.0 interactions to interactions in the classical sense is given by induction on the syntactic structure of its argument as follows:*

$$\begin{aligned} \nu(B) &= \mathsf{None} \\ \nu(\mathsf{strict}(S_1, S_2)) &= \mathsf{alt}(\mathsf{strict}(\nu(S_1), \mathsf{Any}), \mathsf{strict}(\sigma(S_1), \nu(S_2))) \\ \nu(\mathsf{seq}(S_1, S_2)) &= \mathsf{alt}(\mathsf{seq}(\nu(S_1), \mathsf{Any}), \mathsf{seq}(\sigma(S_1), \nu(S_2))) \\ \nu(\mathsf{par}(S_1, S_2)) &= \mathsf{alt}(\mathsf{par}(\nu(S_1), \nu(S_2)), \mathsf{par}(\nu(S_1), \sigma(S_2)), \mathsf{par}(\sigma(S_1), \nu(S_2))) \\ \nu(\mathsf{loop}(m, \overline{n}, S)) &= \mathsf{and}(\mathsf{loop}(m, \overline{n}, \nu(S)), \mathsf{not}(\mathsf{skip})) \\ \nu(\mathsf{ignore}(M, S)) &= \mathsf{ignore}(M, \nu(S)) \\ \nu(\mathsf{alt}(S_1, S_2)) &= \mathsf{and}(\nu(S_1), \nu(S_2)) \\ \nu(\mathsf{neg}(S)) &= \mathsf{and}(\sigma(S), \mathsf{not}(\mathsf{skip})) \\ \nu(\mathsf{assert}(S)) &= \mathsf{not}(\sigma(S)) \end{aligned}$$

where B ranges over basic interactions, S, S_1, and S_2 over interactions, M over sets of messages, m over the natural numbers, and $\overline{n}$ over the natural numbers or ∞.

Lemma 8. *Let S be a UML 2.0 interaction and t be trace. Then $t \models_{\mathrm{n}} S$ iff $t \models_{\mathrm{p}} \nu(S)$.*

Proof. By induction on the partial ordering $\leq$ on UML 2.0 interactions.

Summarising, a closer look at negation leads to the following two results:

$$\begin{aligned} t \models_{\mathrm{p}} S \quad & \text{if } t \models_{\mathrm{p}} \sigma(S) \\ t \models_{\mathrm{n}} S \quad & \text{if } t \models_{\mathrm{p}} \nu(S) \end{aligned}$$

where S is an arbitrary UML 2.0 interaction, $\sigma(S)$ is an interaction of the positive fragment of the language of UML 2.0 interactions obtained in terms of S, and $\nu(S)$ is an interaction in the classical sense in terms of S. Notice that the positive fragment of the language of UML 2.0 interactions is also the positive fragment of the language of interactions in classical sense. More importantly, by means of these two transformations, σ and ν, we do not need to test negative satisfaction. This observation may be useful for checking overspecification, but we defer a closer investigation to future work.

6 Implementation and Refinement

Having a formal semantics for interactions, further concepts can be defined in terms of it. We introduce the notions of implementation of an interaction by a process, of equivalence of interactions, and of refinement of an interaction by another one. These notions show a number of useful properties, and are intended for formal verification.

Definition 6. *A process I is an* implementation *of an interaction S, written $I \models S$, if*

1. *there exists $t \in lin(I)$ with $t \models_{\mathrm{p}} S$, and*
2. *$t \not\models_{\mathrm{n}} S$ for every $t \in lin(I)$.*

An interaction S is implementable *if there is a process I such that $I \models S$; it is* contradictory *if it is not implementable.*

The following lemma ensures that any interaction admits positive traces and thus that the first condition of the implementation relation is always satisfiable.

Lemma 9. *For every interaction S there exists a trace t with $t \models_{\mathrm{p}} S$.*

Proof. By induction on the partial ordering $\leq$ and the fact that $\varepsilon \models_{\mathrm{p}} \mathsf{neg}(S)$.

This lemma, however, does not imply that any interaction is implementable. Indeed, having a positive trace is not enough, since this very trace may also be negative for the same interaction. Take for instance the overspecified interaction $\mathsf{strict}(\mathsf{neg}(B_2), B_2)$ of Sect. 4.4: its only positive trace t_2 is at the same time negative. Nonetheless, an overspecified interaction may be implementable, that is, overspecified interactions are not necessarily contradictory. Take for instance the interaction $S = \mathsf{alt}(\mathsf{seq}(\mathsf{neg}(B_2), B_1), \mathsf{seq}(\mathsf{neg}(B_2), B_2))$ with B_1 and B_2 as in Sect. 4.4. The trace t_2 is both positive and negative for S, i.e., both $t_2 \models_{\mathrm{p}} S$ and $t_2 \models_{\mathrm{n}} S$, whereas the trace t_1 is only positive for S, i.e., $t_1 \models_{\mathrm{p}} S$ and $t_1 \not\models_{\mathrm{n}} S$. Thus $\{t_1\} \models S$.

Moreover, note that a combination of interactions, each equipped with its own implementation, not necessarily is implemented by the same combination of the corresponding implementations. Take for instance $S_1 = \mathsf{neg}(B_1)$, $S_2 = \mathsf{neg}(B_2)$, $I_1 = \{t_2, \varepsilon\}$, and $I_2 = \{t_1, \varepsilon\}$, with B_i and t_i as defined in Sect. 4.4 ($i = 1, 2$). It is easy to check that, while $I_i \models S_i$ ($i = 1, 2$), it is not true that $I_1 \parallel I_2 \models \mathsf{par}(S_1, S_2)$.

A notion of implementation allows the definition of an equivalence relation.

Definition 7. *Two interactions S_1 and S_2 are* equivalent, *denoted by $S_1 \equiv S_2$, whenever $I \models S_1$ iff $I \models S_2$ for any process I.*

Furthermore, the implementation relation gives rise to a model-theoretic notion of refinement.

Definition 8. *An interaction S'* refines *an interaction S, written $S \rightsquigarrow S'$, if any implementation of S' is also an implementation of S, i.e., if $I \models S'$ implies $I \models S$ for any implementation I.*

Lemma 10. *Refinement is a partial order w.r.t. the equivalence on interactions, i.e., refinement is reflexive, transitive, and antisymmetric w.r.t. $\equiv$.*

An example of an interaction refinement is provided by the removal of disjunctions.

Lemma 11. $\mathsf{alt}(S_1, S_2) \rightsquigarrow S_i$ *for* $i = 1, 2$.

Proof. Let $I \models S_1$. On the one hand, there exists $t \in lin(I)$ with $t \models_{\mathrm{p}} S_1$ and thus $t \models_{\mathrm{p}} \mathsf{alt}(S_1, S_2)$. On the other hand, $t \not\models_{\mathrm{n}} S_1$ and hence $t \not\models_{\mathrm{n}} \mathsf{alt}(S_1, S_2)$ for all $t \in lin(I)$. The case $I \models S_2$ is treated analogously.

Let us now investigate the properties of the refinement relation. As the following lemma shows, in refinement the set of genuine positive traces cannot be enlarged, negative traces remain negative, and at least one positive trace is kept.

Lemma 12. *Let S and S' be interaction with $S \rightsquigarrow S'$.*

1. *For all traces t, if $t \not\models_{\mathrm{p}} S$ or $t \models_{\mathrm{n}} S$, then $t \not\models_{\mathrm{p}} S'$ or $t \models_{\mathrm{n}} S'$.*
2. *If S' is implementable, then for all traces t, $t \models_{\mathrm{n}} S$ implies $t \models_{\mathrm{n}} S'$.*
3. *If S' is implementable, then there is a trace t such that $t \models_{\mathrm{p}} S$ and $t \models_{\mathrm{p}} S'$.*

Proof. For claim (1), suppose $t \models_{\mathrm{p}} S'$ and $t \not\models_{\mathrm{n}} S'$. Then $\{t\} \models S'$, and also $\{t\} \models S$ since $S \rightsquigarrow S'$. Thus $t \models_{\mathrm{p}} S$ and $t \not\models_{\mathrm{n}} S$ which contradicts $t \not\models_{\mathrm{p}} S$ or $t \models_{\mathrm{n}} S$.

For claim (2), suppose $t \not\models_{\mathrm{n}} S'$ and let I be any process such that $I \models S'$. Then also $I \cup \{t\} \models S'$, and thus $I \cup \{t\} \models S$ because $S \rightsquigarrow S'$, which contradicts $t \models_{\mathrm{n}} S$.

For claim (3), assume that $t \not\models_{\mathrm{p}} S$ for all $t \models_{\mathrm{p}} S'$. Since S' is implementable, there is a trace such that $t \models_{\mathrm{p}} S'$ but $t \not\models_{\mathrm{n}} S'$. Then $\{t\} \models S'$, but $\{t\} \not\models S$.

An inconclusive trace can indeed become negative. Recall for instance the interaction B_2 and the trace t_2 from Sect. 4.4: trace t_2 is inconclusive for skip and negative for $\mathsf{neg}(B_2)$, where $\mathsf{skip} \rightsquigarrow \mathsf{neg}(B_2)$. On the other hand, a positive trace may become inconclusive, as witnessed by Lemma 11.

A desirable property of refinement is that the operators be monotonic with respect to it. For instance, for a proof of monotonicity of disjunction w.r.t. refinement, we need to show that a process implementing $\mathsf{alt}(S_1', S_2)$ also implements $\mathsf{alt}(S_1, S_2)$ if $S_1 \rightsquigarrow S_1'$. Unfortunately this is not true. Consider the following constellation:

$$S_1 = B_1 \qquad S_1' = \mathsf{alt}(\mathsf{seq}(\mathsf{neg}(B_2), B_1), \mathsf{seq}(\mathsf{neg}(B_2), B_2))$$
$$S_2 = B_3 \qquad t = t_2$$

where B_1, B_2, B_3, and t_2 are the interactions resp. trace of Sect. 4.4; in particular, $S_1 \rightsquigarrow S_1'$. We have then the following facts:

$$t \models_{\mathrm{p}} S_1' \text{ and } t \models_{\mathrm{n}} S_1' \qquad t \not\models_{\mathrm{p}} S_1$$
$$t \not\models_{\mathrm{p}} S_2 \text{ and } t \not\models_{\mathrm{n}} S_2$$

$$\frac{S_1 \rightsquigarrow_{\mathrm{p}} S_1'}{\mathsf{strict}(S_1, S_2) \rightsquigarrow_{\mathrm{p}} \mathsf{strict}(S_1', S_2)} \qquad \frac{S_2 \rightsquigarrow S_2'}{\mathsf{strict}(S_1, S_2) \rightsquigarrow \mathsf{strict}(S_1, S_2')}$$

$$\frac{S_1 \rightsquigarrow_{\mathrm{p}} S_1'}{\mathsf{seq}(S_1, S_2) \rightsquigarrow_{\mathrm{p}} \mathsf{seq}(S_1', S_2)} \qquad \frac{S_2 \rightsquigarrow S_2'}{\mathsf{seq}(S_1, S_2) \rightsquigarrow \mathsf{seq}(S_1, S_2')}$$

$$\frac{S_1 \rightsquigarrow_{\mathrm{p}} S_1'}{\mathsf{par}(S_1, S_2) \rightsquigarrow_{\mathrm{p}} \mathsf{par}(S_1', S_2)} \qquad \frac{S_1 \rightsquigarrow S_1'}{\mathsf{alt}(S_1, S_2) \rightsquigarrow \mathsf{alt}(S_1', S_2)}$$

$$\frac{S \rightsquigarrow S'}{\mathsf{neg}(S') \rightsquigarrow \mathsf{neg}(S)} \qquad \frac{S \rightsquigarrow S'}{\mathsf{assert}(S) \rightsquigarrow \mathsf{assert}(S')}$$

Table 3. Compositional refinements of interactions

that is, $\{t\} \models \mathsf{alt}(S_1', S_2)$ and $\{t\} \not\models \mathsf{alt}(S_1, S_2)$, i.e., $\mathsf{alt}(S_1, S_2) \not\rightsquigarrow \mathsf{alt}(S_1', S_2)$.

When restricting ourselves to refinements by non-overspecified interactions, disjunction indeed is monotonic w.r.t. refinement.

Lemma 13. *Let S_1, S_1', and S_2 be interactions and let S_1' be implementable and not overspecified. If $S_1 \rightsquigarrow S_1'$, then* $\mathsf{alt}(S_1, S_2) \rightsquigarrow \mathsf{alt}(S_1', S_2)$.

Proof. Let I be a process such that $I \models \mathsf{alt}(S_1', S_2)$. Let $t \in lin(I)$ be a trace of I. Then $t \not\models_{\mathrm{n}} \mathsf{alt}(S_1', S_2)$. In particular, $t \not\models_{\mathrm{n}} S_1'$ or $t \not\models_{\mathrm{n}} S_2$ and thus, by Lemma 12(2), $t \not\models_{\mathrm{n}} S_1$ or $t \not\models_{\mathrm{n}} S_2$, that is, $t \not\models_{\mathrm{n}} \mathsf{alt}(S_1, S_2)$.

Moreover, there is a $t \in lin(I)$ with $t \models_{\mathrm{p}} \mathsf{alt}(S_1', S_2)$, i.e., $t \models_{\mathrm{p}} S_1'$ or $t \models_{\mathrm{p}} S_2$. If $t \models_{\mathrm{p}} S_2$ then $t \models_{\mathrm{p}} \mathsf{alt}(S_1, S_2)$. If $t \models_{\mathrm{p}} S_1'$ then $t \not\models_{\mathrm{n}} S_1'$, as S_1' is not overspecified, and thus $t \models_{\mathrm{p}} S_1$ by Lemma 12(1); hence again $t \models_{\mathrm{p}} \mathsf{alt}(S_1, S_2)$.

However, for proving the monotonicity of the sequential operators in the first argument w.r.t. refinement, the restriction to refinements by non-overspecified interactions is not enough. In fact, in demonstrating that $S_1 \rightsquigarrow S_1'$ implies $\mathsf{strict}(S_1, S_2) \rightsquigarrow \mathsf{strict}(S_1', S_2)$, we have to assume that all positive traces of S_1 are still positive in S_1': If a positive trace of S_1 becomes inconclusive in S_1', there may be more negative traces in $\mathsf{strict}(S_1, S_2)$ than in $\mathsf{strict}(S_1', S_2)$. We therefore introduce a restricted refinement relation $\rightsquigarrow_{\mathrm{p}}$ that keeps all positive traces.

Definition 9. *An interaction S'* positively refines *an interaction S, written $S \rightsquigarrow_{\mathrm{p}} S'$, if S' refines S and for all traces t it holds: if $t \models_{\mathrm{p}} S$ then $t \models_{\mathrm{p}} S'$.*

Some results on the monotonicity of interaction operators w.r.t. the refinement relations $\rightsquigarrow$ and $\rightsquigarrow_{\mathrm{p}}$ are summarised in Tab. 3, where S, S_1, S_2, S', S_1', and S_2' are interactions and S', S_1' and S_2' are implementable and not overspecified. A more complete calculus for interaction refinement is subject of future study.

7 Discussion

Lemma 7 concludes that, for a given interaction, any trace is negative if it completely traverses a negative region, independently of the steps performed afterwards, if any. The

proposal of Haugen and Stølen [3] states that "[. . .] any trace that [completely traverses a negative region] is a negative scenario. Anything may happen [afterwards], it will never make it positive." It is not explicitly said that further steps cannot make the trace inconclusive. If in particular their proposal allows the trace to become inconclusive, then the semantics of Sect. 4 above is more restrictive.

Indeed, this is not a merely speculation of ours. The example used there is that of a restaurant, where a customer orders and is served a beef, including an inbetween negative subinteraction that forbids to burn the meat. Intuitively, hence, if in fact the meat burns in the oven, the obvious thing to do is to take it from the oven and not to bring it to the customer's table. This means that a trace is only negative if, after traversing the negative region, the next positive region is exhaustively traversed as well. It therefore seems that a trace is negative if it traverses all positive regions plus at least one negative region. A big disadvantage of this interpretation is that a semantic definition for it cannot be compositional. Compositionality is not just a comfortable mathematical property, it allows for instance an on-the-fly recognition of a negative trace (or to warn a running system from generating a negative trace), since decisions are taken locally, i.e., independently of what happened before or what will happen henceforth.

The semantics of Sect. 4, plainly worded, states that "the trace is *bad* as soon as it leaves a negative region, it is *good* if both it is exhaustive (i.e., the interaction does not specify any event beyond the trace's last event) and it only traverses positive regions, and it is *inconclusive* otherwise." The key point here is that a trace, which has completely traversed a negative region, is definitively negative. We do think that this is a better choice, and hence put it at the community's disposal for discussion.

A further deviation from the proposal of Haugen and Stølen [3] is the existence of overspecified interactions. The cited work states that "the same trace cannot be both positive and negative." We dispute the convenience of this requirement. Consider any of the overspecified interactions shown above, and a trace that is both positive and negative for the interaction. It is by far not obvious how to rule out one of both possibilities (i.e., deciding if the trace is positive or negative) in a non-arbitrary manner, and making this trace inconclusive is capricious.

Let us finally consider the concepts of supplementing, narrowing and detailing by Haugen and Stølen [3]. Supplementing means reducing the set of inconclusive traces by making some of them either positive or negative; in doing so, positive (negative) traces remain positive (negative). Narrowing means reducing the set of positive traces by making some of them negative; inconclusive (negative) traces remain inconclusive (negative). Detailing consists in providing a translation from a more detailed (concrete) interaction to a given (abstract) interaction; it leaves the sets of positive, negative, and inconclusive traces unchanged. These notions are colloquially defined using the three types of traces associated with an interaction; as with our refinement relation, there is no clue on how to define those in syntactical terms. Our refinement relation is somehow supplementing and narrowing at the same time; supplementing cannot be defined in terms of refinement, since supplementing may make positive an inconclusive trace. The spirit behind all these concepts, however, makes them difficult to compare, since supplementing and narrowing address design evolution, whereas refinement is a tool for formal verification.

8 Conclusions and Outlook

The contribution of the present article is twofold. On the one hand, it defines a semantics for UML 2.0 interactions that is both formal and consistent with the standard [6]. This proposal is compared with earlier ones. On the other hand, a formal semantics allows a mathematically precise definition of implementation and of refinement, such that these relations can be formally proved. These notions show some desirable and some questionable properties, so that they may be subject to further adjustments. They nevertheless set the ground for lifting UML 2.0 to a formal design technique, a sine qua non for its use in the development of critical systems.

Some UML 2.0 operators for interactions were disregarded, namely break and critical, and also message parameters, conditions, and time. We plan to extend the semantics above to include these other features of UML 2.0. The semantics for OCL/RT of [1] can be a good starting point for traces which include time and on which conditions are checkable. A calculus for formal verification is the utmost challenge. This matter can be addressed once implementation and refinement have reached a stable, i.e., broadly accepted, definition.

Acknowledgements. We thank Øystein Haugen and Harald Störrle for fruitful discussions.

References

1. María Victoria Cengarle and Alexander Knapp. Towards OCL/RT. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Proc. 11th Int. Symp. Formal Methods Europe (FME'02)*, volume 2391 of *Lect. Notes Comp. Sci.*, pages 390–409. Springer, Berlin, 2002.
2. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
3. Øystein Haugen and Ketil Stølen. STAIRS — Steps to Analyze Interactions with Refinement Semantics. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proc. 6th Int. Conf. Unified Modeling Language (UML'03)*, volume 2863 of *Lect. Notes Comp. Sci.*, pages 388–402. Springer, Berlin, 2003.
4. International Telecommunication Union. Message Sequence Chart (MSC). Recommendation Z.120, ITU-T, Genève, 1999.
5. Alexander Knapp. A Formal Semantics for UML Interactions. In Robert France and Bernhard Rumpe, editors, *Proc. 2nd Int. Conf. Unified Modeling Language (UML'99)*, volume 1723 of *Lect. Notes Comp. Sci.*, pages 116–130. Springer, Berlin, 1999.
6. Object Management Group. Unified Modeling Language Specification, Version 2.0 (Superstructure). Adopted draft, OMG, 2003. `http://www.omg.org/cgi-bin/doc?ptc/03-08-02`.
7. Vaughan Pratt. Modeling Concurrency with Partial Orders. *Int. J. Parallel Program.*, 15(1):33–71, 1986.
8. Harald Störrle. Assert, Negate and Refinement in UML-2 Interactions. In Jan Jürjens, Bernhard Rumpe, Robert France, and Eduardo B. Fernandez, editors, *Proc. Wsh. Critical Systems Development with UML (CSDUML'03)*, San Francisco, 2003. Technische Universität München, Technical report TUM-I0317.
9. Harald Störrle. Semantics of Interactions in UML 2.0. In *Proc. IEEE Symp. Visual Languages and Formal Methods (VLFM'03)*, Auckland, 2003.

A UML Class Diagram Analyzer

Tiago Massoni, Rohit Gheyi, and Paulo Borba

Informatics Center
Federal University of Pernambuco, Brazil
{tlm,rg,phmb}@cin.ufpe.br

Abstract. Automatic analysis of UML models constrained by OCL invariants is still an open research topic. Especially for critical systems, such tool support is important for early identification of errors in modeling, before functional requirements are implemented. In this paper, we present ideas on an approach for automatic analysis of UML class diagrams, according to a precise semantics based on Alloy, a formal object-oriented modeling language. This semantics permits the use of Alloy's tool support for class diagrams, by applying constraint solving for automatically finding valid snapshots of models. This kind of automation helps the identification of inconsistencies or under-specified models of critical software, besides allowing checking of properties about these models.

1 Introduction

As in other engineering fields, modeling can be a useful activity for tackling significant problems in software development. As a de-facto standard, the Unified Modeling Language (UML) [1] plays a significant role. In particular, the development of high-quality critical systems can benefit from features offered by a standard visual modeling notation like UML, since traditional code-driven approaches are highly risky and often error-prone for such complexity. Further, business rules can be precisely expressed by Object Constraint Language (OCL) [2] invariants attached to class diagrams, enabling the specification of constraints over complex states of critical systems.

As the use of OCL for critical systems substantially grows, the lack of standardized formal semantics for the language does not stimulate the development of tools supporting analysis and verification of OCL expressions in UML models. When using class diagrams for modeling critical systems, the absence of tool support restrains the developer's task, since subtle structural modeling errors may be considerably hard to detect. For instance, inappropriate OCL invariants may turn a model over-constrained, or even inconsistent (i.e. the model allows no implementation). Likewise, models lacking important constraints may allow incorrect implementations.

In this paper, we propose an approach for automatic analysis of class diagrams, according to a precise semantics based on Alloy [3], a formal object-oriented modeling language founded on first-order logic. Alloy is suitable for

object modeling, employing sets and relations as a simple semantic basis for objects and its relationships. We defined a number of mapping rules between UML/OCL and Alloy elements, resulting in equivalent Alloy models from UML class diagrams annotated with OCL invariants. By using this approach, we can leverage to UML/OCL the benefits from the powerful tool support offered by the Alloy Analyzer [4].

Alloy's simple semantics allowed powerful tool support represented by the Alloy Analyzer, which is a constraint solver that finds instances of formulae representing models, backed by an off-the-shelf SAT solver [4]. This approach allows automatic generation of snapshots satisfying model constraints, which can be significantly useful for verifying whether models are over or under-constrained. Similarly, assertions can be made against models, which are checked by exhaustive search for counterexamples refuting the assertions. Due to the undecidability of such analysis, they are parameterized by a scope (provided by the user), which assigns a bound of objects to each entity.

Since the search for a solution is limited by a scope, the absence of an instance does not automatically show that a formula is inconsistent. However, such level of automation can cover significantly more cases than any kind of testing, allowing early identification of bugs in functional requirements of critical systems. More specifically, this analysis is able to generate all valid snapshots of a model within a given scope. The Alloy language and its analysis has been successfully used for modeling critical systems, including air-traffic control [5] and a proton therapy machine [6]. We believe that similar benefits can be achieved by applying this automatic analysis to UML/OCL.

A proposed tool support is closely related [7]. The USE tool offers a useful evaluation of a user-provided snapshot of a model, checking whether this instance satisfies the model invariants. In contrast, the Alloy Analyzer offers solving, which involves searching for instances satisfying a given constraint. The latter may be more effective as an instrument for finding unexpected problems in scenarios where it is unfeasible to supply a representative set of test cases.

The remainder of this paper is organized as follows. Section 2 describes the Alloy language and the underlying tool support. In Section 3, we show the mapping rules for transforming UML class diagrams into Alloy specifications. Section 4 describes how the Alloy's tool support can be useful for analyzing UML class diagrams. Section 5 describes related work, whereas Section 6 presents our conclusions and future work.

2 Alloy

Alloy is a formal modeling language based on first-order logic, allowing specification of – primarily structural – properties in a declarative fashion. In general, models in Alloy are described at a high level of abstraction, ignoring implementation details. With Alloy, one can apply object modeling in a similar fashion to UML class diagrams, with the additional benefit of a simple semantics, allowing automatic analysis. Logical formulae can be used to enforce business rules,

playing a role similar to OCL invariants. In this section, we first discuss the language, and then provide more detail on its automatic analysis.

2.1 The Alloy Language

The language is strongly typed, assuming a universe of objects partitioned into subsets, each of which associated with a basic type. An Alloy model is a sequence of paragraphs of two kinds: signatures, used for defining new types; and formula paragraphs, such as facts and predicates, used to record invariants. Analogous to classes, each signature denotes a set of objects. These objects can be mapped by the relations (associations) declared in the signatures. A signature paragraph may introduce a collection of relations.

As an example, we show an Alloy model for part of the banking system, where each bank contains a set of accounts and a set of customers. An account can only be a checking account. The next Alloy fragment declares four signatures representing system entities, along with their relations:

```
sig Bank {
  accs: set Account,
  custs: set Customer
}
sig Customer {}
sig Account {}
sig ChAcc extends Account {}
```

In the declaration of `Bank`, the `set` keyword specifies that the `accs` relation maps each object in `Bank` to a set of objects in `Account`, exactly as a 1-N association in UML. `ChAcc` denotes one kind of account. In Alloy, one signature can extend another one by establishing that the extended signature is a subset of the parent signature. For example, the set of `ChAcc` objects is a subset of the `Account` objects.

A fact is a formula paragraph. It is used to package invariants about certain sets. Differently from OCL invariants, a fact may not introduce a context for its formulae, allowing expression of global properties on models. Also, fact formulae are declared as a conjunction. The following code introduces a fact named `BankProperties`, establishing general properties about the previously declared signatures and relations.

```
fact BankProperties {
  Account = ChAcc
  all a:Account | lone a.~accs
}
```

The first formula states that every account is a checking account; the second one states that every account is related to at most one bank by `accs`. The `all` keyword is the universal quantifier. The expression `~accs` denotes the transpose of `accs`, while `lone` states that `a.~accs` yields a relation with at most one object.

The join of relations[1]. `a.~accs` is the set of all elements for which there exists an element of `a` related to it by the relation `~accs` (direct relation image). The expression yields the bank in which the `a` account is stored. Figure 1 shows how this model could be represented by a UML class diagram annotated with OCL invariants.

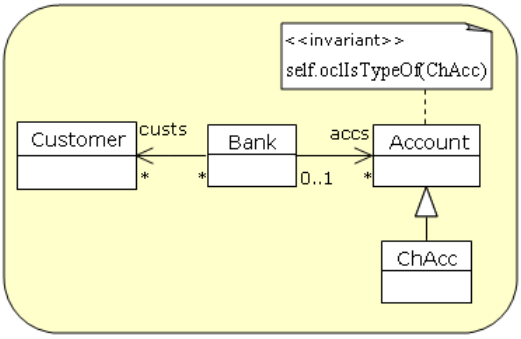


Fig. 1. Banking Analogous UML Class Diagram.

2.2 The Alloy Analyzer

Alloy was simultaneously designed with a fully automatic tool that can simulate models and check properties about them. The tool translates the model to be analyzed into a boolean formula. This formula is transferred to an SAT solver, and the solution is translated back by the Alloy Analyzer into Alloy. The two kinds of analysis consist in binding objects to signatures and relations, searching for a combination of values that make the translated boolean formula true [4].

In particular, simulation generates structures without requiring the user to provide sample inputs or test cases. If the tool finds a binding of objects making the formula true, this binding constitutes a valid snapshot. If we consider, from the previous banking example, the `Bank` and `Account` signatures, along with `accs` relation, a valid snapshot is shown below:

```
Bank = {(B1),(B2),(B3),(B4),(B5)}
Account = {(A1),(A2),(A3),(A4),(A5)}
accs = {(B1,A1),(B1,A2),(B2,A3),(B3,A4),(B5,A5)}
```

In this representation, the Bx and Ax symbols represent Bank and Account objects, respectively. The parentheses are used for tuples (scalar elements are one-element tuples). Notice that every `Account` object is related to at most one

[1] In Alloy, set elements are designed as singleton unary relations.

`Bank` object in `accs`, as modeled in the second formula of the `BankProperties` fact.

On the other hand, assertion checking generates counterexamples - valid snapshots for which an expected property does not hold. The tool searches for a binding of objects which makes true the formula representing the model conjoined with the negated formula of a given logical assertion. A valid snapshot is a counterexample to that assertion. Considering the previous Alloy model, and an assertion stating that each account is associated with one bank (in Alloy: `Account in Bank.accs`), a possible counterexample could be as follows, where the `A3` account is not related to any bank:

```
Bank = {(B1),(B2)}
Account = {(A1),(A2),(A3)}
accs = {(B1,A1),(B2,A2)}
```

The tool does not provide a complete analysis. Instead, it conducts a search within a finite scope chosen by the user, bounding the number of elements in each basic type [4]. For instance, the examples above were analyzed within a scope of five (5). The output is either a snapshot or a message that no snapshot was found in the given scope. When checking a given property, a snapshot is a counterexample and indicates that the asserted formula was not valid. Theoretically, nothing can be inferred when no snapshot is found. However, gradually increasing the scope can give the user a greater confidence during the modeling of critical systems, which helps finding inconsistencies or lack of constraints before implementation. Billions of state combinations for a model's signatures and relations can be covered within a predetermined scope in a matter of seconds, not requiring any input of test cases from the user. This bounded analysis might still constrain models involving numeric types [8].

3 A Semantics for UML Class Diagrams

We offer a semantics for UML class diagrams by translation to correspondent Alloy models. Alloy constructs can represent a number of UML static or dynamic constructs, contributing with a semantics to a subset of UML that may be automatically analyzed. Furthermore, translating a representative subset of OCL expressions to Alloy is straightforward, since they are both defined for expressing constraints in object modeling, based on first-order logic.

In order to define mapping rules between UML and Alloy, we focused on structural properties of class diagrams including OCL invariants, thus exploiting Alloy's expressiveness for modeling complex state properties of a system. For that reason, we did not consider some UML constructs, such as operations (methods) and their effects, besides timing constraints (e.g. {`frozen`}). Although this may seem restrictive, we initially focused on aspects that may be extremely useful in finding problems when modeling complex states. Moreover, Alloy and its supporting tool have also been used for modeling properties over state transitions [6, 9], which shows the language's usefulness in behavioral modeling as well.

We believe that related UML constructs, even other diagrams, can be similarly analyzed by means of analogous mapping.

Regarding OCL, we only consider class invariants, due to the reasons explained above. Since recursive operations have an undefined semantics [10], we do not deal with those in our semantics. Also, numeric types (except integer) are ignored, due to the nature of Alloy's analysis, which is scope-limited. Nevertheless, the latest version of the language introduces an improved support for integer types [8]. Likewise, string type and operations, loop constructs or packaging operations are not considered, since they are implementation-specific (we are primarily concerned with abstract models). The keyword `self` is considered mandatory when expressing context-dependent invariants. We also require role names for each navigable end in binary associations, which simplifies the translation.

Our translation rules are divided into two categories: from UML diagrammatic constructs to Alloy constructs and from OCL invariants to logically-equivalent Alloy formulae. Most UML diagrams do not offer an agreed precise semantics, due to its broad applicability in unlimited modeling contexts [11]. We adapted class diagrams to Alloy semantics (sets of objects with relations as fields), with the purpose of modeling abstract structural aspects of critical systems. Regarding OCL constraints, we based our translation on the OCL specification version 1.5 [10]. Basic set theory and predicate calculus guided the translation rules, neglecting OCL formulae that may present undefined semantics (such as recursion).

Figure 2 depicts a class diagram describing an extended version of the banking system. The invariant over `Customer` states that a customer identifier must be unique, while the invariant over `Account` states that it is an abstract class.

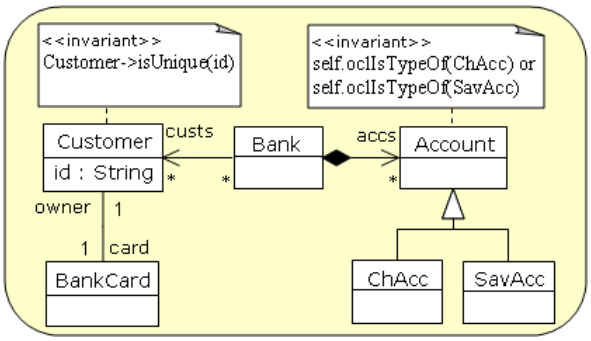


Fig. 2. Extended Class Diagram for the Banking System.

Regarding diagrammatic constructs, classes and interfaces are translated to signatures in Alloy. In addition, binary associations are translated to relations

declared with the `set` qualifier. Similarly, attributes also translate to relations. In our example, `Customer` and `BankCard` can be represented in Alloy as follows:

```
sig Customer {
  card: set BankCard,
  id: set String
}
sig BankCard {
  owner: set Customer
}
```

A relation is created for each navigable association end, using the opposite role name. This rule is required due to the limitation in representing navigability constraints with binary relations in Alloy (a binary relation is bidirectional by definition). In case we have two navigable ends for an association, each signature declares a relation for its opposite navigable end, as exemplified by `owner` and `card`. A constraint is added, stating that a relation is the transpose of its opposite counterpart. Furthermore, multiplicity constraints, such as one (1) for `card` and `id`, are expressed as formulae over signatures' relations. The composition between `Bank` and `Accounts` is transformed into a constrained binary relationship. The added multiplicity constraint ('1') on `Bank` represents coincident lifetimes between a bank and its accounts, in addition to forbidden sharing of an account between different banks. These constraints will be all within the `BankProperties` fact. The following Alloy fragment describes this fact.

```
fact BankProperties {
  card = ~owner
  all c:Customer | #(c.card) = 1
  all a:Account | one a.~accs
  ...
}
```

The `#` symbol is the cardinality operator. For example, the second formula in `BankProperties` states that there is exactly one object of `BankCard` mapped by each customer. In addition, generalization is translated to Alloy's `extends`. The subsequent Alloy fragment shows `SavAcc` translated signature:

```
sig SavAcc extends Account {}
```

OCL invariants are translated to equivalent Alloy formulae, being universally quantified on `self`. This limits expression of global properties in OCL, since universally quantified formulae can be true either if the formula is valid or the quantified set is empty. Even though Alloy allows global properties without quantification, we do not intend to fix the problem, as we provide a semantics for OCL.

Next, we show an Alloy fragment that translates the invariants from the `Account` and `Customer` contexts, which will be defined within `BankProperties`. The `oclIsTypeOf` operation is translated to set membership and `isUnique` to an equivalent quantified formula.

```
all self: Account | (self in ChAcc) || (self in SavAcc)
all self: Customer | all disj c,c':Customer | c.id != c'.id
```

The `||` operator corresponds to logical disjunction, while `in` denotes set membership and `!` denotes negation. The `disj` keyword states that the declared variables are distinct. As an example, Table 1 formulates these and aditional mapping rules for OCL expressions, where `X,Y` denote collections, `P,Q` denote logical formulae, `a` and `r` denotes a variable and a attribute, respectively.

Table 1. Examples of Mapping Rules from OCL to Alloy.

OCL	Alloy
`X.oclIsTypeOf(Y)`	`X in Y`
`X->includes(Y)`	`Y in X`
`X.allInstances`	`X`
`X.isUnique(r)`	`all disj a,a':X | a.r != a'.r`
`P or Q`	`P || Q`
`P and Q`	`P && Q`
`X->isEmpty()`	`no X`
`X->exists(a|P)`	`some a:X | P`
`X->forAll(a|P)`	`all a:X| P`
`X.size()`	`#X`

Currently, the translation is performed systematically, but manually. Nevertheless, the process was designed to be decidable, allowing automatization. Both languages can be defined by meta-modeling, and transformations can be implemented as a correspondence between meta-model elements, using an approach similar to OMG's Model-Driven Architecture [12]. XML-based representations, such as XMI [13], can certainly help obtaining Alloy paragraphs from UML classes and OCL invariants. Furthermore, OCL constraints can be translated into Alloy formulae as source-to-source transformations, involving a parser for OCL and manipulation of abstract syntax trees. Having an automatic translator from UML to Alloy, constraint solving from the Alloy Analyzer can be provided for a subset of UML class diagrams, as explained in the next section. As the last step of the process, the results yielded by the Analyzer must be transformed back to UML. They could be appropriately represented by object diagrams [1], by integrating instances yielded by the Alloy analyzer with UML CASE tools, such as Rational Rose [14].

4 Class Diagram Analysis

In this section, we present how our approach could be used by developers to improve modeling, adding reliability to the development of critical systems. First, we present, through our banking example, some of the benefits of constraint

solving for UML class diagrams. Next, we describe a number of case studies using Alloy and its tool support on modeling of critical systems, which could apply to UML/OCL by our approach.

4.1 Example

Considering the class diagram depicted in Figure 2, a number of unspecified properties could be uncovered by constraint solving in a straightforward way. For instance, if we translate this diagram into an Alloy model with same meaning, by the mapping rules given in Section 3, the simulation of the resulting model may yield the snapshot depicted in Figure 3 (represented as a UML object diagram).

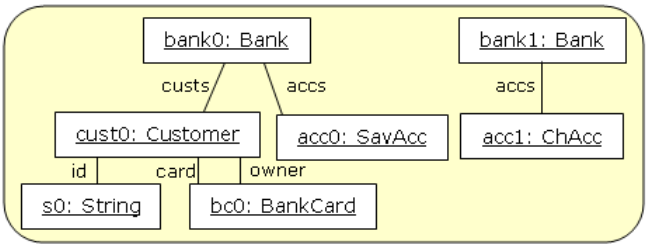


Fig. 3. First Generated Snapshot.

This simulation uses a scope of at most two objects for each entity. The generated snapshot shows that customers and their personal accounts are not related within one bank. Assuming that this is a functional requirement, the model should be modified in order to accommodate the required relationship. The modeler could, for instance, add a directed binary association between these two classes (as depicted in Figure 4), with no multiplicity constraints, then executing a new simulation. A possibly generated snapshot for the new model is showed in Figure 5.

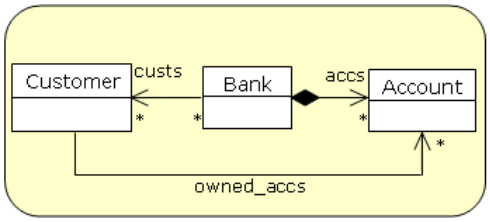


Fig. 4. Modification to the Banking Class Diagram.

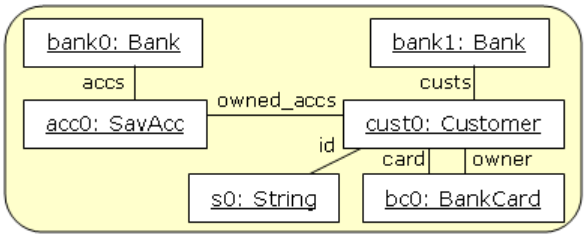


Fig. 5. Second Generated Snapshot.

Although customers and accounts are related, the model allows a snapshot where a customer and its (savings) account are within distinct banks. Assuming that it is a undesirable situation, the modeler can then include the following OCL invariant to the `Bank` context:

```
context Bank inv customersAccountsInBank:
  self.custs.owned_accs->includes(self.accs)
```

which, according to the mapping rules, translates to Alloy:

```
all self:Bank | self.accs in self.custs.owned_accs
```

The translated model possibly yields the snapshot depicted in Figure 6. A number of similar executions, with gradually greater scopes, can increase confidence on the modeled properties.

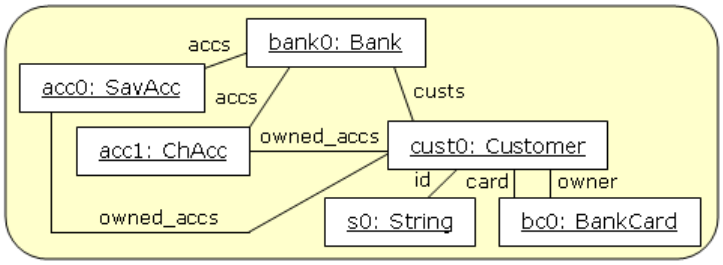


Fig. 6. Third Generated Snapshot.

Additionally, our approach can help identification of over-constrained class diagrams, usually leading to inconsistent models that are not implementable. In the resulting class diagram from the previous example, a functional requirement could be added to the system, allowing customers temporarily without a bank card to be registered into any bank. If we add the following OCL constraint to the `Customer` context:

```
context Customer inv customersNoBankCard:
  Customer.allInstances->exists(c | c.card->isEmpty())
```

No snapshots are found by the Analyzer for the translated Alloy model, even if we increase the scope. This result is due to the newly-added constraint, which contradicts with the multiplicity constraints between `Customer` and `BankCard` objects, stating the existence of an one-to-one correspondence. One of the two constraints must be removed in order to include the desired property to the model.

Finally, one may want to investigate the relationship between `BankCard` and `Account` objects. For a tool implementing our approach, users should be able to enter OCL formulae by means of a dialog box, which may be similar to the USE tool [7]. These formulae would be translated to Alloy assertions, then passed on to the Alloy Analyzer as input. In our example, the following OCL formula could be checked against the diagram:

```
context Customer inv cardsAndAccountsAssertion:
  self.card->notEmpty() implies self.owned_accs->notEmpty()
```

This assertion tests whether every customer that has a card owns at least one account, which could be a new functional requirement. The analysis yields a counterexample, depicted in Figure 7. The snapshot denotes a state with one customer owning a card and no accounts, refuting the assertion. Further changes can fix this requirement, possibly a new constraint relating accounts and customers' bank cards.

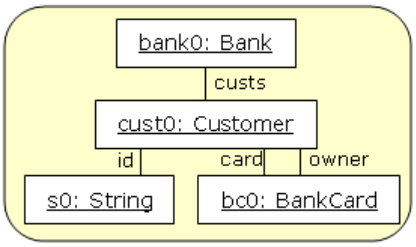


Fig. 7. Generated Counterexample.

Besides showing how our approach can be useful for finding problems in class diagrams, this example shows an additional benefit of this automatic analysis: incremental modeling. A diagram can be written, along with some constraints. Simulation can then give feedback to the modeler regarding the current status of the constraints, indicating the next changes that will accommodate the desired functional requirements. In the same way, assertions offer greater confidence that the model follows those requirements adequately.

4.2 Applications of Alloy and Constraint Solving to Critical Systems

The banking example was used as a simple illustration of our ideas. However, Alloy has been used as a useful approach for finding problems in safety-critical systems. These examples are applicable in the context of this paper, due to the semantics proposed for class diagrams and OCL constraints.

In a recent case study, Alloy was used for specifying a system controlling a radiation therapy machine [6]. This machine produces beams of photons for treating a number of diseases. The system was previously modeled in UML/OCL, and its translation to Alloy uncovered a number of errors. Automatic analysis helped verify whether machine operations were commutative (the order of execution do not affect the final result). It was observed that several pairs of operations did not commute, revealing potential problems during system activity.

Related examples of applying Alloy to critical systems include modeling of access control for information systems [15] and a railway system [9]. In the latter, an Alloy model was built for checking the policy for controlling the motion of trains of the Bay Area Rapid Transit (BART). The analysis was performed in order to check the existence of any condition (state configuration) which could cause train collision.

In addition, an air-traffic control system build by NASA has been modeled using Alloy, which explored new ideas for the design and analysis of such systems [5]. These applications show the potential use of constraint solving and Alloy for understanding critical systems, which can lead to the early discovery of bugs in modeling.

5 Related Work

Our approach allows generation of snapshots and counterexamples to claims over UML class diagrams and OCL invariants, by means of the Alloy Analyzer tool [4]. Related tool support have been developed for similar purposes [16, 17], although allowing syntax and type checking for OCL formulae. In particular, the USE tool [7] also makes OCL machine-analyzable. It contains a useful and easy-to-use model animator, which can uncover bugs by checking snapshots against UML class diagrams constrained with OCL. Our approach differs from USE in the sense that in the latter modelers must provide the verified test cases. Using our approach, snapshots can be automatically generated within a given scope, becoming amenable to automatic simulation. Besides, this analysis allows assertion checking by searching a snapshot that refutes the asserted property (which could be provided as an OCL formula by the user).

Regarding the semantics we provided for UML/OCL, Bordeau and Cheng [18] define a similar approach for a related modeling notation. They automatically map models to algebraic specifications, allowing formal reasoning on the semantics of the translated specification. In contrast, Alloy admits a more direct mapping from UML, since both are similarly suitable to object modeling, as

reported by another work [6]. Also, automatic simulation and analysis in Alloy may be more appealing to software architects and designers. In other related approach [19], a systematic approach for translating UML class diagrams with OCL constraints is provided. The translation rules are similar to ours, although they use an additional intermediate language for the translation.

There have also been a number of efforts on proposing formal semantics for UML and related modeling languages, in order to clarify the meaning of its diagrammatic constructs, supporting tool development. For example, related approaches [20, 21] give a formal semantics to a subset of UML class diagrams. We defined a semantics for UML class diagrams and OCL invariants with the specific purpose of leveraging Alloy's automatic analysis to UML.

6 Conclusion

In this paper, we have proposed an approach for automatic analysis of UML class diagrams, according to a semantics based on Alloy, employing the capabilities offered by its analysis tool. This approach provides an option for automation regarding UML/OCL, which can be useful for early identification of problems in critical systems' models, including under- and over-constrained models and undesired properties, which may compromise correctness.

The analysis provided by the Alloy Analyzer is sound but not complete. Since it is based on a user-provided scope, if no snapshot is found, nothing can be inferred (same with counterexamples for assertions). However, small scopes may suffice for improving confidence in modeling and finding relevant problems [22]. Furthermore, the analyzer can generate all possible snapshots for a model within the scope, not requiring any user input. Therefore, many more states can be covered than any testing activity. In this context, Alloy was used as a test case generator for Java programs [23]. We believe that our approach can leverage the benefits of this analysis to UML class diagrams, improving modeling of critical systems using UML. Although the support for numeric types is limited in Alloy, in order to make bounded analysis feasible, a wide range of complex structural properties can still be modeled and analyzed. Such analysis may not even be possible in class diagrams using all features from UML and OCL specifications.

It may be argued that constraint solving might be applied to UML directly, without the need of Alloy as an intermediate language. For example, this can be done through an analysis tool that maps UML class diagrams to a SAT solver, allowing simulation and analysis. However, due to their complex and unresolved semantics, we would still need to map UML and OCL to a formal language, whose boolean formulae are given as input to a SAT solver. For effective analysis, this language should also be sufficiently simple to allow scope-limited analysis over classes and associations. In this context, Alloy is a reasonable choice, since it was developed for constraint solving. Furthermore, we have on-going work on defining an equivalence notion for object models in Alloy, which support the specification of semantics-preserving transformations and refactorings for models [24]. These results can be directly leveraged to UML by our translation to Alloy.

Alloy provides a simple semantics to UML class diagrams and OCL invariants. However, Alloy cannot represent implementation-oriented UML constructs. For instance, attributes are mapped to simple binary relations, disregarding properties such as visibility and default value. Nevertheless, the chosen subset is representative to a number of problems that can be abstractly modelled. Similarly, our approach regards only a subset of UML at first, since Alloy was designed to primarily model structural properties. However, recent work [6, 9] shows use of Alloy for modeling behavioral properties. The extension of our semantics on that topic is considered as future work.

The automation of our approach is an on-going work. The mapping rules between UML/OCL and Alloy can be implemented using well-founded transformation techniques. Analysis is performed on the resulting Alloy model, and the results can be presented as UML object diagrams. This tool support may be integrated to leading UML tools, such as Rational Rose [14], aiming at straightforward application to UML class diagrams with OCL annotations.

7 Acknowledgements

This work benefitted from comments by members of the Software Productivity Group. Also, we give special thanks to Augusto Sampaio, Rodrigo Ramos and Franklin Ramalho for their valuable comments. In addition, we would like to thank all the anonymous referees, whose appropriate suggestions helped improving the paper. This work was partially funded by CAPES and CNPQ, grant 521994/96-9.

References

1. Booch, G., et al.: The Unified Modeling Language User Guide. Object Technology. Addison-Wesley (1999)
2. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Second edn. Addison-Wesley (2003)
3. Jackson, D.: Alloy: A Lightweight Object Modelling Notation. ACM Transactions on Software Engineering and Methodology (TOSEM) **11** (2002) 256–290
4. Jackson, D., Schechter, I., Shlyahter, H.: Alcoa: the Alloy Constraint Analyzer. In: Proceedings of the 22nd International Conference on Software Engineering, ACM Press (2000) 730–733
5. Dennis, G.: TSAFE: Building a Trusted Computing Base for Air Traffic Control Software. Master's thesis, MIT (2003)
6. Dennis, G., Seater, R., Rayside, D., Jackson, D.: Automating Commutativity Analysis at the Design Level. In: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. (2004) 165–174
7. Richters, M., Gogolla, M.: Validating UML Models and OCL Constraints. In Evans, A., Kent, S., Selic, B., eds.: UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings. Volume 1939 of LNCS., Springer (2000) 265–277
8. Jackson, D.: Alloy 3.0 Reference Manual. Available at http://alloy.mit.edu/beta/-reference-manual.pdf (2004)

9. Jackson, D.: Railway Safety. Available at http://alloy.mit.edu/contributions/-railway.pdf (2002)
10. Object Management Group: OMG Unified Modeling Language Specification Version 1.5 (2003)
11. Gogolla, M., Radfelder, O., Richters, M.: A UML semantics FAQ - the view from bremen. In Kent, S.J.H., Evans, A., Rumpe, B., eds.: Proc. ECOOP'99 Workshop UML Semantics FAQ, University of Brighton (1999)
12. Kleppe, A., Warmer, J., Bast, W.: MDA Explained: the Practice and Promise of The Model Driven Architecture. Addison-Wesley (2003)
13. OMG: XML Metadata Interchange (XMI) Specification (2001) OMG Document formal/02-01-01.
14. Rational Software: Rational Rose XDE Developer (2004) http://www-306.ibm.com/software/awdtools/developer/rosexde/.
15. Zao, J., Wee, H., Chu, J., Jackson, D.: RBAC Schema Verification Using Lightweight Formal Model and Constraint Analysis. Submitted to SACMAT 2003 (2002)
16. Tigris.org: ArgoUML (2004) http://argouml.tigris.org/.
17. Klasse Objecten: Octopus: OCL Tool for Precise Uml Specifications (2004) http://-www.klasse.nl/ocl/octopus-intro.html.
18. Robert H. Bourdeau and Betty H. C. Cheng: A Formal Semantics for Object Model Diagrams. IEEE Transactions on Software Engineering **21** (1995) 799–821
19. Lano, K., Clark, D., Androutsopoulos, K.: UML to B: Formal verification of object-oriented models. In Boiten, E.A., Derrick, J., Smith, G., eds.: Proceedings, International Conference on Integrated Formal Methods, Canterbury, UK. Volume 2999 of LNCS., Springer (2004) 187–206
20. Clark, T., Evans, A., Kent, S.: The Metamodelling Language Calculus: Foundation Semantics for UML. In Hussmann, H., ed.: Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001, Genova, Italy, April 2-6, 2001. Volume 2029 of LNCS., Springer (2001) 17–31
21. Evans, A., France, R.B., Lano, K., Rumpe, B.: The UML As a Formal Modeling Notation. In Bézivin, J., Muller, P.A., eds.: The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998. Volume 1618 of LNCS., Springer (1999) 336–348
22. Khurshid, S., Marinov, D., Jackson, D.: An Analyzable Annotation Language. In: Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, ACM Press (2002) 231–245
23. Marinov, D., Khurshid, S.: TestEra: A Novel Framework for Automated Testing of Java Programs. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering, IEEE Computer Society (2001) 22
24. Gheyi, R., Borba, P.: Refactoring Alloy Specifications. In Cavalcanti, A., Machado, P., eds.: Electronic Notes in Theoretical Computer Science, Proceedings of the Brazilian Workshop on Formal Methods. Volume 95., Elsevier (2004) 227–243

Rigorous development of reusable, domain-specific components, for complex applications*

I. Johnson[1], C. Snook[2], A Edmunds[2] & M. Butler[2]

[1]AT Engine Controls Ltd., Portsmouth, UK.
[2] University of Southampton, Southampton, UK.

Abstract. The reuse of reliable, domain-specific software components is a strategy commonly used in the avionics industry to develop safety critical airborne systems. One method of achieving reuse is to use domain specific languages that map closely onto abstractions in the problem domain. While this works well for control algorithms, it is less successful for some complex ancillary functions such as failure management. The characteristics of device failures are often difficult to predict resulting in late requirements changes. Hence a small semantic gap is especially desirable but difficult to achieve. Object-oriented design techniques include mechanisms, such as inheritance, that cater well for variations in behaviour. However, object-oriented notations such as the UML lack the precision, and rigor, needed for safety critical software. UML-B is a profile of the UML for formal modelling. In this paper we show how UML-B can be used to model failure management systems via progressive refinement, and indicate how this approach could utilise UML concepts to cope with high variability, while providing rigorous verification.

1 Introduction

Developers in the avionics industry are interested in the use of object-orientated technology (OOT) [1, 2] as a way to increase productivity. In particular, concepts, such as inheritance and design patterns, enable more flexible reuse of software. The emergence of the UML [3] as the de-facto standard modelling language for object-oriented design and analysis, and the subsequent development of supporting tools, has promoted the modelling and design of applications in the UML. Due to concerns over safety certification issues, OOT has not seen widespread use in avionics applications. One reason for this is that the controlling standards used in the industry, such as RTCA DO-178B [6] and its European equivalent, EUROCAE ED-12B [7], do not consider OOT. In order to address this, a new version of the standard, DO-178C/ED 12C, is planned. A significant contribution to this new standard will come from the findings of the Object Orientated Technology in Aviation (OOTiA) initiative [8], which was set up by the FAA and NASA to develop guidelines for the safe use of object-oriented technology in avionics software development.

* This research is being carried out as part of the EU funded research project: IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

Application development based on a combination of UML and formal methods will improve safety and provide flexibility in the design of software for aviation certification. The combination of UML and formal methods at an abstract modelling stage will enable the reuse of reliable software components both at the specification and code levels. We indicate how we can exploit the reuse features of the UML and the reliability provided by formal methods. The development process will benefit from a reduction in the semantic gap by defining a vocabulary of entities that maps closely onto abstractions in the problem domain. UML class diagrams assist greatly with this, especially in complex application domains where the use of features such as inheritance caters for variation of subtypes. The use of formal methods to address the rigorous verification required for safety critical applications has been advocated before [5] but adoption within industry has been limited partly due to the need for industrial strength tool support. One formal method that has been developed for practical use in industry and enjoys good tool support is the B method [9].

1.1 B and B tools

The B method is based on a set theoretic approach and provides the ability to perform rigorous proof, thus ensuring a self-consistent specification. The B method's Abstract Machine Notation (AMN) is used to describe the state and behaviour. Under-specification in assignments or choices is possible via non-deterministic constructs, which must be resolved in later refinements prior to implementation. An invariant clause describes properties of the system that must hold at all times. The B verification tools [11] generate proof obligations (POs) and then attempt to automatically discharge (prove) them. Invariably there are a number of PO's that remain to be discharged semi-automatically using the interactive prover [10]. The user guides the proof by suggesting strategies and sub-goals. Discharging POs with the interactive prover often leads to a greater insight into the specification and inaccuracies can be identified and addressed at an early stage of development. Discharging proof obligations can be difficult and time consuming, but once complete the specification is known to be self-consistent. A model checker, ProB [13], that searches for deadlocks and invariant violations may be used as a preliminary verification of the specification before commencing proof. Refinements are related to the previous level of abstraction in such a way that a valid refinement always satisfies the abstract specification. Proof provides confidence that the refinement is not only self-consistent but also reflects the behaviour of the abstract specification it refines.
Event B [12] is a derivation of the original B Method that uses the notion of predicate guards that enable or disable events. The event driven approach of Event B begins with the abstraction of the observable events that 'may' occur in a system. The abstraction is refined in a number of steps, adding new events and state information at each iteration. The aim is to move towards a consistent model, with enough detail to fully describe the behaviour of the system. There are a number of additional requirements for the event driven approach, firstly, the added events of a refinement are not allowed to permanently take control of the system so that the events of the more abstract model are eventually enabled; secondly, a concrete event must not be enabled more often than its abstract counterpart; but the abstract event must not be

enabled more than the disjunction of the concrete event together with the new events. That is, the abstract event is replaced by a sequence of new events culminating in the refined event.
Validation of specifications is just as important as verification. ProB [13] can also be used to animate B machines. The list of currently enabled operations is displayed in a pane. The current state of variables is displayed in another pane. The user may choose sequences of enabled operations in order to explore the behaviour of the specification.

2 Failure management

A common functionality required of many systems is to manage failure of its inputs. This is particularly pertinent in aviation applications where lack of tolerance to failed system inputs could lead to loss of aircraft. The role of failure management in an embedded control system is shown in Fig.1.

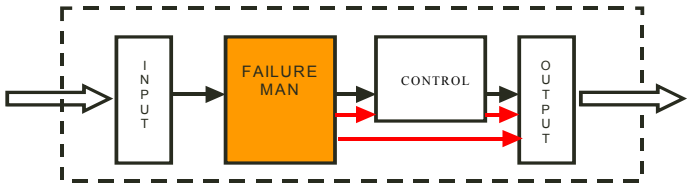


Fig. 1. Context diagram for failure management subsystem

The inputs are tested and, if good, are passed unaltered to the control subsystem; otherwise the failure of the input is managed. This may involve substituting values, and taking alternative actions. There are two aspects to the subsystem; detection and remedial action. Failure detection involves checking for input validity, including out of range checks, rate of change checks, and comparison with other conditions in the system. A failed condition must persist for a period of time before a failure is confirmed. If the invalid condition is not confirmed the input recovers and is used again. When setting the persistence conditions for confirmation of a failure, a balance must be sought between achieving a fast response to failures and over sensitivity to spurious interference. Once a failure is confirmed it is latched until power is reset. Remedial actions vary, depending on the input's function and importance within the system, and the state of the system when the failure occurred. Temporary remedial actions, such as relying on the last good value, or suppressing control behaviour, may be taken while a failure is being confirmed. Once a failure is confirmed, more permanent actions are taken such as switching to an alternative source, altering or degrading the method of control, engaging a backup system or freezing the controlled equipment. Tables 1 and 2, show some typical failure management activities.

Table 1. Detection

Signal	High	Low	Rate	Conditions for test
ESa	120%	0%	100%/sec	Engine Stood
	120%	10%	100%/sec	Engine Starting
	120%	50%	100%/sec	Engine Running
ESb	120%	0%	100%/sec	Engine Stood
	120%	10%	100%/sec	Engine Starting
	120%	50%	100%/sec	Engine Running
ESa - ESb	5%	-5%	-	ESa or ESb >30%
ESa – Engine speed (main input) ESb – Engine speed (alternative input)				

Table 2. Remedial Actions

Signal	Procedure	code
ESa	Select ESb if available	ES1
	else Switch to backup system	
ESb	ESb not available	ES2
ESa/ESb diff	Use highest value sensor	ES3

Experience has shown that failure management systems can be difficult, and expensive, to develop and maintain due to their complexity and vulnerability to change. Changes often occur late in the development cycle, since requirements are redefined based on empirical performance under failure conditions. The semantic gap between control algorithm design notations and coding constructs has been addressed successfully by the development of domain specific languages. Unlike control algorithms failure management has no successful domain specific language. The nature of failure management is that different control actions and behaviour are required, dependent on the outcome of conditional logic for each of many inputs; this can result in complex overall behaviour. Failure management can become functionally complex due to the following: the rate of decay of an input depends on sensor device characteristics, the application of a test depends on engine and input conditions, a test may depend on the sampling rates of inputs, a test may vary according to outcomes of other tests, the detection of a failure may require hysteresis to avoid oscillation, the sequence of tests may depend on temporal interdependence of input sampling, remedial actions depend on the system state.

One approach, to improve flexibility, is to model a failure management subsystem using the UML; this will improve configurability and, if combined with formal methods to ensure consistency, will be particularly suited to the safety critical applications found in aviation. Modelling functional behaviour will provide the ontology to convey functional understanding and, through formal techniques, provide a way to map this to the code, reducing the semantic gap.

3 Overview of the UML-B profile and U2B translator

The UML-B [17] is a profile of UML that defines a formal modelling notation. It has a mapping to, and is therefore suitable for translation into, the B language. UML-B consists of class diagrams with attached statecharts, and an integrated constraint and action language based on the B AMN notation. The UML-B profile uses stereotypes to specialise the meaning of UML entities to enhance correspondence with B concepts. The profile also defines tagged values (UML-B clauses) that may be attached to UML entities to provide formal modelling details that have no counterpart in UML. Several styles of modelling are available within UML-B. Here we use its event systems mode, which corresponds with the Event B modelling paradigm. In event systems mode, UML operations represent spontaneous events. Since events are parameterless, operation parameters represent non-deterministic selection of local variables. UML-B provides a diagrammatic, formal modelling notation based on UML. It has a well defined semantics, as a direct result of its mapping to B. There are barriers to the acceptance of formal methods in industry. The popularity of the UML enables UML-B to overcome some of these barriers. Its familiar diagrammatic notations make specifications accessible to domain experts who may not be familiar with formal notations. UML-B hides B's infrastructure, it packages mathematical constraints and action specifications into small sections, each being set in the context of its owning UML entity. The U2B [18] translator converts UML-B models into B components (abstract machines and their refinements). Translation from UML-B into B enables verification and validation tools to be utilised.

In many respects B components resemble an encapsulation and modularisation mechanism suitable for representing classes. A component encapsulates variables that may only be modified by the operations of the component. However, to ensure compositionality of proof, B imposes restrictions on the way variables can be modified by other components (even via local operations). Translating classes into B components imposes corresponding restrictions on the relationships between classes. Therefore we translate a complete UML package (i.e. many classes and their relationships) into a single B component. This option allows unconstrained (non-hierarchical) class relationship structures to be modelled. Since the B language is not object-oriented, class instances must be modelled explicitly. Attributes and associations are translated into variables whose type is a function from the class instances to the attribute type or associated class. For example a class A with attribute x of type X would generate the following B:

```
SETS          A
VARIABLES     x
INVARIANT     x : A --> (X)
```

Operation behaviour may be represented textually in a notation based on B, as a state chart attached to the class, or as a simultaneous combination of both. Further details of UML-B are given in [17]. Examples of previous case studies using UML-B and U2B are given in [14,15,16 and 19].

4 UML-B model of failure management

As an example of using UML-B to develop failure management systems we show a simplified model and its verification. Our first abstract model captures the overall states of the system. in subsequent refinements we model the stages in confirming a failure, the mechanism for freezing the system and the relationship between individual inputs and the collective state of the system. In these early stages we leave many aspects of the system under specified, saying only, for example, that an input may be detected as an unconfirmed failure and then may either recover or become a confirmed failure; but saying nothing about how or why these choices are made. Despite this (non-deterministic) under-specification the model embodies important properties about the interaction of the states of inputs that we verify by proof. In practice, inputs have differing levels of sensitiveness and importance to the control system operation. However, to simplify the example we only consider inputs to which the controller is sensitive (i.e. freezes while a failure is unconfirmed) and can not continue to control without (i.e. hard fault).

4.1 Machine fman_a

This first abstract model of failure management considers the overall state of the system. It defines the three main states of the controller in response to input validity conditions, which are; a) normal operation, b) frozen while attempting to confirm a possible input error, and c) hardfaulted when an input error has been confirmed. Note that once the system has hard faulted no further events may occur (the model is intentionally deadlocked). The following state diagram is attached to a class utility (within the fman_a <<machine>> package) and hence represents a simple variable.

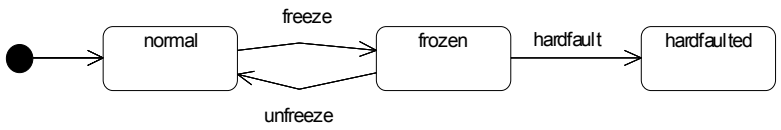


Fig. 2. Statechart diagram of the abstract machine

Since this level is a simple expression of the permitted transitions between the three states of the system, the only verification is that there are no other states. The B produced by U2B for this model is shown in the appendix.

4.2 Refinement fman_r1

In this refinement we recognise that the system state is actually an abstraction of the states of many instances of input failure management (Fig.4). Each input has three possible states; `ok`, `suspect`, and `confirmed`. Each input can have a `good` event (corresponding to a valid input value being detected) or a `bad` event (when an invalid

value is detected). Some of these `good` and `bad` events (depending on the state of the full collection of inputs) refine the `freeze`, `unfreeze` and `hardfault` events from the abstract model. These are `first_bad` (the first input to enter the suspect state), `last_good` and `confirm` respectively. When an input has confirmed detection of an invalid value, a guard on each transition prevents further events from being enabled. This models the intentional deadlock in the abstract model. The refinement relation (Fig. 6.) specified in a `REFINEMENT_RELATION` clause attached to the package, `fman_r1`, gives the correspondence between the equivalent states of the two models. The system is `normal` when no inputs have detected invalid values (`confirmed` or `suspect`), `frozen` when at least one input has detected an invalid value but no inputs have been confirmed invalid, and `hardfaulted` when an input has detected and confirmed an invalid value. Note that we use a 'Petri' style interpretation of the state model (where each state is a variable whose value is the set of instances in that state) since this makes it easier to specify the collective state of the class in transition guards. Verification proves that the collective state of the inputs behaves in accordance with the overall system states; `normal`, `frozen` and `hardfaulted`. The B produced by U2B for this model is shown in the appendix.

Fig. 3. Class diagram of the first refinement

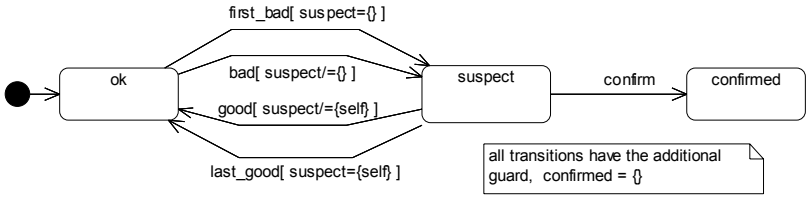


Fig. 4. Statechart diagram of the INPUT class

```
REFINEMENT_RELATION
((control_state=normal) <=> (ok=INPUT)) &
((control_state=frozen) <=> (suspect/={} & confirmed={})) &
((control_state=hardfaulted) <=> (confirmed/={}))
```

Fig. 5. Refinement relation between abstraction and first refinement.

Table 3. - Event refinement in first refinment

event in refinement r1	**refines event in abstraction a**
first_bad	freeze
Bad	(new event)
first_good	unfreeze
Good	(new event)
Confirm	hardfault

4.3 Refinement fman_r2

In this refinement we introduce the idea that each input consists of several tests and each test is in a passed, failed or latched state. This is modelled as a class `TEST` which has a state model, and an association to its owning input. `INPUT` no longer has any events or a state model. Its state is derived from its associated collection of `TEST`s. The state of an input is `ok` when all its tests are `passed`, `confirmed` when one of its tests is `latched` and `suspect` otherwise.

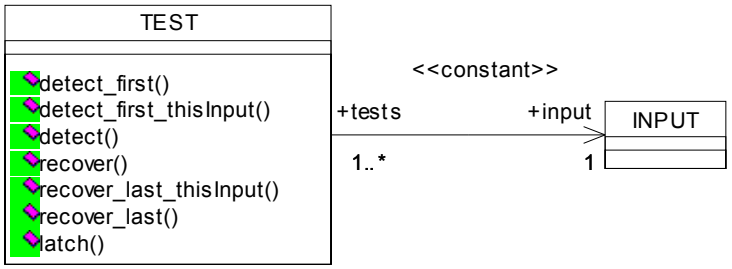


Fig. 6. Class diagram of the second refinement

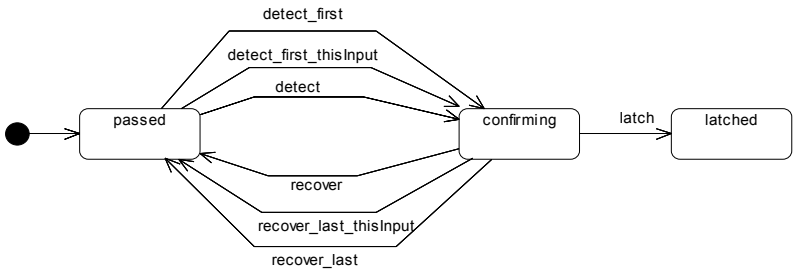


Fig. 7. state diagram of the TEST class

The guards for the transitions (events) are given in Fig 9. (Where `a|>s` restricts the association `a` to links whose targets belong to the set `s`. E.g. `(tests |> confirming)[{i}]` is the set of instances of `TEST` that are associated with the input, `i`, that are in the state `confirming`).

```
detect_first:       latched={} & confirming={}
detect_first_thisInput: latched={} & confirming/={} &
                                (tests|>confirming)[{input}]={}
detect:             latched={} &

          (tests|>confirming)[{input}]/={}
recover_last:       latched={} & confirming={self}
recover_last_thisInput: latched={} & confirming/={self} &

          (tests|>confirming)[{input}]={self}
recover:          latched={} &

          (tests|>confirming)[{input}]/={self}
latch:              latched={}
```

Fig. 8. Guards on events in second refinement

The refinement relation defines the set of inputs in each of the r1 level states based on the state of its collection of tests. (where `s<<|a` restricts the association a to links whose source do not belong to the set `s`).

```
REFINES fman_r1
REFINEMENT_RELATION
          ok =       ran(confirming\/latched<<| input) &
          suspect =     ran(confirming-latched <| input) &
          confirmed =ran(latched <| input)
```

Fig. 9. Refinement relation between first and second refinement.

The events of the class `INPUT` are re-specified as events of the class `TEST` and in terms of the conditions of the collection of tests belonging to the input. Two new events, `detect` and `recover`, are added to model the transitions of subsequent tests detecting failures when another test on the same input has already done so. These new events had no effect in the previous level of refinement.

Table 4. Event refinement in second refinement

event in refinement r2	**event in refinement r1**	**event in abstraction a**
detect_first	first_bad	freeze
detect_first_thisInput	bad	-
detect	-	-
recover_last	last_good	unfreeze
recover_last_thisInput	good	-
recover	-	-
latch	confirm	hardfault

4.4 Adding further details to Test

In subsequent refinements we introduce further detail to the model in many stages that we summarise here. This includes events and a counter attribute for confirming (or recovery of) a test. We add a parameter, `pval`, and a corresponding `limit` for deciding when `detect`, `recover` and `latch` events occur for a particular test. Having verified the previous refinement stages we no longer need to distinguish separate events for the differing conditions when a test detects an invalid input. We therefore merge `detect`, `detect_first` and `detect_first_thisInput` using a three branch guarded choice (SELECT substitution). Subsequently we also merge in the confirmation counting events and latching event with further guarded branches of the same event. In this way the correspondence of actions taken under different conditions is verified to represent the abstract event model before being merged into a conditional single event, as the model is refined towards an implementation. The B produced by U2B for the merged operation, `test`, is shown in the appendix.

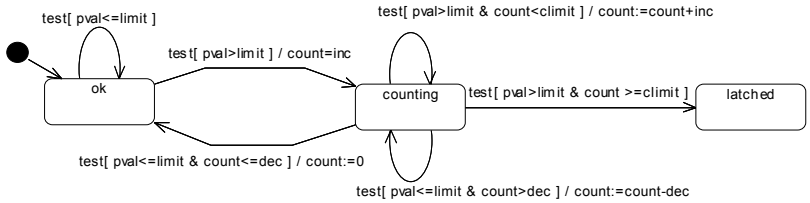


Fig. 10. Statechart diagram of refinement with counting and merged events

4.5 Defining subtypes of Test

Having rigorously developed a generic test class this can be specialised in further refinements to perform several sub-types of test, such as magnitude tests, rate of change tests and difference comparison tests. There are now three subclasses of `TEST`. `TEST` is an abstract class having no instances other than those that belong to one of its subclasses. The inheritance is modelled in the B produced by U2B as disjoint subsets of `TEST`. `MAG` represents a magnitude test that reuses the behaviour of its superclass. `DIFF` is a test that compares the associated input with another input (represented by association, `comp`). It overrides the test event by 'calling' the superclass' `test` passing it the difference in the values of the 2 inputs. To achieve this we add a `value` attribute to the class `INPUT` with an `update` event to change its value. `RATE` is a rate of change test that compares the current value of its associated input with the last value it tested. It has an additional attribute to record the `last` value and overrides the `test` operation as shown.

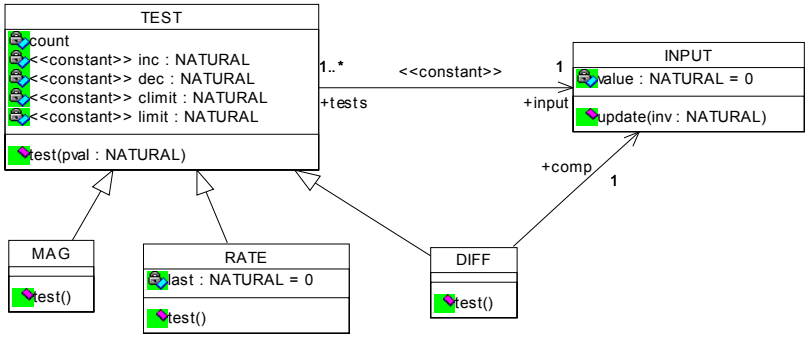


Fig. 11. Class diagram for refinement with subclasses of test

The overriding of the `test` event is shown below. This is modelled in the corresponding B by collating the three specialised test events into a single event with three guarded branches. The guard for each branch tests the instance for membership of a subclass.

```
MAG:test   = super.test(input.value)
DIFF:test  = super.test(abs(input.value-comp.value))
RATE:test  = last:=input.value ||
                        super.test(abs(input.value-last))

   where abs(i,j)= max(i-j,j-i)
```

Fig. 12. Overriding of the event, test

5 Discussion

To verify an event refinement it is required to show that any new events lead (eventually) to the enabling of one of the original events. When we attempted to verify our first refinement we found that this wasn't possible. Our initial attempt at a model as presented above doesn't ensure that if the system enters the freeze state it will ever be able to leave it, (a requirement imposed by Event B). The problem is that, although an individual input must leave the unconfirmed state after entering it, another input might also enter the unconfirmed state before the original one leaves it. In this way, inputs could take it in turns to be unconfirmed leaving the system permanently frozen. This would be an undesirable outcome since the frozen state is intended to be a temporary stage before confirming or recovering. A limiting mechanism that forced the system into the hardfaulted state after a certain number of unconfirmed inputs would prevent this and enable us to prove the event refinement. Even with this grossly oversimplified example the event modelling approach increased our understanding of the problem at the earliest stages of development.

6 Future work

This paper describes our first attempt using an event based modelling approach, with UML-B, to improve the reusability and portability of failure management systems. We are in the preliminary stages of a three-year research programme that aims to investigate this area. Within the project we will develop UML-B, to better support the use of UML features such as inheritance, and to provide modelling mechanisms that aid the refinement and transformation of UML-B models. We plan to re-implement the U2B translator within eclipse in order to achieve better integration with the B validation and verification tools, which will also be ported to eclipse. We will test and develop the ideas presented in this paper, on a larger scale example, in the following stages. Model a small but realistic, imaginary failure management application. The model may consist of several levels of refinement but be platform independent. We will then validate and verify the model via translation to B. Following validation and verification; we will investigate methods for abstracting away from the specific example to obtain a generic UML-B model that is application independent. New ideas for the development, using B# [20], may be used in this stage. Further development of UML-B and U2B may be needed to support this kind of model. Finally, we will investigate mechanisms for model transformation to obtain application and domain specific models from the generic model.

7 Conclusions

In this paper we have illustrated an approach to rigorous development of critical complex systems, such as failure management, in a manner that results in a high degree of re-usable verified components. The approach provides rigorous consistency verification through a sequence of refinement steps starting from a very abstract expression of overall system behaviour. This process of refinement can be continued through requirements development, design and into implementation. The process involves a, UML based, formal modelling notation, UML-B and utilises the tools available for the formal notation B. The refinement process inherently provides a high degree of reuse of verified specifications due to the deferment of specific application details. The genericity features of the UML may be used to provide re-usable common components within each refinement level. In this way we hope to provide a library of component classes that have a flexible but simple mapping with application domain concepts. The hierarchy of class models can then be instantiated with an object model for different applications thus achieving a specification and implementation language with small semantic gap that is suitable for the target complex problem domain; in this case, failure management systems. This approach could similarly be used for other complex systems problems.

References

1. Bertrand Meyer, *Object-Oriented Software Construction*, Prentice Hall PTR, Upper Saddle River, NJ 1997.
2. Grady Booch, *Object Oriented Analysis and Design with Applications* , The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA 1994.
3. Object Management Group, *OMG Unified Modeling Language Specification*, version 1.3, June 1999, from http://www.omg.org/technology/documents/modeling_spec_catalog.htm.
4. Jishnu Mukerji, Joaquin Miller, *MDA guide version 1.0*, available from Object Management Group website http://www.omg.org.
5. Ministry of Defence (1997) *Def Stan 00-55: Requirements for safety related software in defence equipment, Issue 2.* http://www.dstan.mod.uk/data/00/055/02000200.pdf
6. Radio Technical Commission for Aeronautics, *RTCA DO 178B -Software considerations in Airborne Systems and Equipment Certification,* from http://www.rtca.org.
7. Eurocae ED12B, *Software considerations in Airborne Systems and Equipment Certification*, from http://www.eurocae.org.
8. FAA/NASA, *OOTiA - Object Orientated Technology in Aviation Program*, from http://shemesh.larc.nasa.gov/foot/.
9. J-R Abrial, *The B-Method*, Cambridge University Press, 1996.
10. J-R Abrial_ and D Cansell, *Click'n Prove: Interactive Proofs Within Set Theory,* from http://www.loria.fr/~cansell/cnp.html.
11. B4free is a set of tools for the development of B models from http://www.b4free.com.
12. J-R Abrial, *Event Driven Construction*, 1999, from http://www.atelierb.com/documents.htm
13. M.Butler, and M. Leuschel, *ProB: A Model-Checker for B,* Proceedings of FME 2003: Formal Methods - LNCS 2805, from http://www.ecs.soton.ac.uk/~mal/systems/prob.html
14. C. Snook, K. Sandstrom, *Using UML-B and U2B for formal refinement of digital components*, Proceedings of Forum on specification & design languages, Frankfurt, 2003.
15. C. Snook and M. Butler, *Using a Graphical Design Tool for Formal Specification,* Proceedings of the 13th Annual Workshop of the Psychology of Programming Interest Group (PPIG).
16. M. Butler, C. Snook, *Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit*, Proceedings of UML 2000 Workshop, Dynamic Behaviour in UML Models: Semantic Questions.
17. C. Snook, I. Oliver and M. Butler, The UML-B profile for formal systems modelling in UML, In *UML-B Specification for Proven Embedded Systems Design*, Springer (In press 2004)
18. C. Snook, and M. Butler, U2B - A tool for translating UML-B models into B, *In UML-B Specification for Proven Embedded Systems Design*, Springer (In press 2004)
19. Mermet, J. (ed.) *UML-B Specification for Proven Embedded Systems Design*, Springer (In press 2004)
20. J-R Abrial, B#: Toward a synthesis between Z and B, In D.Bert, J.Bowen, S.King, M.Walden, editors, *ZB 2003: Formal Specification and Development in Z and B. Third International Conference of B and Z Users,* Lecture Notes in Computer Science, Vol.2651, Springer, pp.168-178.

Appendix - B produced by U2B

B machine for first abstract level

```
MACHINE         fman_a
SETS            CONTROL_STATE={normal,frozen,hardfaulted}
DEFINITIONS
   type_invariant == ( control_state : CONTROL_STATE ) ;
   invariant == (type_invariant)
VARIABLES       control_state
INVARIANT       invariant
INITIALISATION
   control_state :(invariant & control_state = normal )
OPERATIONS /*EVENTS*/
   hardfault =
      SELECT control_state=frozen THEN
         control_state:=hardfaulted
      END ;
   freeze =
      SELECT control_state=normal THEN
         control_state:=frozen
      END ;
   unfreeze =
      SELECT control_state=frozen THEN
         control_state:=normal
      END
END
```

B refinement for the first refinement:

```
REFINEMENT   fman_r1
REFINES         fman_a
SETS            INPUT
DEFINITIONS
   tests == input~ ;
   type_invariant == (
      ok : POW(INPUT) &
      suspect : POW(INPUT) &
      confirmed : POW(INPUT)      ) ;
   INPUT_invariant == (
      ok /\ suspect={} & ok /\ confirmed={} &
      suspect /\ confirmed={} &
      ok \/ suspect \/ confirmed = INPUT ) ;

   invariant == (type_invariant & INPUT_invariant) ;
   refinement_relation == (
      ((control_state=normal) <=> (ok=INPUT)) &
      ((control_state=frozen) <=> (suspect/={} &
                                   confirmed={})) &
      ((control_state=hardfaulted) <=> (confirmed/={})) )
VARIABLES    ok, suspect, confirmed
INVARIANT    invariant & refinement_relation
INITIALISATION
   ok, suspect, confirmed :(invariant &
            ok=INPUT & suspect={} & confirmed={} )
OPERATIONS /*EVENTS*/
   first_bad =
      ANY thisINPUT WHERE thisINPUT:INPUT THEN
         SELECT  confirmed={} THEN
```

```
          SELECT thisINPUT : ok & suspect={} THEN
             ok:=ok-{thisINPUT} ||
             suspect:=suspect\/{thisINPUT}
          END
       END
  END ;
etc.
```

B for the operation test:

```
ANY thisTEST,pval WHERE thisTEST:TEST & pval:NATURAL
THEN
   SELECT  thisTEST : ok & pval>limit(thisTEST)
   THEN    ok:=ok-{thisTEST} ||
           counting:=counting\/{thisTEST} ||
           count(thisTEST)=inc(thisTEST)
   WHEN   thisTEST : ok & pval<=limit(thisTEST)
   THEN   skip
   WHEN   thisTEST : counting & pval>limit(thisTEST) &
           count(thisTEST)<climit(thisTEST)
   THEN    count(thisTEST):=count(thisTEST)+inc(thisTEST)
   WHEN   thisTEST : counting & pval>limit(thisTEST) &
           count(thisTEST) >=climit(thisTEST)
   THEN   counting:=counting-{thisTEST} ||
           latched:=latched\/{thisTEST}
   WHEN   thisTEST : counting & pval<=limit(thisTEST) &
           count(thisTEST)<=dec(thisTEST)
   THEN   counting:=counting-{thisTEST} ||
           ok:=ok\/{thisTEST} || count(thisTEST):=0
   WHEN   thisTEST : counting & pval<=limit(thisTEST) &
           count(thisTEST)>dec(thisTEST)
   THEN   count(thisTEST):=count(thisTEST)- dec(thisTEST)
   END
END
```

From Misuse Cases to Collaboration Diagrams in UML

Zaid Dwaikat[1,2] and Francesco Parisi-Presicce[1]

[1] George Mason University, Fairfax, VA
[2] Software Productivity Consortium, Herndon, VA
zaldwaik@gmu.edu , fparisi@ise.gmu.edu

Abstract. Misuse and abuse cases are important concepts in software security, as they provide a major benefit to systems designers: the ability to think about abnormal scenarios. We present an approach that extends the misuse concept to software design artifacts. Through well–defined steps, we provide a mechanism to describe misuse behavior in collaboration diagrams. The formal semantics of such diagrams are positive and negative graphical constraints based on typed attributed graphs. The methodology can be effectively used, in conjunction with misuse cases, to develop more secure systems. The underlying semantics can be used to detect and remove redundancies and conflicts.

1 Introduction

Information security is mostly concerned with malicious behavior and unexpected failures of systems. Thinking only about functionality does not address most of the security aspects of software systems. The security of a system constitutes a subset of the non-functional requirements (NFR); other NFRs include availability, reliability and safety [1] . Functional requirements are well understood in the software community and there is a wealth of literature on how to derive and document them. NFRs are less understood and there is no agreed-upon standard that can be used uniformly by system's engineers.

From our experience in the field, NFRs typically take a lower priority than functional requirements, as customers are driven by the need for certain functionality. It is the job of the system designer to ensure that the system adheres to a certain set of NFRs. Security requirements in software systems are often overlooked and customers become aware of the need for secure systems only after a security incident (the system is attacked). This *after the fact* realization is often costly, resulting in attempts to retrofit the system with security controls.

Misuse cases were presented in [17]. The concept of misuse or "what a system should not do" is applicable to a variety of systems including software systems. Unexpected failures and flaws in the software design are hard to detect, and just thinking about potential abuse and misuse of system functions provides a good starting point to address security flaws in software systems. Use case relationships are extended with *prevents* and *detects* in [17] and *mitigates* and

threatens in [2]. These additional relationships are needed to express the new concept of misuse and describe counter measures to prevent (or at least reduce the effect of) such misuse.

Software security is a process that is tied to all phases of the Software Development Life Cycle (SDLC). Misuse cases provide a mechanism where developers can think about the system security during requirements analysis. It can be argued that use and misuse cases generally benefit all phases of the SDLC, as they provide a reference for architects and developers to validate many of their design decisions. To our knowledge, misuse and abuse have only been addressed in use and misuse cases and there have been no attempts to extend misuse concepts to other UML artifacts. Identifying bad actors and their potential behavior is only a start: to make this approach effective, software engineers need to address misuse cases in their design, development and testing activities. Hence the need to extend misuse cases to other SDLC phases and related UML artifacts.

Security flaws and bugs are hard to prevent by looking at one phase of the SDLC. While proper requirements engineering is essential for developing secure systems, design flaws and code bugs make up the majority of the security vulnerabilities that exist today. The need for secure coding practices cannot be overemphasized. In [20] and [6] the authors present valuable practical knowledge to designers and developers, but offer little formal methods.

Different methods have been recommended to address software security at various levels. Principles and best practices [20] are discussed at length and should be used in all software systems development. There are generic principles that apply to all systems and platform- or language-specific best practices [15] that need to be followed.

Other approaches deal with the need to check the artifacts produced by the different phases of the SDLC. Architectural reviews are used to locate potential flaws that may lead to security vulnerabilities. Code reviews present a way to eliminate potentially dangerous calls in programming languages and security bugs. Security testing verifies the proper functionality of the system's security controls, while penetration testing, a special case of security testing, simulates attacks on a target system by insiders and outsiders to evaluate its defenses.

2 Related Work

In [16] the authors propose a reusable approach using standard use and misuse cases. It is a novel approach and may gain some ground based on the potential savings as a result of reuse. The difficulty arises from defining a standard set of reusable use and misuse cases. Such standard set should be large enough to cover most common situations, but at the same time practical enough for average systems designers to use. Discussion on misuse case templates and structure is detailed in [18]. Proper templates and descriptions are essential to the use of this concept. While primarily text-based, use and misuse cases accompanied by diagrams are easier to understand and follow. Both [18] and [5] provide useful

suggestions on what to include in misuse cases, but their analysis stops at the requirements level.

In [1] further analysis is done that elaborates on the feasibility of misuse cases in non-functional requirements. Specifically, security requirements, described in *Shall not* statements or *Shall* statement with qualifiers, are derived based on the analysis of misuse cases. Interactions between misuse and abuse cases is also discussed in [13] and it is easy to see why misuse cases may be integrated with use cases. Since a software system's own functionality is often used to break into the system, it is necessary to utilize use cases as well as misuse cases in describing potential malicious behavior.

With regards to security engineering, we have identified two attempts at extending UML: UMLsec and SecureUML. In [8] UML is extended as UMLsec to provide useful mechanisms for describing security properties for software systems. UMLsec does not perform a security analysis, but rather aids in conducting such analysis. SecureUML [12] defines a Meta model for the specification and modeling of security aspects of a software system in UML diagrams. Access control modeling is analyzed in [12] using the well known Role-Based Access Control Model (RBAC).

3 Trust and *Mistrust*

Trust lies at the heart of security engineering and software systems are often built with many assumptions about trust that may cause costly security problems, as developers often mishandle trust relationships leading to a variety of security bugs [19]. Decisions about trust can be made at different levels in a software system: software components trust each other, trust the operating system and most often trust the users of the system. Questions about whom to trust when developing software systems are numerous, and there usually are no simple answers.

Trust is context sensitive. To make valid assumptions, it is important to analyze the environment in which a software system is intended to operate. In many cases, this environment is not well defined at the early stages of the SDLC, and a complete implementation description is often not available while design decisions about the system are being made. In other cases, users deploy systems in ways never imagined by its creators. Ultimately, it is the users who define how a system is deployed based on their needs and environmental and budget constraints. Both situations lead to the same outcome: assumptions about trust relationships (and other assumptions) may no longer be valid. This exposes the system to attacks as a result of misplaced trust.

Since some trust is inevitable, how does a software architect define how much trust is appropriate? Incorporating misuse concepts in this scenario adds great value, as misuse cases clarify many trust relationships and their susceptibility to abuse. Once an initial set of use cases has been developed, another pass is needed to analyze misuse cases. A thorough analysis should reveal unacceptable trust assumptions and allow software designers either to incorporate countermeasures,

or to remove such trust relationships. This process is iterative and can be extended into collaboration diagrams that can also be employed to express misuse behavior. However, even the use of this methodology does not account for the lack of information on environmental details or on users' actions that violate *recommended* implementation parameters.

4 Misuse in Collaboration Diagrams

Collaboration diagrams [1] show detailed interactions between objects [14]. In combination with sequence diagrams, class diagrams and state machines, they allow the visualization of data flow paths and information exchange between system objects and components. A collaboration diagram is typically developed for a certain task, and defines the participants with their roles and interactions to accomplish a common objective. Its ability to show participants as well as messages exchanged allows us to analyze trust relationships more easily.

4.1 Methodology

In this subsection, we present our methodology to extend misuse behavior to collaboration diagrams. The misuse concept can be extended into collaboration diagrams in the following manner:

- Use cases should be fully reflected in collaboration diagrams detailing the functionality of the system
- Misuse cases should be incorporated into collaboration diagrams using an inverted format [17]
- The mapping of mis–actors does not constitute designing such actors and their actions in the system, but rather an acknowledgement of their existence and their potential behavior. This serves two purposes: first, the ability to visualize malicious behavior by the system designers should prompt them to design mitigation techniques to counter such behavior; second, the expression of mis–actors and their behavior provides a record that helps in explaining certain design decisions
- Collaboration diagrams that represent undesired behavior should be flagged as such. Security controls to eliminate or reduce the effects of the undesired behavior should be incorporated in further design refinements
- In cases where a potential mitigation mechanism is feasible and can be architected into the system, such mitigation should be implemented. Mis–actors need not exist in further refinements of the collaboration diagrams once such mitigation has been implemented.
- In cases where mitigations are not feasible (e.g. too costly) mis–actors and their behavior should remain in further refinements of the collaboration diagrams. This allows for a real reflection of the system's behavior and accepted risks that are attached to such behavior

These steps may be applied iteratively until an acceptable design is reached.

[1] communication diagrams in UML 2.0 [3]

4.2 Example

To illustrate our methodology, we present a simple example that includes a use case and a misuse case integrated into one diagram. Figure 1 shows a typical access control mechanism where a user is asked to login to the system. The user supplies the login credentials to be validated by the system, and the system either accepts or rejects the login attempt. In the same diagram, a potential

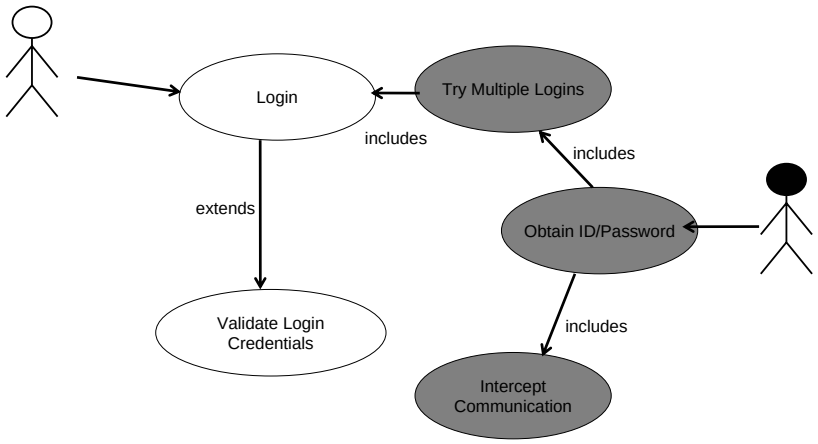


Fig. 1. *Use and Misuse Case Diagram.*

misuse is illustrated. This is a common situation we run into in software systems that we analyze for various organizations. A mis–actor may intercept the login credentials of a valid user and subsequently use such credentials to login to the system. Alternatively, the mis–actor may try a brute force attack on the system trying every possible password in successive login attempts. Both scenarios may lead to unauthorized access to the system. Further technical details about these types of attacks including feasibility, protocols and infrastructure are beyond the scope of this paper. Figure 2 shows two collaboration diagrams illustrating use and misuse behavior[2]. The misuse diagram represents the brute force attack scenario, showing, out of an unbounded number of login attempts, one with only 3 attempts. Based on this diagram, a design decision may be made to limit login attempts to two attempts (the number of attempts would be based on a careful evaluation of the system users, of the environment and of the support mechanisms). The objective is to mitigate a brute force attack. Notice that this behavior can also be a legitimate one, since users often forget (or misplace) their passwords. Figure 3 shows a further refinement of our collaboration dia-

[2] Other related,non misuse, diagrams (Class, sequence and state machine) are omitted for space considerations

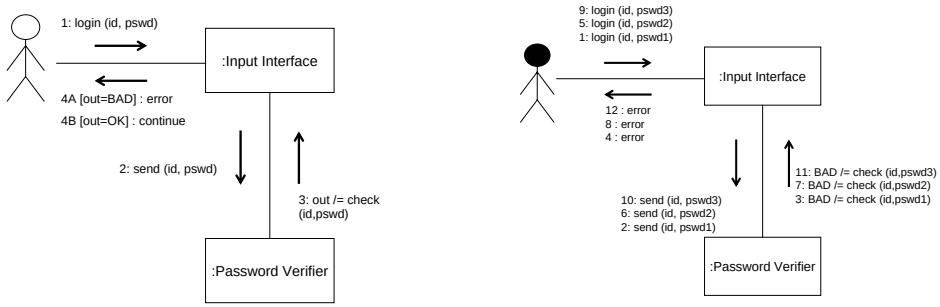


Fig. 2. *Normal Logins and Repeated Attempts.*

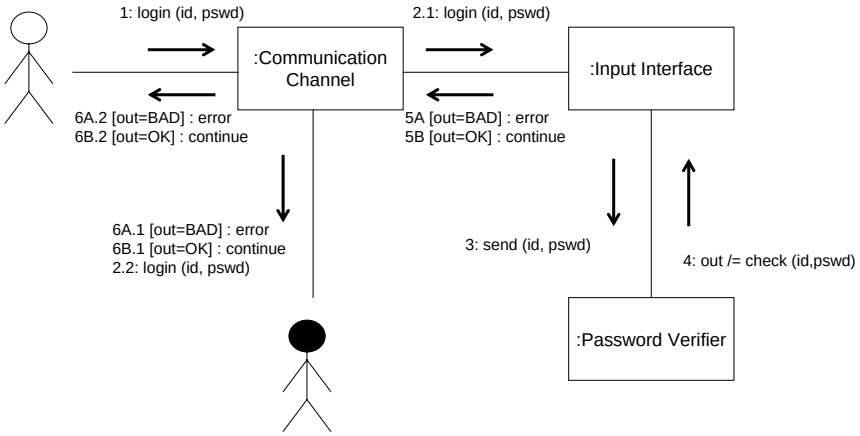


Fig. 3. *Collaboration Diagram: eavesdropping.*

gram in Fig. 2, obtained by adding another object, Communication Channel (a conceptualization of several communication channels, including keyboard cable, computer bus, electromagnetic waves (wireless networks) and network cables), to illustrate the ability of an attacker to intercept traffic between users and the system. To mitigate the risk presented in Fig. 3, we add two new objects, Encrypter and Decrypter, to the diagram. Figure 4 shows how the addition of these two objects reduces the risk of interception through the use of encrypted traffic. The previous collaboration diagrams illustrate two general views on misuse behavior: the diagram on the right of Fig. 2 indicates an unwanted behavior and can be viewed as a negative constraint on the system; the diagrams in Fig. 3 and Fig. 4 can be viewed, combined, as a positive constraint, requiring a mitigating factor (encryption in Fig. 4) on a dangerous scenario (eavesdropping in Fig. 3).

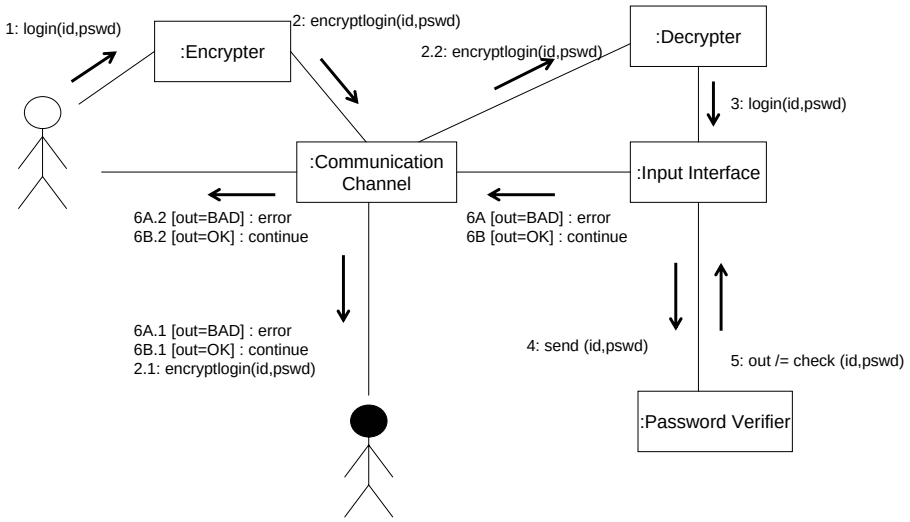


Fig. 4. *Collaboration Diagram: mitigating intercept.*

4.3 Semantics as Graphical Constraints

Constraints put limitations on the acceptable system states (in this context represented by collaboration diagrams) by either requiring a property or by forbidding a certain configuration. Model constraints can be either specified graphically or by OCL constraints. The latter ones, although part of UML, lack the visual appeal and intuition provided by diagrams and graphs.

Viewing the collaboration diagrams resulting from misuse cases as constraints on the system allows the (re)use of established results on graphical constraints [11, 10]. Due to the graphical notation, UML diagrams can be represented as attributed typed graphs.

Each class in a UML class diagram is a node of type *class*. The class name is stored in the attribute *name*, the stereotype(s) in the attribute *stereotype*. The UML class attributes and operations are represented as sets of tuples in the attributes *attributes* and *operations*, respectively. Each tuple for a UML class attribute contains information about the attribute's name and the attribute type. The tuple for the UML class operations has tuple elements for the operation's name, the parameter list and the return type. UML objects are represented in a similar way. The graph node for a UML object has the graph attributes *name* for the object name, *class* for the class name of which the object is an instance, and *attribute* for the attribute values.

A positive graphical constraint is a total and injective graph morphism $c : X \to Y$ and a graph G *satisfies* c if for all total and injective graph morphisms $p : X \to G$ there exists a total and injective graph morphism $q : Y \to G$ so that $X \xrightarrow{c} Y \xrightarrow{q} G = X \xrightarrow{p} G$. Informally, if G contains X, then it must also contain

$Y \setminus c(X)$. A negative graphical constraint is a graph C and a graph G satisfies C if there does not exist a total, injective graph morphism $p : C \to G$.

The use of this formalism, then, allows the analysis of the constraints to determine possible conflicts between wanted and unwanted behavior, and to discover redundancies among constraints. Although previously applied to access control policies [10], there are several results to systematically determine whether a positive constraint requires entities or patterns prohibited by negative constraints and, if so, how to modify either one to remove the conflict.

The graphical representation can be given by a collaboration diagram using the stereotype `<<exists>>` to distinguish between positive and negative constraints:

positive constraint A collaboration diagram is a *positive constraint* if it contains an object or a link with stereotype `<<exists>>`. The intended meaning of this diagram is that whenever the object structure without the stereotype `<<exists>>` occurs, then the object structure with the stereotype `<<exists>>` must also be present.

negative constraint A collaboration diagram is a *negative constraint* if there are no objects or links with stereotype `<<exists>>`. The intended meaning of this diagram is that the diagram in its entirety must not occur.

The framework in [10] can also be used by considering a refinement step (such as the one from Fig. 3 to Fig. 4) as a rule (in the sense of [9, 11]) and then applying rule-resolution algorithms in the presence of conflicts [10]. More work in this direction is needed to transform these preliminary ideas into a usable tool.

5 Concluding Remarks

Misuse cases present a valuable tool for security engineering. Building secure systems requires designers to think beyond the system functionality and its intended users, and to consider potential behavior that violates the system's security policy. In this paper we propose to extend the misuse concept to other phases of the SDLC. Specifically, we incorporate misuse behavior into collaboration diagrams, and show, using a simple example, how our methodology can be applied. So far misuse has only been applied to use cases UML artifacts.

The next step is to use a more complex and realistic example to provide us with feedback on the validity and feasibility of our approach. Another area that requires further investigation is a formal definition of our methodology. In the last section we present an overview of a methodology that can be used to define our approach. Further discussion will apply such methodology and derive specific semantics for incorporating misuse into collaboration diagrams.

We also believe that further extensions of misuse behavior should be expressed in other UML diagrams. Class diagrams and state diagrams are two important artifacts where security engineering can benefit from misuse analysis. There is also a need to adapt our work to the more recent standard UML2.0. At some point during the design, the boundaries between use and misuse may disappear, especially when describing regular users that may abuse the system.

References

1. I. Alexander. Misuse Cases help to elicit Nonfunctional Requirements. In *Proc. 8 th Int. Workshop REFSQ'2002*
2. I. Alexander. Misuse cases: Use cases with hostile intent. In *IEEE Software*, vol.20, no.1, 2003, 58–66.
3. S. Ambler. What is new in UML2. In *Software Development Magazine*, Feb. 2004.
4. E. Fernandez-Medina, A. Martinez, C. Medina, and M. Piattini. Uml for the design of secure databases: Integrating security levels, user roles, and constraints in the database design process. In Jürjens et al. [7], pages 93–106.
5. D.G. Firesmith. Security Use Cases. In *Journal of Object Technology*, 2(3), May/June2003, 53–64.
6. M. Howard and D. LeBlanc. Writing Secure Code. 2nd ed. *Microsoft Press*, 2002.
7. Jürjens, Cengarle, Fernandez, Rumpe, and Sandner, editors. *Critical Systems Development with UML*, number TUM-I0208 in Technical Report TU München, 2002.
8. J. Jürjens. UMLsec: extending UML for Secure Systems Development. In *Proc. UML'02*, number 2460 in LNCS, pages 412–425. Springer, 2002.
9. M. Koch, L. Mancini, and F. Parisi-Presicce. Foundations for a graph-based approach to the Specification of Access Control Policies. In F.Honsell and M.Miculan, eds., *Proc. FoSSaCS 2001*, Lect. Notes in Comp. Sci. Springer, March 2001.
10. M. Koch, L. Mancini, F. Parisi-Presicce. Conflict Detection and Resolution in Access Control Specifications. In M.Nielsen and U.Engberg, eds., *Proc. FoSSaCS 2002*, Lect. Notes in Comp. Sci., pages 223–237. Springer, 2002.
11. M. Koch, and F. Parisi-Presicce. Access Control Policy Specification in UML. In Jürjens et al. [7], pages 63–78
12. T. Lodderstedt, D. Basin, and J. Doser. SecureUML:A UML-Based Modeling Language for Model-Driven Security. In *Proc. of UML 2002*, number 2460 in LNCS, pages 426–441. Springer, 2002
13. J. McDermott and C. Fox. Using Abuse-case Models for Security Requirements Analysis. In *Proc. Annual Computer Security Applications Conf. ACSAC'99*, 1999
14. Object Management Group (OMG). Unified Modeling Language UML. version 1.5, March2003.
15. The Open Web Application Security Project. Guide to building Secure Web Applications. www.owasp.org, 2004.
16. G. Sindre, D.G. Firesmith, and A. Opdahl. A Reuse-Based Approach to determining Security Requirements. In *Proc. 9 th Int. Workshop REFSQ'2003*
17. G. Sindre, and A. Opdahl. Eliciting Security Requirements by Misuse Cases. In *Proc. IEEE TOOLS-37*, pages 120-131, IEEE CS Press, 2000
18. G. Sindre, and A. Opdahl. Templates for Misuse Case Description. In *Proc.7 th Int. Workshop REFSQ'2001*
19. J. Viega, T.Knono and B.Potter. Trust (and Mistrust) in Secure Applications. *Communication of the ACM*, 44:2, pages 31–37, 2001.
20. J. Viega and G. McGraw. Building Secure Software: How to Avoid Security Problems the Right Way. *Addison-Wesley*, 2001.

Formal Specification of Security-relevant Properties of User Interfaces[1]

Bernhard Beckert Gerd Beuster

{beckert|gb}@uni-koblenz.de

University of Koblenz
Department of Computer Science

Abstract. When sensitive information is exchanged with the user of a computer system, the security of the system's user interface must be considered. In this paper, we show how security relevant properties of a user interface can be modelled and specified using the Object Constraint Language (OCL).

1 Introduction

A large part of the specification of interactive applications is concerned with the relation between user input and the information shown to the user. For example, when editing a text, the current (internal) state of the text should be shown to the user, and user input should cause changes to the text. Usually, the specification of user input and system output is rather informal. Specifications declare that something "is shown on the screen" and the user "enters a text." In most cases, this informal description is sufficient. However, in security-critical applications, a precise and formal definition is desirable. In this paper, we show how security relevant properties of a user interface can be modelled, investigated, and ensured using formal methods.

2 Environment and Notation

In this paper, we model a text-based user interface. Input comes from the keyboard and output goes to a terminal with a fixed number of rows and columns for display of characters. Assuming no additional input from other sources (like a mouse or network card), the behavior of a text-based application can be described as a function from a (finite) sequence of keystrokes to a screen output. That is, the behavior is specified by what is supposed to appear on the screen after a particular sequence of keystrokes. We use *keyboard* to refer to keyboard input and *screenAt* to refer to screen output. When we want to refer to a specific

[1] This work was partially funded by the German Federal Ministry of Education, Science, Research and Technology (BMBF) in the framework of the Verisoft project under grant 01 IS C38. The responsibility for this article lies with the authors. See `http://www.verisoft.de` for more information about Verisoft.

screen position, we use the notation $screenAt[x, y]$ for the character shown at screen position (x, y).

To refer to keyboard input up to resp. screen output at a particular point t in time, we use $keyboard(t)$ to denote the list of keystrokes entered up to time t, and $screenAt(t)$ to denote the screen output at time t.

In a post-condition, t refers to the current time, i.e., the point in time when the function terminates, while $t@pre$ refers to the point in time when the function is entered. In this, we follow the common OCL syntax (though standard OCL does usually not contain explicit references to particular points in time).

3 Specifying Operating System Requirements

3.1 Overview

The operating system provides interfaces between application programs and the hardware. In the case of simple, text-based user interfaces, which we are examining in this paper, the operating system has to provide access to two resources: the keyboard and the screen. Most work on secure interface design assumes that the application runs in a safe and friendly environment. Although some work takes attacks on input/output facilities from the outside into account, interference with the input/output facilities from within the system (by trojans, worms, viruses, etc.) is usually not part of the attack scenarios. Here, we assume that the security critical application is running in a multi-process environment, where hostile processes may launch attacks on input/output facilities. We provide a formal method that *guarantees* security against software based man-in-the-middle attacks.

3.2 Specifying Screen Output Functions

Below, we give constraints specifying the operating system functions for accessing the screen.

context `setChar(character, x,y)`
post **if** $((\mathtt{x} \geq 0)$ and $(\mathtt{x} < screenWidth())$ and
$(\mathtt{y} \geq 0)$ and $(\mathtt{y} < screenHeight()))$
then
$screenAt(t)[\mathtt{x}, \mathtt{y}] = \mathtt{character}$ and
result = CHAR_SET_OK
else
result = POSITION_OUT_OF_BOUNDS
endif

3.3 Specifying Keyboard Input Functions

Since user input comes from the keyboard, we can identify all user input during the lifetime of the application with a list of keystrokes, where a keystroke is a character (a character code) associated with a timestamp.

Usually, computer systems have an input buffer. This buffer is filled with the user's keystrokes independently of the current application's activity. When the application calls the operating system function for retrieving the next keystroke, the first keystroke of the keyboard buffer is returned. In the scenarios we are modeling, however, the use of a keyboard buffer is often not advisable. From the view of security, we want that the user approves or denies an activity only *after* he or she is aware of the options available. With a keyboard buffer, a user may enter commands that are executed at a later point in time. It could then happen that the user approves or denies an activity *before* the available options are shown to him or her. Therefore, we define the operating system function `getkeystroke` without using an input buffer. The function `getkeystroke` is specified to return the next character typed *after* its invocation.

context `getkeystroke()`
post result $\in$ $keyboard(t)$ and
$timestamp$(result) $>$ $t@pre$ and
not $\exists k \in keyboard(t)$:
($timestamp(k) > t@pre$ and
$timestamp(k) <$ $timestamp$(result))

3.4 Specifying Security-relevant Properties

Under security aspects, a key requirement for a system using keyboard input and screen output is the impossibility of man-in-the-middle attacks against the keyboard and the screen. If an attacker can get in between the legitimate application and its input/output facilities, the attacker can manipulate the user at will.

There is no easy way to prevent physical man-in-the-middle attacks like, for example, covering the real keyboard with a faked keyboard as described in [2]. However, the prevention of software-based attacks with trojans, worms, viruses etc. is possible if the operating system provides means to guarantee exclusive access to the keyboard and screen. We call the process of acquiring exclusive access "locking" and the release of the lock "unlocking."

We consider information on whether screen and keyboard are locked and by which process to be part of the current configuration (i.e., the status) of the operating system. In the specification of requirements for the operating system, one has to refer to this information and other configuration details. For that purpose, we assume the relevant parts of the operating system configuration to be stored in a data structure (a class) `OSConf` with the following class attributes:

```
OSConf.screenLocked
OSConf.keyboardLocked
OSConf.ioStatus
```

`OSConf.screenLocked` and `OSConf.keyboardLocked` contain the process IDs (PIDs) of the processes locking the screen resp. the keyboard. A PID of 0 means that the resource is not locked. The third attribute `OSConf.ioStatus` can have the values `busy` and `waiting`. It indicates whether the system is busy or is

`Conf.command`	Last issued command
`Conf.commandResult`	Result of last command
`Conf.applicConf`	Application-specific part of configuration

Table 1. Configuration of an application.

waiting for input. While it is busy, all input is discarded (see comment about input buffers in Chapter 3.3).

Locking a resource is not sufficient to guarantee security. The user must also *know* which process locks a resource and whether the system is busy or not. Therefore, the operating system configuration must be shown to the user represented by a string of characters. We assume this string representation to be given by the function $OSConfString$: `OSConf` $\rightarrow$ `String` , which we do not further specify here. It must return a string that allows the user to determine the exact operating system configuration. Its actual implementation depends, for example, on the language(s) the user is supposed to understand.

We assume that the first line of the screen is reserved for information on the operating system configuration, i.e., the first line should be identical to $OSConfString$(`OSConf`).

We specify the correct display of the operating configuration resources as an invariant of `OSConf`:

context `OSConf`
inv $stringAt(t)[0,0] = OSConfString($`OSConf`$)$

In the following we give a constraint for the operating system call for locking the screen. The calls for locking the keyboard and unlocking the resources are equivalent. Here `PID` refers to the PID of the current application.

context `lockscreen()`
post **if** `OSConf.screenLocked@pre` = 0
then
`OSConf.screenLocked` = `PID` and
result = SCREEN_LOCKED_OK
else
result = SCREEN_LOCKED_BY_OTHER
endif

4 Security of Interactive Applications

4.1 Overview

In Chapter 3, we showed how to specify security-relevant properties of input/output functions provided by an operating system. By ensuring and verifying these properties, certain types of software-based man-in-the-middle attacks can be prevented.

There are, however, other essential aspects of a secure software system. In this chapter, we are going to introduce a method for specifying properties of applications that are desirable both for security and usability. Namely, the following properties are considered:

1. The user is always aware of the state of the system.
2. User input is only possible if the screen output is consistent.
3. Results of user actions are communicated to the user.

On an abstract level, the behavior of text-based interactive applications can be described using state charts. Edges are labeled with keystrokes, guard conditions, or both, as shown in Figure 1. In this example, the system transits from state Not Signed to state Signed if the command "Sign Text" is issued and the guard condition "Key Available" is satisfied. Of course, the states in such as state chart are abstractions of the application's actual internal configuration, which is much richer in detail. Nevertheless, we assume that these states are the *right* abstraction in that the user has sufficient information about the internal configuration of the application if he or she knows in what abstract state the application is. Since, as said above, we also want the user to know what the result of the last issued command was, we define the configuration `Conf` of the application to contain—besides an application-dependend part `Conf.applicConf`—the last issued command `Conf.command`, and the result `Conf.commandResult` of that command, which can take the special valued `none` if the command is not yet completed (see Table 1).

Fig. 1. State Chart Example

Now, two aspects of the application have to be specified:

1. The way in which the configuration is related to screen output; and how keyboard input corresponds to commands.
2. The effect that the execution of a command has, which must implement the abstract behavior specified by the state chart.

4.2 Specification of Input/Output Behavior

For the specifiation of the first aspect (input and output), we assume the following to be given (see Table 2):

- *stateAsString*(`state`) is a string that allows the user to determine what the state of the application is.
- *resultAsString*(`commandResult`) is a string that allows the user to determine what the result of the last issued command is.
- *screenOutput*(`applicConf`) is a two-dimensional array of characters. It contains the correct screen output corresponding to `applicConf`. Its dimensions are $screenWidth()$ and $screenHeight() - 3$.
- *command*(`char`) is the command that is issued by entering `char` on the keyboard.

Name	Description
stateAsString	Textual representation of the state
resultAsString	Textual representation of a command result
screenOutput	Screen output for a configuration
command	Command issued by entering a character
state	State abstraction of a configuration
newState	Next state when a command is issued in a certain configuration
result	Result of a command in a certain configuration

Table 2. Functions specifying an application.

We demand that *stateAsString*(`state`) is shown on the second line of the screen, and *resultAsString*(`commandResult`) on the third line (remember that the first line is reserved for the operating system's status line), which is why *screenOutput*(`applicConf`) must have a height of $screenHeight() - 3$.

Thus, the function `updateScreen` can be specified as follows. It is the application's function for updating the screen contents (using the operating system function `setChar`).

context `updateScreen()`
post $stringAt(t)[0,1] =$
$\quad stateAsString(state(\texttt{Conf.applicConf}))$ and
$stringAt(t)[0,2] =$
$\quad resultAsString(\texttt{Conf.commandResult})$ and
$\forall k \in \{3, \ldots, screenHeight() - 1\} :$
$\quad stringAt(t)[0,k] =$
$\quad\quad screenOutput(\texttt{Conf.applicConf})[\text{k}]$

4.3 Specification of Command Execution

For the specification of the second aspect (command execution), we assume the following to be given (see Table 2):

- *state*(`applicConf`) is the state abstraction of the application configuration.
- *newState*(`applicConf`, `command`) specifies the state transition. (It has `applicConf` as an argument and not, as one might expect, the abstraction *state*(`applicConf`), because it depends on guard conditions that can only be evaluated using the concrete application configuration.
- *result*(`applicConf`, `command`) is the result of executing `command` when the application is in configuration `applicConf`.

Now, the function `execute` can be specified. It executes a command and implements the state transition by changing the application configuration.

```
context execute()
post   state(Conf.applicConf) =
           newState(Conf.applicConf@pre,
                    Conf.command@pre)
       Conf.commandResult =
           result(Conf.applicConf@pre,
                  Conf.command@pre)
```

4.4 The Application's Main Algorithm

Now, we have everything at hand to describe how the main algorithm of the application works: First, screen and keyboard are locked. Then, in the main loop, commands are read and executed while keeping the screen updated. These steps are arranged in the following way:

- Screen and keyboard are locked immediately on program start and unlocked when the program quits. If locking the screen or the keyboard fails, the program terminates.
- Whenever the program is waiting for user input, the screen is up to date. Commands can be issued only when the system is waiting. All keystrokes entered during processing are discarded. By this we ensure that the user issues a command only when the current configuration of the system is visible on the screen.
- When processing is finished, the loop starts over again unless the user has issued the command "quit."

Pseudo for the main execution loop is given in Algorithm 1.

Algorithm 1 The application's main algorithm

```
 1: if not (lockkeyboard() = KEYBOARD_LOCKED_OK) then
 2:    Exit
 3: end if
 4: if not (lockscreen() = SCREEN_LOCKED_OK) then
 5:    Exit
 6: end if
 7: {OSConf.screenLocked = PID and OSConf.keyboardLocked = PID}
 8: repeat
 9:    updateScreen()
10:    Conf.command = command(key(getkeystroke()))
11:    Conf.commandResult = none
12:    updateScreen()
13:    execute()
14: until Conf.command = QUIT
```

The consistency of the screen output follows from the algorithm and the specification of `getkeystroke`. The screen is up to date when the system is waiting for user input, and immediately after user input, and it may be inconsistent in

between. Since the operating system displays status information "waiting" when the system is waiting for user input and "busy" when it is not, the user knows when the display must be consistent (whenever the system is waiting for user input). The situation would become more complicated if we used an input buffer. In that case, there is no longer a direct relationship between waiting/busy status and the consistency of screen output. It would be necessary to show an extra "consistency flag" on the screen.

5 Conclusions and Future Work

In Chapter 3 we gave a formal specification for text-based input/output functions of an operating system. This formalism can be extended to other input/output devices, e.g., card readers and graphical terminals. Additionally, we showed how to protect against software-based attacks on input/output resources. These security measurements require special functionality of the operating system. It must be able to grant processes exclusive access to input/output resources. Moreover, dedicated screen areas must be provided for information on who is locking the resources. This area must not be writable for anybody except the operating system.

The method we propose does not make any claims about what happens outside the realm of software. It cannot guarantee that an output device operates as intended, nor can it prevent tempering with the hardware of input/output devices.

In Chapter 4, we described a state-chart-based method for the formal specification of interactive applications. This formalism takes both security and usability aspects into consideration.

Our future work will go into two directions: As part of the Verisoft project (`http://www.verisoft.de`), the methods introduced in this paper are used to formally specify an email client. In Verisoft, both the operating system and the application program will be formally verified based on that specification.

The other direction of further work is to develop formal methods for the specification of applications that have richer user interfaces than a purely text based interface.

References

1. G. D. Abowd, J. P. Bowen, A. J. Dix, M. D. Harrison, and R. Took. User interface languages: A survey of existing methods. Technical Report PRG-TR-5-89, Oxford University Computing Laboratory, October 1989.
2. L. Bussard and Y. Roudier. Authentication in ubiquitous computing. In *UBICOMP 2002, Workshop on Security in Ubiquitous Computing*, Göteborg, Sweden, September 2002.
3. A. Dix and G. Abowd. Modelling status and event behaviour of interactive systems. *Software Engineering Journal*, 11(6):334–346, 1996.
4. V. Jain. User interface description formalisms. Technical report, McGill University School of Computer Science, Montréal, Canada, 1994.
5. B. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, pages 157–202, 1982.

Verification Tool for the Active Behavior of UML

M. Encarnación Beato[1], Manuel Barrio-Solórzano[2], Carlos E. Cuesta[2], and Pablo de la Fuente[2]

[1] EUI. Universidad Pontificia de Salamanca, Salamanca, Spain
ebeato@upsa.es
[2] ETSII. Universidad de Valladolid, Valladolid, Spain
{mbarrio,cecuesta,pfuente}@infor.uva.es

Abstract. The use of the UML specification language is very widespread due to some of its features. However, the ever more complex systems of today require modelling methods that allow errors to be detected in the initial phases of development. The use of formal methods make such error detection possible but the learning cost is high.
This paper presents a tool which avoids this learning cost, enabling the active behaviour of a system expressed in UML to be verified in a completely automatic way by means of formal method techniques. It incorporates an assistant for the verification that acts as a user guide for writing properties.

The Unified Modeling Language (UML) [4] has unquestionable advantages as a visual modeling technique, and this has meant that its applications have multiplied rapidly since its inception. To the characteristics of UML itself must be added numerous tools that exist in the market to help in its use (Rational Rose, Argo UML, Rhapsody ...). However, unfortunately, none of them guarantee specification correctness. It is widely accepted that error detection in the early phases of development substantially reduces cost and development time. It would thus be very useful to have a tool that would allow the integration of this semi-formal development method with a formal method to enable system verification. This paper presents a tool —TABU (*Tool for the Active Behaviour of UML*)— to carry out this integration by providing a formal framework in which to verify the UML active behaviour.

The tool uses SMV [2] (*Symbolic Model Verifier*) like formal specification, as it has the adequate characteristics for representing the active behaviour of a specification in UML. The main reason for this is that it is based on labelled transition systems and because it allows the user's own defined data types to be used, thus facilitating the definition of variables. It also uses symbolic model checking for the verification, which means that the test is automatic, always obtains an answer and more importantly, should the property not be satisfied generates a means of identifying the originating error.

The tool carries out, with no intervention on the user's part, a complete, automatic transformation of the active behaviour specified in UML into an SMV

specification. XMI [3] (*XML Metadata Interchange*) is used as the input format, thus making it independent of the tool used for the system specification.

On the other hand, our tool has a versatile assistant that guides the user in writing properties to be verified using temporal logic. Property writing is not a trivial problem. To write them correctly, advanced knowledge of logics and the type of specification obtained from the system is necessary. Our tool overcomes this problem using the assistant. Assistant starting point was the pattern classification proposed by Dwyer et al. [1] to which our own cataloguing of the different properties to be automatically verified has been added. The established property types are: A state machine with one active object is in a particular state, an object activity is in a particular state, a signal or event is produced or value comparison of an attribute.

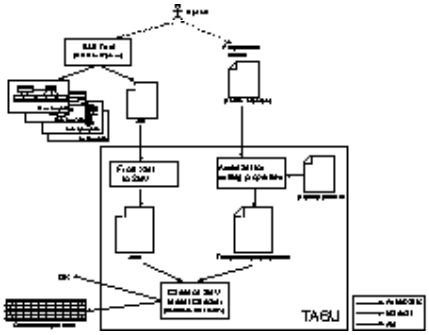


Fig. 1. Tool architecture

The verification is carried out in such a way that the user needs no knowledge of either formal languages or temporal logic to be able to take advantage of its potential. In addition, notions of the form of the specification obtained are unnecessary. Figure 1 is a graphical representation of the tool's architecture, the engineer only needs knowledge of UML and the system studied, the tool obtains automatically the formal representation in SMV from textual representation in XMI. Parallel, a wizard helps to write properties to verified, moreover if the property is not satisfied, the tool shows a counterexample trace.

References

1. Dwyer M. B., Avrunin G. S. and Corbett J. C.: Patterns in Property Specifications for Finite-State Verification. (1999)
2. McMillan K. L.: Symbolic Model Checking. An approach to the state explosion problem. Carnegie Mellon University. CMU-CS-92-131. (1992)
3. OMG: XML Metadata Interchange (XMI) v2.0. OMG Document 2003-05-02. (2003)
4. OMG: UML 2.0. OMG Document ptc/03-09-01. (2003)

Towards Engineering Development of Distributed Stochastic Hybrid Systems with UML

Manuela L. Bujorianu
Department of Engineering
University of Cambridge,
Email: lmb56@eng.cam.ac.uk

Marius C. Bujorianu
Computing Laboratory
University of Kent
Email mcb8@kent.ac.uk

Abstract

We propose a UML implementation of a parallel and communication extension of stochastic hybrid system. For stochastic hybrid systems we use the model introduced in [4], namely GSHS (general stochastic hybrid systems). It generalizes the most used models of stochastic hybrid systems used in control engineering. The stochastic features of the model make difficult this attempt. From a computer science perspective our approach uses a mixture of ACP and CCS techniques.
We use the concept of *distributed stochastic hybrid systems* (DSHS) [5] as an automata formalism for compositional specifications of GSHS. A DSHS can be thought of as an automaton representation of a GSHS, with an extra possibility to interact with other processes via so-called *passive transitions* (which are *discrete* transitions). We sketche how to extend the syntax of the Charon system, that is an UML implementation, in order to implement DSHS.

We present the DSHS formalism in a simplified version (a detailed presentation can be found in [5]). A DSHS, denoted by $\mathcal{DH}$, is a collection $((Q, d, m, \mathcal{X}), (f, \sigma), L, A, P)$ where: (i) $(Q, d, m, \mathcal{X})$ describes the state space, which is countable union of open sets from an euclidean space (modes), each one corresponding to a discrete location. Note that the dimension of embedding euclidean space might be different for different locations. (ii) (f, σ) gives the continuous dynamics between jumps of the continuous state within the locations. (iii) L is the set of labels. (iv) A are the set of *active transitions*. These are the union of the boundary-hit transitions B and the spontaneous transitions S. The boundary-hit transitions B depend on the transition-choice function C.
A realization of a DSHS generates a stochastic process. The structure of a DSHS assures that this process is a GSHS. We will refer to this process as the associated GSHS to the given DSHS. For the generation of the DSHS executions we assume that no communication takes place, therefore the passive transitions do not play any role in the generation of executions.

Parallelism is introduced in an axiomatic manner as in the ACP tradition. Communication is based on dually labelled transitions (where duality means sent/received transitions or active/passive transitions). Communication takes place as a handshake between dually labelled transitions. A detail presentations of the parallel composition and communication for DSHS can be found in [5].

General models of stochastic systems are specified with enough accuracy in the classical language of continuous mathematics. After adding concurrency the system dynamics are getting very complex, thus there is the danger of confusion. In this context system specification is becoming increasingly important. A specific implementation of UML,

considering extensions to support control and hybrid systems is the Charon system [1].

We sketch how it is possible to implement our model in an extension of the modelling language Charon obtained in a similar manner as in [2]. The language supports the operations of composition of agents to model concurrency, hiding of variables to restrict sharing of information, and instantiation of agents to support reuse. To make Charon suitable for implementation of DSHS, it is possible to extend the current version with syntax for specifying initial probabilities, jumps, and stochastic differential equations. In the terminology of Charon any DSHS is an agent.

The syntax for specifying an invariant is `inv <condition>` where condition can depend on the variables of the agent.

A jump can be specified as follows

`jump from <source_mode> when <guard>`
`(to <destination_mode> do {<update_cv>} weight <weight>)`$^+$

where the guard depends on the variables of the agent and defines a part of the complement of the invariant assigned to the source mode. The union of the guards of all jumps from a mode must be equivalent to the complement of the invariant of the mode. A jump has multiple transition branches. Each branch is specified by its destination mode, post jump location, and the weight assigned to it. The weight can depend on the variables of the agent. The post jump location `<update_cv>` is an assignment of the following form: `variable_name=f(...)` where f is a function specifying the distribution of the transition measure. Function f can depend on the variables of the agent. The post jump location specifies the probability measure on the set of valuations of the destination mode. The function f is built using the predefined distributions.

An SDE is specified by

`SDE {d(<variable name>)==f(...)*dt+`σ`(...)*dW(t)}`

where $f(...)$ and $\sigma(...)$ are functions which depend on the variables of the agent.

Distribution is defined using the intimate structure of sequential processes. This model requires a very rich concept of abstract state machine.

References

[1] R. Alur, R. Grosu, Y. Hur, V. Kumar, I. Lee. *Modular Specifications of Hybrid Systems in CHARON.* Proceedings of Third International Workshop on Hybrid Systems: Computation and Control, LNCS 1790, pp. 6-19, 2000.

[2] M. Bernadskiy, R. Sharykin, R. Alur. *Structured Modelling of Concurrent Stochastic Hybrid Systems.* Formats 2004. Available from http://www.cis.upenn.edu/~alur/Formats04.ps

[3] E.A. Boiten, M.C. Bujorianu. *Exploring UML Refinement through Unification.* Workshop on Critical Systems Development with UML, <<U M L>> 2003, San Francisco, California, USA, October 20 - 24, 2003.

[4] M.L. Bujorianu, J. Lygeros. *General Stochastic Hybrid Systems.* IEEE Mediterranean Conference on Control and Automation MED'04, 2004.

[5] M.L. Bujorianu, M.C. Bujorianu. *A Distributed Extension of Stochastic Hybrid Systems.* submitted.

[6] S.N. Strubbe, A.A. Julius, A.J. van der Schaft. *Communicating Piecewise Deterministic Markov Processes.* Preprints Conference on Analysis and Design of Hybrid Systems ADHS 03, pp.349-354, 2003.

Integrating an UML tool in an Industrial Development Process - a Case Study

John Knudsenjohnk@cs.auc.dk, Rene Gøttler, Michael Jacobsen, Mads W. Jensen, Jens G. Rye-Andersen, and Anders P. Ravn

CISS Aalborg University
DK-9220 Aalborg
{johnk,reneg,shapeles,blonde,jgra,apr}@cs.auc.dk

Abstract. This paper presents our experience with successfully integrating the UML CASE-tool, Rhapsody, into the software development process of a small company producing high quality embedded systems. The company had not been using Object Oriented Analysis Design and Programming, and this problem was attacked by developing a prototypical application that can be used as a pattern for developers in the company. Documentation of the application, including test specifications and generated and hand coded programs, is kept within the tool, such that the tool truly assists in the full development process. However, the most challenging aspect in our case study was to customize the tool such that it could support concurrent engineering and special platforms. We discuss these matters in detail.

Many small and medium sized companies are producing embedded systems which are increasingly software intensive. This software needs to be highly dependable, because it is an integral part of a mass-produced product which the customers expect to work smoothly for their lifetime. For this reason, such a company tend to have a defined software process with documented procedures for design, development and test; they appreciate the cost of software development, and are keen to find ways of making the process more efficient.

When we were approached by the company, the subject of this case-study, they formulated their expectations as follows: "We have invested in a license for the Rhapsody tool and we would like you to help us to fit it into our software development process, such that this process is made just as dependable and manageable as our hardware development process (which is done using Cadence tools)".

The contribution of this paper is thus a description and discussion of the steps that may be taken to successfully integrate tool supported, model based design and development methods into a small company producing high quality embedded systems.

Our analysis of the requirements for a successful integration gave us three important success criteria:

1. smooth embedding of *model-driven development* into the existing software process,

2. better control of *test and validation* efforts, in particular at the system level, and
3. version controlled *integration*.

Code generation is not a key issue; but keeping track of *code-design correspondence* is very important.

In our work we give an account of a successful integration of tool supported model based design and development into the company. The success is documented by the fact that the company now is in the process of adapting the procedures for future software development. We believe that one key ingredient in achieving take-over is to provide a realistic demonstration system based on the problem domain of the company, which is recognizable by the developers, and make it complete with respect to documentation, tests and with target platform code generation. Especially where developers have no or little experience in the Object-Oriented method.

With respect to the concrete tool, we note that Rhapsody did allow a successful integration. We encountered difficulties in building a cross-compiler tool-chain for Cygwin/Windows, where integration with Rhapsody went rather smoothly. However, with respect to configuration management, we managed to integrate basic CVS functionality with the configuration menu in Rhapsody, but the final solution is based on the usage of WinCVS. DiffMerge provides a graphical merge functionality, but because it is an absolute merge and the merge process can be very tedious, it is not recommended for daily usage. The plain CVS merge is a feasible alternative to the DiffMerge tool, but is still troublesome to use due to the complexity of the text format of some model elements. The lock mechanism, also provided in CVS, combined with model organization and process management, is the chosen solution. This solution does limit concurrent development, but is found to be the better choice.

Verification and Test of Critical Systems with Patterns and Scenarios in UML

Matthias Sand

University of Erlangen-Nuremberg
Department of Computer Science 3
matthias.sand@informatik.uni-erlangen.de

1 Introduction

When dealing with complex developement tasks, numerous requirements have to be met. Patterns and scenarios are a useful means to reflect both the functional requirements imposed to a system under developement as well as standardized general solutions to problems that arise from functional and non-functional requirements. When combined with a reasonable common notation for scenarios and the system model, it is possible to provide automated tools for the simulation or verification of the model against the requirements. In the following, the parts concerning the notational and formal aspects of such an approach, targeting mainly at the scope of hardware near embedded systems, are briefly described.

2 Approach

Due to its increasing popularity, great versatility and its focus on object oriented developement, the Unified Modeling Language (UML) is used for modeling the system as well as for the representation of its requirements and patterns. Primarily for practical reasons we fall back upon elements from UML 1.X [OMG01]. For the representation of the model serves a restricted subset of the class and state chart diagrams as well as possibly object diagrams. The static structure of the system is specified by a number of active classes and their relationships (i.e. associations) with each other. Each class must have a state chart diagram attached, which describes the behavior of its instances. For flexibility, a big subset of typical elements of the UML state charts is supported. Messages between instances are sent in the course of actions associated to transitions or state entries or exits. To this end, and for guarding expressions of transitions, an appropriate textual language had to be introduced, which clarifies the UML syntax.

Patterns and requirements are formulated by parametrised collaborations. Collaborations describe sets of objects, which interact with each other in order to achieve a certain goal. Each member of a pattern corresponds to a role of the collaboration. Roles can be associated to each other to provide the paths of communication between them. The scenarios of the collaboration are described using interaction diagrams, representing sequences of messages passed between

the roles. Patterns are typically laid out independently of concrete models and are bound to the current design later, whereas those collaborations that directly represent requirements imposed to the system under developement become part of the model at a relatively early stage of development, as soon as they arise in the progress of analysis and design. Binding patterns to the system model is equivalent to the instanciation of parametrised collaborations using the UML template mechanism.

The lack of a sufficiently formal semantics typical of the UML can be compensated by defining a transformation to a notation that is already endowed with one. In our case, a transformation to VHDL [IEE93] is proposed. This language provides a clear semantics, a complete sequential programming language, a rich set of data types, supports concurrency, instance-level hierarchy, and offers an elementary concept for the separation of interface and implementation. The rules of our mapping were particularly laid out in such a way that they reflect the informal semantics described in the UML standard as closely as possible, especially for the behavioral parts of the model. Each class within the UML model is mapped to a pair consisting of an ENTITY and an ARCHITECTURE. The ENTITY specifies the classes interface for communication, the ARCHITECTURE consists of at least one subcomponent, which is derived by transformation from the associated state chart and contains the actual logic of the class, realised by complex sequential descriptions. To achieve the message semantics of the UML state charts, a message buffer and a sequentialiser have to be added to the classes ARCHITECTURE in most of the cases. The default strategies of these infrastructural parts – FIFO and round robin – can easily be modified by providing different implementations for their respective ARCHITECTURE.

For simulation purposes it is possible to automatically generate a test bench based on the directed acyclic graph of send and receive events that are derived from an interaction connected to the respective collaboration. As an additional approach it is currently considered to formally verify a system model against its scenarios using model checking techniques.

3 Ongoing and Future Work

Based on this technical toolkit, a set of stereotypes has already been elaborated and is more and more extended, addressing the needs of scenario-based test and verification in the scope of the developement of embedded systems. These restrictions and extendsions to the UML metamodel are to be collected to form a UML profile, which is to be complemented by methodical guidance for its deployment in the design process.

References

[IEE93] *IEEE Standard VHDL Language Reference Manual*. New York, NY, 1993.

[OMG01] Unified Modeling Language Specification 1.4. Technical report, The Object Management Group, 2001.

A Case Study of Software Development using Embedded UML and VDM

Kazuya Tabata, Keijiro Araki, Shigeru Kusakabe, and Yoichi Omori

Graduate School of Information Science and Electrical Engineering,
Kyushu University, Hakozaki 6-10-1 Fukuoka, JAPAN
tabata, araki, kusakabe, yomori@ale.csce.kyushu-u.ac.jp

It is effective for improvement of software quality to utilize models of the target system and confirm their behavior in early stages of development. Models specified in VDM++, which is one of model oriented specification languages, include pre, post, and invariant conditions and statically typed classes and variables. The models also can be simulated by CASE tools, which enables us to comprehend their behavior. We can derive confirmation about verification and validation by these characters. These features are especially important for embedded systems, because errors may cause serious damages.

The goal of this research is to propose a software development method for embedded systems with a formal method. We had performed a trial using a miniature elevator system as a case study. In this paper, we combined VDM++ models and Embedded UML which is an existing model-based software development method. Embedded UML is a best practice including several processes and practical guides when applying UML to the development of embedded systems.

We added novel phases in which construction and analysis of VDM++ model are done between the phases of Embedded UML (Fig.1). We used UML models to analyze the structure and the interaction of models. VDM++ models are helpful to confirm the models' behavior. We constructed UML model following Embedded UML at first. Then, we constructed VDM++ model referring the corresponding UML model and added more details to the VDM++ model. The intuitive understandability of UML model was useful when we consider abstract models in each early phases of Embedded UML.

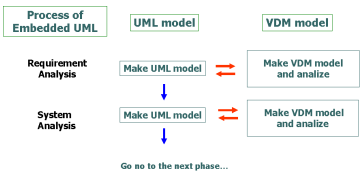


Fig. 1. Design flow of this study

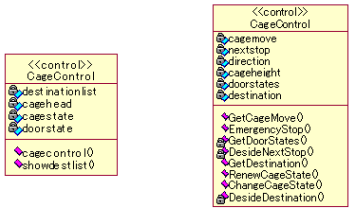


Fig. 2. Increase of operation and variables

However, a drawback of Embedded UML is that there are no means except manual review to confirm behavior of UML models. Otherwise these VDM++ models are automatically verified and validated by tools. For example, there was a request about the elevator that "Do not move the cage during any door is opened". In UML models, a model analyzer examined that the models kept the request by checking consistency of several UML diagrams (e.g. state diagram, activity diagram). On the other hand, the request in VDM++ models was always examined mathematically. Therefore, VDM++ models were efficient as they went more complex. We could find several pieces of leakage in these VDM++ models and fixed the discrepancies (Fig.2). We discovered that we had overlooked to define some operations and had read the values of variables at incorrect timings. And else, some objects were extracted from the inconsistencies between the behavior of models and what we thought. For example, there was a request "The cage prefers to move along when two calls from upper and lower floors were received at once". We saw the VDM++ model did not meet the request, thus reconsidered the algorithm and appended a state about "direction".

We conclude the benefits derived from the combination of Embedded UML and VDM++ as follows:

Benefits of Embedded UML

- Graphical expressions of models
  Embedded UML afford graphical expression of models which help us to understand the system intuitively.
- Analyzing method for embedded systems
  Embedded UML describes specialized methods to identify targets in embedded systems.
- Guide for proper model abstraction and hierarchization
  We could design both UML models and VDM models with proper abstraction and hierarchization at each phases of Embedded UML.

Benefits of VDM

- Assurance of specified conditions
  We can mathematically prove that models satisfy the specified conditions to increase reliability.
- Confirmation of model behavior by simulation
  We could confirm behavior of models by simulation. This let us discover boundary problems and inconsistencies among operations.

In accordance with this combination, we are considering about the metrics to choose a UML model or a VDM model in more detailed method.

References

1. John Fitzgerald and Peter gorm Larsen : Modelling Systems. the Press of the University of Cambridge. (1998)
2. H. Watanabe, M. Watanabe, K. Horimatu and K. Tomotake : Embedded UML. Shoeisha. (2002) (in Japanese)
3. Bruce Douglass : Real-Time UML. Addison-Wesley (1999)

A Semantics of UML Sequence Diagrams Based on Causality between Actions

O. Tahir, C. Sibertin-Blanc, J. Cardoso

Université Toulouse 1 / IRIT
21 allée de Brienne F-31042 Toulouse cedex
{otahir, sibertin, jcardoso}@univ-tlse1.fr

The unified modeling language (UML) is widely used to describe the behavior of systems. It provides a suitable framework for requirements elicitation by means of Use Case diagrams and Interaction diagrams (Sequence diagram, Collaboration diagram or Communication diagram), and for behavioral specification by means of Statechart (State-Transition diagrams) as an abstract executable specification of their behavior.

Unfortunately, UML lacks a formal semantics and does not offer semantic relationships between the diagrams of dynamics. In particular, Sequence Diagrams do not have an operational semantics describing formally how to execute such diagrams. Therefore, it is not possible to apply formal techniques for deriving Statecharts from Sequence Diagrams (SD). This lack has attracted the attention of researchers interested in automatically deriving the behavior of the components of a system from their interactions. To reach this goal, this article proposes a new semantics for the UML sequence diagrams based on a relation of causality between the actions of emission and reception of messages.

Indeed, the definition of SD in UML deals only with the scheduling of messages and does not say anything about the scheduling of the actions of sending (!m) and receiving (?m) a message m. While the scheduling of the actions of sending and receiving messages is unambiguous in the case of a procedural SD including a single focus of control (only one object is active at the same time), it is not obvious in the context of distributed systems including several objects simultaneously active. For instance, the SD depicted in figure 1 says that message m precedes message m', but it does not say e.g. whether the receiving action ?m is to be performed before the sending action !m'. The object O3 does not have enough information to known at which time it must emit the message m'.

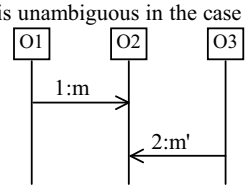


Figure 1. An UML SD

To provide SDs with an operational semantics, we propose to define a partial order relation among the actions of sending and receiving messages. This semantics is based on the abstract syntax of Interaction diagrams, i.e. on their definition in term of the metamodel of UML [1]. Any scheduling of actions must take into account the fact that a message has to be sent before it is received, thus we always have for any message m: !m before ?m.

There are several semantics – or ways to schedule the sending and receiving actions. It is not easy to give an exhaustive synthesis. Mainly, there are two kinds of semantics.

The first one consists in to read a SD as a Message Sequence Chart (MSC) [2], that is to ignore the global ordering of messages and to just consider the total ordering of the actions along the life-line of each object. According to this definition, we just have for the SD of Figure 1: ?m before ?m' in addition to !m before ?m and !m' before ?m' .

A second way to schedule the actions is defined and studied in a detailed way in [3, 4]. This semantics is mainly based on the scheduling of messages. It considers that whenever a message m precedes a message m', m must exist before m' and thus we have: !m before !m'. In particular, the operational semantics proposed by [3] imposes a total serialization on the emission and reception actions of successive messages sent by different objects. According to this definition, we have an additional constraint for the SD of Figure 1: !m before !m'.

In this paper, we introduce a third way to schedule the actions of a SD, based on the causality relationship between the actions. This semantics is particularly relevant in the context of reactive systems where two actions a1 and a2 are serialized only if a1 causes a2 in some way. According to this approach, we just have for the SD of Figure 1: !m before ?m and !m' before ?m'. Indeed, there is nothing that enables to relate !m performed by O1 and !m' performed by O3, and as a result there is no reason to assert that the reception ?m of m by O2 is the cause of the reception ?m' of m'. The main property of this new scheduling of actions is that it orders two actions only if there is a necessity reason for that.

Beyond the obvious clause: "a message can be received only if it were sent previously", we briefly describe the three rules of the causality notion :

i) the sending of a message is caused by the reception of the precedent message and thus it is to be postponed after the reception of all these precedent messages. The intuition under this rule is that the reception of a message is a call for a reaction that is achieved by sending a message;

ii) if two directly successive messages are sent by the same object, then their respective emission actions must be ordered. But if they are emitted by two different objects, the emission actions are independent and thus are not ordered. Thus, this operational semantics does not impose a systematic serialization on the emissions and receptions of successive messages sent by different objects;

iii) in the particular case where two messages sent by the same object O are addressed to the same object O', they are received by O' in the very same order as they were sent by O, to take into account the intention of the sender's ordering.

There is a family of SDs known as locally controllable [3]: a SD is locally controllable iff for two directly consecutive messages, either they are sent by the same object or the second message is sent by the recipient of the first one. A locally controllable SD includes a single thread of control and each object has enough information to know when a send action must be performed, so that the global ordering of actions can be ensured by the local controls within each object.

The semantics described in [3] is strongly related to the UML definition of SD. When it is applied to a SD, if the resulting global ordering reveals the addition of synchronizations, it means that the SD is not locally controllable. In this case, this ordering does not make sense because it needs either a centralized controller that activates each object when it has to perform a send action or the addition of a synchronization message.

The other semantics, the MCS-like and the causality-based ones, are relevant for any SD (locally controllable or not). In fact, the causality-based semantics is a subset of the MSC-like semantics; it includes an ordering constraint between two actions only if there is really some functional necessity for that. Thus, defining the semantics of a SD according to the causality-based semantics proposed in this paper results in a distributed system having a more robust behaviour, because it has to satisfy a smaller number of synchronisation constraints.

References

[1] OMG *Unified Modeling Language Specification* : version 1.5 Mars 2003. http://www.omg.org/.

[2] *ITU-T recommendation Z.120. Message Sequence Charts*, May 1996, ITU Telecommunication Standardization Sector.

[3] J. Cardoso, C. Sibertin Blanc. "*An operational semantics for UML interaction: sequencing of actions and local control*." European Journal of Automatised Systems, APII-JESA 36, p 1015-1028, ISBN 2-7462-0573-4, Hermès-Lavoisier, 2002.

[4] Alexander Knapp. "A Formal Semantics for UML Interactions". In Proc. 2nd Int. Conf. on the Unified Modelling Language UML'99. October 28-30, 1999 Fort Collins, Colorado, USA. LNCS 1723, pp. 116-130, Springer 1999.