T U M

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

Critical Systems Development with UML - Proceedings of the UML'03 workshop

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

TUM-I0317
September 03

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

TUM-INFO-09-I0317-80/1.-FI
Alle Rechte vorbehalten
Nachdruck auch auszugsweise verboten

©2003

Druck: Institut für Informatik der
Technischen Universität München

Critical Systems Development with UML - Proceedings of the UML'03 workshop

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

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 and time-to-market. Where thorough methods of system design pose high cost through personnel training and use, they are all too often avoided. The 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)" aims to gather practitioners and researchers to contribute to overcoming the challenges one faces when trying to exploit this opportunity.

The previous, and first, edition of the series was the CSDUML'02 satellite workshop of the UML'02 conference in Dresden (Germany), which had been very successful, and had the largest number of participants of all workshops around UML'02.

The proceedings at hand now present the accepted contributions for the CSDUML'03 workshop, which takes place on October 21, 2003, as part of the UML'03 conference (October 20 - 24, 2003, in San Francisco, California). 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, six were selected to be presented in talks at the workshop and included as full papers in the proceedings, and five were selected to be presented as posters and included as short papers. This corresponds to a selective full paper acceptance rate of 37.5% out of the submitted contributions, which 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 Bran Selic (IBM Software Group – Rational Software) on "The Use of Modeling Techniques in Safety-Critical System Design". Also, there will be a panel with the (provocative) title "What's wrong with UML for critical systems design ?"

with panelists including Robert France, Holger Hermanns, Heinrich Hussmann, Alexander Knapp, Ingolf Krüger, and Bran Selic, which will surely create lively discussions on the subject. Finally, it is planned to have a tool session to demonstrate tools for critical systems development with UML, and to discuss related questions.

As with the CSDUML'02 workshop, 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 home-page.[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'03 conference chair Jon Whittle (QSS Group Inc, NASA Ames Research Center), the workshop chair Ana Moreira (New University of Lisbon), the local arrangements chair Sherif Yacoub (Hewlett Packard Laboratories), and in particular Ben Glocker and the other students involved in the organization at TU Munich for their help. In addition, some of the organizers thank their various projects (including, ViSEK, NAME, and Verisoft) for their personal funding.

Jan Jürjens
Bernhard Rumpe
Robert France
Eduardo B. Fernandez
(Organization team for CSDUML'03)

September 2003

[1] http://www4.in.tum.de/~csduml03

Program committee

Joao Araújo, Universidade Nova de Lisboa
David Basin, ETH Zuřich
Marko Boger, Gentleware
Ruth Breu, University of Innsbruck
Manfred Broy, TU Munich
Gregor Engels, University of Paderborn
Martin Gogolla, University of Bremen
Radu Grosu, State University of New York at Stony Brook
Polar Humenn, Adiron LLC and Syracuse University
Heinrich Hußmann, LMU Munich
Alexander Knapp, LMU Munich
Ingolf Krüger, UCSD
Richard Paige, University of York
Noël Plouzeau, IRISA-INRIA Rennes
Gianna Reggio, University of Genova
Andy Schürr, TU Darmstadt
Bran Selic, IBM / Rational
Ketil Stølen, SINTEF Norway
Jon Whittle, NASA Ames Research Center

. . . and the organizers.

Additional referees

Folker den Braber, SINTEF
Phil Brooke, University of Plymouth
Joanna Chimiak-Opoka, University of Innsbruck
Jürgen Doser, University of Freiburg
Geri Georg, Colorado State University
Jan Hendrik Hausmann, University of Paderborn
Siv-Hilde Houmb, NTNU
Knut-Eilif Husa, Ericsson
Jochen Küster, University of Paderborn
Torsten Lodderstedt, University of Freiburg
Mass Soldal Lund, SINTEF
Atle Refsdal, University of Oslo
Thorsten Sturm, Gentleware
Jesco von Voss, Gentleware

Technical Organization

Ben Glocker, TU Munich

Organizers

Jan Jürjens, TU Munich
Bernhard Rumpe, TU Munich
Robert France, Colorado State University
Eduardo B. Fernandez, Florida Atlantic University

Table of Contents

Preliminary Program

Session: Critical Systems Design with UML

9:00 Richard Hawkins, Ian Toyn, Iain Bate: An Approach to Designing Safety Critical Systems using the Unified Modelling Language.

9:30 Sergio Cigoli, Philippe Leblanc, Salvatore Malaponti, Dino Mandrioli, Marco Mazzucchelli, Angelo Morzenti, Paola Spoletini: An Experiment in Applying UML 2.0 to the Development of an Industrial Critical Application

10:00 Geri Georg, Robert France, Indrakshi Ray: Creating security mechanism aspect models from abstract security aspect models

10:30 **Coffee Break and Poster Session**

Session: UML as a Formal Design Notation

11:00 Eerke Boiten, Marius Bujorianu: Exploring UML development through unification

11:30 Daniel Varro, Andrs Pataricza: Automated Formal Verification of Model Transformations

12:00 Harald Störrle: Assert, Negate and Refinement in UML-2 Interactions

12:30 **Lunch and Poster Session**

14:00 Bran Selic: The Use of Modeling Techniques in Safety-Critical System Design (**invited talk**)

15:00 Discussion

15:30 **Coffee Break and Poster Session**

16:00 **Panel** (R. France, H. Hermanns, H. Hußmann, A. Knapp, I. Krüger, B. Selic; moderation J. Jürjens): "What's wrong with UML for critical systems design?"

17:00 **Tool Session** (informal discussion and demo session)

17:30 **Closing**

19:00 Dinner (and kickoff of organization for CSDUML'04)

Posters:

Siv Hilde Houmb, Kine Kvernstad Hansen: Towards a UML Profile for Model-based Risk Assessment of Security Critical Systems

Zhongfu Xu, Johannes Luethi, Axel Lehmann: Predicting Software Performance Based on UML Models during the Unified Software Development Process

Luigi Lavazza, Sandro Morasca, Angelo Morzenti: A Dual Language Approach to the Development of Time-Critical

Jan van Katwijk, Bo Sanden, Janusz Zalewski: An Approach to Evaluate Real-Time Software Architectures for Safety-Critical Systems

Joao Barros, Luis Gomes: Actions as Activities and Activities as Petri nets

The Use of Modeling Techniques in Safety-Critical System Design

Bran Selic

IBM Software Group – Rational Software
Kanata, Ontario, Canada
bselic@ca.ibm.com

Safety critical software systems impose additional requirements to the fundamentally difficult design problem of embedded/real-time software. Traditional code-driven development approaches are highly risk laden and often inadequate in the face of such intimidating complexity. In this talk, we examine how model-driven methods based can reduce much of this risk. In particular, we focus on two significant new developments of the Unified Modeling Language can remove much of the accidental complexity of traditional development methods.

An Approach to Designing Safety Critical Systems using the Unified Modelling Language

Richard Hawkins, Ian Toyn, Iain Bate

Department of Computer Science
The University of York
Heslington, York
YO10 5DD, UK
{richard.hawkins, ian.toyn, iain.bate}@cs.york.ac.uk

Abstract. In this paper an approach to using the UML for developing safety critical systems is presented. We describe how safety analysis may be performed on a UML system model and how this analysis can derive safety requirements for classes in the system. We show how these requirements can be expressed in the form of *safety contracts* using the OCL. This makes it possible to reason about the safety of individual elements of the UML model and thus makes it easier to safely change the UML design, as well as facilitating maintenance and reuse of classes or components in the system. A tool is also described which has been developed to automate some aspects of this analysis.

1 Introduction

There is increasing interest in the use of an Object Oriented (OO) approach for developing safety critical systems. OO systems have improved maintainability due to encapsulation, high cohesion and low coupling, and the facility for reuse through inheritance and design patterns. These benefits could bring potentially large savings in terms of time and cost for developers of safety critical systems. However they also raise specific challenges which must be addressed if the full advantage of these OO features are to be realised for safety critical systems. To realise these benefits requires an ability to reason about the safety of individual classes or components in a system. This requires firstly that safety and hazard analysis be successfully performed on OO designs.

The Unified Modelling Language (UML) has become a de-facto standard for modelling OO systems and is widely used throughout industry. In this paper we initially look at performing hazard analysis of UML models. Moreover, however, most existing safety analysis techniques tend to decompose hazards in a functional manner. It is therefore difficult to reason about the safety of individual classes or components. If the benefits discussed earlier are to be realised, these existing techniques must be adapted such that the required results are obtained. Therefore we go on to look at how safety requirements may be captured in the form of safety contracts. These safety contracts can be specified using the Object Constraint

Language (OCL) and incorporated into the UML model of the system. This could contribute towards the development of a safety critical profile for UML. We show how the safety contracts can be generated through analysis of different aspects of the UML model, covering functional, timing and value behaviours.

2 System Safety Analysis

Leveson provides us with the following definitions [1]. *Safety* is the freedom from accidents or loss. Safety critical systems are systems which have a direct influence on the safety of their users and the public. A *hazard* is a state or set of conditions of a system that, together with other conditions in the environment of the system, will lead inevitably to an accident. The primary concern of system safety analysis is the management of hazards: their identification, evaluation, elimination, and control through analysis, design and management procedures. One aspect that distinguishes system safety from other approaches to safety is its primary emphasis on the early identification and classification of hazards so that corrective action can be taken to eliminate or minimize those hazards as part of the design process. The system safety process culminates in a safety case being produced that gathers all the necessary evidence to justify the system is safe.

The system safety analysis process can be basically split into the following steps:

- Hazard identification – This step identifies the potential hazards in the proposed system.
- Risk assessment – This examines each of the identified hazards to determine how much of a threat they pose. This assists in deciding the steps required to reduce the risks to acceptable levels. Many initial safety requirements are set at this stage.
- Preliminary system safety assessment (PSSA) – This phase is concerned with ensuring that a proposed design can meet its safety requirements and also with refining these safety requirements as necessary
- System safety assessment – This stage is concerned with producing the evidence that demonstrates the safety requirements have been met by the implementation.
- Safety case delivery – This involves producing a comprehensive and defensible argument that the system is safe to use in a given context. The analysis performed as part of the PSSA stage will form part of this argument.

In this paper we are primarily concerned with the PSSA. This is very closely linked with design activities. A key part of this is the identification of specific derived safety requirements to guide the detailed design of the system. The majority of the safety analysis techniques used in PSSA are deductive. This means that they investigate possible causes of a specific hazard or condition, starting from system level hazards and requirements identified during the hazard identification phase. The majority of techniques used for PSSA tend to decompose hazards functionally. System level hazards are identified, and for each, functional failures which may contribute to that hazard are elicited.

For a functionally decomposed system it is much easier than in an OO system to allocate a functional failure to the failure of a system component. This is because there is often a direct mapping between a functional failure and a subsystem. An OO design is not decomposed functionally, but rather into classes and objects. The functionality of the system is realised by many objects collaborating through message passing to achieve a system function. Therefore a functional failure will not map easily onto a single element of the design. In theory it would be possible to put all functionality into just one class and to apply standard safety analysis techniques. In doing this however, all advantages of adopting an OO approach would be lost. It is important therefore, for OO systems and indeed for UML system models that existing techniques can be adapted such that failures can be allocated between classes.

Our approach achieves this by allocating derived safety requirements to individual classes and controlling the interactions between classes through the use of contracts. This allows the impact of changes to be understood. An appropriate allocation of constraints would also allow better support for change in that change would be contained within a class or within a small hierarchy of classes. Other work looks at this in more detail.

3 Safety Analysis for UML

There have been attempts to adapt safety analysis techniques to apply them to UML. In [2], Lano et al examine how Hazard and Operability Studies (HAZOP) can be adapted to UML. HAZOP is a predictive safety analysis technique which uses a set of guidewords to consider the behaviour of 'flows' between system components. Lano et al take the standard guidewords for HAZOP defined in Defence Standard 00-58 [3] and reinterpret them such that they are applicable to different views of the UML model. For example, when considering state transition diagrams, the 'flow' is taken to be the transition and the guidewords are interpreted accordingly. Similarly, for class diagrams 'flows' are taken to be relationships. This is a very useful approach to analysing failures, however it would demand that the technique be applied to every class, attribute, state transition and interaction in the system to be effective. Even for a fairly small system this could be prohibitively time consuming.

Nowicki and Gorski have also developed analysis based largely around state charts. In [4], three methods are introduced. The first method, detailed in [5] centres around the development of a hazard model, which explicitly models safe and unsafe states and the transitions between these states. Reachability analysis can then be used to check if the hazardous state is reachable. This method assumes that the system and environment are reliable in the sense that they behave as specified. The second method [6] accepts that this assumption isn't always valid as objects are exposed to random failures and the environment can violate assumptions made about it. This method provides a set of templates of faulty behaviour, which are deviations from the normal behaviour of the object. The templates consider possible faults in transitions and are applied to the state chart of the object under consideration. This generates a state chart for an 'unreliable object'. Reachability analysis is then performed to see if a hazard can occur for the unreliable object. Whereas the first two methods were

concerned with analysing the system design, the final method aims to strengthen the safety guarantees of the system by enriching the system with a 'safety monitor' object. This is of less interest here. It is the second method which is perhaps most useful. There are certain weaknesses, particularly in identifying which objects to apply the templates to. Again, applying them to all objects in the system would be potentially very time consuming. It is also unclear how hazardous states are correctly identified.

In the remainder of this section we take some of the ideas discussed above and incorporate other techniques such as Fault Tree Analysis (FTA) to produce a more focussed approach to analysing UML models such that safety requirements can be derived for classes in the system in the form of safety contracts. Safety contracts constrain the design of the interactions which occur between objects, and hence can ensure system behaviour is safe. The properties of an interaction that we are interested in from a safety perspective are function, timing and value. Analysis of each of these aspects results in requirements which are included in the safety contract. Safety is a system property and therefore, the analysis process will begin with the consideration of a system level hazard.

3.1 Functional Aspects

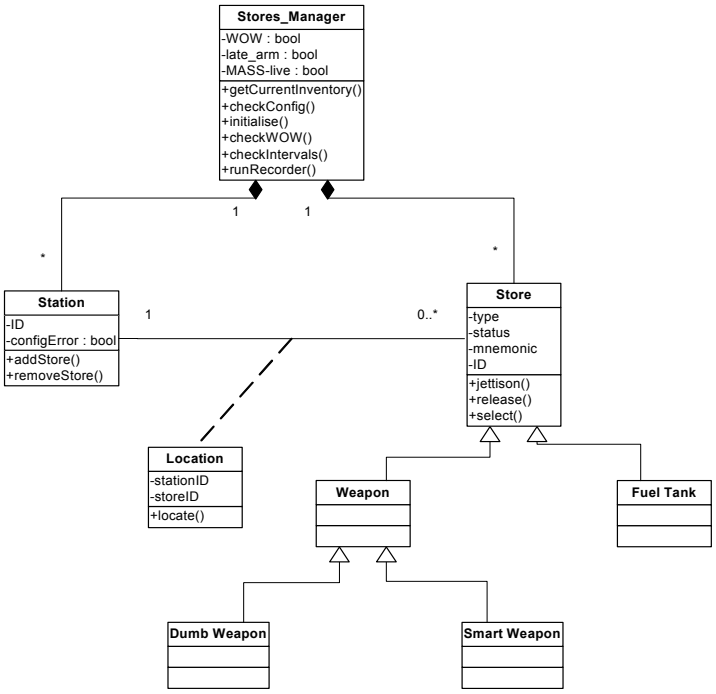


Fig. 1 – Class Diagram for Part of a Stores Management System

To illustrate the analysis we use an example of a highly simplified aircraft Stores Management System (SMS). In the context of an aircraft, a store could be such things as a weapon or a fuel tank. These are connected to the aircraft via stations on the wings. The UML class diagram for this system is shown in figure 1. The initial hazard identification process performed on the SMS identified a number of general system hazards including:

- Inadvertent release of store
- Release of store whilst on the ground
- Inadequate temporal separation of store releases
- Unbalanced stores configuration
- Release of incorrect store

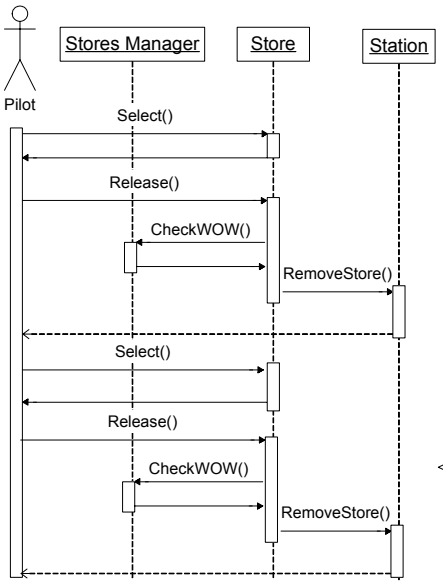


Fig. 2. UML Sequence Diagram for a Normal Operation Scenario.

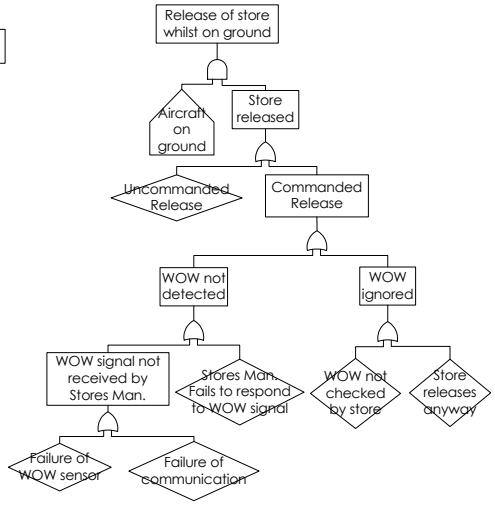


Fig. 3. Simplified Fault Tree for the Release of Store Whilst on the Ground

For all hazards identified it is necessary to perform analysis to identify how the hazard may be brought about. For this example we will look solely at the release of store whilst on the ground hazard. A UML sequence diagram is developed to illustrate the dynamic behaviour of the system for the relevant normal operation scenario. This can be seen in figure 2. A fault tree is constructed using system information collected from the UML diagrams and domain knowledge. Fault Tree Analysis (FTA) represents graphically the combination of events and conditions which contribute to a single undesirable event. Starting from the top event, the immediate causes of that event are identified. The manner in which these causes contribute to the event is expressed with the use of logic gates (basically AND and OR). This continues down the tree until the tree consists entirely of basic events, i.e. events which cannot be decomposed further, or until the desired level of detail has been reached. The fault tree in figure 3 shows the faults that can occur to bring about the top event “Release

of store whilst on ground". It should be noted that WOW is 'Weight on Wheels', used to indicate when the aircraft is on the ground.

It is possible to relate leaf nodes (undeveloped failure events) in the fault tree, represented by diamonds, to classes in the system. For example the 'WOW not checked by store' event can be associated with the Store class in the system design. Not all leaf nodes in the fault tree will relate to classes in the system design. Of the six leaf nodes in figure 3, three can be related to classes in the system design. For example 'Failure of WOW sensor' cannot be attributed to a class, since the WOW sensor is not part of the Stores Management System. For the three leaf nodes identified as related to the SMS' classes, the information from the fault tree can be used to generate a definition of the hazardous behaviour of the system. This information is recorded in a table as shown in table 1.

Table 1. Table of hazardous class behaviour

Hazardous event (from FT)	**Class**	**Interaction**	**Role**	**Hazardous class behaviour**
WOW not checked by store	Store	checkWOW()	client	Stores_Manager.checkWOW() call not made as required
Store releases anyway	Store	release()	supplier	Store moves to released state inappropriately
SM fails to respond to WOW signal	Stores Manager	WOW(true) – signal	supplier	Stores_Manager fails to move to WOW state

In this table, the hazardous event field is the event identified in the fault tree. The class field is the class with which that event is associated. The interaction is the operation identified from the interaction diagram as corresponding to this event. The role is that played by the class in that interaction (either client or supplier of that operation call), and the hazardous class behaviour is that which would be exhibited by an instance of the class to bring about the hazardous event.

3.1.1 Analysing State Charts

We have now derived hazardous conditions for classes in the system. It is therefore possible to start constructing safety requirements and safety contracts for these classes. However to identify more detailed derived safety requirements it is necessary to understand how the class may behave such that these conditions can occur. This can be done by studying the state charts of these classes. For the purposes of this example we will consider just the Store class. A simple state chart has been developed for this and is shown in figure 4. A store may be jettisoned or released, jettison is a special case of release when a store is in a 'safe' state (e.g. dropping an unarmed weapon). The hazardous class behaviour for the Store class taken from table 1 can be summarised as:

- State = release ^ ¬checkWOW
- State = release ^ WOW = true

Now that hazardous states have been defined it is possible to apply the ideas of Gorski and Nowicki discussed earlier [6]. Firstly it is assumed that the system behaves as specified in the design, that is that the class exhibits no faulty behaviour. In this simple example it can be seen from examining the state chart that this design does not exhibit any of the hazardous behaviour defined above. Checking that a proposed design does not exhibit hazardous behaviour can be achieved using a reachability analysis tool. The effects on the safety of the system if an object were to behave in an unexpected manner, that is to behave in a way other than that specified in the design due to mistakes in implementation, must also be investigated. To do this we mutate the transitions in the state chart [6]. Transitions in a state chart are of the general form *event[condition]/action*. The event triggers the state transition, the condition is a Boolean expression which must evaluate to true for the transition to occur and the action is triggered when the transition fires. Transitions may have any, all, or none of these elements. In order to identify possible faulty behaviours for the transitions we can apply guidewords to each of the elements of the relevant transitions. In order to be able to simulate these faulty behaviours, extra transitions must be added to represent these deviations in the state chart. Applying the guidewords omission, commission and value to each of the elements results in five distinct transitions:

1 - e[c] self-transition – event or condition is ignored (omission)

2 - not e[c] / a – event spuriously generated or action performed without initiating event (commission)

3 - e[not c] / a – condition taken as true when false (value)

4 - e[c] – action is ignored (omission)

5 - e[c] / b (where b is an action other than a of the initiator object) – wrong action performed (value)

For each of the transitions in the state chart relevant to the hazardous behaviour, these five extra transitions are added to the diagram to simulate faulty behaviour. The results of this can be seen in figure 5. The transitions between the unselected and selected states have not been included as they are not relevant to the hazardous behaviour we are interested in. It is now possible to identify if any of the faulty behaviours are unsafe. These are the faulty behaviours that can lead to the hazardous object behaviour which was defined previously.

The faulty transitions that could lead to the hazard ‘Release of store whilst on the ground’ can now be analysed. The results of this are shown below.

A1. release – *Not Hazardous*

A2. **not** release / check WOW - *Not Hazardous*

A4. release – ***Hazardous*** – WOW is not checked but class may enter release state

A5. release / remove store – *Not Hazardous*

B1. [WOW=false] – *Not Hazardous*

B3. [WOW=true] – ***Hazardous*** – class enters release state when WOW is true

C1. [WOW=true] - *Not Hazardous*

C3. [WOW=false] - *Not Hazardous*

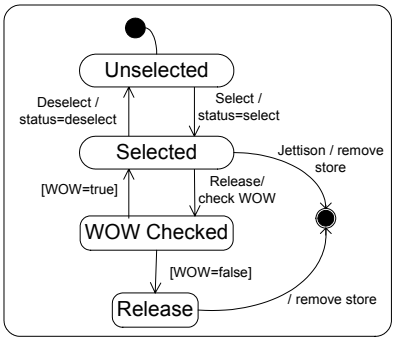


Fig. 4. State Chart for Store Class

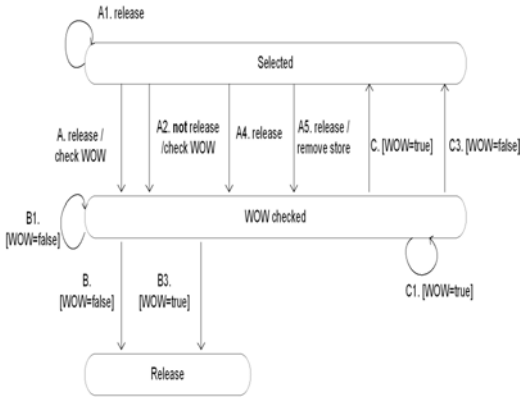


Fig. 5. Mutated State Chart Showing Faulty Transitions

It is possible for more complex system designs to use an automation tool to generate the mutated transitions and to check which of these transitions may bring about the hazardous condition. Such a tool is under development at the University of York and is discussed in section 5. Although most of the faulty transitions identified through this method are not hazardous (i.e. will not contribute to the system hazard) they still result in incorrect operation. In defining a safety contract we are only interested however in constraining the hazardous behaviour and not all correct behaviour. There is now sufficient information about the intended and faulty behaviour of the class to begin to construct a contract for operations which may contribute to the system hazard. We look at constructing these contracts in section 4.

3.2 Timing Aspects

Although a large part of the safety requirements generated for any given system will be functional in nature, it is important to also consider the impact of non-functional properties on the safety of the system. Firstly the timing of the interactions is investigated. This analysis process hinges on identifying deadlines, separations and priorities for tasks performed by the system. A task is an encapsulated sequence of operations that executes independently of other tasks [7]. Therefore a task will consist of a number of interactions between classes in the system. Again the analysis begins with the identified system level hazards and at this point we focus on the normal scenario for releasing a store as shown in the sequence diagram in figure 2. This scenario can be broken into the following tasks:

- Select store – This task begins with the pilot choosing a store and ends with that store being selected
- Release store – This task begins with the pilot requesting a release and ends with the store being removed from its station. This task also includes a subtask of checking WOW.

The analysis involves investigating the effect of deviations on the tasks that are performed to identify which of these deviations may contribute to a system hazard. The deviations considered are tasks occurring too quickly or too slowly and early or late. Early and late correspond to a task occurring too soon or too long after the previous task or event. The result of applying these deviations to the identified tasks is shown in table 2.

Table 2 – The effects of timeliness of tasks on system hazards

Task	Deviation	Effect
Select Store	Quick	**No safety consequence (positive effect) –** It is desirable that the selection of the correct store occur as quickly as possible
	Slow	**Potential safety impact** – Delays in selecting the appropriate store for release may delay release
	Early	This task is triggered by the pilot who's decision to select a store will impact safety only if incorrect store is chosen
	Late	
Release Store	Quick	**No safety consequence (positive effect)** – It is desirable that the store be released as quickly as possible when requested
	Slow	**Potential safety impact –** A delay in releasing a store could be hazardous to the aircraft under certain circumstances
	Early	**Hazardous –** A weapon released too soon after a previous weapon could be catastrophic
	Late	**No safety consequence**

Those tasks whose timeliness may have an impact on the safety of the system have now been identified and constraints must now be specified for these tasks. For quick and slow interactions it is necessary to constrain the response time of the task. If necessary a minimum response time and a maximum response time, or deadline can be specified for a task. A minimum response time will be specified for those tasks where too quick is identified as being hazardous and a deadline is specified for those where too slow could be hazardous. For tasks where early or late may be hazardous, minimum and maximum separations respectively between the completion of one task and the triggering of the next or between an event and the triggering of a task must be specified. These constraints can be used to define a safe scenario of tasks.

Domain knowledge is used to place the following requirements on the tasks identified above as being hazardous or potentially hazardous. It should be noted that this analysis is not trying to produce accurate estimates of execution times for operations or transactions, but to specify the minimum requirements for a safe system. Requirements should therefore have as much tolerance as possible whilst still constraining the task sufficiently to ensure it is safe. This will allow maximum flexibility to the implemented system. In [8] we presented an approach and framework for identifying, in control systems, an appropriate and valid set of timing requirements, and their corresponding control parameters. The approach is based on decomposing the systems objectives to a number of design choices and assessment criteria. Each design choice can be evaluated using a combination of static analysis and simulation. Heuristic search techniques can then be used to search the design space for the design solution considered most appropriate. The figures given below are for illustration purposes only.

- Select store – From pilot choosing a store to that store being selected should be no longer than 200ms – Deadline = 200ms

- Release store – The minimum permissible time between store releases will vary depending on the type of store being released. For this example we will specify – Min Separation = 100ms
- Release store – The time from the pilot requesting a store release to that store's removal from the station should not exceed 50ms – Deadline = 50ms

Up to this point only the normal scenario has been identified. A scenario is a sequence of actions that illustrates the execution of a use case. Therefore a normal scenario simply represents the normal or expected sequence of actions which occurs for a particular use case, in this example releasing a store. When considering safety however it is important to consider alternative scenarios which may occur as these could potentially be hazardous, but may also lead to a requirement for extra timing constraints. In order to illustrate the scenarios clearly, the UML notation of activity diagrams can be used to show the different sequences of tasks which may realise the use case. Although activity states in an activity diagram are normally used to model a step in the execution of a procedure, here each activity state is used to represent a task or sub-task. We have found activity diagrams to be particularly suited to this application as they emphasize the sequential and concurrent nature of the tasks in a scenario. However, it is acknowledged that sequence diagrams could also potentially be adapted for this purpose. The alternative scenarios can be identified by omitting tasks from the normal scenario, adding in extra tasks (i.e. repetition of existing tasks), tasks occurring concurrently with other tasks or tasks occurring in an alternate order. It is necessary to identify if any of the alternative scenarios identified could be hazardous. That is to say that they could provide an additional contribution to the hazard. These scenarios could also necessitate additional timing requirements, which will form part of the safety contract (see section 4).

3.3 Value Aspects

The data represented in the system can also contribute to system hazards if important data attributes are incorrect. It is important for each system hazard to identify which data attributes are critical. These critical data items must be constrained to ensure that they won't contribute to the hazard. It is possible to take advantage of the information hiding principle inherent in OO when trying to place constraints. If the attributes of a class are private, it is only possible for them to be manipulated by operations provided by the class. It is therefore possible to protect the accuracy of data items by constraining the interactions that may manipulate that data. It is therefore important for safety critical systems that attributes of classes are declared as private such that they may be constrained in this manner.

For the system hazard 'incorrect store released' it can be identified (through a fault tree) that the pilot selecting an incorrect store, or the wrong store information being displayed to the pilot could cause incorrect store selection. This would be caused by the incorrect store being associated with a particular station. The critical attributes here (as identified from the class diagram in figure 2) are the station ID and store ID, which are associated through the location class. The only operation specified in this system design which can manipulate this data is the addStore() operation of the station class. When this operation is called on a station, the store ID passed as a

parameter is associated with the station through the creation of a location object. By constructing constraints it is possible to assure the store ID being passed is correct.

The nature of constraints can only be properly specified with a great deal of understanding about the system under consideration. Even more so than with functional and timing aspects of the system, the data within a system is very dependant on domain knowledge for deriving effective safety requirements

4 Specifying Safety Contracts

Once the safety requirements for the classes in the system have been derived it is necessary to specify them in a useful and meaningful way. There are a myriad of techniques such as Douglass' real-time annotations [9] suitable for specifying constraints in UML. In this work we have chosen to use OCL [10], which is a constraint language compatible with UML. The pre- and post-condition constraints from OCL can be used as the basis for safety contracts on operations. An OCL expression for an operation can be expressed as follows:

```
context Typename::opName(param1 : Type1,…):ReturnType
        pre: param1 > …
        post: result = …
```

The constraints expressed in this manner are all requirements on static aspects of the system. As can be seen with the example in section 2, it is often necessary from a safety perspective to express that events have happened or will happen, that signals have or will be sent, or that operations are or will be called. An extension to OCL then known as an **action clause** was proposed by Kleppe and Warmer [11] to address this problem. This has formed part of the response to the UML 2.0 OCL request for proposals submission where it has become known as a **message expression** [12]. To specify that communication has taken place, the *hasSent* (^) operator is used. A simple example is given below:

```
context Subject::hasChanged()
        post: observer ^ update(12,14)
```

The post condition here results in true if an update message with arguments 12 and 14 was sent to *observer* during the execution of the *hasChanged()* operation. *Update()* is either an operation that is defined in the class of observer, or it is a signal specified in the UML model. The arguments of the message expression must conform to the parameters of the operation/signal definition. Messages in OCL are particularly useful for describing the functional aspects of the safety requirements. From the results of the analysis carried out in the example, a safety contract can be defined for the store class which restricts the hazardous behaviour. This safety contract is shown below:

```
context Store ::release()
        pre: none
        post: WOW=false
              and
              Stores_Manager ^ checkWOW()
```

A further limitation of OCL is that no way is provided for representing constraints over the dynamic behavior of a system. Again an extension to OCL has been proposed for modeling real-time systems [13]. This provides a mechanism for representing deadlines and delays. Deadlines for operations can be represented in the following manner:

```
context
Typename::operationName(param1:Type1,…):ReturnType
          pre: …
          post: Time.now <= Time.now@pre + timeLimit
```

Where Time is a primitive data type that represents the global system time and *timeLimit* is a variable representing a time interval. In our examples we take the unit of time to be ms. The above constraint represents a maximum permissible execution time equal to *timeLimit* for the operation *operationName()*.

Delays in reactions to signals or events can be represented in the following manner:

```
context
Typename::operationName(param1:Type1,…):ReturnType
          pre: lastEvent.at + timeLimit >= Time.now
          post: …
```

Where *lastEvent.at* is the arrival time of the last event. This represents a maximum delay equal to *timeLimit* for reaction to the *lastEvent*. The requirements derived for the timing aspects of the store class can thus also be represented as part of the contract. The safety contract for the store classs of the SMS would therefore be of the form:

```
context Store ::release()
          pre: previous_release.at + 100 <= Time.now
          post: WOW=false
          and
          Time.now <= Time.now@pre + 50
          and
          Stores_Manager ^ checkWOW()
```

By using OCL to specify contracts in this way the safety requirements on each class in the system design are explicit. For all our case studies, OCL with the extensions described has proved suitably expressive. If necessary however, other constraint languages or extensions could be considered. To ensure it will not impact on the safety of the system, the class must meet the set of safety requirements consisting of all preconditions of interactions for which it is the client, and all post conditions of interactions for which it is the supplier. To know that a system will be safe it is necessary to show that the complete set of safety requirements are met. Specifying contracts in this way also makes it easier to deal safely with maintenance, change, and reuse. This is discussed more thoroughly in [14].

5 Tool Support

Our tool support, as developed to date addresses the analysis discussed in section 3.1.1. Tool support for this part of the analysis is particularly important given its potential complexity for real systems. The tools developed provide assistance in a number of ways. They can identify sequences of transitions that could lead to an identified hazard. The tools are also capable of generating mutations of transitions which may arise due to mistakes in implementation. It is possible to perform these in combination to identify which mistakes in implementation could lead to an identified hazard. The tools can also be used for test data generation by identifying what input values would trigger a transition, and what output values would result from that transition and inputs. This allows us to create test cases for any of the hazardous potential mistakes in our implementation. Before any of this can be done however it is first necessary to check that the state chart under consideration uses only notations that are analysable by the tools, and to formalise the state chart ready for analysis.

5.1 Well-formedness and Formalisation

The tool first checks that the state chart provided is analysable, that is that the state chart is well-formed. If this is the case then the state chart is translated into an ISO standard Z representation [15]. Both of these steps are fully automated. Once the tool has been initiated with the state chart design no user intervention is necessary. The rest of the toolset works from the resulting Z representation. This should ensure that the analysis is less dependant on any particular state chart dialect. We currently cope with Statemate and Stateflow. Each of the transitions in the state chart is represented by a Z schema. The inputs and outputs of the state charts become inputs and outputs of the schema. The labels on transitions become predicates over these inputs and outputs.

5.2 Mutant Generation

In section 3.1.1 it was shown how extra transitions can be added to the state chart to simulate faulty behaviour. These extra transitions are referred to in the toolset as mutant transitions. The five basic cases in 3.1.1 are the basis for the mutant generation by the tool.

1 - e[c] self-transition – This involves changing the destination state to be the same as the source state and dropping the action.

2 - not e[c] / a and 3 - e[not c] / a – These are generalised to negate any conjunct of the e[c] trigger, and also to negate the entire trigger. Where the trigger involves ordering relations, each can be mutated to use a different relational operator, e.g. < becomes <=.

4 - e[c] – This involves deleting the action. If the action is a sequential composition, further mutants are generated by deleting one of those actions.

5 - e[c] / b – This involves changing the actions. Where an action is the assignment of a Boolean, the Boolean assignment can be negated. Where an action is an

assignment of a number, the number assigned can be incremented/decremented, and any arithmetic operator can also be changed, e.g. + to –.

The tools generate all mutants that they can, for all transitions. The user merely initiates the mutant generation. It should be noted that using the tools allows many more mutants to be considered than would otherwise be feasible.

5.3 Hazard Detection

Section 3.1 illustrated the identification of hazards for a class in the system. In section 3.1.1 these were shown in a notation that is basically Z predicates, but which need declarations of the names to be legal Z, these were:

- State = release ^ ¬checkWOW
- State = release ^ WOW = true

By writing them in a Z schema with inclusion of the state chart's inputs and outputs, the identified hazards become available to the tools. This small and relatively simple step currently has to be done manually, but since it involves manipulation of Z could be automated. Hazard detection involves conjecturing whether a composition of transitions implies the hazard. The hazard might be implied never, sometimes or always. This can be done for the original state chart and for the mutant transitions. When composing only original transitions, *sometimes* is sufficient for concern. When composing transitions involving mutants, it may be preferable to focus on the *always* cases. The user chooses the maximum length of compositions to be considered, then the tools perform automatically. The cost increases rapidly with the number of alternative transitions exiting each state. Generating a large number of mutants can also have a devastating effect on the cost of hazard detection. It would be possible to allow the user to limit the number of mutants generated by the tool in order to ensure hazard detection remained practical. This could be based on the complexity of the initial state chart and the severity of the resulting hazard. Further work may look at this.

5.4 Test Data Generation

Test data generation for a transition amounts to finding a binding that exists in the schema. This determines input and output values simultaneously. The solutions are written out in a form suitable as input to a test harness tool. The user has only to choose the transition for which to generate test data. This allows tests to be constructed to check for the existence in the implemented design of any of the hazardous mutant transitions identified previously by the toolset.

6 Conclusions and Future Work

In this paper we have outlined an approach to developing safety critical systems using UML. This is based on analysis of the UML system model. We have described with a simple example how this analysis may be carried out. We have also described a

tool for automating some aspects of this analysis. This analysis derives safety requirements for classes in the system. We have shown how these requirements can be expressed in the form of safety contracts using OCL. This approach makes it possible to reason about the safety of individual elements of the UML model and thus makes it easier to safely change the UML design, as well as facilitating maintenance and reuse of classes or components in the system.

Future work will focus on verification that a design will meet derived safety requirements arising from safety contracts, and deal with violation of conditions. We will also look at supporting traceability, particularly with a changing or evolving UML model. It will also be necessary to define safety arguments in support of this approach such that UML may be successfully adopted into safety critical domains.

The work presented in this paper was funded by BAE Systems through the DCSC, and by the EPSRC through the MATISSE project. A great deal of work on developing the toolset was carried out by Stuart Bass, an intern student at the University of York.

References

1. Leveson, N., Safeware - System Safety and Computers. 1995: Addison-Wesley.
2. Lano, K., D. Clark, and K. Androutsopoulos, Safety and Security Analysis of Object Oriented Models. Lecture Notes in Computer Science, 2002. **2434**: p. 82 - 93.
3. MoD, Defence Standard 00-58: Hazop Studies on Systems Containing Programmable Electronics. 1996: HMSO.
4. Nowicki, B. and J. Gorski, Object Oriented Safety Analysis of an Extra High Voltage Substation Bay. Lecture Notes in Computer Science, 1998. **1516**: p. 306-315.
5. Gorski, J. and B. Nowicki, Safety Analysis Based on Object-Oriented Modelling of Critical Systems. Proc. SAFECOMP '96, 1996: p. 46 - 60.
6. Gorski, J. and B. Nowicki, Object Oriented Approach to Safety Analysis. Proc. ENCRESS '95, 1995: p. 338-350.
7. Douglass, B.P., Real-Time UML - Developing Efficient Objects for Embedded Systems. 1998: Addison-Wesley.
8. Bate, I., J. McDermid, and P. Nightingale, Establishing Timing Requirements for Control Loops in Real-Time Systems. Journal of Microprocessors and Microsystems, 2003. **27**(4): p. 159 - 169.
9. Douglass, B.P., Doing Hard Time - Developing Real-Time Systems with UML, Objects, Frameworks, And Patterns. 1999: Addison-Wesley.
10. OMG, Object Constraint Language Specification, in Unified Modeling Language Specification version 1.4. 2001, Object Management Group.
11. Kleppe, A. and J. Warmer, Extending OCL to Include Actions. Lecture Notes in Computer Science, 2000. **1939**: p. 440-450.
12. Warmer, J., et al., Response to the UML 2.0 OCL RfP - Revised Submission, Version 1.6. 2003, OMG.
13. Cengarle, M. and A. Knapp, Towards OCL/RT. Lecture Notes in Computer Science, 2002. **2391**: p. 390-409.
14. Hawkins, R.D. and J. McDermid, Developing Safety Contracts for OO Systems. Proc. 21st International System Safety Conference, 2003.
15. ISO, Information Technology - Z formal Specification Notation - Syntax, Type System and Semantics. First Edition ed. 2002: ISO / IEC

An Experiment in Applying UML2.0 to the Development of an Industrial Critical Application [1]

Sergio Cigoli[1], Philippe Leblanc[2], Salvatore Malaponti[1], Dino Mandrioli[3], Marco Mazzucchelli[3] Angelo Morzenti[3], Paola Spoletini[3]

[1]PARVIS Systems and Services s.r.l.
email: {cigoli, malaponti}@parvis.it

[2]Telelogic France
email: Philippe.Leblanc@telelogic.com

[3]Dipartimento di Elettronica e Informazione, Politecnico di Milano,
email: {mandrioli, morzenti, spoleti}@elet.polimi.it

Abstract. We relate on an experiment in applying the novel notation UML2.0 and a toolset centered on it to a case study in an industrial setting. After an overall description of the case study and of the project, we discuss the support provided by the notation and the toolset to the various phases of the test case development, with a special focus on timing features, which allows us to speculate on possible enhancements to the real-time simulator. **Keywords**: UML, UML2.0, critical system, real time, validation and verification, tool and method evaluation.

Introduction

For over a decade the members of the formal methods community asserted with great emphasis that the development of critical systems deserves the application of systematic and rigorous methods supported by effective automated tools, due to the seriousness of the consequences of their faults, in terms of economic, environmental, or human losses. In the area of industrial applications, for long time and to a large extent still now, critical systems are however designed and implemented adopting ad hoc techniques or at best manual procedures with scarce automated support, so that their design, albeit correct, may be poorly documented and full of loopholes. This was, in various circumstances and contexts, explained or justified by a series of reasons, among which we cite the following:

* Work partially supported by the ITEA EU project "UMsdL: the powerful UML-RT" and by the MIUR project: "QUACK: Piattaforma per la qualità di sistemi embedded integrati di nuova generazione".

a) the absence of true *methods*: we only have notations and algorithms, sometimes even tools, but these are not integrated into a unifying approach able to provide a systematic and manageable development process;
b) the absence of really industrial-strength tools: notations and algorithms are inadequate, from the point of readability and of computational complexity, so that a formal approach is not really scalable and it is impossible to apply it in an industrial setting;
c) the lack of standards universally recognized and adopted, which prevents the creation of a common base of knowledge and practice and the emergence of a small set of notations, methods and tools whose acceptance is wide enough to reach a critical mass and justify adequate investments in software tool development by the tool constructors and vendors.

In recent years a few positive events actually took place in the field of languages, methods and tools for design, validation and verification. We mention model-checking, which, in some application fields, has gained wide acceptance as an effective verification technique [4].

Another important outcome is the advent of UML2.0 [12], as it is called the "second version" of the well known Unified Modeling Language [6] which is the result of a rather long and laborious standardization effort carried out by the OMG. UML2.0 provides a sound semantic foundation to many notations that in the original UML were left (sometime intentionally) imprecise and as a consequence tend to be used in an informal, exploratory fashion, thus making the notation inappropriate for the development of critical systems. In parallel with the definition of the new UML standard, some tool constructors have developed and released tool suites for the development of critical systems that are compliant with UML2.0.

The present work relates on the experience of a research project financed by the European Union, centered on the application of UML2.0, and of the tool set Telelogic Tau Generation2 (TauG2) [13] supporting it, in the development of a critical computer based application in an industrial setting. This experience has therefore been achieved in a favorable context, where, for instance, the impediments to the adoption of formal methods outlined in points (a)-(c) above could potentially be overcome.

The main indication we obtained from our experience are: that formal methods can be actually exploited in industrial environments; that industrial strength tools are indeed available but still need important enhancements; that timing issues are one of the most relevant fields where such enhancements are needed; that an interesting alternative of the traditional path for building industrial tools and methods by evolving them from academic prototypes is also the other way around, i.e. starting from a well established and engineered industrial tool and enriching it by adding advanced and sophisticated features.

We relate on our experience through a description of the project in a rather informal way, without entering into the details of how the various notations have been applied to the case study, but presenting methodological remarks on how the features of the UML2.0 notations affected, mostly positively and sometimes negatively, the development of the systems that was the object of the experiment. Section 2 provides a wide-ranging description of the project; Section 3 deals in greater detail with matters concerning the UML2.0 notation and the employed tool suite; Section 4

delves more deeply into some semantic features of the notation and tools concerning the treatment of time (for the sake of shortness many technical details are omitted; the interested reader can find them in [8]); finally, Section 5 draws conclusions and outlines directions of future research. We assume a slender acquaintance of the reader with UML and UML2.0.

The project

Three partners participated to this project. Telelogic France is the French subsidiary of Telelogic A.B. that is the constructor of the TauG2 tool and the provider of the original methodology. PARVIS is a small enterprise, based in Italy, whose mission is to provide systems and services for parallel vision in a variety of industrial settings, the principal one being the quality control of banknotes printing. These systems are highly critical due to the economic relevance of the performed activity: they must be highly available and, due to the massive amounts of banknotes checks to be performed in a relatively short period of time, they have high throughput requirements and hence they must satisfy hard real-time constraints. Therefore, PARVIS supplied the case study used in the project. The third partner, Politecnico di Milano, is academic and contributed to the experimentation of tools and methodology and to their critical evaluation.

The project lasted about two years; the total effort spent by Telelogic was 21.5 person*year; that of PARVIS was 4,2 person*year, and that of Politecnico di Milano was 3.2 person*year. It had a very satisfactory outcome from the technical viewpoint, (in fact it has been selected as one of the three most successful projects of the current EU ITEA project round and it will receive a special award at the coming ITEA workshop, to be held in Leuven, Belgium, on October 10-11, 2003).

The system developed in the project is a so-called tracking unit that is in charge of tracing the position of banknotes inside the machine that performs the quality check. Banknotes are fed into the machine and travel, at a considerable speed, along a path that successively brings them to a set of points where the various checks for quality are performed (for instance, the front, back and transparency image of banknotes are collected by sophisticated cameras and analyzed by data- and computation intensive computer-graphics algorithms). At the end of the process the banknotes are distributed among a set of containers depending on the outcome of the quality control. The tracking must be quite precise to allow for a strict synchronization among data acquisition and processing and mechanical actions necessary to drive the banknote through its path to the checkpoints and to its final destination. The part of the overall project related to the application of the tools and methods to the case study lasted about one year, from February 2002 to March 2003, and was structured in the following phases:

- detailed definition of the case study and informal requirements analysis and identification;
- informal modeling of the system architecture in UML2.0 (by means of structure diagrams) and allocation of roles and activities to the identified parts;

- detailed, formal modeling of the complete system (structure diagrams defined in full detail and behavior of every component modeled via state-transition machines) and, simultaneously,
- validation of the model by means of static analysis (basically consisting of static type checking) and simulation;
- automatic code generation from the model;
- porting and integration of the code with the real-time operating system on the target machine, and final testing.

The transition among the various phases was rather smooth and easily manageable, so that the overall development process can be described with good approximation as having taken a true waterfall shape: no major revision or amendment was ever necessary, as all design decisions proved correct. This was certainly facilitated by the so called Model-Driven Development, promoted by UML2.0 and supported by TauG2, where most of the effort is focused on modeling and analysis activities and artifact construction is quite systematic or even automated.

UML2.0 and tool support to the development

Requirements elicitation and analysis were performed with the visual aid Use Case Diagrams, Sequence Diagrams and Activity Diagrams. These diagrams proved to be useful to represent in a graphical and standard fashion some information that is often provided in natural language together with generic and mostly informal graphical schemas or constraints expressed in algebraic form. For instance informal timing diagram can be translated in sequence diagram with timing information. (Notice that in this phase Sequence Diagrams are constructed manually, while in the following phases of the development they can be generated automatically from behavior descriptions provided in the modeling phase as state-transition diagrams.) Similarly, informal diagrams with boxes, arrays, and informative identifiers are typically rendered by structure diagrams.

Unfortunately the nominal data on performance, that are often well known from the beginning of the system development and constitute the main requirement to be satisfied by the design, could not be directly translated into UML2.0 due to the lack of a general enough descriptive notation allowing the designer to express timing properties independently of any operational model. For instance the following performance requirements, which are taken verbatim from the "Case Study Description" document [3], written mostly in English,

Transport speed range: full speed 10 m/sec (40 banknotes/s)
Frequency of tracking encoder: 4 kHz
Encoder tracking precision: 2.5 mm
Rate of decision sending (inspection results): every 25ms (full speed)
Decision point offset (distance from SP to DP): 2.4m

did not find any equivalent corresponding part inside the formal model of the system, not in the requirements analysis phase, nor in the detailed version produced during the formal modeling phase. These time-related issues could be at least partially addressed inside the OMG profile for schedulability, performance and time, which however is

just partially supported by TauG2 suite. In fact the scheduling, performance and time profile is very large, thus is hard to be fully implemented in an industrial modeling tool. In addition or alternative inside a formal model one would like to be able to write simple requirement formulas in a descriptive notation like linear temporal logic [10] such as the following

$$\text{signalToBeProcessed} \Longrightarrow \text{WithinF(SignalProcessed, 25 msec)}$$

that accounts for the first requirement above (the Inspection System must be able to process 40 banknotes/sec; since the processing of any signal is done in sequence for the various banknotes by a single physical device or software process, this requirement implies that each such processing must be completed within 25 milliseconds), or the following

$$\forall \text{BnkntId (Enter(BnkntId)} \Longrightarrow \text{WithinF(Output(BnkntId, pocket), 240 msec))}$$

encompassing the first and last line above (the full speed of banknotes on the conveyor is 10m/sec, and that the decision point offset, i.e., the distance from the start and finish points of the inspection path, is 2.4 meters).

In the definition of the system architecture and of the high-level design, we found extremely useful the use of structure diagrams (notice that this kind of diagram is a new feature of UML2.0): Parts (graphically depicted as boxes) were used to represent either data abstractions, or functional abstractions, or electrical and electronic devices such as sensors and actuators, and in general any physical component of the system under design. We used structure diagrams in a rather open and unconstrained fashion, paying little attention to follow the principles and suggestions of any Object-Oriented analysis and design methodology. For instance, the set of objects and relations among them, were defined before the classes to which they belong, but this could be managed smoothly with the support of the automatic checks provided by the tools, which proved extremely valuable under this respect.

Structure diagrams lead the designer to decompose the system under development into a hierarchy of state-transition machines that communicate via one-to-one connectors, through ports that define precisely the interface and protocols, thus avoiding the semantic complexities that arise, in some formal notations, from the notion of broadcasting signals to all the components of a specification.

The TauG2 tool suit supports UML2.0 structure diagrams by partitioning classes (and their objects) into active and passive ones: active objects have their own thread of control and can actively interact with other objects through signal exchange; passive objects, on the other hand, simply encapsulate data. TauG2 permits the direct mapping of active objects into threads of the real-time operating system of the target. We took advantage with some concern of this possibility, as we were worried about a possible overhead in the communication among active classes that would originate in the target architecture. After the system construction (supported by the tool through the automatic code generation from the model) and the porting on the target environment we could verify that in fact such overhead was quite limited and the synchronization among concurrent threads of execution took place as expected. This fortunate circumstance occurred, in our opinion, because the amount of resources in the target platform had been chosen adequately, so that marginal variations of the resource consumption (time and memory) did not affect the overall system's ability to meet its performance requirements.

The Parts at the "leaf level" in the structure diagrams hierarchy contain state-transition machines, whose input and output tokens consumed and generated by the transitions correspond to signals entering and exiting through the Ports and Connectors of the Part. The tool allows for two modes of visualization of the state-transition machine: a *state centered* mode that uses a textual representation of the UML action semantics and a *transition centered* mode where the action associated with a single state transition can be described in detail using a visual representation of the UML action semantics, inspired by SDL [9]; this notation has an abstraction level similar to that of a programming language: in fact, completing this part of the model amounts to coding the most minute details of the system being designed. It must be pointed out, however, that such a coding usually involves a quite limited effort and is not error prone as traditional coding in a purely textual programming language, both because of the visual aids (almost all programming constructs are rendered via some graphic object) and because the code corresponding to a single machine transition is in fact embedded in the context of the machine itself and of the Part that contains it, hence these program fragments are usually rather simple, for instance they rarely include an iteration. Both notations, textual and graphical, conform to the UML action semantics [11].

As anticipated in Section 2, the project reached its natural termination rather smoothly, satisfying its budget of resources and time. The overall effort was about uniformly balanced between the following activities: requirements analysis and specification, architecture definition and high-level design, detailed design, final testing. It should be noticed that this is NOT the usual division of effort in a typical industrial project. Notice also that in the above effort estimation we do not consider the implementation phase; this is due to essentially two reasons. First, the code is not produced by hand, but is completely generated automatically by the tool; second, and more precisely, there is in fact a (limited) coding activity that corresponds, as remarked above, to the specification of state transitions via transition-centered diagrams inside the leaf-level Parts.

We consider this a great improvement with respect to normal industrial practice, where the final testing phase fatally consumes a prevailing part of the total human resources, because it includes re-design and debugging that are typically necessary when errors are found and, possibly, some design decisions must be reconsidered. This scenario matches the usual figures that are reported in the literature on (successful!) applications of formal methods in an industrial setting, which leads to the common place belief that a judicious usage of Formal Methods makes the entire development process simpler and more manageable.

We must observe, however, that the exploitation of Formal Methods in the present project has been rather "light" [5] because validation and verification were carried out exclusively through simulation, and therefore they provided results that were only "partial" (i.e., non exhaustive, being related to a subset of all possible system executions) and "qualitative", especially from a timing and performance point of view. Indeed, a quantitative treatment of time, and therefore the possibility of specifying/modeling real-time and performance features, was present in the modeling language, through the *Timer* construct that is a tool extension inspired by SDL, but not in the simulation tool, which did not support a notion of real-time measured in a quantitative fashion. As a consequence, the model validation obtained through

simulation and the test cases derivable from the Sequence Diagrams generated by the simulation tool could only account for the correctness of the relative ordering among events, without providing a possibility to verify *a priori* the actual value of the performance times under defined hypotheses to be assumed on the target platform, and therefore the satisfaction of the hard real-time requirements that were in fact present in the case study. Although this limitation did not affect severely the success of this project -essentially thanks to the rather abundant sizing of the target architecture- in general one may need deeper analysis features from the point of view of time behavior. This is the object of the analysis and proposal described in the next section. Also, the verification of the desired system properties in a complete and exhaustive fashion (comparable, e.g., to model-checking or property proof) could not be performed on the model developed in this project, as the model itself lacks a notation for expressing properties. Concerning this issue, we mention that several research lines are actively pursued [7], aiming at the enhancement of OCL (the Object Constraint Language) of UML and of the UML2.0 notation itself with constructs for describing formally, directly, and succinctly the desired real-time properties.

Model simulation and timing features

We stated in the previous section that the tool suite does not provide a complete real-time simulation, where each event occurrence is labeled by a time stamp that represents its actual time as computed by the simulator. In the present section we show how we analyzed the tool's features from the point of view of real-time simulation and how we designed its evolution to enhance such features.

In the original version of the simulator that is included in the commercial tool suite the UML ordinary signals and the signals originating from the expiry of timers were treated in the very same way: timers' signals did not have any priority over other signals, so that timers lose their ability to drive the timing of the execution by "forcing" the firing of other transitions depending on their expiry.

In strict cooperation with Telelogic as the tool provider, however, we were able to make slight but crucial modifications of the simulator that allow for a different treatment of the ordinary signals -which are still managed in a FIFO queue- and of signals originated by the expiry of a timer -which are managed in a separate queue with a priority related to the respective expiration time. These modifications permit the management of real time simulation in a more complete and realistic way.

We designed an evolution path for the tool through several *tool versions*. In the following we briefly account of such an evolution by referring to version alpha as the original, commercially available one. We illustrate our experience through a couple of examples specifically devised for this presentation

Throughout the discussion, variable "Now" will represent the value of the real time of the various event occurrences, as computed by the simulation tool.

Example 1

As a first example we consider a simple echo server that receives from the environment messages associated with a numerical value indicating when the server

has to give a reply, consisting of a formatted echo, to the environment. The echo server uses a timer (the TauG2 construct to model time-outs) to set the time of its reply.

More in detail, as shown in Figure 1, the system is composed of an active object of class Server, that receives signals “Say” and sends signals “Echo”.

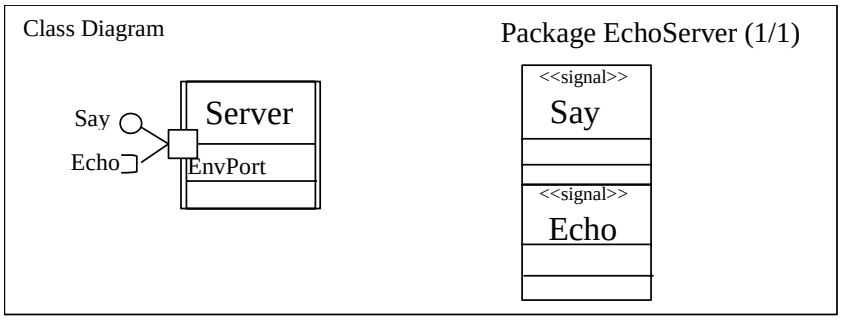


Figure 1. Class diagram for the Echo Server.

The class Server is composed of the active class RequestHandler and its associated EchoFormatter (Figure 2). Each instance of the class RequestHandler manages an echo request, sending to the environment the signal echo, of which the parameters have been created through the operation format of the EchoFormatter class.

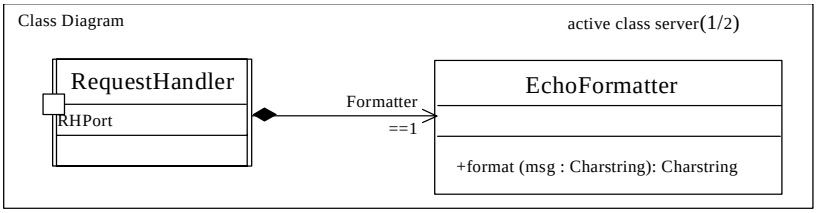


Figure 2. Class Diagram for Server

The structure diagram in Figure 3 shows that the server receives the request from the environment through the signal Say, while the echo signal is sent to the environment from the RequestHandler.

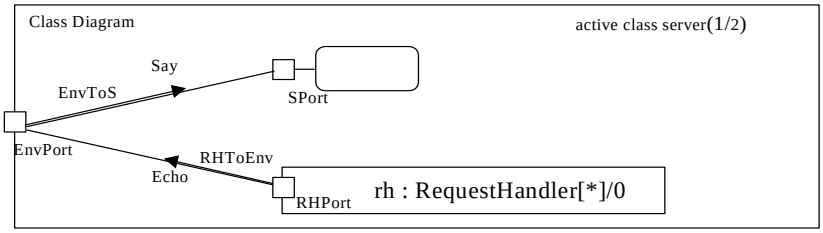


Figure 3. Structure diagram for class Server.

The behavior of EchoServer is described by the state diagrams in Figures 4 and 5, for classes Server and RequestHandler, respectively. The system behaves as follows. The environment sends a Say signal including a string and a numerical value. These are received by the server that passes the control to an instance of the class RequestHandler and then returns to the idle state. The RequestHandler sets a timer at the numerical value passed through the signal Say; when the timer expires a new string composed by doubling the original string through the operation format of the class EchoFormatter is sent to the environment.

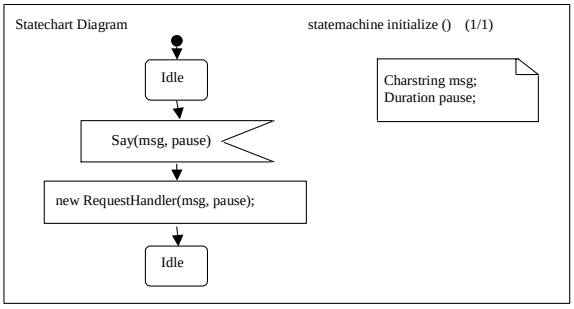


Figure 4. State diagram of Echo Sever, page 1.

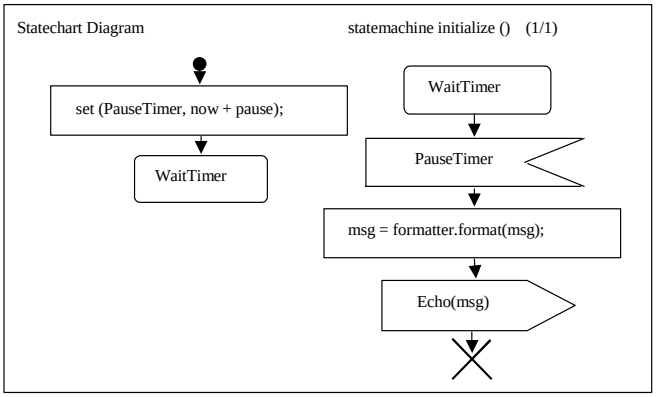


Figure 5. State diagram of Echo Server, page 2.

We first generated simulations of this system by means of version alpha.

By inserting a breakpoint on the timeout or on the reception of the signal “pauseTimer” it is possible to check that when N signals “Say” are simultaneously input and the simulator is in a stable state (i.e., with no signal to be processed) all signals are processed before the first timeout occurs even if (looking at the value of the variable Now) it seems that the signals are processed *t* time units after the timer setting.

For instance, if we supply the tool with the following sequence of input signals (say(<string, timer value>)):

say(john,10)

say(mary,4)

say(mary,4)
say(mary,4)
say(mary,4)
say(mary,4)
say(mary,4)
say(mary,4).

The Sequence Diagram generated by the simulation with version alpha contains eight signals “Say“ that are sent by the environment and received by the server when the variable Now is zero. Then, still with Now equal to zero, an instance of the class RequestHandler (RH) for each signal Say is activated and each timer is set by the corresponding instance of RH. When the timers with the least setting value expire, i.e. the timers set by the seven signals Say(mary,4), the variable Now assumes the value 4 and then changes again only when the last timer expires. Hence, except for the incrementing due to the timers, time does not pass, as if each instance of the classes was evolving in its own timing environment rather than in a global one. In fact the signals are sent and received by the different instances of RequestHandler in zero time and every change of states of the instances takes zero time and does not influence the other ones.

This remark led to a first modification of the tool, i.e., version beta. In the beta version the timeouts are not treated in the same way as other signals. However the variable “Now” increases randomly, giving sometimes rise to unexpected results even if the time consistence is kept. For instance the Sequence Diagram produced by version beta in correspondence of the above input sequence, exhibits a correct relative order of the expiry signals of the timers with respect to the values of their settings, but the timeout signals themselves occur with a delay that is not predictable. Moreover the expiry signals arrive long time before the state changes caused by the timer expiry. Thus, version beta produces a behavior that is consistent with the logic sequencing of the events, but the random increasing of variable Now is not what the user would expect as a final result.

A further minor modification, however, could easily produce the “right time semantics”: it would suffice to allow the user to specify explicitly the time duration of every transition. Such a duration would define explicitly the increment of the Now variable in the tool. To achieve the maximum generality such an increment could be any nonnegative value, even depending on other system variables. Such a generality, however, would have a highest price in terms of executability since an infinite number of system behaviors would be possible even in a finite time interval. As a reasonable compromise, instead, version gamma was designed, where the possibility was given to the user to associate a constant time interval with the transitions. A large literature, e.g. in the field of time Petri nets [1, 2] shows that such an approach allows the user to deal with most cases of practical interest. On the other hand version gamma of the tool supports a structured way of performing simulations, whereby one can build a finite number of execution traces that represent the totality of all the traces of the system being modeled, which are potentially infinite in number.

The possible execution traces of a system are characterized by the transitions occurred, the relative order of their firing and the absolute firing time of each transition. One can easily define an equivalence relation over the set of the execution

traces, where traces are equivalent if they contain the same transitions firing in the same order; hence the traces can be grouped into equivalence classes, and the whole behavior of a system can be in principle be described by a simulator generating one trace per class.

Next, we briefly describe the behavior of version gamma by means of a second example.

Example 2

As second example we consider the following simple system, a simplified version of the classical "dining philosophers" problem, shown in Figure 6. Three cyclic processes, P1, P2 and P3, compete for two resources (represented in the diagram with the variable Res), and switch between two states: running and waiting. When a process i is in the waiting state and there is an unassigned resource, i.e. Res>0, the process can take the resource. When process i goes from the running state to waiting it releases the resource. Note that in the system there are only two available resources, hence only two processes could be enabled at the same time. The hexagons (containing the general interval $[m_i, M_i]$) near the transitions in Figure 6 represent an additional constraint that can be provided by means of the tool; they mean that when a transition is enabled it can fire only in the interval $[m_i, M_i]$ from its enabling time, if it is not disabled in the meantime.

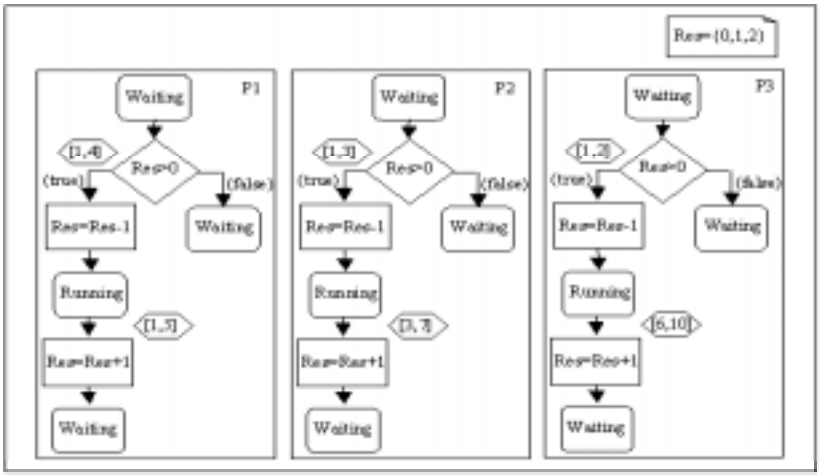


Figure 6. State diagrams of the three processes.

In our illustration of the simulator in version gamma we will use the following notation. The transitions of processes P1. P2,. and P3 from waiting state to running state are called A, B, and C respectively, while the transitions from the running state to the waiting state are respectively called D, E and F.

Since the three processes have the same behavior but the resources are only two, the transitions A, B and C need the firing of D, E and F respectively and at most one process in the running state. If two processes are in the running state and the other one is in waiting state, the transition of this latter process to running state is not enabled (Res=0). On this system we now apply the algorithm incorporated in version gamma

of the tool, which is reported in full detail in [8]; in the presented example we build traces with 4 fired transitions.

First we build the tree of all the possible traces, called relative tree and reported in Figure 7, whose edges are marked with the name of the fired transitions and whose nodes contain the relative intervals with respect to the last fired transition of all the enabled transitions and the ordering constraints among the enabled transitions (these constraints are omitted in our example since they are redundant). Note that the relative interval of transitions that remain enabled in more than one node, i.e. transitions enabled in a node that are not effected by the firing of another transition, changes from one node to the next in a given trace.

From the relative tree, considering only relative intervals and ignoring the additional constraints, we can build the absolute tree. The absolute tree is a structure whose nodes contain absolute firing intervals for the enabled transitions, and whose edges are marked with the absolute interval in which a transition must fire to reach the child node. The root of the absolute tree is simply created by copying the relative firing intervals from the relative tree and the edges coming out from the root contains the relative intervals of the fired transition in which the maximum is limited by the least maximum value of the transitions enabled in the root. Then to create the absolute firing constraints for a generic node at depth w+1 we change the relative intervals in the following way:

- If a transition is enabled both in the considered node and in its parent node then the interval is copied changing the minimum if the minimum of the fired transition is greater than that of the considered transition.
- If a transition is newly enabled in the considered node its absolute interval is obtained by adding the absolute interval of the fired transition to the relative interval of the transition.

As an example, the trace CBFE of the relative tree in the absolute tree becomes:
$[1\leq A\leq 4,1\leq B\leq 3,1\leq C\leq 2] \rightarrow 1\leq C\leq 2$
$\rightarrow [1\leq A\leq 4,1\leq B\leq 3,7\leq F\leq 12] \rightarrow 1\leq B\leq 3$
$\rightarrow [4\leq E\leq 10,7\leq F\leq 12] \rightarrow 7\leq F\leq 10$
$\rightarrow [8\leq A\leq 14,8\leq C\leq 12,7\leq E\leq 10] \rightarrow 7\leq E\leq 10$ (the inequalities in the square parentheses represent the key of the nodes, while the inequalities between arrows are the labels of the edges)

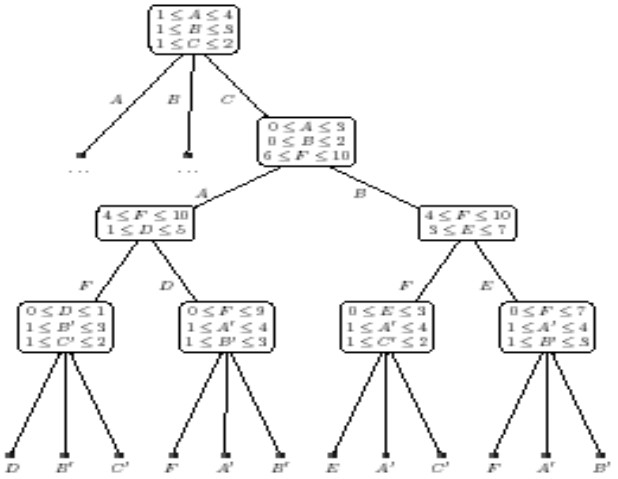


Figure 7. The tree of all possible traces (relative tree).

Now starting from the relative and the absolute tree we can build the admissible domain of each trace, i.e., the domain where the firings can actually take place. We present a method for creating evolution constraints, so that a user can fix the firing time of each transition for a considered evolution trace. The described procedure must be repeated L times where L is the total number of evolution traces, which is equal to the number of leaves in the tree. The procedure computes the intersection of all the constraints regarding a fired transition in order to reduce its absolute domain to the admissible domain. Continuing with the previous example, let us consider again the trace CBFE. The trace has to be considered starting from the last fired transition in the trace. So we start with E. We group the absolute interval of E with the constraints obtained by the relative tree, i.e. starting from the node that generates the edges on which E fires we visit the tree toward the root until the transition E is present in the node and we add the constraint in the node that connects E with the just fired transition. The constraints of E are

$$\begin{cases} 7 \le E \le 10 \\ F \le E \le F + 2 \\ B + 3 \le E \le B + 7 \end{cases}$$

From it we can deduce the constraints $B \le F \le B+7$, $0 \le B \le 7$ and $4 \le F \le 10$. Now we consider the constraints of F adding also the constraints deduced by the constraints of E and so on. If the set of constraints includes two constraints on the same variables they are replaced by their intersection, computed by taking the greatest minimum and the least maximum values. As result of this procedure we obtain the following trace

$$1 \le C \le 2 \rightarrow \begin{cases} 1 \le B \le 3 \\ C \le B \le C + 2 \end{cases} \rightarrow \begin{cases} 7 \le F \le 10 \\ B + 4 \le F \le B + 7 \\ C + 6 \le F \le C + 10 \end{cases} \rightarrow \begin{cases} 7 \le E \le 10 \\ F \le E \le F + 2 \\ B + 3 \le E \le B + 7 \end{cases}.$$

Hence starting from C the user can fix arbitrarily a point in the interval of C creating a numerical interval for B, and so on. A possible member of this trace can be the following: C=1.5 → 1.5≤B≤3: B=2→ 7.5≤F≤9: F=8.5→ 8.5≤E≤9: E=8.5.

Conclusions

We reported on "yet another experience in the application of Formal methods to industrial projects". Previous experiences, however, were mostly ignited by a research institution that tried to promote its own approach within the industrial environment. This project, instead, was originated in an industrial environment and based on a commercial tool, whereas the academic partner played the role of critical evaluator and enhancements proponent.

The overall results of the project are largely positive: the design goals were reached within schedule and the industrial partner who provided the case study is satisfied by the fact that the development process was cheaper, smoother, and better manageable than in previous projects of similar systems. In our opinion, the favorable result comes from the ability of UML2.0 to provide a semantically coherent view of the system under development, that covers all phases of the development and allows one to support it through a nicely integrated set of tools.

UML2.0 and its related toolset thus make an important step towards *model-driven development*, that is a development process where the design effort is mainly devoted to modeling activities while the construction of the actual artifact is greatly facilitated or even completely automated by tools that translate the model into executable code.

We have also pointed out the main weaknesses of the adopted method and toolset:

- the absence of a descriptive notation for expressing performance and real-time requirements; this prevents the construction of verification tools that provide more guarantee than, and complement, traditional simulation and testing,
- the lack, in the current version of the simulator tool, of a feature for real-time analysis.

The former issue involves the very definition of the UML2.0 notation itself; therefore, it is addressed in the context of medium/long term research, essentially centered around the development of the OCL standard; the latter instead can be addressed adequately by relatively simple modifications of tool's internals that manage UML signals and the value of the "Now" variable. Suitable algorithms to implement version gamma as described in Section 4 have already been implemented, though not yet plugged into the tool.

References

1. B. Berthomieu, M. Diaz, Modeling and verification of time dependent systems using time Petri nets. IEEE Transactions on Software Engineering, 17(3), 1991.
2. G.Bucci, E.Vicario: Compositional Validation of Time-Critical Systems Using Communicating Time Petri Nets. Transactions on Software Engineering, 21 (12): 969-992 (1995)

3. S.Cigoli, S. Malaponti, UMsdL Case Study Description, PARVIS Systems and Services, Milano, Feb 2003.
4. E. M. Clarke, O.Grumberg, D. Peled, Model Checking, MIT Press, 2000.
5. D.Jackson and J.Wing, *Lightweight Formal Methods, 29(4):21,* IEEE Computer, April 1996.
6. I.Jacobson, G.Booch, J.Rambaugh, The Unified Software Development Process, Addison-Wesley, 1999
7. L. Lavazza, S. Morasca, A.Morzenti, A Dual Language Approach Extension to UML for the Development of Time-Critical Component-Based Systems, TACoS International Workshop on Test and Analysis of Component Based Systems, Warsaw, April 13th, 2003, in conjunction with ETAPS 2003.
8. D.Mandrioli, M.Mazzucchelli, A.Morzenti, P.Spoletini, Generating simulation traces for systems with interval transitions, Technical Report N.2003.29, Dipartimento di Elettronica, Politecnico di Milano, July 2003, http://www.elet.polimi.it/upload/morzenti/publications/tr2003.29.pdf
9. A.Mitschele-Thiel, System Engineering with SDL – Developing Peerformance-Critical Communication Systems, John Wiley, 2001.
10. A. Morzenti, P. San Pietro, "Object-Oriented Logic Specifications of Time Critical Systems", ACM TOSEM-Transactions on Software Engineering and Methodology, vol.3, n.1, January 1994, pp. 56-98.
11. OMG, Action Semantics for the UML, OMG ad/2001-03-01, www.omg.org, 2001.
12. Unified Modeling Language: UML 2.0 Superstructure Final Adopted specification, ptc/03-08-02, Aug-2003.
13. www.taug2.com .

Creating Security Mechanism Aspect Models from Abstract Security Aspect Models

Geri Georg, Robert France, and Indrakshi Ray

Department of Computer Science
Colorado State University, Fort Collins, CO 80523

Abstract. Aspect-oriented modeling (AOM) techniques allow system architects to design the most important decompositions of complex systems to create a primary system modularization. These techniques can also be used to design additional system concerns that are not part of the primary system modularization. Aspect-oriented modeling techniques can be used to compose different aspect models with the primary decomposition models in order to analyze the complete system design. The results of analyses can be used to compare potential design realizations of multiple competing concerns. Aspect models, composition, and analysis techniques must be available at different levels of abstraction to enable comprehensive trade-off analysis among competing concern realizations.

Different levels of abstraction are particularly important when multiple mechanisms are available to realize a concern, such as in the area of security. Architects need to experiment with different security mechanisms in order to choose those that best meet overall system goals while providing minimal interference with other design considerations. Abstract aspect models can be used to develop more detailed mechanism models that are still independent of implementation considerations. These detailed models can be used for mechanism analysis and trade-off experimentation.

We have created two detailed authentication mechanism models using an abstract aspect model, and we demonstrate the steps used to create the detailed model for one of these mechanisms in this paper. Although not discussed in this paper, we have composed these different mechanism models with primary decomposition models using the same AOM composition techniques that we use to compose abstract models. The resulting compositions allow system architects to analyze different mechanisms available to realize a particular abstract concern, such as authentication. Architects can use analysis results to make design trade-off decisions and choose the mechanisms that best meet overall system requirements. We are continuing to evolve this work to define a refinement mechanism for our prototype tool.

1. Introduction

Software architects decompose problems and their solutions to manage complexity during software system development. Decomposition partitions multiple competing concerns in the problem and solution space. Key design decisions made early in the design process identify some of these concerns as most crucial to the system, and realizing these concerns determines the modular structure of the design. We call the result of these early design decisions the primary decomposition, or primary modularization of the system. Models of this decomposition are called primary models. Other concerns that are not used to determine this primary structure may be equally important to meeting all the system goals. Realizing these additional concerns often involves adding functionality that cross-cuts the primary modularization. The way these concerns are realized can impact each other, and sometimes the primary system functionality.

We can treat an additional system concern realization as an aspect that cross-cuts functionality of the primary system modularization, then use composition techniques to integrate it into a primary modularization model for the purpose of analysis. We model these concern realizations as design aspect models, using the UML [20] to specify the structures and behaviors that realize the concern. A structural model identifies required entities and their relationships with each other, and dynamic models define behavior. We have found that design aspect models assist in separating concerns, while *composing* multiple models provides the necessary integration of different pervasive concern realizations to allow architects to experiment with different designs that realize a concern. Composition also allows developers to create a complete system model if it is needed for continued system development.

We have developed design aspect models that specify abstract realizations of security concerns such as authentication, access control, and auditing. These models are abstract in that they do not specify any particular mechanisms to realize the security concern. Some conflicts can be identified when abstract models are composed with primary models. These kinds of conflicts are often the result of conflicting relations between model classes or multiplicities between models. There are more subtle conflicts that are not apparent at high levels of abstraction, such as when a portion of a required behavior is compromised as the result of model composition. These kinds of conflicts may not be apparent until more detailed aspect models are available. In addition, system designers must be pragmatic during design, and they must choose a particular mechanism to realize a security concern. For example, access control can be realized using mechanisms such as mandatory access control (MAC), role-based access control (RBAC0, RAC1, etc.), or discretionary access control (DAC). Each mechanism realizes access control in a different way, with different effects on the system. Some may cause conflicts with other design concerns in the system, while others may not. Therefore developers need to be able to experiment with different mechanisms and analyze their impact on an overall system design. When different mechanism models are composed with a primary model, developers are able to analyze various design choices and choose the mechanisms that cause the fewest composition conflicts, or those that are simplest to resolve.

We can use abstract aspect models to create different detailed mechanism models for comparison purposes. Each model can be composed with the primary model, and then be analyzed in order to determine which mechanism best meets the overall system goals. It is important to note that although mechanism models are more specific than an abstract model, mechanism models are still abstract in the sense that they are platform independent. No particular implementations are reflected in the models.

We demonstrate our method using an abstract authentication aspect model to create a mechanism model. The abstract authentication model is used to create a mechanism model that uses a shared-secret mechanism to provide two-way authentication between two entities in a system. The mechanism is Secure Remote Password (SRP). (See [21] for details of the SRP mechanism.) Two-way authentication means that both entities in a communication authenticate each other. We have successfully composed these models with primary system models. However, the composition step and results are very similar to our previous composition results (e.g. [6,7]) and are not repeated in this paper.

The rest of the paper is structured as follows. Section 2 describes design aspect models in the AOM method and presents the abstract model of two-way authentication. Section 3 presents the refined model that specifies the Secure Remote Password (SRP) authentication mechanism. Related research is discussed in Section 4, and conclusions and future work are presented in Section 5.

2. Abstract Design Aspect Models

We define design aspect models in terms of structures of roles called *Role Models*. We developed Role Models initially to specify behavioral and structural properties captured by design patterns [5]. Our treatment of aspect models as patterns allows us to use a template form of Role Models to represent design aspect model elements. The template form of Role Models was developed to facilitate tool-supported composition of design aspect models and primary models.

A Role Model defines a pattern of UML model structures. It is a structure of roles, where a role defines properties that must be satisfied by conforming UML model elements. A UML model is said to conform to (or realize) a Role Model if (1) all of its model elements conform to the roles in the Role Model and (2) the structure of the UML model is consistent with the structure characterized by the Role Model.

The roles in our aspect models specify the properties that are to be incorporated into user designated points of a primary model. Each role can be considered a potential integration point with the primary model. Thus, each design aspect model specifies its own integration model, which allows considerable flexibility and reuse in aspect model definitions. The specific integration points with a primary model are not specified as part of a generic aspect model. Instead, before

model composition occurs, the modeler must indicate which primary model elements are intended to "play" the roles specified in the aspect model by creating a context-specific aspect model. These model elements become the integration points between the aspect model and the primary model.

We are able to accomplish two goals by describing aspects as patterns containing templates. First, we are able to create composed models that correctly incorporate aspects by construction. This is due to the fact that composition is essentially a model transformation process in which a non-conforming primary model is transformed to a conforming model. Second, the use of templates allows us to make simplifying decisions in our prototype tool. We can simply "stamp out" portions of an aspect that are missing in the primary model.

Abstract aspect models of two-way authentication are shown in Figure 1 and Figure 2. Figure 1 presents a static model of this aspect, while Figure 2 presents the dynamic behavior associated with authenticating an entity.

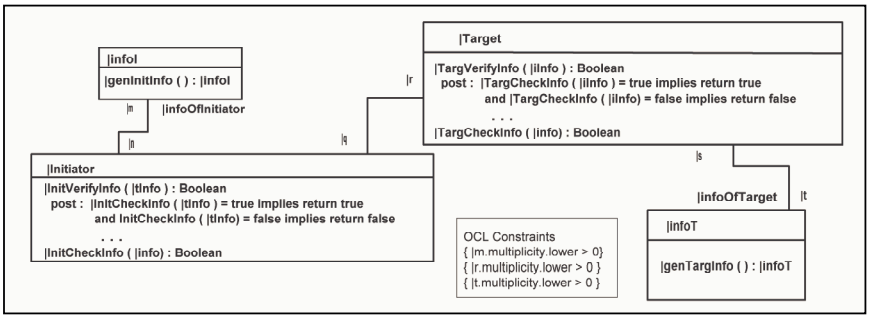


Figure 1. Static model of an abstract authentication aspect.

Figure 1 shows a static model of an abstract authentication aspect. Unlike the authentication aspect presented in our previous papers [6,7], this authentication aspect shows two-way authentication, which is often used in peer-to-peer distributed systems. Two-way authentication allows both peers to authenticate each other before a communication link is established. If either party fails the authentication, the communication link is not established. Authentication is based on some information that is generated (or perhaps just stored) in an *initiator*, and is then passed to a *target* for verification. The *target* in turn also generates some information that is passed to the *initiator* for verification. The information can be something that each entity shares, such as a secret password, or it can be an identification certificate, or some other information.

Figure 1 shows the use of a | preceding a name to indicate the presence of a role in the aspect model. Roles are played by existing entities in a primary model, or are created during composition with a primary model if no entity previously exists that to play the role. OCL constraints are used to place bounds on the acceptable values of association multiplicities. If no constraints are given for a multiplicity variable, any value is acceptable. Multiplicity role values are substituted with values that meet the OCL constraints as part of the composition with the primary model.

Figure 2 shows the associated behavioral diagram for performing two-way authentication.

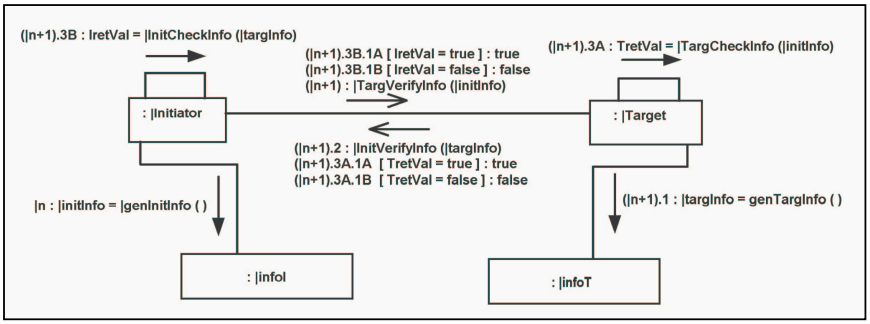


Figure 2. Dynamic behavior model of the abstract authentication aspect.

Figure 2 shows one dynamic behavior of the abstract aspect shown in Figure 1. The model shown is a portion of a UML 1.4 collaboration diagram, with numbered messages indicating the behavior flow. Note that in this diagram, although the initiator begins the authentication process, the target sends its own authentication information back before beginning the verification of the initiator. Verification then proceeds concurrently in the target and initiator. The role |n is substituted with the message number that needs to begin the authentication process during composition with a primary model. Composition then involves some re-numbering of other messages in the primary model to properly position the entire authentication sequence. (See [7] for a discussion of dynamic model composition in the AOM method.)

3. Creating Mechanism Models

Abstract aspect models such as those shown in Figures 1 and 2 can be composed with a primary model for analysis purposes. When multiple aspect models are composed with a primary model, gross composition conflicts can be identified. For example, in a previous paper [6], we demonstrated how the composition of an auditing aspect with a primary model, followed by a one-way authentication aspect composition led to a conflict. In this case, the authentication methods would not be audited, unless the auditing aspect was reapplied. Other conflicts that can be detected from very abstract aspects have to do with association multiplicities. An example of this kind of conflict is when a single repository in a primary model is replaced with a set of repositories during composition with a fault tolerant aspect using replicated repositories. The resulting conflict is the multiplicity of the one repository in the primary model versus multiple repositories in the aspect model. Since the entire reason for using replicated repositories is so that more than one repository can be used in the event of failure, the conflict must be resolved in favor of the aspect model

multiplicity. These types of conflicts are examples of cases where a property in one aspect contradicts a property in another aspect or the primary model.

Conflicts can also result from interference between aspect behaviors. Such compromised behavior occurs when a behavior required by an aspect cannot be performed as specified because some of its sub-behaviors have been modified (or deleted) as a result of merging behaviors with other aspects or the primary model. For example, an aspect may introduce a relationship between two entities that is prohibited by another aspect, or an operation introduced by an aspect may result in behavior that violates requirements previously satisfied by the original operation.

In order to identify these latter kinds of composition conflicts, more detailed mechanism aspects are needed. Abstract aspect models can be used to create these mechanism models. When a more detailed design is composed with a primary model, either conflicts that appear at lower levels of abstraction can be identified, or there is more evidence that conflicts will not be present as a result of choosing a particular mechanism to realize a concern. In addition, abstract aspects that can be realized using several different mechanisms can be used to create different mechanism models. When different mechanism models are composed with a primary model, developers are able to analyze various design choices and choose the mechanisms that cause the fewest composition conflicts, or those that are simplest to resolve. This paper describes mechanism models for authentication. Like access control, authentication can be realized using several different mechanisms. Examples are mechanisms that use something the entity knows (e.g. a password, or some other shared secret), mechanisms that use some characteristic of the entity (e.g. fingerprint or retinal prints of a user), or mechanisms that use something the entity has (e.g. a security card). Mechanisms that use something the entity knows include password and certificate mechanisms. Of these, password mechanisms are generally less complex than certificate mechanisms due to the additional setup needed for certificates (in particular, a trusted certificate authority that is needed to create certificates and distribute them to all entities that request a certificate).

We have created detailed aspect models for two authentication mechanisms to use in composition conflict analysis.[1] The mechanisms are Secure Remote Password (SRP) which is a shared secret based authentication mechanism, and Secure Sockets Layer (SSL) which is a certificate-based authentication mechanism. (SSL actually contains multiple security mechanisms, only one of which is authentication; SSL is not discussed further in this paper.) An example of a portion of the static structure of the SRP mechanism is shown in Figure 3.

[1] Aspect models were created from design descriptions developed with Julio, Garcia, Agilent Laboratories when one of the authors was at Agilent Laboratories.

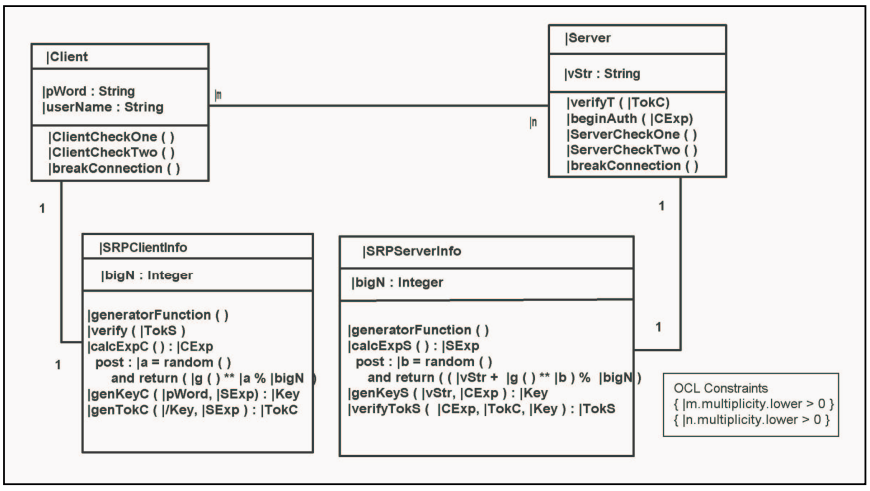


Figure 3. Two-way authentication aspect static diagram for the SRP mechanism.

The diagram in Figure 3 shows a portion of the SRP static structure diagram. There is a set up portion of the SRP mechanism that is not shown. This setup consists of a string generator that takes the client's password and user name, and creates a verifier string that is used by the server during the authentication process. The classes and methods involved in this setup are not shown in Figure 3.

A conceptual explanation of the method used to create the SRP mechanism model is as follows. First, the *initiator* and *target* in Figure 1 corresponds to the *client* and *server* in Figure 3, respectively. Similarly, the *infoI* and *infoT* classes in Figure 1 correspond to the *SRPCLientInfo* and *SRPServerInfo* classes in Figure 3, respectively.

Method correspondence is more complex. The *TargVerifyInfo* method in Figure 1 corresponds to the *beginAuth* method of the *server* class in Figure 3. There is no direct correspondence between the *InitVerifyInfo* method of Figure 1 and any method in Figure 3. This is because the function of the *InitVerifyInfo* method is simply to use the equivalent of the *InitCheckInfo* method directly. However, the *InitCheckInfo* method in Figure 1 corresponds to *ClientCheckOne* and *ClientCheckTwo* in Figure 3. This is because in the SRP protocol, there are actually two checks that occur on data transferred from the server. Similarly, there are two checks performed in the server for data transferred from the client, so *TargCheckInfo* in Figure 1 corresponds to *ServerCheckOne* and *ServerCheckTwo* in Figure 3. (Additional data transfer and calculations occur between these two checks in both the client and the server in the actual SRP protocol.) The *genInitInfo* and *genTargInfo* methods in Figure 1 correspond to the *calcExpC* and *calcExpS* methods in Figure 3, respectively. The other methods shown in Figure 3 are further refinements of the

methods previously discussed. A refinement has been developed for the SRP authentication protocol, based on the abstract behavioral diagram shown in Figure 2. It is not included in this paper due to space considerations.

Once detailed models are created for different aspect mechanisms, the composition process can identify additional conflicts, or provide more assurance that conflicts do not exist between composed aspect and primary models. Composing different mechanism models thus gives developers the data needed to make design decision tradeoffs. (We do not discuss composition further in this paper since it is described in our other papers.)

4. Related Research

Model-Based Development (MBD), sometimes referred to as Model-Driven Development (MDD, see [2]) is concerned with using models as the primary artifacts of software development, and is supported by techniques for rigorously analyzing models and for generating production-strength implementations from models. MBD methods raise the level of abstraction at which developers compose and implement systems. Examples of MBD approaches are (1) the Model Driven Architecture (MDA) initiative of the Object Management Group (OMG) that emphasizes the use of the Unified Modeling Language, model transformations, and code generation (see http://www.omg.org/mda); and (2) Model-Integrated Computing (MIC) for embedded systems that emphasizes rigorous analyses of models and code generation (see http://www.isis.vanderbilt.edu/research/research.html). Our AOM method directly enables these model-based approaches since it is based on standard modeling techniques.

Aspect-Oriented Development (AOD) supports the separation of concerns principle that has proven to be effective at tackling complexity [8]. AOD methods allow developers to represent pervasive design and implementation concerns as *aspects*. In an AOD approach, a design consists of (1) a primary design or implementation artifact (e.g., a UML model or code) in which the pervasive concerns are not included, (2) a set of aspects, each representing a pervasive design concern that impacts the elements of the primary design artifact, and (3) a weaving mechanism that composes aspects with the primary artifact to obtain a view of the design that details how the structures and behaviors modeled in the primary artifact are impacted by the aspects. Examples of AOD approaches are *aspect-oriented programming* (e.g., see [1, 10, 11, 13, 14, 17, 18]) in which the primary design artifacts are code, and aspects are concerns that cross-cut code modules, and *subject-oriented design* (e.g. see [3, 4, 9, 19]) in which aspects are design realizations of requirements, and a design is created by composing aspects. Our AOM method is related in that it addresses design realizations of important system concerns. The AOM method differs in that we concentrate on system concerns other than those involved in the primary system modularization. We also concentrate our efforts on using aspect models to support design analysis and trade-off analysis among competing system concerns. Part of this support includes developing more detailed mechanism models,

such as those described in this paper, to discover potential conflicts at lower levels of abstraction and to provide a basis for mechanism comparison. This is on-going work in our research.

Fiadeiro et al. [4] specify secondary system characteristics related to system coordination using an algebraic approach. Their approach is applicable to detailed design and code, and utilizes a notation that is not widely known by system developers. Gray et al. [9] use aspects to represent secondary system characteristics in domain-specific models. Their research is part of the MIC initiative that targets embedded software systems specifically. Model-Integrated Computing (MIC) extends the scope and usage of models such that they form the backbone of a development process for building embedded software systems. Requirements, architecture, and the environment of a system are captured in the form of formal high-level models that allow the representation of concerns. Suzuki et al. [19] extend the UML so that it can be used to model code level aspects. Their approach is restricted to secondary system characteristics that can be represented as aspects in an aspect-oriented program. Our approach differs since we do not require aspect-oriented programming techniques.

The subject-oriented design approach proposed by Clarke et al. is a UML-based approach that is closest to the AOM method [3]. In the subject-oriented modeling approach a design is created for each system requirement. The design for a system requirement is referred to as a *subject*. A comprehensive design is obtained by composing subjects. In the subject-oriented approach aspects are subjects expressed as UML model views, and composition involves merging the views provided by the subjects. Merging is restricted to adding and overriding named elements in a model. Merging of constraints is not supported, nor is there support for deleting elements from models (except the implicit deletion that occurs when an element is overridden). Conflict resolution mechanisms are limited to defining precedence and override relationships between conflicting elements. In prior work [6, 7] we have shown how secondary system characteristics can be modeled as design aspects, expressed as structural and behavioral patterns specifications, and woven into designs expressed in the UML. We have also demonstrated some conflicts that can occur during composition, and directives that can be used to resolve them.

As part of the Early Aspects initiative, Moreira, Araujo, and Rashid have targeted multi-dimensional separation beginning early in the software cycle [15, 16]. Their work supports modularization of broadly scoped properties at the requirements level to establish early trade-offs, provide decision support and promote traceability to artifacts at later development stages. Our AOM method compliments this work by supporting aspect modeling, composition, and analysis of successively more detailed levels of abstraction needed during system design. To our knowledge, our work is unique in this respect.

5. Conclusions and Future Work

Aspect-oriented modeling provides a straightforward way to approach the design of complex distributed systems that must include multiple competing critical systems concerns such as security and dependability. AOM allows architects to address these concerns separately, and then compose them with primary modularization models and analyze the resulting models. The AOM method allows architects to experiment with different mechanisms to effect realizations of these concerns in order to choose the mechanisms that best meet overall system goals. This is particularly an issue when there are many different mechanisms to realize a particular concern, for example authentication or access control concerns.

We have demonstrated that it is possible to create more specific mechanism models for such experimentation from abstract aspect models, such as a shared secret mechanism like SRP. These mechanism models can be composed with primary system models using the same AOM methods as are used to compose abstract models with primary models (see [6, 7]). Composition allows architects to analyze a complete system model and make tradeoff decisions between various mechanisms. This capability also increases the practicality of using aspect-oriented modeling techniques during the design of complex critical systems with multiple competing additional concerns, since industrial software development is often constrained by the used of specific security and dependability mechanisms provided by COTS middleware.

Continued work in this area includes developing a formal model of refinement in order to automate this critical design step to the extent that automation is possible and practical. We are also in the process of developing additional mechanism models for security and dependability concerns (e.g. access control) and demonstrating their evolution from more abstract aspect models. In addition, we are working on algorithms to demonstrate refinement of behavioral models. These algorithms will be included in the prototype tool we are developing (see [12]).

From a notational point of view, the UML has been adequate to model abstract aspects. However more detailed mechanism models present some modeling challenges. For example, the *calcExpS* and *calcExpC* methods shown in Figure 3 use a random number generator (*random(*), exponents (**), and a mod function (*%*). Conventions must be used to model these concepts so that they are not confused with other OCL concepts (e.g. an exponential function expressed using ^ being confused with the OCL method call notation expressed using ^). Another notational issue involves using UML 2.0 sequence diagrams rather than UML 1.4 collaboration diagrams to specify dynamic aspect behavior. We are now investigating this issue, particularly the impact of the new sequence diagrams on our composition algorithms.

6. References

[1] Bergmans, L. and M. Aksit, M., "Composing multiple concerns using composition filters", *Communications of the ACM*, vol 44, no 10, Oct 2001

[2] Booch, G., "Growing the UML", Software and System Modeling Journal, Vol 1, no 2, Feb 2003

[3] Clarke, S., Harrison, W., Ossher, H., and Tarr, P., "Separating concerns throughout the development lifecycle", *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, June, Lisbon, Portugal, 1999

[4] Fiadeiro, J. L. and Lopes, A., "Algebraic semantics of co-ordination or what is it in a signature?", *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology (AMAST'98)*, Amazonia, Brasil, Lecture Notes in Computer Science, vol 1548, pp 293-307, A. Haeberer , ed, Springer-Verlag, Jan 1999

[5] France, R. B., Kim, D. K., Song, E., and Ghosh, S., "Using Roles to Characterize Model Families", in Practical Foundations of Business and System Specifications, Haim Kilov, ed., Kluwer Academic Publishers, 2002

[6] Georg, Geri, France, Robert, and Ray, Indrakshi, "Designing High Integrity Systems using Aspects", *Proceedings of the Fifth IFIP TC-11 WG 11.5 Working Conference*
on Integrity and Internal Control in Information Systems (IICIS 2002), Bonn, Germany, Nov 2002

[7] Georg, G., Ray, I., and France, R., "Using Aspects to Design a Secure System", *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, ACM Press, Greenbelt, MD, Dec 2002

[8] Ghezzi, C., Jazayeri, M., and Mandrioli, D., Fundamentals of Software Engineering, Prentice Hall, 1991

[9] Gray, J., Bapty, T., Neema, S., and Tuck, J., "Handling crosscutting constraints in domain-specific modeling", *Communications of the ACM*, vol 44, no 10, pp 87-93, Oct 2002

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G., "An Overview of AspectJ", *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '01)*, pp 327-353, Budapest, Hungary, June, 2001

[11] Kieberherr, K., Orleans, D., and Ovlinger J., "Aspect-oriented programming with adaptive methods", Communications of the ACM, vol 44, num 10, pp 39-41, Oct 2001

[12] Mekerke, F., Georg, G., France, R., Alexander, R., "Tool Support for Aspect-Oriented Design", *Advances in Object-Oriented Information Systems: OOIS2002 Workshops*,

[13] Ossher, H. and Tarr, P., "Using multidimensional separation of concerns to (re)shape evolving software", *Communications of the ACM*, vol 44, num 10, p 43-50, Oct, 2001

[14] Pace, J. A. D. and Campo, M. R., "Analyzing the Role of Aspects in Software Design", *Communications of the ACM*, vol 44, no 10, pp 66-73, Oct 2001

[15] Rashid, A. and Chitchyan, R., "Persistence as an Aspect", *2nd International Conference on Aspect-Oriented Software Development*, ACM, pp 120-129, Boston, Mar 2003

[16] Rashid, A., Moreira, A., and Araujo, J., "Modularization and Composition of Aspectual Requirements", *2nd International Conference on Aspect-Oriented Software Development*, ACM, pp 11-20, Boston, Mar 2003

[17] Silva, A. R., "Separation and composition of overlapping and interacting concerns", *OOPSLA '99 First Workshop on Multi-Dimensional separation of Concerns in Object-Oriented Systems*, Denver, Colorado, Nov 1999

[18] Sullivan, G. T., "Aspect-oriented programming using reflection and metaobject protocols", *Communications of the ACM*, vol 44, num 10, pp 95-97, Oct 2001

[19] Suzuki, J. and Yamamoto, Y., “Extending UML with Aspects: Aspect Support in the Design Phase”, *Proceedings of the 3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999

[20] The Object Management Group, “The Unified Modeling Language", OMG, formal/2001-09-67, version 1.4, 2001

[21] Wu, T., “The secure remote password protocol”, *Proceedings of the 1998 Internet Society Symposium on Network and Distributed Systems Security*, pp 97-111, San Diego, CA, March 1998

Exploring UML Refinement through Unification

Eerke Boiten and Marius Bujorianu

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK.
(Email: E.A.Boiten@kent.ac.uk.)

Abstract. One way of making UML more suitable for the development of critical systems is to define a formal notion of development (or *refinement*) for it. We explore refinement indirectly: through *unification*.
Different UML diagrams may contain information on the same system element, which may or may not be contradictory. Such diagrams may be part of the same UML model, or taken from different models representing "viewpoints". A representation of the combined information of diagrams is a *unification*. Implicit in this is a notion of "information content" which needs to be formalised. A unification is not only a representation of combined information, it also witnesses consistency between the models.
The theory of consistency and unifications for viewpoint specification is well-developed for formal methods. In general, such unification methods are parameterised by a notion of refinement (i.e., how to compare information content), and a notion of correspondence (relating the information between specifications). In particular, in Z all of these can be expressed syntactically, and a variety of refinement relations have been developed inspired by different styles of viewpoint specification.
This paper considers a number of small UML models, their intuitive "unifying" diagrams, and how these would relate to the unifications of formalisations of the original diagrams. In this way, desirable properties for a formal development notion in UML emerge.

1 Motivation: Formalists' Pamphlet

For the use of UML in the development of critical systems, we believe that an increased degree of formality is necessary. Although the UML community as a whole may not ever agree on the interpretation of UML, within development teams such agreement is essential. Formal interpretations (plural intended!) of UML provide an economy of scale for such shared interpretations.

In the absence of a formal interpretation, there are a number of reasonable questions that one *might* ask about a UML specification, but which are not answerable. These questions are listed here, with theorems describing how they are interrelated. In essence, they are so closely related that an answer to one of these questions solves all or most of them.

Conformance If we have made the effort of writing a UML model, either before or after writing the program, can we actually check that the specification and the program match? In other words, given a UML model S, and a program P, does P conform to S? Let us write this as $Conf(P, S)$.

Consistency The various diagrams in UML describe systems from different angles, but they are not entirely orthogonal. So we may ask whether, even if each diagram by itself is consistent, their combination imposes constraints that cannot be satisfied. Let us denote the consistency of UML descriptions $S_1, \ldots, S_n$ as $Consist(S_1, \ldots, S_n)$.

Theorem 1 (Consistency for free (1)). *Given conformance, consistency can be defined by*

$$Consist(S_1, S_2) =_{def} \exists P \bullet Conf(P, S_1) \land Conf(P, S_2)$$

Refinement Imagine we have written a first UML specification, and obtained agreement from all parties involved that it correctly reflects all their requirements. Unfortunately, it does not quite describe the system's workings in detail – we would like to change the UML description to reflect such extra detail. Can we determine whether the resulting description bears any relation to the original requirements without renewed consultation? I.e., is our second specification S_2 a correct refinement of S_1, denoted $Ref(S_2, S_1)$?

Theorem 2 (Refinement for free (1)). *Given conformance, refinement can be defined by*

$$Ref(S_2, S_1) =_{def} \forall P \bullet Conf(P, S_2) \Rightarrow Conf(P, S_1)$$

Theorem 3 (Consistency for free (2)). *Given refinement, consistency can be defined by*

$$Consist(S_1, S_2) =_{def} \exists S_3 \bullet Ref(S_3, S_1) \land Ref(S_3, S_2)$$

Also, if we have a notion of "implementation specification" in UML, i.e., specifications which correspond one-to-one to actual implementations (call this $ImpSpec(S, P)$), then conformance also follows from refinement:

Theorem 4 (Conformance for free). *Given refinement and implementation specifications, conformance can be defined by*

$$Conf(P, S) =_{def} \exists S' \bullet Ref(S', S) \land ImpSpec(S', P)$$

Semantics Maybe this question is slightly less reasonable than the earlier ones. Nevertheless . . . we might ask what the acceptable *models* of a UML description S are. Let us denote the set of all of these as $Models(S)$.

Theorem 5 (Refinement for free (2)). *Given a model-based semantics, refinement can be defined by*

$$Ref(S_2, S_1) =_{def} Models(S_2) \subseteq Models(S_1)$$

We might avoid trivialisation by additionally insisting that $Models(S_2) \neq \varnothing$.

Theorem 6 (Consistency for free (3)). *Given a model-based semantics, consistency can be defined by*

$$Consist(S_1, S_2) =_{def} Models(S_1) \cap Models(S_2) \neq \varnothing$$

Obviously, semantics is "for free" when we have conformance (all conformant programs as models) or refinement (models are defined as the set of all possible refinements).

The theorems provide extensive possibilities for defining one of the notions in terms of the other; in particular, defining *refinement* is a strong basis for a full UML formalisation.

The theory represented here, briefly hinted at in [7], is inspired by our earlier work on viewpoint consistency [3, 5]. In that work, another useful notion is that of a *unification*.

Unification Given two specifications, one might create a single specification that contains (exactly) the information contained in both. These we call *(least) unifications*, let us denote the unification of S_1 and S_2 as

$$S_1 \oplus S_2$$

Least unifications can be characterised in terms of each of the preceding notions:

$$Conf(P, S_1 \oplus S_2) \Leftrightarrow Conf(P, S_1) \wedge Conf(P, S_2)$$
$$Consist(S_1, S_2) \Leftrightarrow Consist(S_1 \oplus S_2)$$
$$Ref(S_3, S_1 \oplus S_2) \Leftrightarrow Ref(S_3, S_1) \wedge Ref(S_3, S_2)$$
$$Models(S_1 \oplus S_2) = Models(S_1) \cap Models(S_2)$$

The second equation holds for any unification; the first and third become implications for unifications which are not "least". The last equality needs to be weakened to an inclusion if the image of *Models* is not closed under intersection.

Unifications are most useful if they can be constructed syntactically; it depends on the properties of the specification language whether this is possible. In the state based language Z this is indeed possible [4]; in process algebra, typically the process structuring is lost in unification [25].

In the next section we will argue why consistency and unification are, by themselves, relevant issues for UML, and discuss some of the approaches taken in the literature.

2 Consistency and Viewpoints for UML

The UML combines multiple notations into a single modelling language. This gives modellers the opportunity to apply appropriate notations for different aspects of the system; however, the various UML notations are not orthogonal

or disjoint in scope. As a consequence, UML has a *consistency* issue: different diagrams may put contradictory constraints on some element of the described system. This is in addition to structural consistency requirements (or better: well-formedness conditions) of UML models, which may be represented as OCL constraints in the UML meta-model [14].

For architectural descriptions in UML in particular, a *viewpoints* [16] method has been proposed [11]. In such a context, consistency is an even more prevalent issue [15]. However, the problem of determining consistency between two UML "viewpoint" models is not fundamentally different from that of determining consistency within a single UML model. In either case, the source of any inconsistency will be a collection of diagrams (or OCL constraints, etc.) which, between them, put unsatisfiable restrictions on some element of the system. In a "viewpoints" situation, the possibility increases that these are diagrams of the same type, e.g., state charts describing the same class.

Most approaches to consistency checking for UML rely on a formalisation of all UML diagrams in a single semantic framework, which may be CSP [6, 12], algebraic specification [2], CSP-OZ [23], etc. UML consistency is then reduced to consistency of the underlying framework. Such approaches to consistency checking are non-constructive: they provide no feedback in terms of *UML* indicating whether consistency held or not. The Promela based approach of Lilius [20], where model checking is used to generate a counter-example of consistency, is nominally constructive, as it generates UML sequence diagrams.

In a fully constructive approach to consistency, for example the one for Z [4], the underlying semantic model remains hidden. Inconsistencies are thus reported in terms of the specifications themselves. Moreover, also the result of a successful consistency check is represented as a specification, which is called the *unification*. Unifications can also be used as the basis for incremental consistency checking of $n > 2$ specifications. See [3, 5] for a full discussion of consistency checking based on unifications.

In this paper, we explore some possibilities for constructive consistency checking in UML. The completeness of a method for this would, obviously, require a complete formal semantics for UML, which is not our concern at this point. Instead, we consider intuitive (maybe even naive!) formalisations of UML fragments whose meaning seems non-controversial, and explore whether these formalisations can substantiate, again, intuitive "unifications" in UML. These unifications are in the same spirit as rules for "joining" in Catalysis [10]. Unsurprisingly, we will use Z for the formalisation of UML, as the constructive method for Z may lead to unifications that may have direct counterparts in UML. Indeed, the aim of our work is not to show adequacy of Z as an underlying semantic framework for UML (see [18, 13] and others), but the degree to which UML itself can have constructive unification.

In the next section we present two small UML models, and consider an intuitive unification of them. Then, in Section 4 we translate the example into Z, apply the Z unification techniques to the translation, and evaluate the result. Section 5 presents some conclusions.

3 Example: Flight Booking System

In this example, we consider two views of a flight booking system. In both cases, the UML model consists of a class diagram and a state diagram.

In the first viewpoint, reservations can be made, but not cancelled. Moreover, the flight itself is not subject to cancellation. The class structure of this model (Figure 1) comprises three classes:

- BookingS describes a basic booking system: openFlight for opening a new flight, close for closing a flight and reserve for reserving a seat on a flight;
- Flight contains a basic attribute of a flight: the number of free seats seats noFreeSeats;
- Seat represents for a seat, whether it is reserved or free.

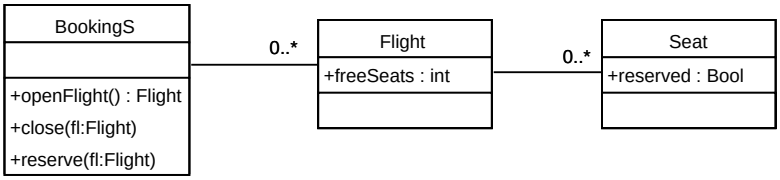


Fig. 1. First viewpoint: class diagram

Associations are defined between the classes: multiple instances of class Seat are associated with an instance of Flight, and multiple flights are associated with the BookingS class.

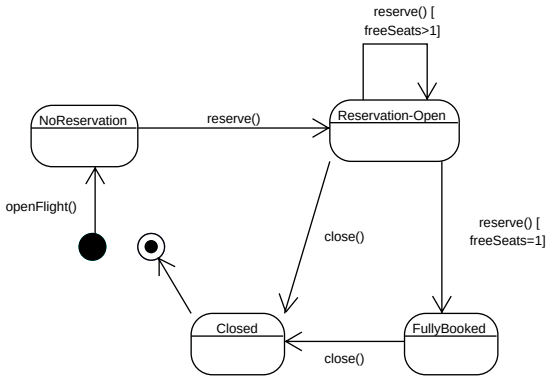


Fig. 2. First viewpoint: state chart

The state diagram (Figure 2) contains four states, apart from the initial and final ones: NoReservation (corresponding to the system state immediately after the flight was opened), ReservationOpen (the state where the flight has seats booked and the booking process is still ongoing), FullyBooked (the state where all seats are booked) and Closed (the seats booking process has stopped – for example two hours before take off). Observe that a flight can be closed even if not all seats are reserved (the transition from ReservedOpen to Closed).

In the second viewpoint, flights and reservations may be cancelled.

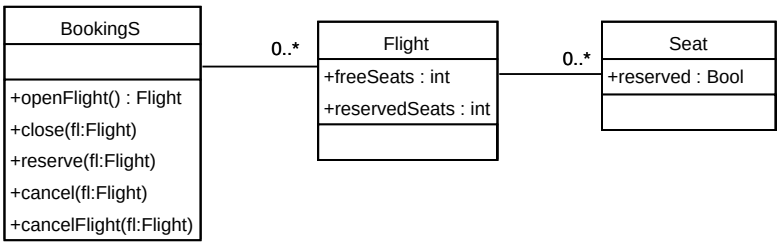


Fig. 3. Second viewpoint: class diagram

The class diagram (Figure 3) has a very similar structure to that of the first viewpoint. Two more methods have been added to the Book_Sys class (corresponding to BookingS): cancel for cancelling a seat reservation and cancelFlight for cancelling a flight. Also, rather than recording just the number of free seats, we also record the number of reserved seats. The state diagram (Figure 4), com-

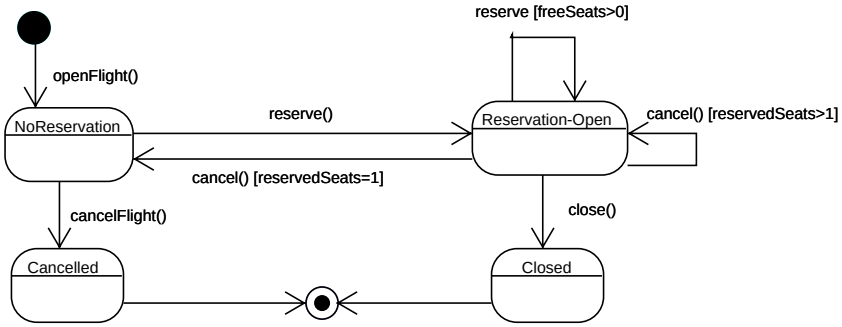


Fig. 4. Second viewpoint: state chart

pared to the previous one, has an extra state for cancelled flight, but does not

identify fully booked flights. It also adds transitions corresponding to cancellations of seats or entire flights.

An intuitive unification of the two viewpoints consists of the second class diagram (Figure 3, which contains more information than the first), and a state diagram (Figure 5) which represents all the states and transitions available in the previous ones.

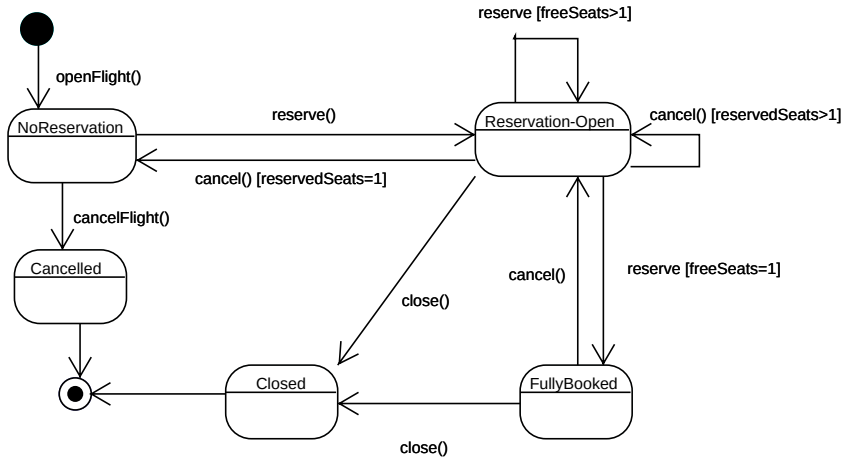


Fig. 5. Unified state chart

4 Unification from Z

Here, we first formalise the UML diagrams in Z. We will concentrate on the behaviour of an individual flight. The translation is similar in spirit to that by France et al. [18], formalising the behaviour of an individual object in Z. In order to also model aggregations, associations, object creation, etc., one would either need to create a layer of object instances on top of this as in [18], or move to an object-oriented variant of Z such as Object-Z [24]. Other formalisations in Z of UML, such as by Evans et al. [13], model UML concepts (such as objects, classes, etc.) as Z schema types – that kind of formalisation is less useful here, as we are not interested in refining UML *concepts* but UML models.

After formalising the models, we derive a unification of these formalisations, illuminating desirable properties for UML refinement relations as they appear.

4.1 First viewpoint

The class diagram of the first viewpoint (Figure 1) tells us it has an attribute $freeSeats : int$; according to the state diagram (Figure 2), it can also be in a number of states. We abbreviate their names in the Z specification.

$$States_1 ::= NoRes \mid ResOpen \mid Full \mid Closed$$

As there are no transitions leading into the initial state, nor labelled transitions into the final state, we can omit those for now. The transition leading out of the initial state corresponds to the initialisation.

$Flight_1$

$s_1 : States_1$

$freeSeats : \mathbb{Z}$

$Init_1$

$\Delta Flight_1$

$s_1' = NoRes$

When doing such formalisations "by hand", it is very tempting to introduce extra information which is not present in the UML though evident from our "understanding" of the real-life situation represented. However, we do not include any constraint on $freeSeats'$ in *Init* as none is implied in the diagrams.

Operations correspond to the disjunctions of *all* transitions in the state diagram which have the same label.

$Reserve_1$

$\Delta Flight_1$

$(s_1 = NoRes \land s_1' = ResOpen) \lor$
$(s_1 = ResOpen \land freeSeats > 1 \land s_1' = ResOpen) \lor$
$(s_1 = ResOpen \land freeSeats = 1 \land s_1' = Full)$

Again, it is tempting to include a predicate like $freeSeats' = freeSeats - 1$, or to assume $freeSeats' > 0$ for the initial state, but it is not actually implied.

$Close_1$

$\Delta Flight_1$

$s_1 = ResOpen \lor s_1 = Full$
$s_1' = Closed$

In our class diagrams, all attributes were declared public. This was done, however, because the decision on visibility should be postponed until a later stage. In order to fully represent the attributes being public, we would need to include Z operations representing accessors and mutators on all attributes as well.

4.2 Second viewpoint

There are four states in the second state chart (Figure 4), with the initial and final states omitted for the same reason as above.

$$States_2 ::= NoRsv \mid RsvOpen \mid Clsd \mid Cancld$$

Note that we use different abbreviations for the state names, in order to distinguish between the two viewpoints.

$Flight_2$

$s_2 : States_2$

$free, reserved : \mathbb{Z}$

$Init_2$

$\Delta Flight_2$

$s_2' = NoRsv$

The operations are once again disjunctions of transitions with the same label.

$Reserve_2$

$\Delta Flight_2$

$(s_2 = NoRsv \land s_2' = RsvOpen) \lor$
$(s_2 = RsvOpen \land free > 0 \land s_2' = RsvOpen)$

$Cancel$

$\Delta Flight_2$

$s_2 = RsvOpen$
$reserved > 1 \Rightarrow s_2' = RsvOpen$
$reserved = 1 \Rightarrow s_2' = NoRsv$

$CancelFlight$

$\Delta Flight_2$

$s_2 = NoRsv \land s_2' = Cancld$

$Close_2$

$\Delta Flight_2$

$s_2 = RsvOpen \land s_2' = Clsd$

4.3 Unification

For the states-and-operations style in Z, the process of unification is described in detail in [4]. Lacking space to repeat all details, we illustrate this process by example.

This sample unification approximates a development relation for UML in the following way. Unification in Z is defined in terms of *refinement*: it is the "least" (-prescriptive) common refinement of two specifications. A hierarchy of refinement relations in Z exist, see [9], leading to slight differences in the rules for unification. Viewing this in reverse: we should select the parameters for Z unification of the UML formalisations in such a way that we end up with a Z formalisation of the combined state diagram above. In this way, we establish desirable properties for UML refinement relations, which we will highlight as they appear. This we consider to be the main contribution of the paper.

State unification As it will turn out later, the crucial step in the unification is the identification of its state space. If the state spaces of the two views are identical, this step can be avoided. However, the schemas $Flight_1$ and $Flight_2$

are different. In terms of the required refinement relation, this implies that we need *data refinement*, rather than the simpler *operation* or *algorithmic* refinement. In other words, in refinement we should not just remove underspecification (non-determinism) from the operations, but also allow change of the internal detail (names and types of private attributes) of the classes.

> **Requirement 1:** UML refinement needs to include *data* refinement, to allow for hidden state with abstraction from internal details.

In order to continue unification, we need to establish the relationship between the variables in the two state schemas. This will be documented in a combined schema, which we call a *correspondence*. We define this in a number of steps, starting with the non-controversial relations between the states.

$$
\begin{array}{|l}
\hline
Corr_0 \\
Flight_1 \\
Flight_2 \\
\hline
free = freeSeats \\
s_1 = NoRes \Leftrightarrow s_2 = NoRsv \\
s_1 = Closed \Leftrightarrow s_2 = Clsd \\
\hline
\end{array}
$$

Next, we are faced with the problem that the second viewpoint has a variable *reserved* that does not occur in the first viewpoint. We might reasonably assume that *free*+*reserved* remains constant in the second view; let us name this constant *capacity*. As reservation is possible from the initial state, its value should be at least 1.

$$
\begin{array}{|l}
capacity : \mathbb{N}_1
\end{array}
$$

The next issue is with the state *RsvOpen*, from which $Reserve_2$ may or may not succeed, depending on the value of *free*; this does not quite match the state *ResOpen* from which $Reserve_1$ is always defined. This is addressed by stating that the state *RsvOpen* is represented by either *ResOpen* (when there are free seats) or *Full* (when there are none). This completes the correspondence relation[1]:

$$
\begin{array}{|l}
\hline
Corr \\
Corr_0 \\
\hline
free + reserved = capacity \\
s_1 = Full \Leftrightarrow s_2 = RsvOpen \land free \leq 0 \\
s_1 = ResOpen \Leftrightarrow s_2 = RsvOpen \land free > 0 \\
\hline
\end{array}
$$

Note that we could have chosen many other correspondences, but the current one represents our intuition about the specifications, including the supposition

[1] Inclusion of a Z schema's name in a new schema implies inclusion of all its declarations *and* predicates.

that they are actually consistent. Unlike in the mechanical process of formalising the UML diagrams, it *is* necessary to insert knowledge about the intention of the specifications at this stage. In general, (automatic) unification or consistency checking requires explicit linking of concepts between specifications more sophisticated than identification by name: identifying $RsvOpen$ and $ResOpen$ would have led to the detection of an inconsistency.

Totalisation A correspondence relation like $Corr$ might serve as the state space for the unification already. However, in this case it will not be sufficient: there is no mention of the state $Cancld$. $Corr$ represents a kind of dependent product of the two state spaces, in relational data base terms: a "theta-join". What is required for the combined state space (in order to guarantee refinement) is an "outer join" which contains representations of every member of both viewpoint state spaces; one might also view this as a *totalisation* of the correspondence relation. In this case, the outer join would link $Cancld$ to a "null" value for $Flight_1$; because the variables *free* and *reserved* play no rôle in $Cancld$, we present a simpler equivalent schema where only $States_2$ is extended with a null value.

$$States_1 ::= NoRes \mid ResOpen \mid Full \mid Closed \mid Null$$

The full state schema for the unification is then (using the latest definition of $States_1$):

TCorr

$$Corr$$

$$s_1 = Null \Leftrightarrow s_2 = Cancld$$

Adaptation Once we have defined a common state schema, all viewpoint operations can be "adapted" to operate on this common state. Technically, this is a data refinement step, determining the least refinement [8] for the retrieve relation which injects the viewpoint state space into the common one. Syntactically, it involves little more than a change of the state name. The first viewpoint's operations become

$AdReserve_1$

$$\Delta TCorr$$

$$(s_1 = NoRes \land s_1' = ResOpen) \lor$$
$$(s_1 = ResOpen \land freeSeats > 1 \land s_1' = ResOpen) \lor$$
$$(s_1 = ResOpen \land freeSeats = 1 \land s_1' = Full)$$

$AdClose_1$

$$\Delta TCorr$$

$$s_1 = ResOpen \lor s_1 = Full$$
$$s_1' = Closed$$

where additional requirements are hidden in $\Delta TCorr$ – in particular, variables from $Flight_2$ are constrained to preserve the correspondence. The operations of the second viewpoint are adapted analogously.

Operation unification Any data refinement can be factored into a "least" data refinement and an operation refinement – operation adaptation handled the former, so we can restrict ourselves to operation unification now. This is a well-known technique, first described for Z by Ainsworth et al [1]. In Catalysis, it is used for "joining" pre/post-specifications, but [10] does not draw the link with refinement.

Two operations Op_1 and Op_2 defined in terms of ΔS are consistent, and can thus be successfully unified [1, 4] whenever

$$\text{pre}\, Op_1 \wedge \text{pre}\, Op_2 \Rightarrow \text{pre}(Op_1 \wedge Op_2)$$

i.e., whenever they are both applicable, a result consistent with both exists. In that case, their unification is

$Op_1 \oplus Op_2$

ΔS

$\text{pre}\, Op_1 \vee \text{pre}\, Op_2$
$\text{pre}\, Op_1 \Rightarrow Op_1$
$\text{pre}\, Op_2 \Rightarrow Op_2$

This definition covers the situation where we assume that a precondition represents a "contract": within the given area, the operation should deliver the right result, and outside this area, we do not care. This is the common assumption for Z; however, for behavioural interpretations we may also view the precondition as a "guard": outside the precondition the operation should be disabled. In this interpretation, we additionally require that $\text{pre}\, Op_1 = \text{pre}\, Op_2$ for consistency, which means that the unification reduces to $Op_1 \wedge Op_2$. For a detailed discussion of "guards" vs. "contracts", see [9].

The preconditions of the two adapted *Reserve* operations coincide: either there are no reservations yet, or reservation is open and there are still free seats. The latter condition is encoded in the state in the first view, and in the variable *free* in the second view. Their outcomes are also consistent, and lead to

$AdReserve_1 \oplus AdReserve_2$

$\Delta TCorr$

$(s_1 = NoRes \wedge s_1' = ResOpen) \vee$
$(s_1 = ResOpen \wedge freeSeats > 1 \wedge s_1' = ResOpen) \vee$
$(s_1 = ResOpen \wedge freeSeats = 1 \wedge s_1' = Full)$

where all the effects of $AdReserve_2$ are already implied by the combination of $AdReserve_1$ and $TCorr'$.

$AdClose_1$ and $AdClose_2$ are also equivalent to each other, and thus also to their unification.

The first viewpoint has no further operations; the second has also *Cancel* and *CancelFlight*. Formally, we can treat this by the operations being defined with a precondition of *false* in the first viewpoint; the unifications would then be *Cancel* and *CancelFlight*. The fact that we would unify two operations with different preconditions shows that we do need to take the "contract" interpretation of preconditions.

> **Requirement 2**: UML refinement should allow widening of preconditions.

As we could not actually distinguish in a state diagram between an operation that is never applicable (appearing as 0 arrows labelled with its name), and one that does not exist, we also have:

> **Requirement 3**: UML refinement should allow introduction of extra operations.

For a solid grounding of this requirement in an OO context, see the "constraint rule" of Liskov and Wing [21].

Finally, observe that the unified Z specification is indeed equivalent to the intuitive formalisation of the state diagram we presented earlier (Figure 5). There are five states, corresponding to the four states of the first view plus the *Cancld* state of the second view. The *RsvOpen* state of the second view is split into two states, depending on whether $free > 0$. The only information that the formal unification adds, compared to Figure 5, is that $freeSeats = 0$ in state FullyBooked. It is not surprising that this information is absent from the intuitive UML unification, as it was propagated from one state into the other via the correspondence relation (which we established during unification only). For more detailed models with OCL constraints specifying the effect of the operations on the given states, there would not have been similar "surprises". All in all, in this simple case study, the UML unification is consistent with the Z unification once the correspondence relation has been accounted for.

5 Concluding Comments

This small case study has substantiated a number of intuitions about the nature of possible refinement relations in UML. In their nature, they should be similar to refinement for Z (and B, and VDM, etcetera). However, in accordance with the Catalysis approach, we believe that a more complicated case study would have introduced additional requirements, leading to a more liberal notion of refinement.

First, the operations we formalised had no inputs or outputs. UML practice appears to allow for inputs and outputs to be elided when they are not relevant – as a consequence, we would need to allow *interface refinement* [22], which is a sound generalisation of normal refinement, see [9, Chapter 10].

Second, decomposition of (inter)actions plays an important rôle in UML. This could be addressed by having unification based on a notion of *action refinement*, which is an independent and compatible generalisation as well [9].

Clearly, issues of consistency, unification, and refinement could have been addressed on the basis of a different formalisation as well. In particular, a CSP formalisation [6, 12] seems an obvious candidate. However, defining the correct refinement relation on the CSP side is non-obvious, for example because most CSP refinement relations do not allow widening of guards, or because unifications in process algebra are likely to be expressible semantically but not syntactically [25].

References

1. M. Ainsworth, A. H. Cruickshank, P. J. L. Wallis, and L. J. Groves. Viewpoint specification and Z. *Information and Software Technology*, 36(1):43–51, February 1994.
2. E. Astesiano and G. Reggio. An attempt at analysing the consistency problems in the UML from a classical algebraic viewpoint. In *Recent Trends in Algebraic Development Techniques, Selected Papers of the 15th International Workshop WADT'02*, Lecture Notes in Computer Science. Springer, 2003. To appear.
3. E.A. Boiten and J. Derrick. Integration of specifications through development relations. In H. Ehrig, B.J. Krämer, and A. Ertas, editors, *IDPT 2002*. SDPS, 2002. Extended version to appear in Journal of Design and Process Science, September 2003.
4. E.A. Boiten, J. Derrick, H. Bowman, and M. Steen. Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29–75, September 1999.
5. H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency. *Formal Methods in Systems Design*, 21:111–166, 2002.
6. J. Davies and C. Crichton. Concurrency and refinement in the Unified Modelling Language. *Formal Aspects of Computing*, 2003. Special issue on REFINE'02, to appear.
7. J. Derrick, D. Akehurst, and E.A. Boiten. A framework for UML consistency. In Kuzniarz et al. [19], pages 30–45.
8. J. Derrick and E.A. Boiten. Calculating upward and downward simulations of state-based specifications. *Information and Software Technology*, 41:917–923, July 1999.
9. J. Derrick and E.A. Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. FACIT. Springer Verlag, May 2001.
10. D.F. D'Souza and A.C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
11. A. Egyed and N. Medvidovici. Extending architectural representation in UML with view integration. In France and Rumpe [17], pages 2–16.
12. G. Engels, J.M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioural models. In V. Gruhn, editor, *Proceedings of the 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 186–195. ACM Press, 2001.

13. A. Evans, R.B. France, K. Lano, and B. Rumpe. The UML as a formal modelling notation. In Jean Bézivin and Pierre-Alain Muller, editors, *UML 1998*, volume 1618 of *Lecture Notes in Computer Science*, pages 336–348. Springer, 1999.
14. A.S. Evans and S. Kent. Meta-modelling semantics of UML: the pUML approach. In France and Rumpe [17], pages 140–155.
15. A.C.W. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multiperspective specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, August 1994.
16. A.C.W. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. *International Journal on Software Engineering and Knowledge Engineering, Special issue on Trends and Research Directions in Software Engineering Environments*, 2(1):31–58, March 1992.
17. R. France and B. Rumpe, editors. *UML'99*, volume 1723 of *Lecture Notes in Computer Science*. Springer, 1999.
18. R.B. France, J.-M. Bruel, and M.M. Larrondo-Petrie. An integrated object-oriented and formal modeling environment. *Journal of Object Oriented Programming*, 10(7):25–34, 1997.
19. L. Kuzniarz, G. Reggio, J.L. Sourrouille, and Z. Huzar, editors. *UML'02 Workshop on Consistency Problems in UML-based Software Development*, Research Report 2002:06. Blekinge Institute of Technology, 2002.
20. J. Lilius and I.P. Paltor. vUML: a tool for verifying UML models. In R.J. Hall and E. Tyugu, editors, *Proceedings of ASE'99*, pages 255–258. IEEE Computer Society, 1999.
21. B. Liskov and J. M. Wing. A behavioural notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.
22. A. Mikhajlova and E. Sekerinski. Class refinement and interface refinement in object-oriented programs. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Application and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 82–101. Springer-Verlag, September 1997.
23. H. Rasch and H. Wehrheim. Consistency between UML classes and associated state machines. In Kuzniarz et al. [19], pages 46–60.
24. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
25. M. W. A. Steen, H. Bowman, and J. Derrick. Composition of LOTOS specifications. In P. Dembinski and M. Sredniawa, editors, *Protocol Specification, Testing and Verification, XV*, pages 73–88, Warsaw, Poland, 1995. Chapman & Hall.

Automated Formal Verification of Model Tranformations*

Dániel Varró and András Pataricza

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1521 Budapest, Magyar tudósok körútja 2.
{varro,pataric}@mit.bme.hu

Abstract. When designing safety critical applications in UML, the system models are frequently projected into various mathematical domains (such as Petri nets, transition systems, process algebras, etc.) to carry out a formal analysis of the system under design by *automatic model transformations*. Automation surely increases the quality of such transformations as errors manually implanted into transformation programs during implementation are eliminated; however, conceptual flaws in transformation design still remain undetected. In this paper, we present a model-level, modeling language independent and highly automated technique to formally verify by model checking that a model transformation from an arbitrary well-formed model instance of the source modeling language into its target equivalent preserves (language specific) dynamic consistency properties. We demonstrate the feasibility of our approach on a complex mathematical model transformation from UML statecharts to Petri nets.
Keywords: model transformation, graph transformation, model checking, formal verification, UML statecharts, Petri nets.

1 Introduction

For most computer controlled systems, especially dependable, real-time systems for critical applications, an effective design process requires an early validation of the concepts and architectural choices, without wasting time and resources to assess whether the system fulfills its requirements or needs some re-design.

The Unified Modeling Language (UML) together with domain specific profiles (e.g., the UML Profile for Schedulability, Performance and Time [16]) provides a standard and easy-to-understand visual way to capture both the requirements and the system model.

However, a standard modeling language does not alone guarantee the correctness of the design. In order to increase the level of confidence that can be put on a system mathematical tools (based on formal methods like Petri nets, dataflow networks, transition systems, process algebras, etc.) are used to assess the most important system parameters

* This work was partially carried out during the visit of the first author to the University of Paderborn (Germany), and it was supported by the SegraVis Research Network, the Hungarian Information and Communication Technologies and Applications Grant (IKTA 065/2000), and the Hungarian National Scientific Foundation Grant (OTKA 038027)

(such as functional correctness, timeliness, performability or dependability). Unfortunately, sophisticated verification tools (such as the SPIN model [11] checker) require a thorough knowledge of the underlying mathematics, and therefore special skills are needed for dependable IT system designers.

In order to bridge the huge abstraction gap, many approaches (e.g., [4, 7, 13, 25]) to automatically transform high-level UML based system models into low-level mathematical models, and then back-annotate the results of the formal analysis into the original UML model of the system in order to hide the underlying mathematics.

In the current paper, we investigate the model transformation problem from a general perspective, i.e., to specify how to transform a well-formed instance of a source modeling language (which is typically UML) into its equivalent in the target modeling language (which can be UML, a target programming language, or a mathematical modeling language).

Related work in model transformations Model transformation methodologies have been under extensive research recently. Existing model transformation approaches can be grouped into two main categories:

- *Relational approaches*: these approaches typically *declare a relationship* between objects (and links) of the source and target language. Such a specification is typically based upon a metamodel with OCL constraints [1, 15].
- *Operational approaches*: these techniques *describe the process* of a model transformation from the source to the target language. Such a specification mainly combines metamodeling with (a) graph transformation [5–8, 25], (b) triple graph grammars [20] or (c) term rewriting rules [26].

Many of the previous approaches already tackle the problem of automating model transformations in order to provide a higher quality of transformation programs compared with manually written ad hoc transformation scripts.

Problem statement However, automation alone cannot protect against conceptual flaws implanted into the specification of a complicated model transformation. Consequently, a mathematical analysis carried out on the UML design after an automatic model transformation might yield false results, and these errors will directly appear in the target application code.

As a summary, it is crucial to realize that *model transformations themselves can also be erroneous* and thus may become a quality bottleneck of a transformation based verification and validation framework (such as [4]). Therefore, prior to analyzing the UML model of a target application, we have to prove that the model transformation itself is free of conceptual errors.

Correctness criteria of model transformations Unfortunately, it is hard to establish a single notion of correctness for model transformations. The most elementary requirements of a model transformation are syntactic.

- The minimal requirement is to assure **syntactic correctness**, i.e., to guarantee that the generated model is a syntactically well–formed instance of the target language.

– An additional requirement (called **syntactic completeness**) is to completely cover the source language by transformation rules, i.e., to prove that there exists a corresponding element in the target model for each construct in the source language.

However, in order to assure a higher quality of model transformations, at least the following *semantic requirements* should also be addressed.

– **Termination:** The first thing we must also guarantee is that a model transformation will terminate. This is a very general, and modeling language independent semantic criterion for model transformations.
– **Uniqueness (Confluence, functionality):** As non-determinism is frequently used in the specification of model transformations (as in the case of graph transformation based approaches) we must also guarantee that the transformation yields a unique result. Again, this is a language independent criterion.
– **Semantic correctness (Dynamic consistency):** In theory, a straightforward correctness criterion would require to prove the semantic equivalence of source and target models. However, as model transformations may also define a *projection* from the source language to the target language (with deliberate loss of information), semantic equivalence between models cannot always be proved. Instead we define *correctness properties* (which are typically transformation specific) *that should be preserved by the transformation*.

Unfortunately, related work addressing these correctness criteria of model transformations is very limited. Syntactic correctness and completeness was attacked in [25] by planner algorithms, and in [9] by graph transformation. Recently in [14], sufficient conditions were set up that guarantee the termination and uniqueness of transformations based upon the static analysis technique of critical pair analysis [10]. However, no approaches exist to reason about the semantic correctness of arbitrary model transformations, when transformation specific properties are aimed to be verified.

Our contribution In this paper, we present a model-level, modeling language independent and highly automated framework (in Sec. 2) to formally verify by model checking that a model transformation (specified by metamodeling and graph transformation techniques) from an arbitrary well-formed model instance of the source modeling language into its target equivalent preserves (language specific) dynamic consistency properties. We demonstrate the feasibility of our approach (in Sec. 3) on verifying a semantic property of a complex model transformation from UML statecharts to Petri nets.

The main benefit of our approach (in contrast to related solutions such as [8]) is that it can be adapted to arbitrary modeling languages taken from both software engineering and mathematical domains on a very high level of abstraction. More specifically, the transformation designers use the same visual notation (based on metamodeling and graph transformation) to capture the semantics of modeling languages and model transformations between them. Then our tools automatically (i) carry out the transformation from the source UML model into the target mathematical domain, and generate (ii) a model checking description to verify the correctness of the model transformation between the source and target model.

2 Automated Formal Verification of Model Transformations

We present an automated technique to formally verify (based on the model checking approach of [22]) the correctness of the model transformation of a specific source model into its target equivalent with respect to semantic properties.

2.1 Conceptual overview

A conceptual overview of our approach is given in Fig. 1 for a model transformation from an fictitious modeling language A (which will be UML statecharts for our demonstrating example later on) to B (Petri nets, in our case).

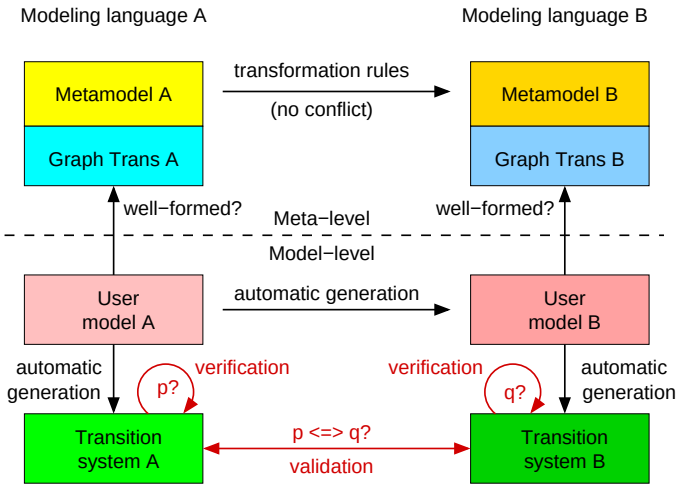


Fig. 1. Model level formal verification of transformations

1. **Specification of modeling languages**. As a prerequisite for the framework, each modeling language (both A and B) should be defined precisely using metamodeling and graph transformation techniques. We demonstrated in, for instance, [21,24] that many (we believe that all) languages in a realization of the MDA may have a semantics defined in this visual way, which is closely related to the UML philosophy.
2. **Specification of model transformations**. The A2B model transformation should be also specified by a set of (non-conflicting) graph transformation rules. The practical feasibility of such a solution has been demonstrated in many papers, see, e.g., [23] for an overview.
3. **Automated model generation**. For any specific (but arbitrary) well-formed model instance of the source language A, we derive the corresponding target model by

automatically generated transformation programs (e.g., generated by VIATRA [5] as tool support). The correctness of this automated generation step is proved in [23].

4. **Generating transition systems**. As the underlying semantic domain, a behaviorally equivalent transition system is generated automatically for both the source and the target model on the basis of the provenly correct encoding presented in [22] (and with a tool support reported in [19]).
5. **Select a semantic correctness property**. We select one semantic property p (at a time) in the source language A which is structurally expressible as a graphical pattern composed of the elements of the source metamodel (and potentially, some temporal logic operators).
   Note that the formalization of these criteria for a specific model transformation is not at all straightforward. In many cases, we can reduce the question to a reachability problem or a safety property, but even in this case finding the appropriate temporal logic formulae is non-trivial. More details on using graphical patterns to capture static well-formedness properties can be found, e.g., in [9].
6. **Model check the source model**. Transition system A is model-checked automatically (by existing model checker tools like SPIN [11] or SAL [3]) to prove property p. This model checking process should succeed, otherwise (i) there are inconsistencies in the source model itself (a *verification* problem occurred), (ii) our informal requirements are not captured properly by property p (a *validation* problem occurred), or (iii) the formal semantics of the source language is inappropriate as a counter example is found which should hold according to our informal expectations (another *validation* problem).
7. **Transform and validate the property**. We transform the property p into a property q in the target language (manually, or using the same transformation program). As a potentially erroneous model transformation might transform incorrectly the property p into property q, domain experts should validate that property q is really the target equivalent of property p or a strengthened variant. Unfortunately, this validation step typically requires human expertise and might not be fully automated.
8. **Model check the target model**. Finally, transition system B is model-checked against property q.
   - If the verification succeeds, then we conclude that the model transformation is correct with respect to the pair (p,q) of properties for the specific pairs of source and target models having semantics defined by a set of graph transformation rules.
   - Otherwise, property p is not preserved by the model transformation and debugging can be initiated based upon the error trace(s) retrieved by the model checker. As before, this debugging phase may fix problems in the model transformation or in the specification of the target language.

Note that at Step 2, we only require to use graph transformation rules to specify model transformations in order to use the automatic program generation facilities of VIATRA. Our verification technique is, in fact, independent of the model transformation approach (only requires to use metamodeling and graph transformation for specifying modeling languages), therefore it is simultaneously applicable to relational model transformation approaches as well.

Naturally, the correctness of a model transformation can only be deduced if the transformation preserves *every* semantic correctness property used in the analysis. Obviously, it requires several runs of the model checker, which can be time-consuming. Therefore, in [22], we assessed the expected run-time performance of our model checking based approach on a verification benchmark. In [2], the same technique was applied on architectural styles to check reachability properties. Both case studies demonstrated that our technique is applicable to non-trivial examples (of medium-size).

Furthermore, it is worth noting that the time related to the model transformation step, or to the automated generation of transition systems is still only a few percentage of the entire verification process in case of non-trivial models.

Prior to presenting the verification case study of a model transformation, we briefly discuss the pros and contras of meta-level and model-level verification of model transformations.

2.2 Meta-level vs. model level verification of model transformations

In theory, it would be advisable to *prove that a model transformation preserves certain predefined semantic properties for any well-formed model instance*, but this typically requires the use of sophisticated theorem proving techniques and tools with a huge verification cost. The reason for that lies in the fact that proving properties even in a highly automated theorem prover require a high-level of user guidance since the invariants derived directly from metamodels should be typically manually strengthened in order to construct the proof. In this sense, the effort (cost and time) related to the verification of a transformation would exceed the efforts of design and implementation which is acceptable only for very specific and critical applications.

However, the overall aim of model transformations is to provide a precise and automated framework for transforming the models of concrete applications (i.e., UML models). Therefore, in practice, *it is sufficient to prove the correctness of the model transformation from the source UML model of the system under design* against a set of properties defined by transformation engineers (while it is typically out of scope to demonstrate that the model transformation is correct for any source model). Thanks to existing model checker tools and the transformation presented in [22], such a model-level verification process can be highly automated. In fact, the selection of a pair (p,q) of corresponding semantic properties is the only part in our framework that requires user interaction and expertise.

Even if the verification of a specific model transformation is practically infeasible due to state space explosion caused by the complexity of the target application, model checkers can act as highly automated debugging aids for model transformations supposing that relatively simply source benchmark models are available as test sets.

3 Case Study: From UML Statecharts to Petri Nets

We present an extract of a complex model transformation case study from UML statecharts to Petri nets (denoted as SC2PN) in order to demonstrate the feasibility of our verification technique for model transformations.

The entire SC2PN transformation was originally designed and implemented as part of a Hungarian research project (IKTA 065/2000 – A framework for the modeling and analysis of dependable and safety critical systems) carried out in cooperation with industrial partners. Here UML statecharts are projected into Petri nets by this transformation in order to carry out (various kinds of) formal analysis such as functional correctness based on semi-decision methods of Petri nets [17].

The primary aim of the project was to formally verify UML models, but we also carried out the verification of the model transformation itself. Due to severe page limitations, we can only provide an overview of the verification case study, the reader is referred to [23] for a more detailed discussion.

3.1 Defining modeling languages by model transformation systems

Prior to reasoning about this model transformation, both the source and target modeling languages (UML statecharts and Petri nets) have to be defined precisely. For that purpose, in [24] we proposed to use a combination of metamodeling and graph transformation techniques: the *static structure* of a language is described by a corresponding *metamodel* clearly separating static and dynamic concepts of the language, while the *dynamic operational semantics* is specified by *graph transformation*.

Graph transformation (see [18] for theoretical foundations) provides a rule-based manipulation of graphs, which is conceptually similar to the well-known Chomsky grammar rules but using graph patterns instead of textual ones. Formally, a **graph transformation rule** (see e.g. addTokenR in Fig. 3) is a triple $Rule = (Lhs, Neg, Rhs)$, where Lhs is the left-hand side graph, Rhs is the right-hand side graph, while Neg is (an optional) negative application condition (grey areas in figures). Informally, Lhs and Neg of a rule define the *precondition* while Rhs defines the *post-condition* for a rule application.

The **application** of a rule to a **model (graph)** M (e.g., a UML model of the user) alters the model by replacing the pattern defined by Lhs with the pattern of the Rhs. This is performed by (i) *finding a match* of the Lhs pattern in model M; (ii) *checking the negative application conditions* Neg which prohibits the presence of certain model elements; (iii) *removing* a part of the model M that can be mapped to the Lhs pattern but not the Rhs pattern yielding an intermediate model IM; (iv) *adding* new elements to the intermediate model IM which exist in the Rhs but cannot be mapped to the Lhs yielding the derived model M'.

In our framework, graph transformation rules serve as elementary operations while the entire operational semantics of a language or a model transformation is defined by a **model transformation system** [25], where the allowed transformation sequences are constrained by a *control flow graph* (CFG) applying a transformation rule in a specific *rule application mode* at each node. A rule can be executed (i) parallelly for all matches as in case *forall* mode; (ii) on a (non-deterministically selected) single matching as in case of *try* mode; or (iii) as long as applicable (in *loop* mode).

UML statecharts as the source modeling language As the formalization of UML statecharts (abbreviated as SC) by using this technique and a model checking case study

were discussed in [21, 22], we only concentrate on the precise handling of the target language (i.e., Petri nets) in this paper. We only introduce below a simple UML model as running example and assume the reader's familiarity with UML and metamodels.

Example 1 (Voting). The simple UML design of Fig. 2) models a voting process which requires a consensus (i.e., unique decision) from the participants.

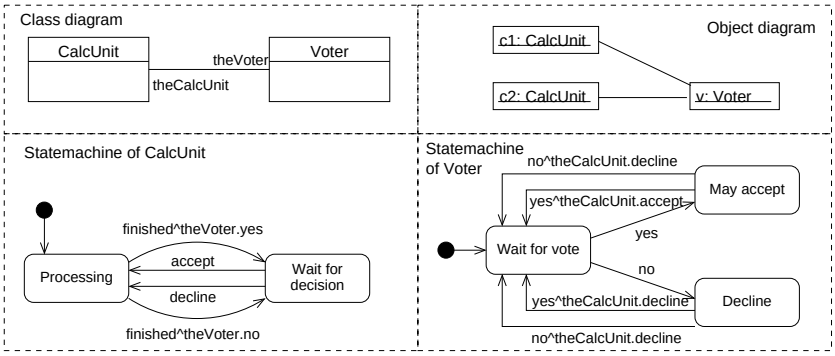


Fig. 2. UML model of a voter system

In the system, a specific task is carried out by multiple calculation units CalcUnit, and they send their local decision to the Voter in the form of a yes or no message. The voter may only accept the result of the calculation if all processing units voted for yes. After the final decision of the voter, all calculation units are notified by an accept or a decline message. In the concrete system, two calculation units are working on the desired task (see the object diagram in the upper right corner of Fig. 2), therefore the statechart of the voter is rather simplified in contrast to a parameterized case.

Petri nets as the target modeling language Petri nets (abbreviated as PN) are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available tools. A precise metamodeling treatment of Petri nets was discussed in [24]. Now we briefly revisit the metamodel and the operational semantics of Petri nets in Fig. 3.

According to the metamodel (the Petri Net package in the upper left corner of Fig. 3), a simple Petri net consists of Places, Transitions, InArcs, and OutArcs as depicted by the corresponding classes. InArcs are leading from (incoming) places to transitions, and OutArcs are leading from transitions to (outgoing) places as shown by the associations. Additionally, each place contains an arbitrary (non-negative) number of tokens). Dynamic concepts, which can be manipulated by rules (i.e., attributes token, and fire) are printed in red.

The operational behavior of Petri net models are captured by the notion of *firing a transition* which is performed as follows.

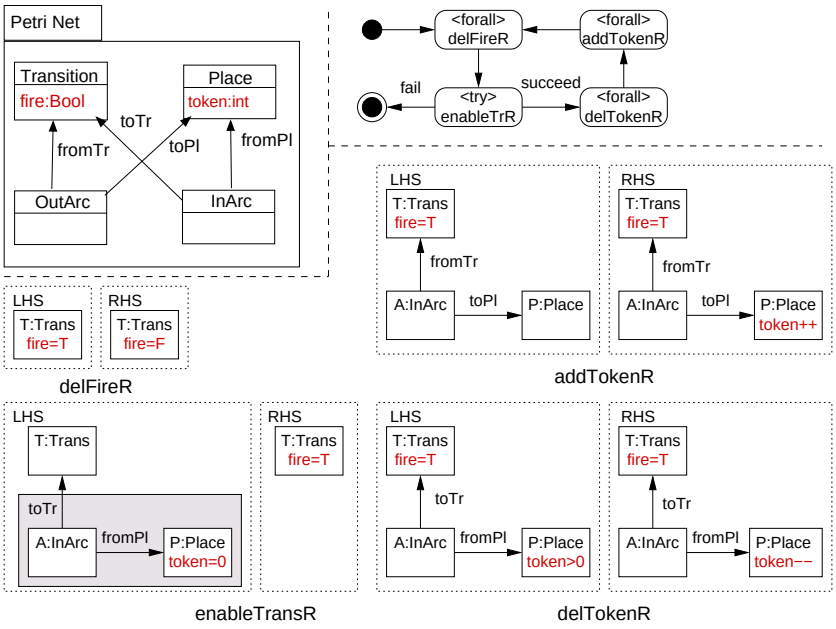


Fig. 3. Operational semantics of Petri nets by graph transformation

1. First, the fire attribute is set to false for each transition of the net by applying rule *delFireR* in *forall* mode.
2. A single enabled transition T (i.e., when all the places P with an incoming arc A to the transition contain at least one token, token>0) is selected to be fired (by setting the fire attribute to true) when applying rule *enableTransR* in *try* mode.
3. When firing a transition, a token is removed (i.e., the counter token is decremented) from each incoming place by applying *delTokenR* in *forall* mode.
4. Then a token is added to each outgoing place of the firing transition (by incrementing the counter token) in a *forall* application of rule *addTokenR*.
5. When no transitions are enabled, the net is dead.

3.2 Defining the SC2PN model transformation

Modeling statecharts by Petri nets Each SC state is modeled with a respective place in the target PN model. A token in such a place marks the corresponding state as active, therefore, a single token is allowed on each level of the state hierarchy (forming a token ring, or more formally, a *place invariant*). In addition, places are generated to model messages stored in event queues of a statemachine. However, the proper handling of event queues is out of the scope of the current paper, the reader is referred to [23].

Each SC step (i.e., a collection of SC transitions that can be fired in parallel) is projected into a PN transition. When such a transition is fired, (i) tokens are removed from source places (i.e., places generated for the source states of the step) and event

queue places, and (ii) new tokens are generated for all the target places and receiver message queues. Therefore, input and output arcs of the transition should be generated in correspondence with this rule.

Example 2.

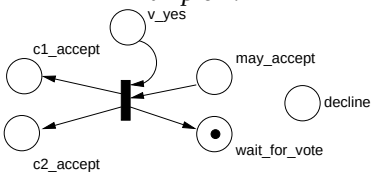


Fig. 4. The Petri net of the voter

In Fig. 4, we present an extract of the Petri net equivalent of the voter's UML model (see Fig. 2). For improving legibility, only a single transition (leading from state may_accept to wait_for_vote and triggered by the yes event) is shown.

The places of the voter subsystem are constituted of the states of the voter (such as wait_for_vote, may_accept, decline) and message queues for valid events (like yes). The initial state is marked by a token in wait_for_vote. The depicted transition has two incoming arcs as well, one from its source state may_accept and one from the message queue of the triggering yes event. Meanwhile, this transition has multiple output places: one for the target state wait_for_vote, and one for each target event queue of the participants that receives the generated accept message.

Formalizing model transformations In [23], we formalize the SC2PN transformation (to handle a meaningful subset of UML statecharts) by model transformation systems consisting of more than 40 graph transformation rules. Feeding these high-level descriptions to VIATRA [5], (an XMI representation of) a transformation program is generated automatically, which would yield the target Petri net model (Fig. 4) as the output when supplying (the XMI representation of) the voter's UML model (Fig. 2) as the input.

Figure 5 gives a brief extract of transforming SC states into PN places. According to this pair of rules, each initial state (i.e., that is active initially) in the source SC model is transformed into a corresponding PN place containing a single token, while each non-initial state (i.e., that is passive initially) is projected into a PN place without a token.

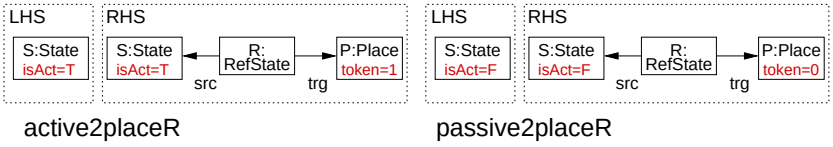


Fig. 5. Transforming SC states into PN places

It is worth noting that a model transformation rule in VIATRA is composed of elements of the source language (like State S in the rule), elements of the target language (like Place P), and reference elements (such as RefState R). The latter ones are also defined by a corresponding metamodel. Moreover, they provide bi-directional transfor-

mations for the *static parts* of the models, thus serving as a basis for back-annotating the results of a Petri net-based analysis into the original UML design.

3.3 Verification of the SC2PN model transformation

For the SC2PN case study, Steps 1–3 in our verification framework have already been completed. Now, a transition system (TS) is generated automatically (according to [22]) for source and target models as an equivalent (model-level) representation of the operational semantics defined by graph transformation rules (on the meta-level).

Generating transition systems Transition systems are a common mathematical formalism that serves as the input specification of various model checker tools. They have certain commonalities with structured programming languages (like C or Pascal) as the system is evolving from a given *initial state* by executing non-deterministic if-then-else like *transitions* (or *guarded commands*) that manipulate *state variables*. In all practical cases, we must restrict the state variables to have finite domains, since model checkers typically traverse the entire state space of the system to decide whether a certain property is satisfied. For the current paper, we use the easy-to-read SAL syntax for the concrete representation of transition systems.

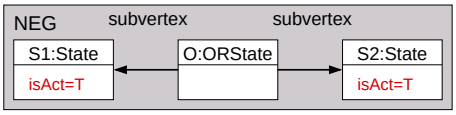
Our generation technique (described in [22] also including feasibility studies from a verification point of view) enables model checking for graph transformation systems by automatically translating them into transitions systems. The main challenge in such a translation is two fold: (i) we have to "step down" automatically from the meta-level to the model-level when generating model-level transition systems from meta-level graph transformation systems, and (ii) a naive encoding of the graph representation of models would easily explode both the state space and the number of transitions in the transition system even for simple models. Therefore our technique applies the following sophisticated optimizations:

- Introducing state variables in the target transition system only for dynamic concepts of a language.
- Including only dynamic parts of the initial model in the initial state of the transition system.
- Collecting potential applications of a graph transformation rule by partially applying them on the static parts of the rule and generating a distinct transition (guarded command) for each of them that only contains dynamic parts as conditions in guards and assignments in actions.

Formalizing the correctness property Now, a semantic criterion is defined for the verification process that should be preserved by the SC2PN model transformation. Note that the term "safety criterion" below refers to a class of temporal logic properties prohibiting the occurrence of an undesired situation (and not to the safety of the source UML design).

Definition 1 (Safety criterion for statecharts). *For all OR-states (non-concurrent composite states) in a UML statechart, only a single substate is allowed to be active at any time during execution.*

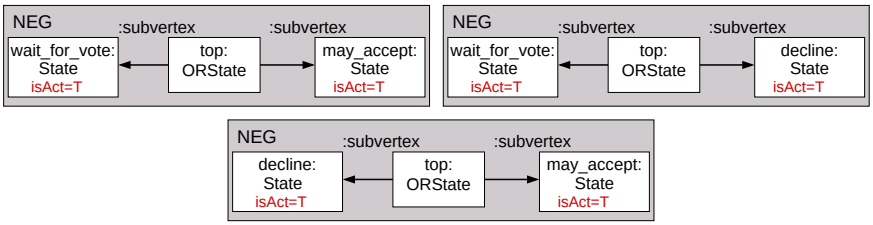
This informal requirement can be formalized by the following graphical invariant in the domain of UML statecharts (cf. Fig. 6 together with its equivalent logic formula). Informally speaking, it prohibits the simultaneous activeness of two distinct substates S1 and S2 of the same OR-state C (i.e., non-concurrent composite state) .



$$\nexists\, O : ORState, S1 : State, S2 : State : subvertex(A, S1) \wedge subvertex(A, S2) \wedge isAct(S1) \wedge isAct(S2) \wedge S1 \neq S2$$

Fig. 6. A sample graphical safety criterion

Unfortunately, it is difficult to establish the same criterion on the meta level in the target language of Petri nets since the SC2PN transformation defines an abstraction in the sense that message queues of objects are also transformed into PN places (in addition to states). However, in order to model check a certain system, this meta-level correctness criterion can be re-introduced on the model level. Therefore, we first automatically instantiate (the static parts of) the criterion on the concrete SC model (as done during the transformation to transitions systems) to obtain the model level criterion of Fig. 7. Note that the different (model level) patterns denote conjunctions, therefore, none of the depicted situations are allowed to occur.



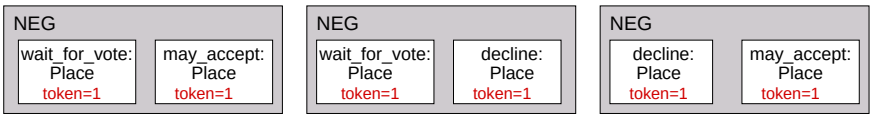
$$\neg(subvertex(top, wait\_for\_vote) \wedge subvertex(top, may\_accept) \wedge isAct(wait\_for\_vote) \wedge isAct(may\_accept)) \wedge ...$$

Fig. 7. Model level safety criterion

Note that our approach is not at all limited to verify only safety criteria. Further verification case studies (e.g., in [2, 22]) also covered reachability and liveness properties or deadlock freedom.

Equivalent property in the target language This model level criterion is appropriate to be transformed into an equivalent criterion for the Petri net model. As the state hierarchy of statecharts is not structurally preserved in Petri nets (as Petri nets are flat) the equivalents of the OR states are not projected into Petri nets. Therefore, the corresponding property (shown in Fig. 8) contain only specific places having a token.

At this point, we need to validate whether the equality (= 1) or inequality checks ($\geq$ 1) are required in the property to be proved (i.e., what to do if there are multiple tokens



$$\neg(token(wait\_for\_vote) = 1 \wedge token(may\_accept) = 1) \wedge ...$$

Fig. 8. The Petri net equivalent of the model level safety criterion

in a single place). We may conclude that checking equality is also sufficient, however, checking the version with inequality definitely strengthens the property, therefore we can also decide to prove something stronger in the Petri net model.

Obviously, constructing the pair of properties to be proved for property preservation is non-trivial and requires a certain insight into the source and target languages and their transformation. Therefore the generation of a target property q from a source property p cannot always be automated.

Model checking the target model Given (i) a system model in the form of a transition system TS (with semantics defined as a Kripke structure), and (ii) a property ϕ, the *model checking problem* can be defined as to decide whether ϕ holds on all execution paths of the system (i.e., whether $TS \models \phi$).

Therefore, as the final step of our framework, the model checker is supplied with the transition system of the Petri net model and the textual representation of the property q. As the places derived from the states of the same OR-state form a place invariant (with a single token circulating around), the model checker easily verifies even the strengthened property.

As a conclusion for our case study, the SC2PN model transformation preserved our sample correctness property for a specific source statechart model and its target Petri net equivalent. Additional correctness properties can be handled similarly. Unfortunately, for space considerations, we omitted the formal verification of property in the source SC model (Step 6), which could be performed identically to the handling of the target PN model.

4 Conclusions and Future Work

We presented a model-level, modeling language independent and highly automated technique to formally verify by model checking that a model transformation from a specific (but arbitrarily chosen) well-formed model instance of a source modeling language into its target equivalent preserves (language specific) dynamic consistency properties. We demonstrated the feasibility of our approach by verifying a semantic correctness property for a complex model transformation from UML statecharts to Petri nets.

Naturally, as based on model checking our technique has practical limitation imposed by the state explosion problem. Therefore, in the future, we aim to improve our automated encoding into transition systems to better exploit the built-in facilities of

model checkers (like partial order reduction or symmetries) to allow the verification of larger scale model transformations.

Further research should also aim at automating the transformation of semantic correctness properties. We think that our model transformation technique can be extended to handle this case as well. As a result, the same specification technique would be used for all transformations in our verification framework.

References

1. D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 243–258. Springer-Verlag, Dresden, Germany, 2002.
2. L. Baresi, R. Heckel, S. Thöne, and D. Varró. Modeling and analysis of architectural styles. In *Proc ESEC 2003: European Software Engineering Conference*. Helsinki, Finland. In press.
3. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway (ed.), *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pp. 187–196. 2000.
4. A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, vol. 16(5):pp. 265–275, 2001.
5. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In *Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, pp. 267–270. IEEE Press, Edinburgh, UK, 2002.
6. J. de Lara and H. Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber (eds.), *5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings*, vol. 2306 of *LNCS*, pp. 174–188. Springer, 2002.
7. G. Engels, R. Heckel, and J. M. Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In M. Gogolla and C. Kobryn (eds.), *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools*, vol. 2185 of *LNCS*, pp. 272–286. Springer, 2001.
8. G. Engels, R. Heckel, J.-M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 212–227. Springer, Dresden, Germany, 2002.
9. J. H. Hausmann, R. Heckel, and S. Sauer. Extended model relations with graphical consistency conditions. In *UML 2002 Workshop on Consistency Problems in UML-based Software Development*, pp. 61–74. Blekinge Institute of Technology, 2002. Research Report 2002:06.
10. R. Heckel, J. M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: First International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 161–176. Springer, Barcelona, Spain, 2002.
11. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, vol. 23(5):pp. 279–295, 1997.
12. G. Huszerl and I. Majzik. Quantitative analysis of dependability critical systems based on UML statechart models. In *HASE 2000, Fifth IEEE International Symposium on High Assurance Systems Engineering*, pp. 83–92. 2000.

13. J.-M. Jézéquel, W.-M. Ho, A. L. Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. In R. J. Hall and E. Tyugu (eds.), *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
14. J. M. Küster, R. Heckel, and G. Engels. Defining and validating transformations of UML models. In *Proc. VLFM'03: International Conference on Visual Languages and Formal Methods*. Submitted.
15. D. Milicev. Automatic model transformations using extended UML object diagrams in modeling environments. *IEEE Transactions on Software Engineering*, vol. 28(4):pp. 413–431, 2002.
16. Object Management Group. *UML Profile for Schedulability, Performance and Time*. `http://www.omg.org`.
17. A. Pataricza. Semi-decisions in the validation of dependable systems. In *Suppl. Proc. DSN 2001: The International IEEE Conference on Dependable Systems and Networks*, pp. 114–115. Göteborg, Sweden, 2001.
18. G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.
19. Á. Schmidt and D. Varró. CheckVML: A tool for model checking visual modeling languages. In *Proc. UML 2003: 6th International Conference on the Unified Modeling Language*. Accepted paper.
20. A. Schürr. Specification of graph translators with triple graph grammars. In . Tinhofer (ed.), *Proc. WG94: International Workshop on Graph-Theoretic Concepts in Computer Science*, no. 903 in LNCS, pp. 151–163. Springer, 1994.
21. D. Varró. A formal semantics of UML Statecharts by model transition systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Proc. ICGT 2002: 1st International Conference on Graph Transformation*, vol. 2505 of *LNCS*, pp. 378–392. Springer-Verlag, Barcelona, Spain, 2002.
22. D. Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modelling*, 2003. Accepted to the Special Issue on Graph Transformation and Visual Modelling Techniques.
23. D. Varró. *Automated Model Transformations for the Verification and Validation of IT Systems*. Ph.D. thesis, Budapest University of Technology and Economics, Department of Measurement and Information Systems, 2003. Submitted.
24. D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modelling*, 2003 (1):pp. 1–24.
25. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, vol. 44(2):pp. 205–227, 2002.
26. J. Whittle. Transformations and software modeling languages: Automating transformations in UML. In J.-M. Jézéquel, H. Hussmann, and S. Cook (eds.), *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, vol. 2460 of *LNCS*, pp. 227–242. Springer-Verlag, Dresden, Germany, 2002.

Assert, Negate and Refinement in UML-2 Interactions

Harald Störrle

Institut für Informatik
Ludwig-Maximilians-Universität München
Oettingenstr. 67, 80538 München, GERMANY
stoerrle@informatik.uni-muenchen.de

Abstract. The Unified Modeling Language (UML) is the industry standard for modeling. With its recent advancement to version 2.0, there have been large amounts of changes and additions. In this paper I study some new features with a view to formal specification and verification, in particular the operators neg and assert, and notions of refinement based upon them.

1 Introduction

The Unified Modeling Language (UML) is the industry standard for modeling, it has even been dubbed "*the lingua franca of software engineering*" (cf. [19, p. v]). However, over the past few years, a number of rather serious shortcomings have been identified, for instance with respect to the formal semantics of UML models, most notably of the dynamic models.

Recently, however, the version 2.0 of UML has been adopted (see [15]), addressing a number of these shortcomings. In particular, Message Sequence Charts (MSC) according to the ISO standard (see [7, 6]) have been integrated. In UML, the concept underlying these notations is called *interaction*.

In a companion paper and technical report (see [23, 22]), I have defined a formal semantics for most of the operators in UML interactions (including time), except those without a straightforward semantics, namely neg and assert. These are also not present in classical MSCs but have precursors in Life Sequence Charts (LSCs, see [3]). The latter, however, have been extensively used in the specification and verification of critical systems, e.g., in the automotive domain. So, the new features of UML might well be exploited for the same purpose.

The semantics of most of UML 2.0-interactions is more or less straightforward. So, I focus on the interesting parts. I discuss a number of alternative interpretations with their advantages and disadvantages with respect to notions of refinement, which would be a natural starting point both for development and verification tasks. I also generalize the interpretations to timed interactions.

2 Interactions in UML 2.0

To level the ground, I start with a brief discussion of the concrete and abstract syntax of interactions in UML 2.0. Here, I shall refer to the UML 2.0 as the "new standard" or simply "the standard" while I refer to the version 1.4 as the "old standard".

2.1 Concrete Syntax

First of all, all diagrams now have a frame around them and a compartment displaying its type and name (see Figure 1) which makes it easier to refer to it, e.g. as a subdiagram or companion diagram.

In the old standard, there were two types of interaction diagrams, namely sequence and collaboration diagrams which both are based on the same metamodel concepts (see below). So called "*metric sequence diagrams*" [16] had been mentioned in UML 1.3, but neither defined nor explained, and have been abandoned in UML 1.4.

In the new standard, collaboration diagrams have been renamed to communication diagrams. A new kind of interaction diagram, timing diagrams as known in many engineering disciplines (see Figure 1) have been introduced. Timing diagrams may be considered as an elaboration of metric sequence diagrams. While communication diagrams and sequence diagrams focus on structure and message exchange, respectively, timing diagrams focus on state and state change across time. Sequence diagrams have been extended considerably, and now have approximately the same expressive power as high-level MSCs.

All these interaction diagrams may be combined ad lib by a given set of InteractionOperators. The notation is similar to that of interaction diagrams in general (see Figure 1). If there are two arguments to an InteractionOperator, they are divided by a dashed line. The notation is strongly reminiscent of MSCs. Also, in the new standard there are now interaction overview diagrams which are basically activity diagrams where the interaction diagrams are activities (see [22]). They correspond to High-Level MSCs (see [7]).

Furthermore, there are also so called overview diagrams which may be used to combine interaction diagrams into a kind of activity diagram, where the places of activty states are taken by interactions.

2.2 Basic semantics

In the new standard, "*an EventOccurrence is the basic semantic unit of Interactions*" (cf. [15, p. 416]), and the "*sequences of EventOccurences are the meanings of Interactions*" (ibid.). Given the domain of Event Occurrences (written $\mathcal{EO}$), the domain of Traces (written $\mathcal{SEQ}$) is $\mathcal{SEQ} = \mathcal{EO}^*$.

More precisely, however, "*the semantics of an Interaction is given as a* pair *of sets of traces*" (cf. [15, p. 419, emphasis added]), representing "*valid traces and invalid traces*" (ibid.), respectively. Thus, the semantic domain for interactions is $\mathcal{SEQ} \times \mathcal{SEQ}$. "*The traces that are not included [in the union of the two]*

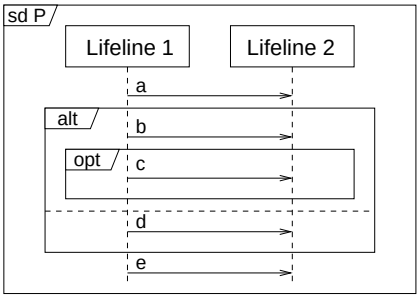


Fig. 1. A sample UML 2.0 interaction diagram, including the high-level operators alt and opt. The language of messages described by this sequence diagram may also be represented by the regular expression `a(b[c]|d)e`.

are not described [...] and we cannot know whether they are valid or invalid" (ibid.). That is, an interaction in UML 2.0 implicitly describes contingency, see Figure 2. It is not obvious, what valid and invalid really mean, however. For the time being, I shall interpret them as necessary vs. forbidden or must vs. must not (see Section 4.2 below).

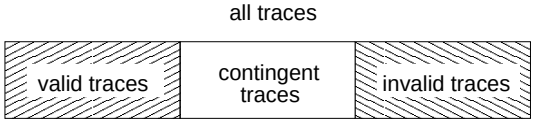


Fig. 2. In UML 2, interactions specify both valid and invalid traces. Unspecified traces are contingent. Probably, valid and invalid traces should be interpreted as necessary and forbidden (or must and must not).

This point of view is obviously adopted from Life Sequence Charts (cf. [3]). With this definition, the mapping from the metamodel to a mathematical domain is now simply

$$\text{(interaction)} \qquad [\![\mathsf{sd}(P)]\!] = \langle [\![P]\!], \emptyset \rangle$$

for an interaction P, with $[\![\_]\!]$ to denote a (denotational) semantic function. In order to distinguish between alternative interpretations of constructs, semantic brackets will be subscripted as in $[\![\_]\!]_{\text{Interpretation}}$.

As a convention, I write the first and second component of a pair X as X^+ and X^-, respectively. Thus, the valid and invalid traces that are the semantics of an interaction P may be written as $[\![P]\!]^+$ and $[\![P]\!]^-$, respectively.

Coming back to the interpretation of sd from above, of course, $[\![\_]\!]$ must also be defined for the other operators (see [23] for details). For example, the alt denotes alternatives in interactions, and its semantics can be defined as

$$(\mathsf{alt}) \qquad [\![\mathsf{alt}(P,Q)]\!] = [\![P]\!] \cup [\![Q]\!].$$

Here I also use the canonical extension of set operators to pairs of sets, that is, I write $\langle A, B\rangle \cup \langle X, Y\rangle$ to mean $\langle A \cup X, B \cup Y\rangle$, and so on for other operators.

2.3 Characterization of interaction semantics

A great number of semantics and equivalences for concurrency have been defined (see e.g. [4, chapter 1] for a comparison). I shall now discuss some of the more popular semantic dimensions.

Interleaving vs. true concurrency This spectrum is usually associated with CSP and CCS on the interleaving end (see [2] and [13], respectively), and Petri-nets and related formalisms on the other (see [17]). Both of these semantic paradigms have been applied to MSCs in the past (cf. [5, 20] for examples and [18] for an overview). The standard avoids a clear statement in favor of interleaving semantics, merely saying that: "*to* explain *Interactions we apply an Interleaving semantics*" (cf. [15, p. 403, emphasis added]). This statement implies that the interleaving semantics provided in the standard is only an explanation, but not a definition. This would mean that other formalisms might be valid definitions (or explanations) of the semantics of interactions in UML, too.

Linear vs. branching time The two ends of this spectrum are often associated with CSP and CCS, respectively. Without the notion of invalid traces, the semantics of UML interactions and the notions of equivalence and refinement would be identical to traditional trace semantics. Considering also the invalid traces simply adds another set of traces, but does not change the semantic paradigm in any way. That is, the UML standard defines a linear time semantics.

Readiness/failure traces In readiness and failure semantics, each trace also carries a set of actions (read: EventOccurrences) which are (not) possible after the trace. For a given set X of conventional traces, it is possible to compute largest common prefixes and determine the respective possible next EventOccurrences. For a given interaction P, it is possible to compute the set of ready traces $\langle t, R\rangle$ from $[\![P]\!]^+$. As an example, suppose that $\Sigma = a, b, c, d, e$ and $X = \{a.b.c, a.b.d, a.e\}$. The common prefixes with their ready EventOccurrences are $\langle a, \{b, e\}\rangle$, $\langle a.b, \{c, d\}\rangle$, $\langle a.b.c, \emptyset\rangle$, $\langle a.b.d, \emptyset\rangle$, and $\langle a.e, \emptyset\rangle$. The set of failure traces of P is $\{\langle t, \overline{R}\rangle\}$, where $\langle t, R\rangle$ is computed like the ready traces, but starting from $[\![P]\!]^-$ rather than $[\![P]\!]^+$. It is unclear, however, how the notion of contingency integrates with this paradigm: in failure and readiness semantics, an action is either possible or not.

3 The neg-operator

The neg-opeator is probably a kind of negation. The standard does not give an example, or explain the intuition or pragmatics of this operator, but flatly declares that "*the interaction operator* neg *designates that the combined fragment represents traces that are defined to be invalid*" (cf. [15, p. 411]). This could be interpreted as

(N.1: loose negate) $[\![\mathsf{neg}(P)]\!]_{N.1} = \langle \emptyset, [\![P]\!]^+ \rangle$.

This interpretation could be formulated intuitively as "*not the traces of P*". However, under this interpretation, all negative traces specified so far would be lost. It thus behaves strangely under double negation, for

$$[\![\mathsf{neg}(\mathsf{neg}(P))]\!]_{N.1} = \langle \emptyset, \emptyset \rangle.$$

Also, the standard declares that "*All InteractionFragments that are different from Negative are considered positive, meaning that they describe traces that are valid [...]* " (cf. [15, p. 370]). This suggests the following interpretation,

(N.2: strict negate) $[\![\mathsf{neg}(P)]\!]_{N.2} = \langle\ \overline{[\![P]\!]^+_{N.2}}, [\![P]\!]^+_{N.2} \rangle$

where $\overline{X} = \Sigma^* - X$, that is, language complement. Observe that $[\![P]\!]^- \subseteq \overline{[\![P]\!]^+}$ and $[\![P]\!]^+ \subseteq \overline{[\![P]\!]^-}$ for all P and for *all* interpretations, and therefore $[\![P]\!]^+ \subseteq [\![\mathsf{neg}(P)]\!]^+_{N.2}$, that is, interpretation N.2 respects invalid traces of P.

Under interpretation N.2, the traces specified by P are marked as invalid, and all other traces as valid. This could be expressed intuitively as "*anything but P*". However, this interpretation makes no sense for double negation. Abbreviating $\mathsf{neg}(P)$ as Q, I have $[\![Q]\!] = \langle\ \overline{[\![P]\!]^+_{N.2}}, [\![P]\!]^+_{N.2} \rangle$ and thus $[\![Q]\!]^+_{N.2} = \overline{[\![P]\!]^+_{N.2}}$ and so

$$\begin{aligned}
[\![\mathsf{neg}(\mathsf{neg}(P))]\!]_{N.2} &= [\![\mathsf{neg}(Q)]\!]_{N.2} \\
&= \langle\ \overline{[\![Q]\!]^+}, [\![Q]\!]^+_{N.2} \rangle \\
&= \langle \overline{\overline{[\![P]\!]^+_{N.2}}}, \overline{[\![P]\!]^+_{N.2}} \rangle \\
&= \langle [\![P]\!]^+_{N.2}, \overline{[\![P]\!]^+_{N.2}} \rangle
\end{aligned}$$

Using $\mathsf{flip}(\langle x, y \rangle) = \langle y, x \rangle$, this means that

$$[\![\mathsf{neg}(\mathsf{neg}(P))]\!]_{N.2} = \mathsf{flip}([\![\mathsf{neg}(P)]\!]_{N.2})$$

and also

$$[\![\mathsf{neg}(\mathsf{neg}(\mathsf{neg}(\mathsf{neg}(P))))]\!]_{N.2} = [\![P]\!]_{N.2}$$

Clearly, this is odd. It leads us to a simpler, more intuitive interpretation of neg.

(N.3: flip negate) $\quad [\![\mathsf{neg}(P)]\!]_{N.3} = \mathsf{flip}([\![P]\!])_{N.3}$.

This interpretation simply reverses valid and invalid traces of P, but preserves the contingent traces. Intuitively, it could be formulated as "*flip valid and invalid*". Note that this interpretation yields an intuitive sense for double negation:

$$[\![\mathsf{neg}(\mathsf{neg}(P))]\!]_{N.3} = [\![P]\!]_{N.3}$$

as $\mathsf{flip} \circ \mathsf{flip}$ obviously is the identity. Obviously, this interpretation is in contradiction to some of the citations from the standard as given above. Still, I shall adopt the interpretation N.3, as it is simply the only consistent approach.

4 The assert-operator

The assert-operator might be a kind of affirmation, implication, or temporal sequence. The standard unfortunately gives only a single (rather unhelpful) example (cf. [15, p. 442]), and does not provide an intuitive explanation of its meaning or usage. In this section, I discuss several possible interpretations fo this operator.

4.1 assert as affirmation

Intuitively, one might interpret assert as an affirmation of the traces of its operand, in the sense of "*P, and only P*". The meaning of the assert-operator is explained by the standard as "*the sequences of the operand are the only valid continuations. All other continuations result in invalid traces.*" (cf. [15, p. 412]) This suggests the following interpretation:

(A.1: affirm) $\quad [\![\mathsf{assert}(P)]\!]_{A.1} = \langle [\![P]\!]^+, \overline{[\![P]\!]^+} \rangle$.

When interpreting assert as an affirmation, one would expect that the meaning of an interaction remains constant, no matter how often it is asserted, that is, a kind of idempotency-property. One would expect that $\mathsf{assert}(P)$ is equivalent to $\mathsf{assert}(\mathsf{assert}(P))$, e.g., in the sense that

$$[\![\mathsf{assert}(P)]\!] = [\![\mathsf{assert}(\mathsf{assert}(P))]\!]$$

This is obviously true for interpretation A.1. Under this interpretation, assert would completely remove contingency, but preserve valid traces and invalid traces, i.e. $[\![\mathsf{assert}(P)]\!]^+_{A.1} = [\![P]\!]^+$ and $[\![\mathsf{assert}(P)]\!]^-_{A.1} \supseteq [\![P]\!]^-$.

In the quotation given above, the standard seems to demand this interpretation quite imperatively. However, there is also a contradictory statement in the standard, declaring that "*the invalid set of traces are associated only with the use of a Negative CombinedInteraction.*" (cf. [15, p. 419]). Interpretation A.1 clearly refers to the set of negative traces in a way that cannot be achieved by understanding assert as syntactic sugar. So, there is some freedom for interpretations of assert, and I shall explore some of them now.

4.2 assert as a binary operator

Another problem with interpretation A.1 is the fact that so far it is described as being unary. It might be understood as a binary operator, too, as the UML standard declares that "*the sequences of the operand (sic!) of the assertion are the only valid continuations.*" (cf. [15, p. 412]). Even though the standard explicitly mentions only a single operand, it talks about it as being a continuation of a preceeding trace. But it is unclear, what the scope of the preceeding trace is (see example in Figure 3).

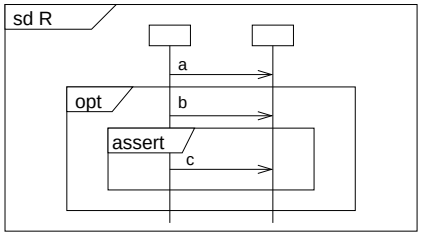


Fig. 3. Which messages constitute the trace preceeding the assert: only b or both a and b?

It would be much easier, if the assert were a binary operator where the first operand declares a condition or trigger (a kind of "*precharts*" known from LSCs, [3]), and the second declares the consequence or result. This is also suggested by the standard when declaring that "*Assertions are often combined with ignore or consider as shown in Figure 345*" (cf. [15, p. 412]). The Figure mentioned is reproduced in a simplified form in Figure 4 (left). A stratified variant of the notation is proposed in Figure 4 (right). The operators ignore and consider are defined to be dual. To simplify my task, I choose ignore in the remainder, thus sparing me another auxiliary filter-function (cf. [23]).

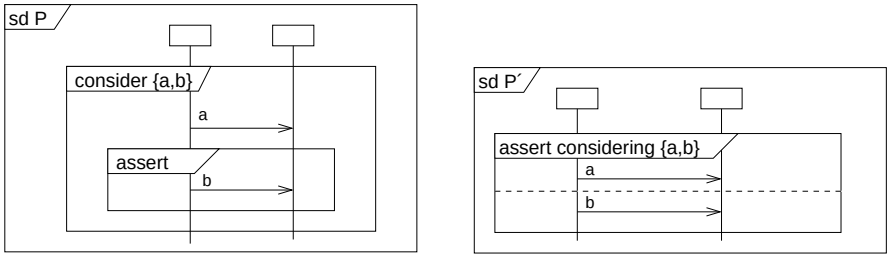


Fig. 4. Usage of assert/consider as suggested by the standard (left). A simple way to cleanly embed this usage into the notation (right).

However, there is a fundamental problem here. Recall the intuitive interpretation of valid and invalid traces as must and must not be possible, as laid out in Section 2.2. Suppose that P is an interaction, Γ a set of messages to be ignored and $\mu : \mathcal{EO} \to \mathcal{MSG}$ a mapping from the event occurrences of P to the messages they belong to. Then what is the meaning of $[\![\mathsf{ignore}(P, \Gamma, \mu)]\!]$?

A first approach might yield

$$[\![\mathsf{ignore}(P, \Gamma, \mu)]\!] = \langle \Sigma^* \sqcup\!\sqcup [\![P]\!]^+, \Sigma^* \sqcup\!\sqcup [\![P]\!]^- \rangle \tag{I.1}$$

such that $\Sigma = \{x \in \mathcal{EO} \mid \mu(x) \in \Gamma\}$. The shuffle-operator $\sqcup\!\sqcup$ is defined for $v, w \in \Sigma^*$ and the empty sequence ϵ as $\epsilon \sqcup\!\sqcup w = w$, $v \sqcup\!\sqcup \epsilon = v$, and

$$xv \sqcup\!\sqcup yw = \{x(v \sqcup\!\sqcup yw), y(xv \sqcup\!\sqcup w)\}.$$

So, for example, shuffling the two sequences $a.b$ and $x.y.z$ yields the following traces: $a.b.x.y.z$, $a.x.b.y.z$, $a.x.y.b.z$, $a.x.y.z.b$, $x.a.b.y.z$, $x.a.y.b.z$, $x.a.y.z.b$, $x.y.a.b.z$, $x.y.a.z.b$, $x.y.z.a.b$. Observe, that the order of symbols from the original traces is respected. The shuffle operator can be extended canonically to sets of words, of course.

Interpreting assert as binary, one might alternatively define

$$[\![\mathsf{assert}(P, Q, \Gamma, \mu)]\!]^+ = [\![P]\!]^+.\Sigma^*.[\![Q]\!]^+, \tag{I.2}$$

with Σ, Γ and μ as before. Recall that Γ denotes messages that are to be ignored, so that in the semantics, all possible event occurrences corresponding to these messages must be considered, which is just $(\mu(\Gamma))^* = \Sigma^*$.

These last two interpretations are problematic, of course, since now, many contingent traces have become valid or invalid, and probably much more than have been intended. Also, there may now be sequences of event occurrences of arbitrary length between the trigger and the consequence, which might also not be intended.

4.3 assert as implication

First of all, it is somewhat reminiscent of implication in classical logic (ignoring the temporal aspect of assert for a moment). Here, I have for instance $(\alpha \Rightarrow \beta) \Leftrightarrow (\neg\alpha \vee \beta)$. The two sides would correspond to $\mathsf{assert}(P, Q)$ and $\mathsf{alt}(P, \mathsf{neg}(Q))$, respectively, and one would expect them to be equivalent in some sense, e.g., as

$$[\![\mathsf{assert}(P, Q)]\!] = [\![\mathsf{alt}(\mathsf{neg}(P), Q)]\!].$$

Using $[\![\mathsf{alt}(P, Q)]\!] = [\![P]\!] \cup [\![Q]\!]$ as described above and interpretation N.3 from above, I would yield

$$[\![\mathsf{assert}(P, Q)]\!]_{A.2} = \langle [\![P]\!]^- \cup [\![Q]\!]^+, [\![P]\!]^+ \cup [\![Q]\!]^- \rangle. \tag{A.2: imply}$$

Note however, that under interpretation A.2, there is implication and disjunction (and even negation), but there is no operator corresponding to conjunction. Also, idempotency is not preserved, since

$$[\![\mathsf{assert}(P,P)]\!]_{A.2} = \langle [\![P]\!]^+ \cup [\![P]\!]^-, [\![P]\!]^- \cup [\![P]\!]^+ \rangle,$$

that is, the specified traces are both valid and invalid, which does not make much sense in general.

4.4 assert as consequence

In another sense, assert might be interpreted in a more temporal way, as is also suggested by the standard, when declaring that "*we expect a q message to occur once a v has occurred*" (cf. [15, p. 442, with a view to Fig. 345, see Figure 4 (left) in this paper]). Intuitively, this could be expressed as "*if P has occurred, then Q and only Q must follow*", i.e. formally

(A.3a) $$[\![\mathsf{assert}(P,Q)]\!]_{A.3a} = \langle \emptyset, [\![P]\!]^+.\overline{[\![Q]\!]^+} \rangle.$$

This is unintuitive in the sense that P only *may* occur, but does not *have to* occur. So, nothing at all needs to happen, and all other traces are contingent (cf. Section 2.2).

One might instead interpret $\mathsf{assert}(P,Q)$ as "*P must occur, and then Q must follow*". This would result in

(A.3b) $$[\![\mathsf{assert}(P,Q)]\!]_{A.3b} = \langle [\![P]\!]^+.[\![Q]\!]^+, [\![P]\!]^+.\overline{[\![Q]\!]^+} \rangle.$$

Arguably, any traces forbidden by P and Q alone should still be forbidden for $\mathsf{assert}(P,Q)$, so that one might want to refine interpretation A.3b to

(A.3c: next) $$[\![\mathsf{assert}(P,Q)]\!]_{A.3c} = \langle [\![P]\!]^+.[\![Q]\!]^+, [\![P]\!]^- \cup [\![Q]\!]^- \cup [\![P]\!]^+.\overline{[\![Q]\!]^+} \rangle.$$

Note that under this interpretation, $\mathsf{assert}(P,Q)$ is not syntactic sugar for $\mathsf{alt}(\mathsf{seq}(P,Q), \mathsf{neg}(\mathsf{seq}(P, \mathsf{neg}(Q))))$. Note also, that idempotency makes no sense under this interpretation.

5 Design steps

In the previous sections I have attempted to provide intuitive and formally satisfying interpretations for the operators assert and neg. These attempts have led to problems with the notions of valid and invalid traces. Therefore, in this section, I try to explore the pragmatic justification of these notions.

One might presume, that the motivation for having two separate sets of valid and invalid traces at the same time—and thus implicitly also a third set

of contingent traces—lies in an idea of consecutive refinement steps. Such a sequence of refinements could carefully carve the desired system behavior out of the totality of all traces.

As an analogy, consider the notion of loose semantics in abstract data types (cf. [24]). There, providing more and more "axioms" for a "specification" in a series of refinement steps allows less and less "implementations" for that "specification". In this analogy, possible and impossible implementations would correspond to valid and invalid traces. Adding axioms would correspond to adding interaction fragments to a design (see Figure 5).

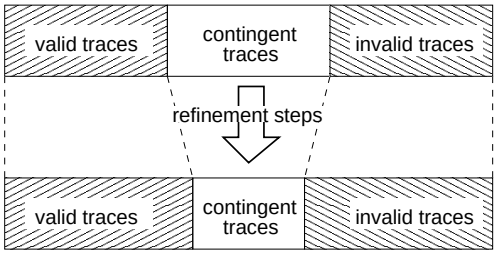


Fig. 5. Refinement reduces uncertainty: valid and invalid traces *must* and *must not* be possible, respectively, while contingent only *may* be possible, but need not be.

5.1 Basic relationships

All relationships between two interactions such that one of them contains more detail and less uncertainty than the other are called **elaborations**. Formally, P elaborates Q (written $P \rightsquigarrow Q$) iff $[\![P]\!]^+ \supseteq [\![Q]\!]^+ \wedge [\![P]\!]^- \supseteq [\![Q]\!]^-$. See Figure 5 for an illustration.

An **enrichment** is a relationship where one interaction has more valid traces than another, while nothing is said about the invalid traces. Formally, P enriches Q (written $P \models^+ Q$) iff $[\![P]\!]^+ \supseteq [\![Q]\!]^+$.

Conversely, a **restriction** is a relationship where one interaction has more invalid traces than another, while nothing is said about the valid traces. Formally, P restricts Q (written $P \models^- Q$) iff $[\![P]\!]^- \supseteq [\![Q]\!]^-$.

Then, a **refinement** is a relationship where one interaction has both more valid and more invalid traces than another, that is P refines Q (written $P \models Q$) iff $([\![P]\!]^- \supseteq [\![Q]\!]^-) \wedge ([\![P]\!]^+ \supseteq [\![Q]\!]^+)$ or $P \models^+ Q \wedge P \models^- Q$, see Figure 5 for an illustration.

Finally, two interactions P and Q are **equivalent** (written $P = Q$) iff $[\![P]\!] = [\![Q]\!]$, or $P \models Q \wedge Q \models P$. Similar to process algebras, constructions like $[\![\mathsf{ignore}(Q, \Gamma, \mu)]\!] = [\![P]\!]$ can be used as a kind of refinement-relationship. Note

that equivalence is the only relationship among interactions the standard mentions ("*Two Interactions are equivalent if their pair[s] of trace-sets are equal*" (cf. [15, p. 420])), and that it is captured by equivalence as defined here.

Observe that only N.2 and A.1 constitute elaborations in the sense of $\mathsf{neg}(P) \rightsquigarrow P$ and $\mathsf{assert}(P) \rightsquigarrow P$. None of the other interpretations constitute any other of the relationships defined here. However, defining an elaboration as $\overline{[\![P]\!]^+ \cup [\![P]\!]^-} \subseteq \overline{[\![Q]\!]^+ \cup [\![Q]\!]^-}$ would also cover N.3 (flip negate).

These two interpretations have in common that they completely remove contingency. Thus, there can be no further (useful) elaborations afterwards, so there is always only exactly one refinement step using assert or neg. So refinement must really be achieved using other means, and so, refinement is not a justification for assert and neg under interpretations N.2 and A.1. All other interpretations, however, are not consistnt with the basic relationships defined above, and those have been very basic indeed.

So either the notion of valid and invalid traces has to be abandoned in favor of a simpler "single set of traces"-semantics. This would also imply to remove the operator neg, and more or less fix interpretation A.3b for assert in the sense that $[\![\mathsf{assert}(P)]\!] = [\![\mathsf{assert}(P)]\!]^+_{A.3b}$. Note, that $[\![\mathsf{assert}(P)]\!]^+_{A.3b} = [\![\mathsf{assert}(P)]\!]^+_{A.3c}$.

Or, alternatively, one might abandon the idea of refinement as a justification for assert and neg and, again, the whole idea of valid and invalid traces. But then, what justification is there?

In both cases, the standard needs clarification. One possible solution (the one I find most convincing from a practitioners point of view) is briefly discussed in Section 6 under the title "metalogical interpretation".

5.2 Applying and tracking design steps

In this section, I look at the impact of design steps. For simplicity, I assume for the time being, that a design consists of (versions of) only a single interaction diagram. This is no real restriction, since a set of interactions can be simulated using the alt-operator. So, the remarks and observations of this section can equally be applied to sets of interactions.

During development and evolution of a system, I distinguish three classes of design steps: detailing, adapting and realizing (see Figure 6).

Detailing means adding details so as to remove uncertainty. It subsumes all activities that leave the set of all traces unchanged, but decreases the number of contingent traces. This meaning is captured formally by the elaboration-relationship.

Adapting means changing a design to accommodate new ideas about a system. It can result in changed or unchanged sets of valid and invalid traces, and it can also change the overall set of traces (or leave it unchanged). There is no formal meaning for this kind of design step, but it can be

Realizing means that an interaction is taken as the specification of a behavioral model or a program. No implementation has contingency -

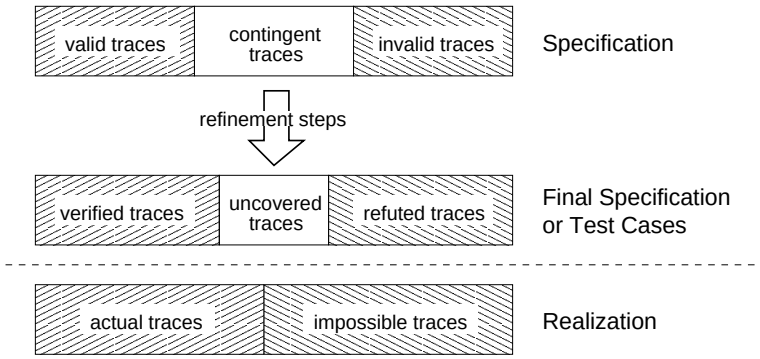


Fig. 6. Design as a sequence of refinement steps. A final step of realization completely removes all remaining contingencies.

traces are either possible, or impossible, and thus valid or invalid. Realizing is also a kind of elaboration-relationship.

These design steps and the underlying relationships between interactions may be used in a number of practical scenarios.

- Suppose that a specification given as an interaction P is realized by a system S, whose behavior is completely described by an interaction R. It can be automatically checked whether actually $R \rightsquigarrow P$, e.g. by model-checking the valid and the invalid traces of P vs. R.
- Seen in a different way, the valid traces of P may be used as test cases for S. The invalid traces can not be tested of course, as this would require a complete test coverage. For this usage, thus, only enrichments are necessary.
- Imagine a maintenance project of a large system, where the people in charge of doing a certain modification do not have adequate knowledge of the system, possibly because the system is poorly documented, and the knowledge is lost. Such systems would typically have quite a large population of interactions, as compared to the modification in question. Here, it is very easy to accidentally introduce an interaction Q such that $[\![Q]\!]^+ \cap [\![Q]\!]^- \neq \emptyset$ (like $\mathsf{alt}(P, \mathsf{neg}(P))$) under interpretation $\mathsf{N}.2$). Obviously, this makes no sense, and thus, Q should be refuted by an automated tool as being inconsistent.
- Suppose, that a series of design steps is being applied to an interaction P_0 resulting in $P_1, \ldots, P_n$. The impact of the design steps is easily determined by comparing $[\![P_i]\!]$ with $[\![P_{i+k}]\!]$. Now assume that at some point, a valid trace t is identified that really should be invalid, or the other way round. When trying to eliminate this trace, one must be careful not to introduce another error instead. For this task, thus, it would be helpful to be able to automatically track the origin of the trace by identifying the design step that introduced t, so as to determin whether it was an accident or not. To this end, one must find i such that $t \in [\![P_i]\!]^+$ but $t \notin [\![P_{i+1}]\!]^+$.

Clearly, these scenarios could be supported by automated tools implementing the relationships defined above. It is currently unclear, whether it is possible to have a calculus of refactoring design steps which might be formally justified.

6 Discussion

In this paper I have discussed several possible interpretations for the operators assert and negate as declared in the new UML 2.0 standard. It turns out that the explanations given in the standard are by no means adequate. It is thus currently not clear what the contribution of these operators in UML 2.0 to verification of and reasoning about software system will be. There are a number of open questions that remain to be explored.

Metalogical interpretation From a pragmatic perspective, a trace is a property, namely, that a given system does (or does not) exhibit a certain behaviour. The assert and neg are of a different kind in that they make statements about traces rather than modifying them. So, they are more like the operators in interaction overview diagrams. For this purpose however, assert and neg are not very powerful. So why not remove them from interactions and embed interactions in a traditional logic, e.g., like in Figure 7.

$\mathtt{Trace} ::= \mathtt{UML\text{-}Interaction}$		
$\mathtt{Expr} ::= \mathtt{Trace}$	$[\![\mathtt{Trace}]\!]$	$=$ the trace
$\mid \neg\ \mathtt{Expr}$	$[\![\neg \mathtt{Expr}]\!]$	$= \overline{[\![\mathtt{Expr}]\!]}$
$\mid \mathtt{Expr} \wedge \mathtt{Expr}$	$[\![\mathtt{Expr}_1 \wedge \mathtt{Expr}_2]\!]$	$= [\![\mathtt{Expr}_1]\!] \cap [\![\mathtt{Expr}_2]\!]$
$\mid \mathtt{Expr} \vee \mathtt{Expr}$	$[\![\mathtt{Expr}_1 \vee \mathtt{Expr}_2]\!]$	$= \overline{[\![\mathtt{Expr}_1]\!]} \cup [\![\mathtt{Expr}_2]\!]$
$\mid \mathtt{Expr} \implies \mathtt{Expr}$	$[\![\mathtt{Expr}_1 \implies \mathtt{Expr}_2]\!]$	$= \overline{[\![\mathtt{Expr}_1]\!]} \cup [\![\mathtt{Expr}_2]\!]$
$\mid \Box\ \mathtt{Expr}$	$[\![\Box \mathtt{Expr}]\!]$	$= \Sigma^*.[\![\mathtt{Expr}]\!]$

Fig. 7. A temporal logic built out of traces: syntax (left) and semantics (right).

This is also very appealing when the underlying semantics is defined using other formalisms than traces, e.g., partial languages or a notation like TTCN, for the notion of a complement is even less trivial there.

Comparative concurrency semantics Section 2.3 very briefly sketches the relationships between the traditional notions of concurrency semantics and those defined by UML. As we have seen, the current definitions as proposed in the standard are somewhat deficient. So, it might be interesting to further study this relationship, and try to adopt notions and tools from that area. As a starter, consider the interaction between complete/partial traces and some interpretations of assert.

Non-classical logics In section 4.3, I attempted a logic interpretation of assert which failed for classical logic. But what about other logics like linear and intuitionistic logics? In intuitionistic logic, I have

$$(1) \qquad \neg(\neg\neg\alpha \implies \alpha) \qquad \text{and} \qquad (2) \qquad \alpha \implies \neg\neg\alpha.$$

When translating the first axiom into $\mathsf{neg}(\mathsf{assert}(\mathsf{neg}(\mathsf{neg}(P)), P))$ and using interpretations A.2 and N.2, this makes sense, in a way, since $[\![\mathsf{assert}(P, P)]\!]_{A.2}$ is contradictory. The other axiom is not true, however.

Another element of classical logics missing in intuitionistic logic is the principle of "*tertium non datur*", which when applied to UML 2.0-interactions would require the set of contingent traces to be empty *always*. This certainly removes a number of problems.

Tools One of the benefits of formal semantics is the possibility of building automated tools using the semantics for validation, verification and visualization purposes. I have already implemented a prototype of such a tool, formalizing the semantics and some equivalence notions. The operators neg and assert, however, need clarification before they can be added to such a tool. Or rather, in the meantime, experiments with different interpretations of these operators might shed light on their meaning and usefulness.

6.1 Related Work

There are many variants of Message Sequence Charts (MSCs), such as the 1996 and 2000 versions of the standard proper, UML 1.4 collaborations [14], Life Sequence Charts [3] and Extended Event Traces [10]. Of course, there is a body of work concerning the formal semantics of these, including [1, 5, 8, 9, 11, 12, 20, 21, 25]. See [10] and [18] for exhaustive surveys.

Acknowledgments Thanks go to Stephan Merz and Alexander Knapp for discussions and proof reading. Also, I'd like to thank the four anonymous referees for their helpful remarks.

References

1. Rajesh Alur, Gerard J. Holzmann, and Doron Peled. An Analyzer for Message Sequence Charts. *Software–Concepts and Tools*, (17):70–77, 1996.
2. S.D. Brookes, Charles A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential programs. *Journal of the ACM*, 31(3):560–599, 1984.
3. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. In *Proc. 3rd Intl. Conf. Formal Methods for Open Object-based Distributed Systems*. IFIP, 1999.
4. Rob J. H. van Glabbeek. *Comparative Concurrency Semantics and Refinement of Actions*. Number 109 in CWI Tracts. CWI, Amsterdam, 1996. 2nd Ed.

5. Peter Graubmann, Ekkart Rudolph, and Jens Grabowski. Towards a Petri-net based semantics definition for Message Sequence Charts. pages 179–190, 1993.
6. ITU-T. *Recommenration Z.120. Message Sequence Charts (MSC)*. International Telecommunication Union, 1996.
7. ITU-T. *Recommenration Z.120. Message Sequence Charts (MSC)*. International Telecommunication Union, 2000.
8. Alexander Knapp. A Formal Semantics for UML Interactions. In Robert France and Bernhard Rumpe, editors, *Proc. 2^{nd} Intl. Conf. on the Unified Modeling Language (≪UML≫ 1999)*., number 1723 in LNCS, pages 116–130. Springer Verlag, 1999.
9. Piotr Kosiuczenko and Martin Wirsing. Towards an Integration of Message Sequence Charts and Timed Maude. In Murat M. Tanik, Jiro Tanaka, Kiyoshi Itoh, Michael Goedicke, Wilhelm Rossak, Hartmut Ehrig, and Franz Kurfeß, editors, *Proc. 3^{rd} Intl. Conf. Integrated Design and Process Technology (IDPT'98)*, Berlin, July 5-9, 1998.
10. Ingolf Krüger. *Distributed System Design with Message Sequence Charts*. PhD thesis, TU München, 2000.
11. Stefan Leue. *Methods and Semantics forTelecommunications Systems Engineering*. PhD thesis, Universität Bern, 1995.
12. Sjouke Mauw and Michel A. Reniers. An Algebraic Semantics of Basic Message Sequence Charts. *The Computer Journal*, 37(4), 1994.
13. Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.
14. OMG. Unified Modeling Language Specification (draft, version 1.4), February 2001.
15. OMG. Unified Modeling Language: Superstructure version 2.0 (Final adopted Specification of August, 2*nd*, 2003), August 2003.
16. OMG Unified Modeling Language Specification (version 1.3), June 1998. Available at uml.shl.com.
17. Wolfgang Reisig. *Petri-Nets: an Introduction*. Springer Verlag, 1985.
18. Michel Adriaan Reniers. *Message Sequence Charts - Syntax and Semantics*. PhD thesis, TU Eindhoven, 1999.
19. Bran Selic, Stuart Kent, and Andy Evans, editors. *Proc. 3^{rd} Intl. Conf. ≪UML≫ 2000—Advancing the Standard*, number 1939 in LNCS. Springer Verlag, October 2000.
20. Harald Störrle. A Petri-Net Semantics for Sequence Diagrams. In Katharina Spies and Bernhard Schätz, editors, *9. GI/ITG Fachgespräch Formale Beschreibungstechniken für verteilte Systeme (FBT'99)*, June 1999.
21. Harald Störrle. *Models of Software Architecture. Design and Analysis with UML and Petri-nets*. PhD thesis, LMU München, Inst. f. Informatik, December 2000. ISBN 3-8311-1330-0.
22. Harald Störrle. Interactions in UML 2. Technical Report 0304, LMU München, Institut für Informatik, August 2003.
23. Harald Störrle. Semantics of Interactions in UML 2.0. In N.N., editor, *Proc. Intl. Symp. Visual Languages and Formal Methods*. IEEE CS Press, October 2003. accepted for publication.
24. Martin Wirsing. Algebraic specification. In Jan van Leeuwen, editor, *Handbook of theoretical Computer Science*, pages 675–788. Elsevier, 1990.
25. Martin Wirsing and Alexander Knapp. A formal approach to object-oriented software engineering. *J. Theoretical Computer Science*, 285:519–560, 2002.

Towards a UML Profile for Security Assessment

Siv Hilde Houmb and Kine Kvernstad Hansen

Norwegian University of Science and Technology, Department of Computer and Information Science
Sem Slands vei 7-9, NO-7491 Trondheim, Norway
(siv.hilde.houmb, kine.kvernstad.hansen)@idi.ntnu.no

Abstract. Security assessment is a multidisciplinary task involving both technical and non-technical stakeholders. One major challenge when performing such assessments is to establish a common understanding of threats, vulnerabilities and security risks among the different groups of stakeholders participating in the assessment. In order to enhance communication, we need easily understandable representation of the different concepts. In this paper we presents SecurityAssessmentUML, a UML profile for model-based security assessments. The main objective of SecurityAssessmentUML is to support documentation of output from risk identification and risk analysis in a security assessment. In particular, the profile supports specification of threat scenarios demonstrating how attacks may occur, as well as fault tree inspired activity diagrams for analysing the frequency of these attacks.
Keywords: UML Profiles, Security Extension to UML, Security Assessment

1 Introduction

Modern society heavily relies on networked information systems. The risks associated with these systems may threaten the economical and physical well-being of people and organisations. Unavailability of a Telemedicine platform may, for instance, result in loss of life, while an organisation conducting business electronically can be subject to major economic loss as a result of a successful denial-of-service attack.

Risk assessments and management techniques have been widely used within the safety domain since World War II [13]. This is not the case within the security domain where there has been more focus on technical solutions rather than providing the arguments for their necessity. There are therefore no largely used standard or practise for using risk assessment to identify and assess security incidents. However, the EU IST-project CORAS [5] has developed a framework for model-based risk assessment of security critical systems based on the Australian/New Zealand Standard for risk management AS/NZS 4360:1999 [2], the security standards ISO/IEC 17799 [9] and ISO/IEC 13355 [8] and the safety standard IEC 61508 [7]. Further, CORAS has developed a UML profile for documenting results from risk assessment as part of their framework [6]. The UML-profile covers part of the activities in the risk management process, but does not provide detailed guidelines and support for documenting threat scenarios.

1.1 Related Work

The OCTAVE framework [1], developed by the NSS Program at SEI, provides guidelines enabling organisations to develop appropriate protection strategies based on identified risks to critical information assets. Others relevant frameworks are CRAMM [3], ATAM [4] and RSDS [12]. However, these methods and frameworks does not provide support for model-based risk assessment (MBRA). Model-based risk assessment make use of models both to describe the system and as input to risk assessment. This strategy is supported by the CORAS framework [16]. CORAS provide a integrated system development and risk management process based on AS/NZS 4360:1999, RM-ODP and RUP. It emphasises reuse and communication in a system development and risk management setting.

There exist some UML profiles targeting security. UMLsec [10] [11], a security extension to UML, provide support for encapsulating security requirements as non-functional requirements in system development. Another example is secureUML [14], which defines a vocabulary to express different aspects of access control, like roles, role permissions and user-role assignments. Within the MBRA-domain CORAS [6] has developed a UML profile for model-based risk assessment based on the integrated process of CORAS. In [15] an extension to UML use case diagrams for documenting unintended behaviour, Misuse Cases, is presented.

This work is inspired by the idea from CRAMM by focusing on revealing possible vulnerabilities in a system, rather than focusing on identifying threats. Further, we make use of the asset-oriented process of CORAS and reuse the steretypes for assets and asset values provided by their UML profile. Both the CORAS UML profile and SecurityAssessmentUML are inspired and reuse notation and ideas from Misuse Cases.

2 SecurityAssessmentUML

SecurityAssessmentUML is a UML profile designed to document result from security assessment. The profile is specified as an extension to UML1.4 and provides support for UML sequence and activity diagrams for risk identification and UML activity diagrams for risk analysis. The aim of the profile is to support non-technical stakeholders in a security assessment.

When deciding on which UML diagrams to support, only the five behavioural diagrams of UML were considered. The reason for this is that security assessment focuses on the behaviour aspects of a system and not on its static structures. Collaboration diagrams are not included as they present similar information as sequence diagrams. Use case diagrams are not suitable in this context as they solely represent functionality of the system and are mainly used when elaborating on functional requirements. Statechart diagrams are already supported by the CORAS UML profile and thus not included in this version of the profile.

2.1 Risk Identification

Figure 1 describes the mapping of concept of risk identification, while Figure 2 illustrated an example of documentation of an attack using the extensions. Figure 2 documents a fabrication attack where an outsider sends a virus to a mail-server through a mail-gateway. In this diagram, events representing threats or unwanted incidents are distinguished from normal events. Further, the misuser is distinguished from a normal actor.

Concept	Mapping to existing UML	UML extension	Description
Vulnerability	Object	<< vulnerability>>  *{vulnerability}	This stereotype is used to distinguish objects containing vulnerabilities from normal objects. The tag vulnerability is used to specify the vulnerability. If more than one vulnerability is present, several vulnerability tags should be used (and numbered).
Unwanted incident	Message	<<unwanted incident>> 	This stereotype is used to distinguish unwanted incidents from normal messages.
Misuser	Actor instance	<<misuser>>  *{type} *{intension}	This stereotype is used to distinguish misusers from normal users (actors). The tag type refers to whether the misuser is insider or outsider. The tag intention refers to whether the misuse is intended or unintended.
Threat	Message	<<threat>> 	This stereotype is used to distinguish threats from normal and abnormal messages.
Asset	Object	<<asset>> 	Stereotype from CORAS UML profile.

Fig. 1. Mapping of concepts to UML sequence diagrams for risk identification

Figure 3 depicts the mapping of concepts of risk identification for activity diagrams, while Figure 4 provides an example of documenting an attack using these extensions. The examples documents how a threat may exploit a vulnerability in the mail-gateway (no virus-wall) leading to an unwanted incident.

2.2 Risk Analysis

For risk analysis SecurityAssessmentUML support UML activity diagrams. The activity diagrams from risk identification is extended with logical gates in order to compute calculations based on the diagrams. For this reason we extend activity diagrams with notation from the safety analysis method Fault Tree Analysis (FTA) [13]. FTA are widely used within the safety domain and there exist tools that perform automatic calculation based on the Boolean logic in the fault trees.

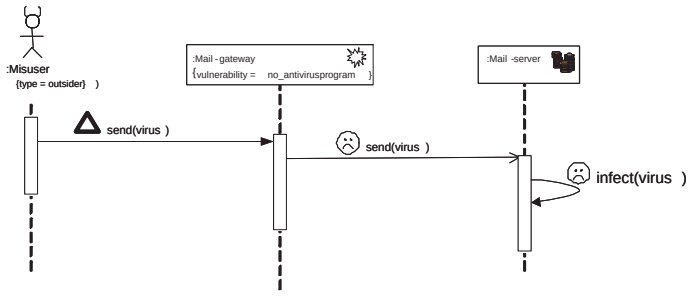


Fig. 2. Example attack documented using extensions to sequence diagrams

Concept	Mapping to existing UML	UML extension	Description
Vulnerability	Guard	<< vulnerability>> *{asset} *{vulnerability}	This stereotype is used to mark that a guard connected to a branch is a test for whether or not a vulnerability exists. The tag asset is used to represent the entity containing the vulnerability. The tag vulnerability can be used to further specify the vulnerability.
Unwanted incident	Activity state	<<unwanted incident>>	This stereotype is used to distinguish activity states involving unwanted incidents from normal activity states
Threat	Activity state	<<threat>>	This stereotype is used to distinguish activity states introducing threats to the system from normal activity states.

Fig. 3. Mapping of concepts to UML activity diagrams for risk identification

FTA is primarily a means for analysing causes of threats, not identifying threats [13]. An undesired system state is specified and the method works backwards to determine its possible causes. The result of an FTA is a hierarchy of undesired events using Boolean logic to depict the logical interrelationships of individual faults. Each level in the tree consist of the combination of undesired events that needs to be present in order to cause the event in the level above. Figure 1 presents the stereotypes for FTA activity diagrams (activity diagrams using stereotypes for FTA notation). The remaining concepts in FTA is subject for inclusion in further work. We will also provide tool-support for direct calculation based on the FTA activity diagram by providing transformation from FTA activity diagrams to fault tree representation.

The tag frequency/likelihood are attached to all activity states, while consequence value and risk level only relates to the unwanted incident (the top event). Rules for estimating risk level using either qualitative or quantitative values for

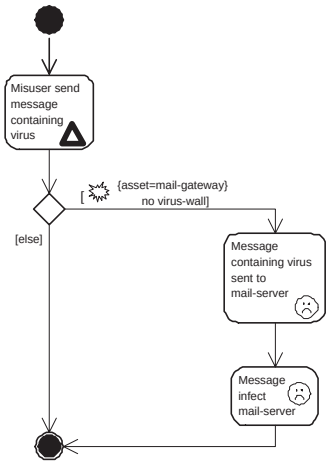


Fig. 4. Example attack documented using extension to activity diagrams

Concept	Mapping to existing UML	UML extension	Description
AND-gate	Fork and join	AND	This stereotype is used to substitute use of fork and join with an AND symbol in order to simplify the notation.
OR-gate	Branch	OR	This stereotype is used to substitute use of branch with an OR symbol in order to simplify the notation.
Transfer down	Activity state	▽	This stereotype is used to distinguish the top event in a decomposed tree from a "normal" top event.
Transfer up	Activity state	△	This stereotype is used to distinguish an event that is further decomposed from normal activity states.
Basic event	Activity state	○	This stereotype is used to distinguish basic events from normal activity states.

Fig. 5. Mapping of concepts to UML activity diagrams for use of fault tree notation

likelihood/frequency and consequence are not included in this paper. For information on this issue the reader is referred to litteratur on FTA [13] (which provides both qualitative and quantitative calculation rules) and CORAS (see publications at [5]). Figure 6 presents an example of a FTA activity diagram. The example documents a hierarchy of events leading to the unwanted incident "Server infected". The unwanted incident will occur if there are accessible open ports in the FireWall and no other security mechanism for detecting viruses are included.

Table 1. Mapping of concepts to UML activity diagrams for frequency, consequence and risk level specification

Concept	Mapping to existing UML	UML extension	Description
Frequency/likelihood	Tagged value on activity state	{frequency = value} {likelihood = value}	This tagged value is used to specify the frequency/likelihood of an event. Frequency is used when quantitative values are available. In other cases likelihood is used.
Consequence	Tagged value on activity state	{consequence = value}	This tagged value is used to specify the consequence value of an unwanted incident.
Risk level	Tagged value on activity state	{risk level = value}	This tagged value is used to specify the risk level estimate of an unwanted incident.

3 Conclusion and Further Work

In this paper we have presented SecurityAssessmentUML, a UML profile supporting documentation of results from security assessment. The profile aims at specifying concrete threat scenarios demonstrating the relationship between undesired events, their frequencies, consequences and impacts.

Based on the extensions provided, three types of threat scenario diagrams are supported. For risk identification the profile provides extensions to sequence and activity diagrams. For risk analysis the profile supports an FTA extension to activity diagrams, including tagged values to document the values of consequence, frequency and risk level. All stereotypes are defined both by text strings in brackets and by icons. Ideally, diagrams should be understandable for non-technical stakeholders and at the same time preserve the semantic of the language. However, this is not the case for the fault-tree inspired activity diagram, which uses a hybrid between FTA and activity diagram notation.

Due to lack of extensive evaluation SecurityAssessmentUML is in a draft version. The preliminary evaluation, using a small field study and expert judgement, of the profile has identified some main areas for improvement. The most significant limitation is the lack of support for specifying the effects of unwanted incidents on asset values. Another important aspect is to incorporate the support for asset identification and asset valuing as specified in the CORAS framework.

The preliminary evaluation identified four main areas for further work:

- **Support for documenting effect on asset value**
  SecurityAssessmentUML does not specify how to document the effects of unwanted incidents on asset values. Since this is essential in security assessments it should be included in the next version of the profile.
- **Extension to support documentation of threats categorised as system failure**

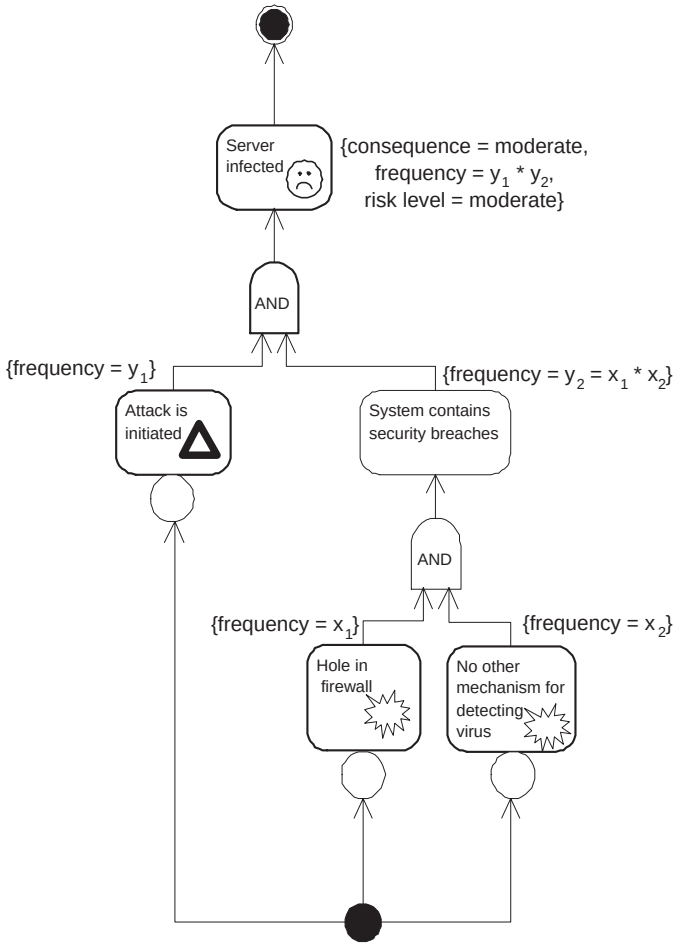


Fig. 6. Example of FTA notation extensions to activity diagrams

SecurityAssessmentUML does only supports documentation of threats categorised as attacks. Support for specifying system failures should be subject for further work.

– **Extension to support all sub-processes in the risk management process**
SecurityAssessmentUML supports documentation of output from two of the sub-processes in the security assessment process. The profile should be extended to allow documentation of output from all sub-processes. For information on the different sub-processes see [6].

– **Extensive evaluation**
In order to draw valid conclusions about the usefulness of SecurityAssess-

mentUML, whether it further communication and interaction among stakeholders with no prior knowledge of UML and whether all necessary concepts are covered, extensive evaluations must be performed. The evaluation should include trials using more detailed and varied assessment results, as well as extending the group of experts used in the expert judgment.

Acknowledgments: The profile presented in this paper is based on the CORAS UML Profile for Model-based risk assessment and other results from the IST-project CORAS. Furthermore, we would like to thank the participants in the judgement study along with Kai Hansen, ABB Research, Norway, for valuable input on the diagrams, Karine Sørby and Ørjan Lillevik for useful discussions on the profile.

References

1. C. J Alberts, S. G. Behrens, R. D. Pethia, and W. R. Wilson. Operationally critical threat, asset, and vulnerability evaluation (octave) framework, version 1.0. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, June 1999.
2. Australian/New Zealand Standard AS/NZS 4360:1999: Risk Management. Strathfield: Standards Australia.
3. B. Barber and J. Davey. The use of the ccta risk analysis and management methodology CRAMM. In *Proc. MEDINFO92, North Holland*, pages 1589–1593, 1992.
4. P. Clements, R. Kazman, and M. Klein. *Evaluating software architectures: Methods and case studies*. Addison-Wesley, 2002. ISBN: 020170482X.
5. CORAS IST-2000-25031 Web Site. http://www.nr.no/coras. 24 February 2003.
6. S. H. Houmb, F. den Braber, M. S. Lund, and K. Stølen. Towards a UML Profile for Model-Based Risk Assessment. In *Critical systems development with UML - Proceedings of the UML'02 workshop*, pages 79–91, September 2002.
7. IEC 61508: 2000 Functional Safety of Electrical/Electronic/Programmable Electronic (E/E/PE) Safety-Related Systems.
8. ISO/IEC 13335: Information Technology - Guidelines for the management of IT Security. http://www.iso.ch.
9. ISO/IEC 17799: 2000 Information technology - Code of practise for information security management.
10. J. Jurjens. UMLsec: Extending UML for Secure Systems Development. Software & Systems Engineering, Dep. of Informatics, Munich University of Technology.
11. Jan Jürjens. Towards development of secure systems using UMLsec. In Heinrich Hussmann, editor, *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, Proceedings*, volume 2029 of *LNCS*, pages 187–200. Springer, 2001.
12. K. Lano, K. Androutsopoulos, and D. Clark. Structuring and design of reactive systems using RSDS and B. proc. fase 2000. In *LNCS*, volume 1783, pages 97–111, 2000.
13. N. G. Leveson. *Safeware: System safety and computers*. Addison-Wesley, 1995. ISBN: 0-201-11972-2.

14. Torsten Lodderstedt, David A. Basin, and Jürgen Doser. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In Jean-Marc Jézéquel, Heinrich Hussmann, and Stephen Cook, editors, *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, volume 2460 of *LNCS*, pages 426–441. Springer, 2002.
15. G. Sindre and A.L. Opdahl. Eliciting security requirements by misuse cases. In *TOOLS-PACIFIC 2000*, pages 120–131. IEEE Computer Society Press Sydney, Australia, 2000. Los Alamitos.
16. 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 F. Barbier, editor, *Business Component-Based Software Engineering*, pages 189–207. Kluwer, 2003.

Predicting Software Performance Based on UML Models during the Unified Software Development Process

Zhongfu Xu[1], Johannes Lüthi[2], and Axel Lehmann[1]

[1] Institute of Technical Informatics
Department of Informatics
University of the Federal Armed Forces Munich
85577 Neubiberg, Germany
Tel: +49(89)6004-2607/2648
Fax: +49(89)6004-2268
Email: xu, lehmann@informatik.unibw-muenchen.de

[2] Business Informatics
FHS KufsteinTirol
6330 Kufstein, Austria
Tel: +43 (5372) 71819-172
Fax: +43(5372) 71819-104
Email: Johannes.luethi@fh-kufstein.ac.at

Abstract: In order to develop high-quality software more efficiently, the Unified Software Development Process (USDP) can be enhanced by predicting software performance concurrently during design. We present a framework for predicting software performance based on UML (Unified Modeling Language) models during USDP. By mapping software execution onto physical computer resources, we define a metamodel to prescribe information required for performance prediction. USDP functional models are extended with performance-related information before building executable and deployable software products. Based on these information, performance models can be constructed and analyzed with respect to software performance properties. Thus, expensive test and tuning costs can be avoided. The proposed approach highlights the integration of the performance prediction process as part of the software analysis and design process. It also emphasizes impacts of implementation details on performance prediction, and aims at automated performance prediction.

Keywords: Unified Software Development Process (USDP), Unified Modeling Language (UML), software performance prediction, extending functional models, performance prediction metamodel

1 Introduction

1.1 Background and Motivation

Software performance analyses reflect the dynamic behavior of programs during execution by measures of the the external effectiveness (such as execution time/response time) and of the internal efficiency (such as resource usage). Performance analysis should be integrated into the software development process in order to develop high-quality software more efficiently. Before implementation, quantitative methods should be used to evaluate whether the selected software architecture and design solutions will satisfy performance requirements. Implementation should be based on validated software design, avoiding expensive post-tuning costs.

The Unified Software Development Process (USDP) is a use case-driven, architecture-centric, iterative and incremental process for developing and deploying software systems [3]. USDP was developed with the intention of being commonly followed by the software development community. Within USDP the Unified Modeling Language (UML) [1, 6] is used as the language for constructing and documenting software models. Each so-called USDP development cycle is composed of four phases. Each phase is further broken down into iterations. An iteration consists of five core workflows. In the requirements workflow, the functional requirements of the software system are captured as use cases. In the analysis workflow, the use cases are refined in order to improve understanding of the functional requirements. In the design workflow, the design- and deployment models are built to provide the input to the subsequent implementation and testing activities. Finally, in the implementation workflow, the executable system is implemented in terms of components. The primary purpose of the test workflow is to verify that the software system correctly provides the functionality described by the use cases. Software builds with defects have to go through earlier workflows such as analysis, design and implementation again. These core workflows are guided by the software system architecture design that addresses the most significant static and dynamic aspects of the software system (see the book [3] for more details about USDP).

The software development project in the Unified Software Development Process is driven by the functional requirements in that the workflows are initiated from the use cases [3]. Performance requirements specify the conditions under which the functional requirements should be fulfilled. No proposal, however, is made in USDP to use quantitative analysis methods to predict and control software performance properties.

1.2 Performance Modeling within the USDP

Most performance problems are caused by fundamental architecture or design problems [5, 11]. When these problems are detected in the test workflow or observed by the users of the software system, fixing them then necessitates substantial revision to the software system and causes high costs. In extreme cases, the project must be canceled since performance objectives can not be met by tuning.

In order to avoid the expensive post-tuning efforts, the software system architecture and software designs should be justified before they are used for implementing, testing, and deploying executable software products. To achieve this goal, the Unified Software Development Process can be enhanced with predicting software performance. Based on the current software system architecture and software designs, performance prediction models can be constructed and analyzed to obtain predictions about software performance. If the prediction results indicate performance bottlenecks, the current software system architecture and software designs are revised, and the performance prediction process is redone. Otherwise, the software development evolves to the implementation and testing.

In USDP, software models are constructed using UML in a functionality-oriented way, and information lacks that is necessary for deriving and specifying variables of the performance prediction models to be built. The functional UML models should be extended to include the required performance-related information. In order to make the design-based performance prediction more applicable and acceptable for software engineers, among others, we consider the following requirements to be important with reference to making performance extensions on functional UML models:

- Performance extensions should be made by using the same modeling language (i.e. UML) and tool as used for constructing functional models. The main advantage is time-saving for the development team (Only UML models are built, evolved, and maintained in one engineering environment).

- Only built-in UML extensibility mechanisms should be employed for special needs in a standardized way without violating the understandability of the resulting UML models.

- Corresponding to the level of abstraction needed by performance prediction, a mechanism should be provided to define: which information in the functional models can be directly used or rewritten, which information should be supplemented, and how these information should be represented in UML. The definition of performance extensions should be described in non-performance-specific terminologies. It should not be mandatory for the software developer to have the knowledge about the performance modeling formalisms and methods to be used.

- With support of the UML tools in use, performance extensions should provide a basis for automated generation of performance prediction models to obtain rapid and informative performance feedbacks. This is also important for achieving consistency between the functional models and the generated performance prediction models in that the updates made on functional models can be automatically imported into the performance prediction models.

In this work, aimed at obtaining estimations about the external effectiveness and internal efficiency of software before implementation, we present a framework (outlined in Figure 1) for extending the functionality-oriented UML-based software models in USDP with information required for generating performance prediction models.

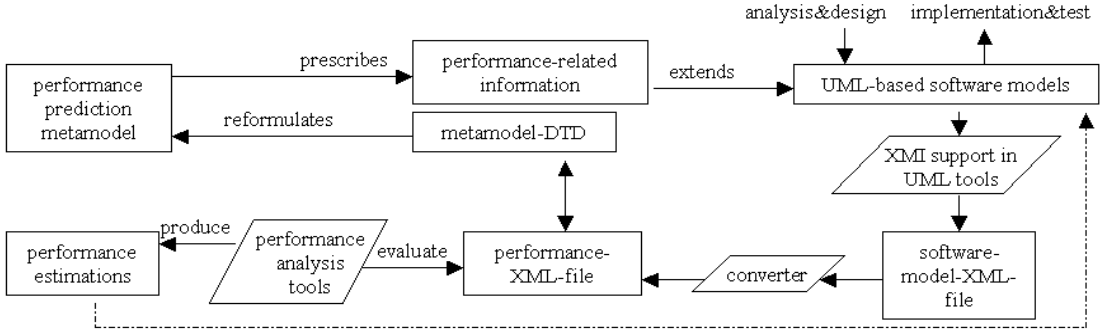


Figure 1: Outline of the framework for extending USDP functional models to predict software performance

In the following we provide a concise description of the presented framework, and conclude the paper by comparing the proposed approach to the related work and discussing the possible future work.

2 Extending USDP Functional Models for Predicting Software Performance

2.1 Performance Prediction Metamodel

We define a metamodel (shown in the UML class diagram in Figure 2) to prescribe information required for software performance prediction. Which computer devices will participate in a use case execution, parameters and quantitative behavior of the participating computer devices, together with the pattern in which the users require a use case, constitute the most important performance-related information. Before implementation, this information is not visible in and should be supplemented to the functional models.

The users impose requests on the software system when they require a use case. Users of a use case are categorized so that the intensity and temporal property of the requests generated by each user category can be described uniquely with a request pattern, which can be open or closed. An open request pattern describes the requests generated by users arriving at the system from the outside. For an open request pattern the request inter-arrival time is specified. A closed request pattern means that the requests are generated by a fixed number of users who continually interact with the software system. The users enter requests, examine responses, and submit the next request after a certain think time. The time duration, during which the requests are generated, is specified for each request pattern (open or closed).

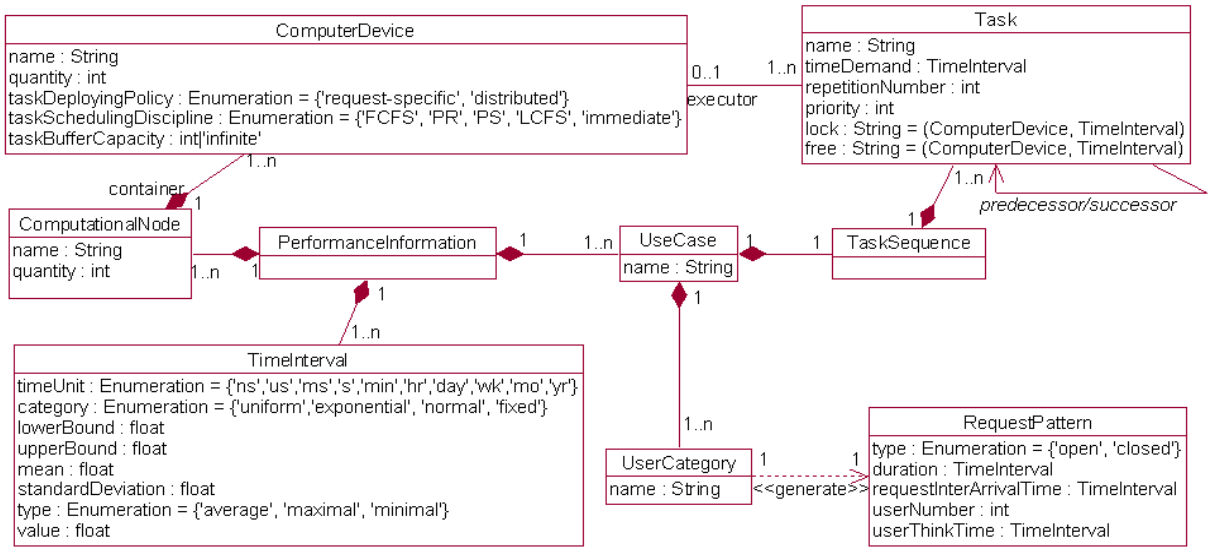


Figure 2: Performance prediction metamodel

In USDP design workflows, the dynamic aspects of a use case realization are represented by a sequence of operations performed by design objects (i.e. instances of design classes). In USDP implementation workflows these operations are realized by suitable algorithms and data structures in the file components implementing the corresponding design classes. The operation functionality will be reified as a set of activities that are performed at runtime by physical computer devices contained in computational nodes. For the purpose of predicting software performance before implementation, these activities are abstracted as a set of atomic tasks, each of which is executed solely by one computer device. An atomic task is not decomposable in that its work is not interruptible. A computer device is one of the physical hardware resources, such as CPU, I/O device, and network device.

The processing of a user request (i.e., execution of a use case) is then represented as a sequence of atomic tasks. Parameters are defined for each atomic task to specify the execution time needed by the host computer device, the number of successive repetitions of the task execution, and the priority of the task. It is possible that a computer device is exclusively used by a user request for a period of time to execute its tasks. The exclusive usage begins when a task of this user request locks the computer device, and terminates when a task of this user request set the computer device free for the other user requests. Two parameters, lock and free, are defined for task related to the exclusive usage of computer devices. They describe which computer device is exclusively

used and the duration of task execution before the computer device is locked or set free by this task. The atomic tasks are related to each other in a predecessor/successor relationship, meaning that only after the execution of a predecessor can the execution of the successor(s) start.

Computer devices are contained in computational nodes. Functionally identical computer devices have a unique name. If the quantity is greater than one, the policy for deploying atomic tasks among these identical computer devices is characterized by the parameter *taskDeployingPolicy*, which may be request-specific or distributed. A request-specific policy indicates that when these identical computer devices are required by a user request for the first time, the user request is processed by one of the multiple computer devices according to the task scheduling discipline, and all the tasks for this user request are routed to the chosen one for executions (when required). A distributed task deploying policy means that tasks for the same user request can be distributed among the identical computer devices. The deployment of tasks among the multiple computer devices is determined by the task scheduling discipline of the computer devices.

The policy in which atomic tasks are executed by a computer device may be FCFS (First-Come-First-Served), or PR (Preempt-Resume), or PS (Processor-Sharing), or LCFS (Last-Come-First-Served). These policies imply that the computer device has a task buffer. An immediate task scheduling discipline indicates that the tasks are executed immediately upon their arrival without waiting, and a task buffer is unnecessary.

Time intervals are defined for specifying parameters of several metamodel elements. A time interval may be specified by a probability distribution function, or be a fixed value whose type may be average, or maximal, or minimal. It is straightforward to extend the definition of time intervals with more probability distribution functions.

2.2 Performance Extensions on USDP Functional Models

1. Extension of the Use Case Model

In the USDP requirements workflows, UML use case diagrams are used to represent the use case model as a whole by showing the use cases, system users, and their relationships. The use case diagram is extended by attaching user request descriptions (including type, duration, intensity and temporal property) to the associations between user categories (represented by actors) and the required use cases.

Figure 3 shows the use case diagram of an ATM system used by bank customers to get account information, to withdraw from and deposit to accounts, and to transfer money between accounts. These functional requirements are captured as four use cases. Two categories of users require *Get_AccountInfo* use case in the daytime and nighttime respectively. The request patterns for two user categories are open and have a maximal duration of 12 hours. Request inter-arrival times are specified by an exponential distribution function with 30 seconds and 600 seconds as mean values. As shown in Figure 2, fixed time intervals are specified as (*"fixed", type, value, timeUnit*), and probability distribution functions as (*probability distribution function name, parameter(s) and value(s), timeUnit*).

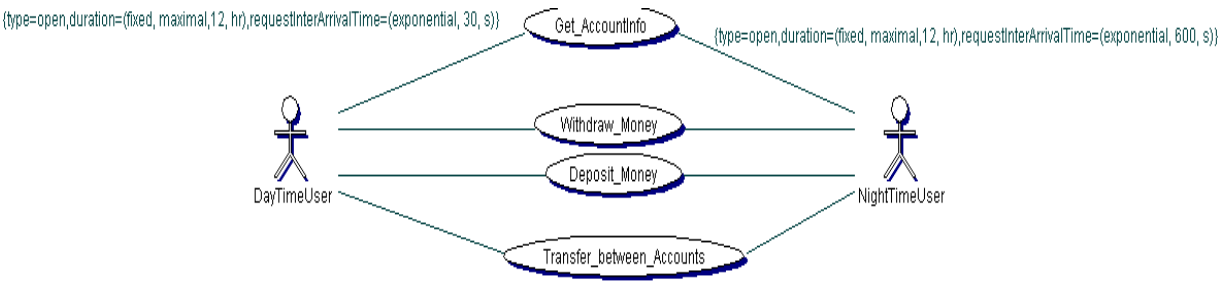


Figure 3: Extended use case diagram for an ATM system

2. Extension of the Design Model

In the design model dynamic realization of a use case is represented by the interactions between design objects in one or more UML sequence diagrams. We emphasize the use of message labels [6] to specify the predecessor/successor relationship of operation invocations caused by message dispatches among design objects, and conditional and iterative operation invocations. The design model is extended by constructing a UML activity diagram for each use case under consideration, as follows. The constructed activity diagram presents the collaboration of computer devices that participate in the use case realization.

First, operations performed by design objects in the sequence diagrams are mapped to decomposable activity states in the activity diagram to be constructed. The time ordering of operations is retained by properly connecting the activity states with transitions. Special building blocks in the activity diagram notation are used to represent concurrent, conditional, and iterative execution of operations: (a) two synchronization bars enclose parallel operation branches. (b) Two decision icons indicate the starting and end points of conditional operation branches. Execution conditions are evaluated to quantify probabilities with which the conditional operation branches are executed. They are specified to the transitions pointing to the first operation in each conditional branch. (c) Two dummy action states represent a loop within which an operation or a sequence of operations is executed iteratively. Looping number is specified for the dummy action states. These special building block pairs are named and numerated to make them uniquely distinguishable.

Second, the functionality of each operation is decomposed into atomic tasks. The decomposition process begins with the operations in the deepest nesting level (i.e. operations without nested operations). The corresponding activity state is substituted by a collection of indecomposable action states (representing atomic tasks within this operation) interconnected by transitions. Pairs of synchronization bars, decision icons, and dummy action states are also used to represent concurrent, conditional, and iterative execution of atomic tasks.

Finally, runtime properties are specified for each atomic task by assigning the parameters of *Task* in the metamodel. The atomic tasks are associated with computer devices by specifying the name of the host computer device for each atomic task. This way, the atomic tasks represent the quantitative behavior of the computer devices that participate in the execution of the use case, and the constructed activity diagram describes the collaboration of the participating computer devices.

Figure 4 shows a fragment of the UML sequence diagram describing the realization of *Get_AccountInfo* use case of the ATM system. The ATM system will be implemented in the client/server architecture. Objects *user* and *client* on ATM side communicate through Internet with remote objects *accountManagerImpl* and *accountImpl* on bank server side. Java is used in system implementation. Java RMI (Remote Method Invocation) is used to achieve transparent object distribution. As shown in the sequence diagram, *readInCardID* of *client* is the first invoked operation followed by *enterPSW* of *user* and *readInPSW* of *client*. (PSW: password)

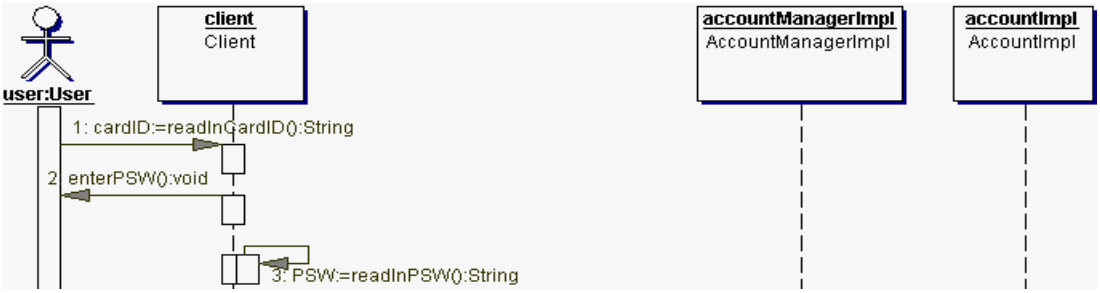


Figure 4: A fragment of the sequence diagram for *Get_AccountInfo*

Figure 5 shows a fragment of the activity diagram constructed on the basis of the sequence diagram for *Get_AccountInfo* use case in Figure 3. Operations *enterPSW* and *readInPSW* are mapped to two atomic tasks: *enterPassword* and *readInPassword*. Operation *readInCardID* is decomposed into two atomic tasks *getCardID*s executed by computer devices *cardReader* and *ATMCPU,* respectively. *CardReader* is locked by *getCardID* before its execution since a user interacts with ATM without interruption.

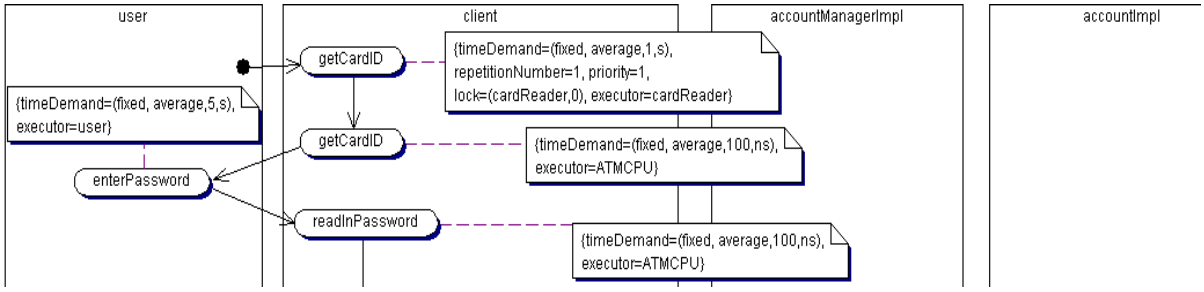


Figure 5: A fragment of the activity diagram constructed for *Get_AccountInfo*

3. Extension of the Deployment Model

The deployment diagram in the deployment model shows the configuration of runtime computational nodes and the interconnecting network. It also shows the distribution of design objects among computational node instances. It is extended to include the computer devices that are identified during decomposing operations of design objects into atomic tasks. The computer devices instantiate *ComputerDevice* in the metamodel. They are contained in the computational node instances and used by design objects. Name and quantity are specified for each computer device and computational node instance. Properties are specified for each computer device by assigning parameters of *ComputerDevice* in the metamodel. Stereotypes [1, 6] based on *ComputerDevice* can be defined and instantiated to represent special computer devices, such as CPUs.

As shown in Figure 6, 400 ATMs communicate with a bank server through Internet. Each ATM instance contains three computer devices: *ATMCPU*, *cardReader*, and *printer*. Based on *ComputerDevice*, a stereotype named *processor* is defined for representing CPUs in ATMs and bank server. It has two additional parameters: MIPS (Million machine Instructions Per Second) and ratio high-level to machine instructions.

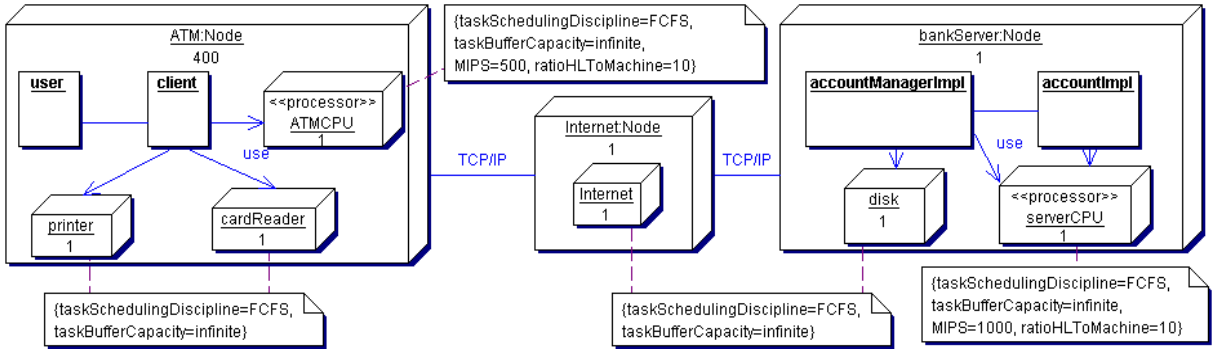


Figure 6: Extended deployment diagram for *Get_AccountInfo*

2.3 Extraction and Reformulation of Performance-Related Information

In this work, Together Control Center 6.1 [2] is used to construct and extend the functional UML models. The Unisys Extension in the Together Control Center automatically exports the resulted UML models into an XML (eXtensible Markup Language) [12] file (called software-model-XML-file). The structure of this software-model-XML-file is defined by a DTD (Document Type Definition) file whose name is UMLX13-11.dtd. This DTD was generated from the UML 1.3 physical metamodel [6] by following the rules described in the XML-based Metadata Interchange (XMI) specification 1.1 [7].

The software-model-XML-file contains performance-related information, together with other information being insignificant for software performance prediction, for example, the graphical model layout. We have implemented in Java a converter to extract performance-related information from the software-model-XML-file, primarily including information about system users and their requests (from use case model), about a user request processing (from activity diagrams in design model), and about physical computer devices participating in the software execution (from deployment model). The extracted information is described concisely in a performance-XML-file whose structure is defined by the metamodel-DTD.

The metamodel-DTD file reformulates the performance prediction metamodel shown in Figure 2. The metamodel elements are represented by XML elements. Their parameters are represented by attributes of the corresponding XML elements. Associations between metamodel elements provide the basis for specifying the nesting and referencing relationship among XML elements.

The performance-XML-file is used as the input for constructing performance prediction models, such as queuing network models. The generated performance models are evaluated to produce estimations of performance properties of software. The performance estimations are used to determine whether the software development evolves to implement the software system based on the selected software architecture and design. If architecture and design alternatives are suggested, they are reevaluated before the executable software is implemented.

3 Conclusion and Future Work

Integrating performance analysis into the software engineering process has been identified already a long time ago to be of substantial importance. However, the lack of overlap between tools for both, software engineering and performance analysis, has prevented the full integration of performance modeling techniques into the software development process. Since the unified modeling language (UML) became a quasi-standard for software engineering, significant research work has been published on methods and tools to evaluate software performance and architecture solutions based on UML models. Some work maps functional UML models to performance modeling mechanisms (e.g. in [9]); UML-based tools and notations are developed for performance modeling and calculation (e.g. in [4]); the built-in UML extensibility mechanisms are used to extend UML models with performance information (e.g. in [8]); several commercial UML tools, e.g. Rational Rose Real-Time [10], support the automatic generation of executable code and direct execution of UML models during the development of real-time and embedded systems.

In addition to existing work on integrating software performance engineering within software design, the work presented in this paper makes the following contributions: **(i)** A performance prediction process is integrated into the software development process. Functional models are extended with performance-related information used as the basis for quantitative performance modeling and calculation. **(ii)** Performance prediction is closely related to the emerging software implementation, in particular, the participating computer resources and their quantitative behavior. This increases the credibility and usefulness of the performance prediction results. **(iii)** Performance-related information is defined in a metamodel in the engineering terms that are easily understandable for software developers. The software developers are sheltered from the knowledge of performance modeling and calculation. They can concentrate on providing performance-related information in a software engineering way and leave the construction and solution of performance prediction models to those having expertise. **(iv)** UML is used both for constructing the functional models and extending them with performance-related information. This is supported by standard UML modeling tools. **(v)** Performance information is described concisely in XML after it is automatically extracted from the extended functional models. This provides the flexibility for choosing available performance analysis tools to obtain performance prediction results quickly.

Our work will continue to adapt the current project to the upcoming UML 2.0, with the expectation that the updates, refinements, and expansions to UML infra- and superstructure will be completely supported by the mainstream UML tools in the near future.

4 References

[1] Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*. Addison-Wesley (1999)

[2] Borland Software Corporation: Together Control Center 6.1. (http://www.borland.com/together/index.html)

[3] Jacobson, I., Booch, G., Rumbaugh, J.: *The Unified Software Development Process*. Addison-Wesley (1999)

[4] Kähkipuro, P.: "UML-Based Performance Modeling Framework for Component-Based Distributed Systems". In: Dumke, R. et al. (eds.): *Performance Engineering - State of the Art and Current Trends*. LNCS 2047. Springer-Verlag, Berlin (2001) 167-184

[5] Kazman, R. et al.: "The Architecture Tradeoff Analysis Method". Technical Report. CMU/SEI-98-TR-008. Software Engineering Institute, Carnegie Mellon University. Pittsburgh (1998)

[6] Object Management Group (OMG): *Unified Modeling Language Specification, version 1.3*. (http://www.omg.org/cgi-bin/doc?formal/00-03-01.pdf)

[7] Object Management Group (OMG): *XML Metadata Interchange (XMI) Specification, version 1.1*. (http://www.omg.org/cgi-bin/doc?formal/00-11-02)

[8] Petriu, D.C., Shen, H.: "Applying the UML Performance Profile: Graph Grammar-Based Derivation of LQN Models from UML Specifications". In: Field, T. et al. (eds.): *Proceedings Performance TOOLS 2002*. London, UK (2002)

[9] Pooley, R., King, P.: "The Unified Modeling Language and Performance Engineering". In: *IEE Proceedings-Software*, Vol. 146, No. 1. (1999) 2-10

[10] Rational Software Corporation. (http://www.rational.com) (2003)

[11] Smith, C.U.: *Performance Engineering of Software Systems*, Addison-Wesley (1990)

[12] World Wide Web Consortium (W3C) Recommendation: *Extensible Markup Language (XML) 1.0 (Second Edition)*.(http://www.w3.org/TR/2000/REC-xml-20001006)

A Dual Language Approach to the Development of Time-Critical Systems with UML

Luigi Lavazza[1,3], Sandro Morasca[2], Angelo Morzenti [1]

[1] Politecnico di Milano, Dipartimento di Elettronica e Informazione,
P.zza Leonardo Da Vinci, 32, 20133 Milano, Italy
{lavazza,morzenti}@elet.polimi.it
[2] Università degli Studi dell'Insubria,
Dipartimento di Scienze Chimiche, Fisiche e Matematiche,
Via Valleggio 11, 22100, Como, Italy
sandro.morasca@uninsubria.it
[3] CEFRIEL,
Via Fucini 2, 20133, Milano, Italy

Abstract. The development of time-critical systems requires the availability of notations that are expressive, rigorous, easy to use, and provided with software tools. In particular it is important that such notations are able to express the time-related features of the systems, in a way that is formal enough to support activities like property verification, test case generation, etc. For this purpose we propose a dual-language approach. In addition to the typical UML (and UML-RT) diagrams, we also propose a descriptive formalism to specify the properties of a system and its components. Specifically, this description consists of a formula of a new logic, called OTL (Object Temporal Logic), which is an extension of OCL. The proposed approach is applied to a case study derived from the industrial experiences of the authors.

1 Introduction

The development of time-critical systems requires the availability of notations that are expressive, rigorous, easy to use, and provided with software tools at the same time. Time-critical software systems are usually complex ones and need to be modeled and analyzed from several different perspectives, such as their functional behavior, their temporal behavior, and their structure. It is unlikely that a single notation may describe all of these aspects adequately, so several notations have been proposed in the past. UML [1] is being increasingly used for the development of complex systems such as real-time software. However, past versions of UML were not able to deal with time-related aspects. Only recently were timing features added to the UML notation, but their introduction is still tentative, incomplete, and not well integrated with the other aspects of UML. In addition, the practical application of UML to the real-time domain is hindered by UML's lack of a complete set of constructs to express time-related constraints and properties, as well as by its lack of formal semantics.

As a first attempt to provide a way to deal with time-related aspects with UML, UML for Real-Time (alias UML-RT) has been defined on the basis of ROOM [6] and has been adopted by several developers. However, the application of UML-RT to the real-time domain still suffers from several problems. In particular, UML-RT is not formally well defined, it is an effective notation for the design and implementation of systems, but not for representing requirements or specifications, finally, time-related information (i.e., the representation of time and time constraints) is not treated at a native level, but often only through ad-hoc components (like timers). These problems are particularly relevant since it has been proposed to include several concepts of UML-RT into the forthcoming UML 2.0. Moreover, since the notion of Time has also been included in the proposal for UML 2.0, it seems quite urgent to enhance OCL with the possibility to deal with time.

An adequate solution to the problem of dealing with time-related aspects with UML will need to go one step forward. In particular, high rigor of syntactic and especially semantic definition is needed, together with high integration and consistency with the rest of the UML notation.

In this paper, we propose an extension to UML that addresses these problems via a set of carefully thought and balanced notations which can be used by practitioners in industrial environments and can support suitable development methods for time-critical systems. As a matter of fact, our proposal for extending/specializing UML for time critical systems is *not* only a notation, unrelated to the development process. This specialized domain requires systematic and rigorous development, centered on explicit, possibly formal requirements specification, and requirement validation and verification are also of crucial importance.

The notation we propose is centered on architectural diagrams that correspond to UML-RT collaboration diagrams. System components are modeled via a small set of fundamental constructs: capsules correspond to components; ports and protocols model abstract interfaces (i.e., they describe only the alphabet, not the behavior); and connectors correspond to communication relations. The partitioning of a complex system into a set of components (i.e., parts) that conceptually evolve in parallel and communicate via connectors can be iterated to an arbitrary level of depth. This results in a tree-shaped hierarchy of parts and sub-parts, where the root corresponds to the overall system being modeled, and the leaves to the components that are not further structured, which are modeled in an operational style with a state-transition machine.

We also propose a descriptive formalism to specify the properties of a system and its components, whose style is complementary to that of statecharts. Specifically, this description consists of a formula of a new logic, called OTL (Object Temporal Logic), which we define in such a way as to make it compatible with OCL (Object Constraint Language).

Thus, we propose a “dual language” approach: the OTL part is an abstract specification of the properties, constraints and temporal relations that must hold among the states, events, and signals of the statechart machine associated with the same capsule. OTL formulas and the statechart associated to a given capsule/component are in the classical specification/implementation relation that is typical of dual language approaches to the development of reactive systems.

2 The OTL language

The Object Constraint Language (OCL) defined in UML can be used to state behavioral properties of a system and its parts. However, when dealing with time-dependent systems, OCL needs to be extended to fully specify temporal aspects, e.g., to specify the time distance between events, that cannot represented with plain OCL.

We propose Object Temporal Logic (OTL) as a temporal logic extension to OCL. Based on one fundamental temporal operator, OTL provides the typical basic temporal operators of temporal logics, i.e., *Always*, *Sometimes*, *Until*, etc. In addition, OTL allows the modeler to reason about time in a quantitative fashion. OTL is a part of a UML-based formalism, so it is totally integrated with the other UML notations. As far as the OCL 2.0 standard library is concerned, OTL extends it by adding two new classes, `Time` and `Duration` (see Fig. 1). Class `Time` models time instants, which are defined based on the current time taken as the time origin. Class `Duration` models duration of time intervals, i.e., the distance between two time instants. `Duration` objects may be positive or negative numbers: a `Duration d` that is added to a `Time` object (see below the '+' operator for class `Time`) is interpreted as a displacement towards the future if `d` is positive, towards the past if `d` is negative. The notion of a time interval can be suitably defined in terms of the concepts of Time and Duration. The existence of both classes `Time` and `Duration` allows for a conceptually proper treatment of time and the definition of sensible operations involving objects of the two classes. For instance, class `Time` provides (1) an operation '≤' that checks the ordering between its objects, so we can say if a time instant precedes or follows another time instant; (2) an operation '`dist`' for finding the time distance between two instants, which returns an object of class `Duration`, positive, null, or negative, depending on the relative position of the considered `Time` points; (3) an operation '+' that takes a parameter `d` of class `Duration` and returns the `Time` object that lies at a time distance `d` in the future if `d` is positive or in the past if `d` is negative; and (4) an operation called `futrInterval` (and a symmetric one for the past, called `pastInterval`) that takes a parameter of class `Duration` and returns a `Collection` all of `Time` points within a distance `d` in the future (respectively, the operator `pastInterval` returns the `Collection` of all `Time` points within a distance `d` in the past). Class `Duration` has sum and subtraction operations between its objects: for instance, the sum of two time distances is a new time distance, whose extension is the sum of the extensions of the original time distances. These operations allow modelers to use quantitative time.

`Time` and `Duration` may be discrete or dense, depending on the application at hand, or on the (sub)system that is described by the given UML model. From a methodological viewpoint, one can note that a continuous time is useful when modeling the evolution over time of intrinsically continuous physical entities (e.g., a temperature or a voltage) that are external to the device or system under development and that must be monitored or controlled. In this case the use of continuous entities is indispensable even for just expressing the user requirements, and *a fortiori* for analyzing and proving their satisfaction on the System Requirements analysis [7]. On the other hand discrete time will suffice to model parts corresponding to digital, synchronous devices

and in general in the UML artifacts related with detailed specification, design and implementation of the device under development.

To allow for the evaluation of a predicate `p` at a time different than the current one, OTL introduces a new semantic primitive as a method of class `Time`, in a way that is consistent with the OCL notation. Thus, given a time instant `t`, represented as an object t of class `Time`, the evaluation of `e` at time `t` is denoted as follows:

```
t.eval(e)
```

Method `eval` receives an `OclExpression` as the parameter (`e`) and returns a value of the type of `e`. Its meaning is that expression `e` is evaluated at time `t`.

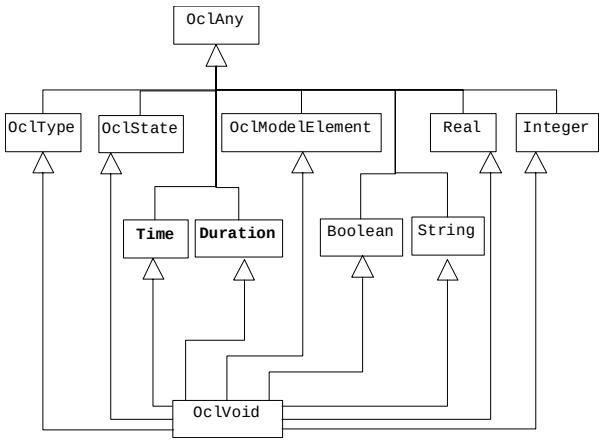


Fig. 1. The OCL standard library extended with types Time, Duration and Interval

Our extensions do not require any change in the metamodel of OCL: types `Time` and `Duration` are simply added to the OCL standard library as specializations of `OclAny`. The full definition of these classes is not reported here for space reasons.

To provide modelers with expressive tools to describe time-critical systems, it is useful and convenient to define a set of temporal operators. For instance,

```
context C
  inv: Lasts(p, d)
```

specifies that predicate `p` holds in the interval from the current time and lasting `d` time units. This statement can be defined as a shorthand for the following expression, where the term `now` denotes the time with reference to which the (sub)formula is interpreted, and, being `T` a set of objects of class `Time`, formula `T->forall(t: Time|t.eval(P))` is true if and only if `p` is true at all time instants in `T`.

```
context C
  inv: let I: Set(Time) = now.futrInterval(d) in
          I->forall(t: Time| t.eval(p))
```

As additional notational conventions, we use `inf` to denote the infinite `Duration` value, and we abbreviate the basic temporal operator `t.eval(p)` into the more

convenient and intuitive syntax `p@t`. With these conventions in mind, other operators can be easily defined in a similar manner, as reported in the following table.

name of operator	intuitive meaning	formal definition
`Futr(p,d)`	`d time units in the future`	`p@(now + d)`
`SomF(p)`	`sometimes`	`let I: Set(Time) = now.futrInterval(inf) in I->exists(t: Time| p@t)`
`AlwF(p)`	`always`	`let I: Set(Time) = now.futrInterval(inf) in I->forall(t: Time| p@t)`
`WithinF(p,d)`	`within d time units`	`let I: Set(Time) = now.futrInterval(d) in I->exists(t: Time| p@t)`
`Until(p,q)`	`p holds until q occurs`	`let I: Set(Time) = now.futrInterval(inf) in I->exists(t:Time| q@t and Lasts(p,t-now))`

Similar operators concerning the past can be defined using the `pastInterval` operator of `Time`: for instance, the past counterparts of the operators defined above would be called: `Lasted(p,d)`, `Past(p,d)`, `SomP(p)`, `AlwP(p)`, `WithinP(p,d)`, and `Since(p,q)`. For operators that refer to time intervals we also adopt the convention of adding a suffix to indicate explicitly whether the extremes of the interval are included or not; we use the letter 'i' to denote inclusion, and letter 'e' to denote exclusion; for instance the formula `Lasts_ie(p,d)` states that property `p` holds starting from `now` (included) to `now+d` (excluded).

As a useful shorthand we will use the constructs `futr(v,d)` and `past(v,d)` (where `v` is a any term, and `d` is a term of class `Duration`) as *terms* to denote the value of any given term `v` at a distance of `d` time units in the future or in the past with respect to `now`. For instance, `futr(v,d)` is defined as `v@(now+d)`.

3 A Case Study

We illustrate our dual language approach by providing a small fragment of the specification of a digital energy and power meter. The meter is composed of a magnetic transducer (a device called "G. Ferraris" after the name of its inventor) that converts the electric energy flow through the line into the rotation of a disc. In the peripheral part of the disc transparent and opaque portions are evenly alternated, with the purpose of permitting the detection of the disc position and its velocity (which are respectively proportional to energy and power consumption) by means of a photocell. The energy meter includes, besides the G. Ferraris, and the disk, a device, called Reader, that provides the sampling signal for the photocells and detects the full/empty position of the disk from the reading of the photocell signal. A further device, called CostAssign, determines the cost for the client of each consumed quantum of energy, based on the current time and date and on the applicable tariff, and a final one, called Totalizer computes the total amount of the invoice to be sent periodically to the client. The UML model of the system, not reported here for space reasons, can be found elsewhere [10].

By means of OTL a few global properties of the system can be specified precisely and clearly.

- The amount of used energy reported by the device (represented by attribute **EnergyUsedReported** of class **CostAssign**) is monotonic.

```
context CostAssign
  inv: EnergyUsedReported() >= past(EnergyUsedReported(), d)
```

  In the statement above **d** is a (implicitly) universally quantified variable, therefore the statement above is equivalent to the following:

```
context CostAssign
  inv: let D: Set(Duration) = All_possible_durations in
       D->forall(d: Duration| EnergyUsedReported() >=
                              past(EnergyUsedReported(), d))
```

- The cost of the energy consumed at constant tariff increases linearly, proportionally to the consumed energy and the tariff:

```
context Totalizer
inv: Lasted(Tariff.CurrentTariff()=tc,d) implies
     (TotalCost - past(TotalCost,d) = tc*
      (CostAssign.EnergyUsedReported()-
       past(CostAssign.EnergyUsedReported(),d))
```

  In the statement above **CurrentTariff** is a method of class **Tariff** , which returns the tariff currently applied. **TotalCost** is an attribute of class **Totalizer** which represents the total cost of the energy consumed up to the current time.

- The difference (absolute value) between the energy reported used and the energy actually used is always less than a given (small) amount:

```
context CostAssign
inv: abs(EnergyUsedReported()-past(EnergyUsedReported(),d)-
(Environment.EnergyUsed-past(Environment.EnergyUsed,d)))<0.01
```

  In the statement above **EnergyUsed** is an attribute of **Environment**, which represents the amount of energy actually consumed.

Note that the above property guarantees that the consumer does not have to pay more than the due, while the energy company does not get paid less than due.

4 Conclusions

The development of real-time critical applications calls for a specific process and rigorous notation. We propose a “dual language” approach, where UML provides the constructs for modeling the structure of the system and the behavior of the system’s components. A new descriptive language based on temporal logic, called OTL, allows the developer to assert properties of the system at an abstract specification level.

The notation we propose should be used in the context of an organized development process to fully take advantage of its characteristics. This process requires that the description of the software system, the environment in which it operates, and the user requirements is complete and precise. In particular, the system model will typically include: the capsules of the collaboration diagrams representing the “domains”

of the problem and the elements of the solution; the statecharts, describing the behavior of each capsule; the OTL specifications representing user requirements at the most abstract level.

Such a system description –once supported by suitable tools–allows developers to: validate and verify the specification through simulation and model checking, to ensure that a model satisfies the properties specified with OTL (i.e., the provided statecharts are a valid "implementation" of the system behavior); support verification by generating functional test cases that can guarantee (to a reasonable extent) that the implementation of the system is consistent with its specifications.

The literature already contains a few proposals for augmenting OCL to deal with time-dependent systems (see [5]). Some of them (e.g., [2,9]) deal with time only from a qualitative viewpoint, i.e., no notion of temporal distance between events is provided. Another proposal allows modelers to deal with time in a quantitative fashion, by extending the set of operators of OCL [3]. However, in the latter approach a discrete time is associated with the system states and events, while in our approach real values can be used to represent time.

References

1. OMG Unified Modeling Language Specification Version 1.5, March 2003, formal/03-03-01. http://www.omg.org.
2. Cengarle, M.V., Knapp, A.: Towards OCL/RT. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, Proc. 11th Int. Symp. Formal Methods Europe, Berlin 2002, Springer LNCS 2391.
3. Flake, S., Müller, W.: An OCL Extension for Real-Time Constraints. In T. Clark and J. Warmer (Eds.), Advances in Object Modelling with OCL, Springer Verlag, October 2001
4. A. Morzenti, P. San Pietro, "Object-Oriented Logic Specifications of Time Critical Systems", ACM TOSEM - Transactions on Software Engineering and Methodology, vol.3, n.1, January 1994, pp. 56-98.
5. Rammig, F.J.: OCL Goes Real-Time. Proc. of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, IEEE 2002.
6. Selic B.,Gullekson G.,Ward P.: Real-Time Object-Oriented Modeling. Wiley 1994
7. A. Gargantini, A. Morzenti, "Automated Deductive Requirements Analysis of Critical Systems", ACM TOSEM - Transactions On Software Engineering and Methodologies, Vol. 10, no. 3, July 2001, pp. 225-307.
8. D. Gabbay, A. Pnueli, S. Shelah, J. Stavi, On the Temporal Analysis of Fairness, 7th ASM Symposium on Principles of Programming Languages, January 1980.
9. Ziemann, P. and Gogolla, M. An Extension of OCL with Temporal Logic. Critical Systems Development with UML – Proceedings of the UML'02 workshop, pages 53–62. TUM, Institut fur Informatik, Septeber 2002, TUM-I0208.
10. Lavazza, L., Morasca, S., Morzenti, A.: "A Dual Language Approach to the Development of Time-Critical Systems with UML", Politecnico di Milano, Dipartimento di Elettronica e Informazione, Technical report 2003.32, September 2003.

An Approach to Evaluate Real-Time Software Architectures for Safety-Critical Systems

Jan van Katwijk[1], Bo Sandén[2], and Janusz Zalewski[3]

[1] Faculty of ITS, Delft University of Technology
2628 CD Delft, The Netherlands
J.vanKatwijk@twi.tudelft.nl
[2] Computer Science, Colorado Technical University
Colorado Springs, CO 80907, USA
bsanden@acm.org
[3] Computer Science, Florida Gulf Coast University
Fort Myers, FL 33928, USA
zalewski@fgcu.edu

Abstract. The authors discuss two new measures and a semi-automated process to evaluate real-time software architectures. First, concurrency level is introduced to evaluate the structure of software architecture and assess its optimality. Next, software sensitivity is presented as a measure of dynamic properties of an architecture, based on the percentage of missed deadlines. Finally, an approach to evaluate software safety is discussed in relation to software architectures. The paper concludes with recommendations for incorporating the processes into a UML-based toolset.

1 Introduction

The objective of this paper is to outline an approach to evaluate real-time software architectures and make recommendations for using them to enhance methods for the development of real-time safety-critical software as well as to improve the effectiveness of corresponding development tools. We can safely say that in recent years UML has become the dominating notation and a standard for expressing software designs. Its design tools, however, while able to produce sophisticated designs, are lacking any significant ability to assess design quality. One definite advantage of enhancing such tools would be the capability of automatic evaluation of architectural properties of software at the design level.

In this study, we address the architecture evaluation aspect, with regard to real-time safety-critical systems. The crucial point is to be able to evaluate the architectural design's structure and the behavior of the design model to help designers in selecting an optimal solution. In this view, we need metrics to evaluate: static aspects of the software architecture, including its structure; dynamic aspects, assessing primarily its behavior; and external aspects to assess the effects that poorly designed architecture may have on dependability.

The rest of the paper is structured as follows. In Section 2, we introduce a new parameter, named concurrency level, to evaluate the structure of real-time software architecture. Section 3 outlines the principles of a new metric called software sensitivity, to evaluate behavior of real-time software. Section 4 discusses the evaluation of dependability properties for software architecture, and Section 5 ends the paper with conclusions.

2 Software Architecture: Concurrency Level and Optimality

Choosing the right overall software architecture is often crucial because one architecture may require a much greater design and programming effort than another, may incur much more run-time overhead, or be susceptible to specific design flaws. Since the very nature of real-time systems is concurrency, the architecture of real-time software must involve concurrency as well. One specific approach, entity-life modeling (ELM) [1] lets you identify the possible architectures without a lengthy and overly detailed process.

In ELM, the tasks in the software are modeled on event threads in the problem domain. These threads are identified by means of the following *partitioning* process:

1. Create an imaginary *trace* by putting all the event occurrences a software system must handle along a time line. Include events that originate in the problem domain and that the software has to react to, as well as events that the software itself must initiate.
2. Partition the events in the trace into *event threads* such that:
(a) Each event occurrence belongs to exactly one thread
(b) Event occurrences in threads are separated by sufficient time for handling.

The events that are the subject of this partitioning occur in the domain, that is, they are independent of a particular software design. They model any sort of input events, which are always independent, and output events, which must be generated by software at specific times to meet functional requirements. One must include *time events* [1].

The result of the partitioning is called a *thread model* of a problem, defined as a set of event threads accounting for all relevant event occurrences in the problem. Each event thread in the model generally becomes a software task. That way, the thread model of the problem domain defines the software architecture.

As an example, consider the Real-Time Data Acquisition System. Such a system will have a number of inputs from various sources, for example:

- T1, a sensor providing data periodically
- T2, another sensor delivering data on a non-periodic basis, in bursts
- T3, irregular commands of certain types (display, compute, etc.) from an operator.

Additional requirements placed on a typical real-time data acquisition system also include:

- T4, data recording by writing to a disk file every so often, and
- T5, computation of results based on data acquired from sensors.

The last two requirements contribute to system's timeliness, since data (results of computations and recording) must be delivered by strict deadlines. At first glance, one may consider doing a computational action in the background, but that is not feasible, since its results must be delivered on time.

Following the partitioning process outlined above, we may produce the sequencing of events as shown in Fig. 2. Since events T1, T2 and T3 are completely independent, they can occur at the same time. Event T4, data recording, is scheduled at some absolute times and can also coincide with the three input events. Moreover, computations T5 are normally scheduled after N sensor readout events and as such may also coincide with the forthcoming events, in the worst case – with all four events previously mentioned.

The partitioning process does not limit the number of threads so in the extreme, each event could have its own thread and task. Such a task would often depend on some other task and could not run until the second task had completed processing. If two tasks always run one at a time, one of them is usually unnecessary. To avoid unnecessary tasks, we need to identify event threads that can be busy at the same time. The intuitive notion of *coincidental simultaneity* captures this idea: event threads are independent if once in a while they all happen to have an event occurring at the same time.

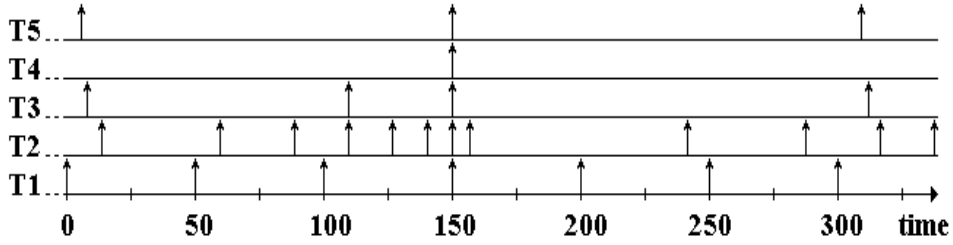


Fig. 1. Illustration of event simultaneity.

Strictly speaking, the likelihood that instantaneous events occur at the same time is zero. To formally capture the notion of coincidental simultaneity, we shall say that threads *co-occur* if an arbitrarily short time interval can be found where each of them has an event occurring. An *optimal* thread model is one where all threads co-occur. There are times – at least theoretically – when every thread in an optimal model has an event occurring. Thus, the optimality criterion to describe the quality of a software architecture can be judged based upon the number of threads constituting such architecture.

In most realistic problems, the number of events that can ever occur at the same time is limited. The *concurrency level* of a problem is defined as the maximum number of events ever occurring in an arbitrarily short interval. The number of threads in an optimal model of a problem is equal to the problem's concurrency level. In a real-time data acquisition example, five events can occur simultaneously, as outlined in Fig. 2, therefore the minimum and optimal number of threads (and tasks) is 5. This means that a correct design with smaller number of tasks is not possible.

Overall, knowing the concurrency level of the problem, it should be possible for the tool to automatically generate the optimal architecture. In the next section, we apply the concept of concurrency level to develop generic real-time software architecture and introduce a new measure called sensitivity, to evaluate real-time software behavior.

3 Evaluating Design Behavior: Software Sensitivity

Concurrency level and optimality deal exclusively with static aspects of software architecture, its structure and composition. Once we have applied the concepts of concurrency level and optimality of architecture to some real-time problem, can we go a bit further and make some conclusions about the behavioral properties of such an architecture?

In [2] we discussed a template for a high-level architecture of real-time software. The approach taken there, combined with the concept of concurrency level, is an excellent starting point for a more systematic development of real-time safety-critical systems. Our architecture of real-time software involves 7 types of events that ultimately map to corresponding tasks, cooperating together towards a common goal. The components in this architecture are derived with respect to the following generic classes of events (Fig. 2):

- measurements (sensors) and control (actuators)
- user interface (often in a form of Graphical User Interface – GUI)
- database storage and retrieval
- processing of the acquired data with some kind of computational algorithms
- communications with other processors over a bus or a network, and
- timing.

For every real-time application and functional requirements, the optimality of this architecture can be easily proved in terms of a minimum number of concurrent events, that is, the problem's concurrency level.

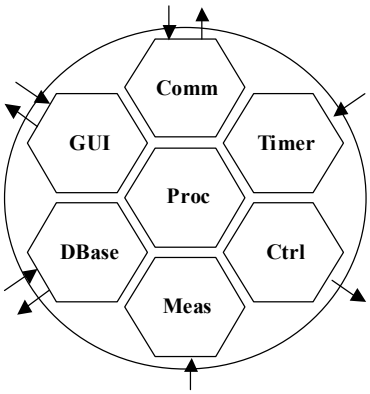


Fig. 2. Architectural template for real-time software.

For this kind of architectural template, we can outline a benchmark to analyze its behavior. Here we include only threads relevant to the data acquisition system, which performs functions mentioned in the previous section: sensor readouts, computations, database access, and user interface. Each of the corresponding tasks may run on the same processor or on separate nodes communicating with other selected nodes as in Fig. 3 [3]:

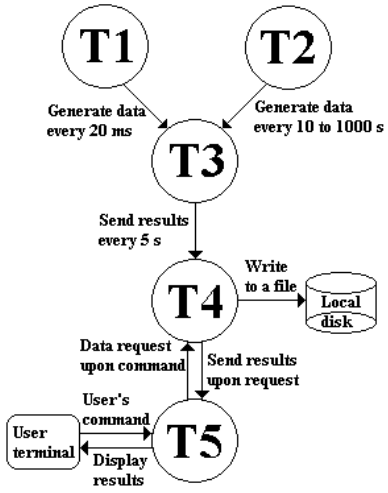


Fig. 3. Five-task benchmark architecture.

- Task T1 is simulating a sensor. It generates a random integer once every 20 milliseconds and sends the data to Task T3.
- Task T2 is simulating another sensor. It generates periodically a random integer in an interval of time, and sends the data to Task T3 for processing. The length of the interval is randomly selected between 10 to 1000 milliseconds.

- Task T3 is a computational task, which accepts all incoming data and makes some calculation. After computation, the task sends its results to Task T4 for storage.
- Task T4 accepts the data from Task T3 and stores it in a local file.
- Task T5 provides an interface for the user. The only commands are: to get the most recent result from Task T3 and display it on the screen, to cause task T5 to terminate all the tasks, including itself.

Under the assumption that a real-time system demonstrates satisfiable performance if it meets its timing constraints (deadlines), we propose the following evaluation measures:

- the number of times the deadlines are missed (percentage of missed deadlines),
- the overall accumulated time by which the deadlines are missed;

which can be evaluated for a particular software module on a particular node.

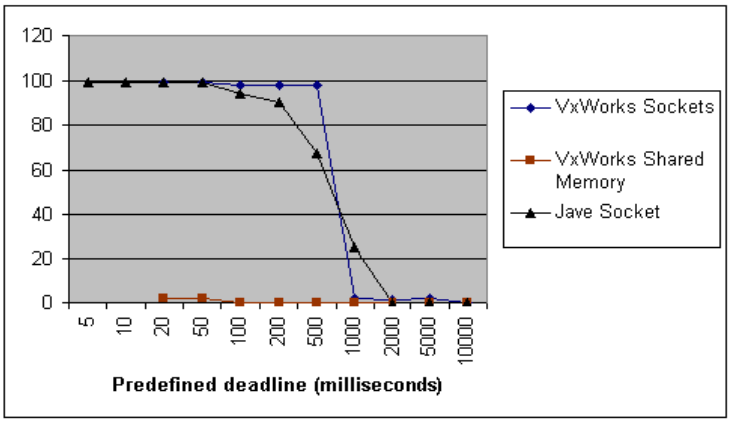


Fig. 4. Percentage of deadlines missed by 2% in VxWorks experiments.

Sample measurements of the former parameter, taken for various configurations of VxWorks and Java sockets and for various CORBA implementations, are presented in Fig. 4 and 5 [3]. Measurements were taken at task T3 for communication with task T2.

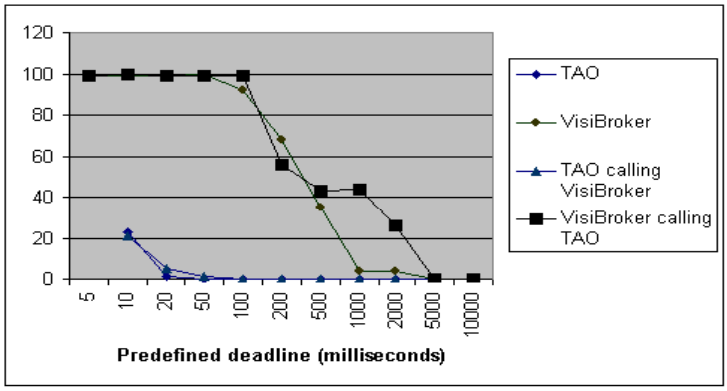


Fig. 5. Percentage of deadlines missed by 2% in CORBA experiments.

Some curves on the graphs descend smoothly, while some others descend sharply. For example, Fig. 4 shows that the performance of VxWorks sockets implementation begins to worsen rapidly when the deadline is shortened below 1 s. Therefore, we may want to study what happens when the deadline is around 1 s. On the other hand, Fig. 5 shows that

CORBA TAO [4] is relatively indifferent to the predefined deadlines, the entire curve remains fairly flat and none of the points exhibits anything special. This means that the system is not very sensitive to changes of deadlines within the range studied.

Based on these observations, we define a new parameter as a metric for software performance, called *sensitivity:*

> *Software sensitivity is a measure of how fast the software responses change when deadlines are increased or decreased*.

The interpretation of sensitivity is that the faster the curve descends the more sensitive the respective system: a small change of a deadline length causes relatively larger changes in the number of deadlines missed. Sensitivity shows whether performance degradation occurs sharply or gracefully. Quantitatively, however, it is not just the slope of the curve. It can be represented by the ratio of the change of response over the range of deadline lengths for the changed interval. Sensitivity is a parameter that takes into account the slope of each curve, in relative terms, to make curves and respective systems comparable.

The first step in calculating the sensitivity is to linearize the curve in the interesting region. Then, to account for relative differences in absolute values of deadlines for different systems, the actual value of sensitivity is calculated from the straight line fits, according to the formula [4], where (*x2,y2*) and (*x1,y1*) are coordinates of respective points on the straight line reflecting the range considered:

$$[(y2-y1)/ (y2+y1)/2] / [(x2-x1)/(x2+x1)/2]$$

That way we allow comparison of different systems, for which deadline lengths are in different ranges, but the system's speed of response, that is, sensitivity, may be equivalent.

Using this method, we can assess sensitivity of various implementations. For example, the five-task model implementation with VxWorks sockets looks as the most sensitive one. In a rather sophisticated air-traffic control simulator [3], sensitivity measurements showed that the least sensitive part of the system. Discovering this relationship had a positive impact on software redesign. Overall, the sensitivity parameter tells us, how fast the system degrades, if the deadlines are shortened, that is, how fast it gets saturated.

4 Software Dependability Perspective

With both metrics introduced in previous sections, we can make evaluations of the architecture for specific functional requirements placed on software. In safety-critical systems, however, there is a number of requirements that are non-functional, such as dependability requirements that include reliability, safety and security. They are relevant to software architecture with respect to external conditions that may lead to software or system malfunctions. Their assessment depends on something more than existing software architecture, so making respective judgments is not possible by evaluating the architecture only. The software's impact on external systems and environment must be involved.

Essentially, there are two ways of dealing with the problem of meeting non-functional requirements by software architecture. First, one can build software architecture specifically designing it for safety. Second, one can verify properties of the architecture as a part of the software development process. The former can be done by introducing a redundancy mechanism into an architecture, so that it will not be optimal in a sense outlined in Section 2, but will meet the additional requirements. One particular approach to do this is to provide a safety guard or safety shell [5] that monitors signals exchanged between the controller and the environment (Fig. 6).

To accomplish its mission, the safety shell must perform two functions: (1) detect conditions that, if unchecked, will lead to a failure, and (2) respond to these conditions in a manner that the system will return to safe operation. Both of these functions may be implemented directly by building and analyzing a fault tree of the entire system.

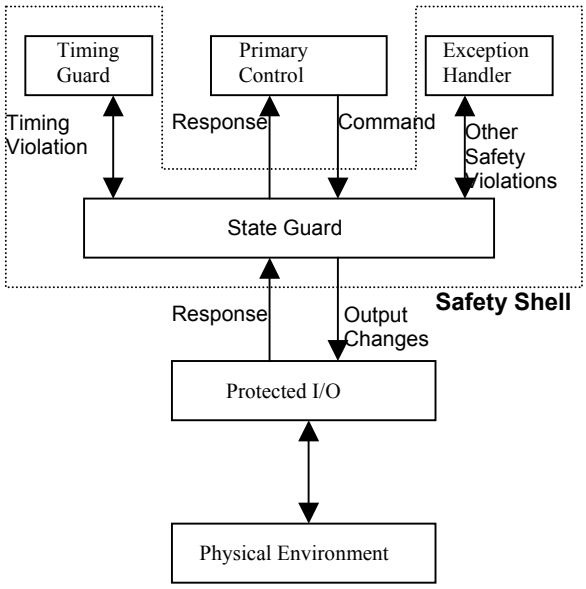


Fig. 6. Illustration of a safety shell concept.

Detection of failures consists of three parts. The shell must detect when a command issued by the controller will force the system into an unsafe state. It must check and 'approve' all commands issued by the controller before they reach the environment. Secondly, the shell must detect when the physical environment has independently entered a failure mode and must issue its own commands to restore the system to a safe state. Lastly, the shell must keep track of all timing constraints that may cause the system to enter failure modes and it must issue its own commands to prevent violation of these constraints. The shell must detect both system state and time variant failure modes.

The second way of treating the problem focuses on means of verifying the software architecture. In case of a safety shell, the safety related functions are separated from the control functions, so verifying the shell independently is sufficient [6]. To accomplish this goal for systems designed with UML, one has to have means of expressing safety properties, usually time related, in a more formal way than current UML notation allows.

In [6] the authors proposed to verify UML models expressed in statecharts with amendments on timing, by converting them to a formal description using extended timed graphs (XTGs). The conversion process is based on mutual correspondence of the notational elements. XTGs are then transformed to code with an automatic tool XTGconverter, and the code is subject to verification by a PMC model checker [7]. The entire design process of a real-time safety-critical software architecture looks as follows:

(a) producing separate designs for safety-critical and non-safety-critical parts
(b) expressing the designs in UML with statecharts
(c) converting the statecharts into XTGs
(d) generating XTG code for the model checker
(e) verifying properties with model checker.

The approach has been tested with safety shell for a number of case studies, including a traffic light controller [5], a railroad crossing system, and various aspects of the Air-Traffic Notification System (ANS) for the Future Air Navigation System (FANS) [6].

5 Conclusion

Real-time safety-critical systems are becoming a pervasive element of contemporary society. Software development methodologies for these systems have made a significant progress in recent years, but this trend has not been sufficiently backed by the development of respective automatic tools. This paper presented a systematic view of designing software architectures for such systems. It is based on two new metrics for evaluating real-time software architectures and results in a procedure separating part of the architecture responsible for safety aspects from the functional controller's part.

The first metric, concurrency level, allows building and evaluation of the optimal architecture, according to the criterion of a minimal number of threads in the problem domain. A generic real-time software architecture can be built, for which a new parameter, called software sensitivity, can be evaluated to estimate dynamic properties based on the percentage of missed deadlines. This technique results in a software design pattern for real-time architecture and can be used in constructing real-time software with UML [8].

Having assessed the software architecture in terms of its concurrency level and sensitivity, we are able to see how it meets functional and performance requirements. For evaluation of the degree of conformance with non-functional requirements, such as safety, we propose amending the architecture by a component called safety shell. This allows for separation of concerns and an independent verification of the safety related part. Design of a safety shell can be normally done in UML using statecharts, that are subsequently converted to Extended Timed Graphs and formally verified using model checking.

Future work should involve a more thorough study of real-time architectural components and their properties, such as coupling, execution dependencies, operational profiles, consequences of requirement changes, fault propagation, and hazard analysis at the architectural level. Studies in real-time software architectures should be complemented by evaluation of automatic software tools used for developing safety-critical software.

References

1. Sandén, B., Entity-Life Modeling: Modeling a Thread Architecture on the Problem Environment, IEEE Software, Vol. 20, No. 4, pp. 70-78, July/August 2003
2. Zalewski, J., Real-Time Software Architectures and Design Patters: Fundamental Concepts and Their Consequences, Annual Reviews in Control, Vol. 25, pp. 133-146, 2001
3. Guo, G., J. van Katwijk, J. Zalewski, A New Benchmark for Distributed Real-Time Systems: Some Experimental Results, Proc. WRTP'03, 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, Łagów, Poland, May 14-17, 2003, pp. 141-146
4. Schmidt, D.C., A.S. Gokhale, T.H. Harrison, A High-Performance End System Architecture for Real-Time CORBA, IEEE Communications Magazine, Vol. 35, No. 2, pp. 72-77, February 1997
5. Anderson, E., J. van Katwijk, J. Zalewski, New Method of Improving Software Safety in Mission-Critical Real-Time Systems, Proc. ISSC'99, 17th Int'l System Safety Conference, Orlando, Fla., August 16-21, 1999, pp. 587-596
6. van Katwijk, J., H. Toetenel, A.E.K. Sahraoui, E. Anderson, J. Zalewski, Specification and Verification of a Safety Shell with Statecharts and Extended Timed Graphs, Proc. SAFECOMP 2000 - 19th Int'l Conf. on Safety, Reliability and Security, Rotterdam, The Netherlands, October 25-27, 2000, Springer-Verlag, Berlin, pp. 37-52
7. Al-Daraiseh, A., J. Zalewski, H. Toetenel, Expressing and Verifying Timing Requirements with UML, Proc. SCI2001, 5th World Multi-Conference on Systemics, Cybernetics and Informatics, Orlando, Fla., July 22-25, 2001, pp. 461-466
8. Sanz, R., J. Zalewski, Pattern-Based Control Systems Engineering, IEEE Control Systems, Vol. 23, No. 3, pp. 43-60, July 2003

Actions as Activities and Activities as Petri nets

João Paulo Barros[1,2] ‡ and Luís Gomes[1]

[1] Universidade Nova de Lisboa,
Faculdade de Ciências e Tecnologia, Dep. de Engenharia Electrotécnica,
Campus da FCT, 2825 Monte de Caparica - Portugal
[2] Instituto Politécnico de Beja,
Escola Superior de Tecnologia e Gestão, Área Departamental de Engenharia,
Rua Afonso III, n. 1, 7800-050 Beja - Portugal
{jpb, lugo}@uninova.pt

Abstract. This paper proposes ways to view the rich semantics of actions as activities. This is achieved by unfolding the invoked activities and the implicit joins and forks. The unfolding offers an explanation for the semantics of actions by activities and also allows a simple translation from an activities subset to a well-known Petri net class. One example of an activity diagram and the respective Petri net is included.

1 Introduction

Contrary to UML 1.x, the UML 2.0 Activities packages are autonomous to statecharts [1] and explicitly relate the respective flow model with Petri nets: "Activities are redesigned to use a Petri-like semantics instead of state machines." [1, page 292]. Although one author has advised that this statement is "only a metaphor for flow modeling without implying a complete mapping to Petri nets" [2] we believe this metaphor can be made concrete to the point of being useful. This paper, takes the analogy further by presenting ways to view the semantics of actions as activities amenable to be translatable to the well-known Place/Transition nets semantics.

Activities present significant similarities to Petri nets, namely the use of a token-based semantics and the support for forks and joins. This makes Petri nets a natural choice for the formalisation of well-identified activities subsets. This formalisation has two distinct but related applications and associated advantages: (1) to improve the activity specification by explicitly relating activity classes to well-known Petri nets concepts; (2) to allow a simple translation from a subset of activity diagrams to Petri nets and also a reverse translation, from Petri nets to the same activity diagrams subset.

We strongly believe that the new proposal for activities can significantly widen the use of activity diagrams. Included in their extremely rich semantics, the support for all common Petri nets concepts is especially noteworthy

‡ Work partially supported by a PRODEP III grant (Concurso 2/5.3/ PRODEP/2001, ref. 188.011/01).

as it makes activity diagrams amenable to several important areas where Petri nets have been traditionally applied. These include critical systems development, namely, embedded systems, hardware design, telecommunications, and manufacturing systems. Moreover, activity diagrams benefit from the additional advantage of being included in a UML environment. The comparison with Petri nets also forces a classification in local and non-local behaviours. This separation is especially important when implementation concerns are at stake. Namely, the exclusive use of local behaviours can be of extreme importance for some parts of the system, due to the known difficulties in implementing the synchronous models in distributed and heterogeneous software/hardware systems.

As already stated, the paper uses Place/Transition nets and it is a known fact that Place/Transition nets are easily extendable to high-level Petri nets, namely to Coloured Petri nets [3]. Yet, due to space limitations and also to avoid cluttering the paper with additional notations, we restrict ourselves to low-level nets. It should be clear, though, that high-level tokens, and related notational extensions, could be readily added to the used Petri nets. This makes the model more suitable for embedded system applications, which, besides the reactive part, also include data processing capabilities and real-time constraints.

2 Actions as activities, and activities as Petri nets

In this paper, we restrict to intermediate activities but without activity creation and destruction, signals, and token termination across the activity or part of it. Activity termination includes termination of all the token flows inside the activity. This "termination of tokens" across the activity, or part of it, is sometimes called abortion, and can not be specified by a Petri net due to its implicit non-local effect. Yet, it is important to note that activity destruction, per se, is only important if system resources have to be freed. If not, the resources for activity implementation can remain available waiting for new tokens. For some critical systems, for example if implemented in some hardware programmable devices such as FPGAs, that static allocation of resources is probably preferred, as one cannot afford the requirements (or the associated risk) of a dynamic invocation implying dynamic memory allocation, the use of a call stack, or both. If there is no real activity termination, tokens can always flow in and out of any activity. This can be implemented by relying on the exclusive use of stream parameters. Yet, there is no real need for that restriction: we allow the use of non-stream object nodes and control nodes. As the activities do not really start or end, but always exist, we will see the activity start pre-conditions as one possible *token reception* semantics, and the activity end pre-conditions as one possible *token emission* semantics. The other possible token reception and token emission semantics are offered by stream input and stream output pins, respectively.

Next we present the paper's main contribution: how actions can be seen as static activities, making them amenable to a translation to Petri nets. Note that due to major significant space restrictions, we do not present with the desirable detail the mapping, from activity diagram nodes and edges, to Petri

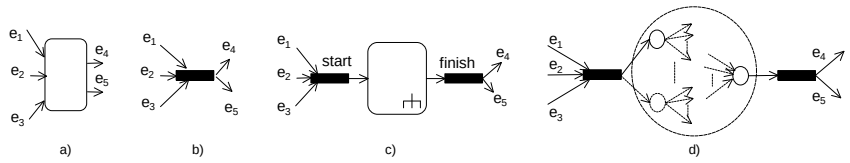


Fig. 1. a) One action; b) and c) two possibly equivalent activity diagrams; d) Hierarchical Petri net for activity diagram in c).

nets constructs. As a minimal, but hopefully sufficient, information we only state the following simple but important mappings: object nodes are made places; forks, joins, initial nodes, activity final, and flow final nodes are transitions; edges are made transitions with one input and one output arc; and, for the sake of simplicity, hyperedges are not considered.

Actions Actions are "(...)not further decomposed within the activity." but "(...)may be complex in its effect and not be atomic" [1, p. 283]. We also know that the action execution implies, with a few exceptions, an "implicit join" and an "implicit fork" [1, page 281]. These are applied to the input and output pins, respectively. Fig. 1a illustrates the definition of an action connected, exclusively, to control edges. The simplest activity diagram for an atomic and instantaneous action is given by simply joining the implicit join with the implicit fork as illustrated in Fig. 1b. Yet, the semantics of actions clearly points to the possibility of a non-instantaneous execution: "An action continues executing until it has completed." [1, page 281]. This suggests that an action is better defined by a *start* join and a *finish* fork with an arbitrary activity diagram in between (see Fig. 1c). As we also know that an action can be a call behaviour action, which may reference an activity definition, we propose to see an action as a higher level representation for an implicit join, an implicit fork, and an *implicit activity*.

In the Petri net domain, the call behaviour action corresponds to a macro place. A macroplace gets its name from being a subnet connected to transitions in the supernet, just like a Petri net place (see Fig. 1d). The Petri net has a begin and a finish join/fork pair (in Petri net terms, transitions with input and output arcs). The start transition models the possibility of multiple *InitialNode*s inside the called activity, the same is to say, the possibility of multiple initial token flows.

Parameters If the action has non-stream input (output) pins, all must have tokens for the action to start (finish). This is defined by including the pins in the input (output) of the implicit join (fork). Note that pins must match activity parameters. Action input pins are needed for an action to start, and action output pins are needed for an action to finish (see Fig. 2). Note also that the join and fork are now join/fork pairs, or transitions in Petri nets terms. The equivalent Petri net is similar enough to be readily obtainable from the equivalent activity diagram in Fig. 2b.

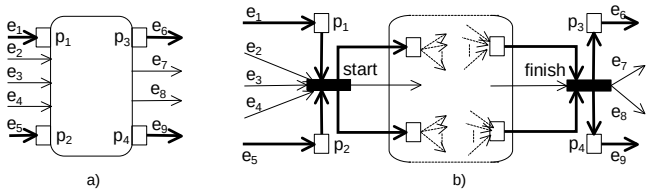


Fig. 2. a) Action with input and output pins and b) the respective activity diagram.

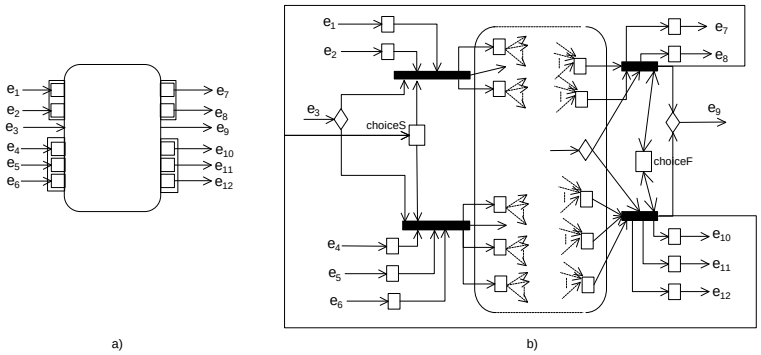


Fig. 3. a) An action with two input parameter sets, two output parameter sets (the action in [1, page 354] but with additional input and output control edges; b) the corresponding activity diagram.

Parameter set An action calling an activity can group its *Pin*s in *ParameterSet*s: "A behaviour with input parameter sets can only accept inputs from parameters in one of the sets per execution. A behaviour with output parameter sets can only post outputs to the parameters in one of the sets per execution."[1, page 320]. Parameter sets can be defined by a generalisation of the idiom in Fig. 2b: each input (and output) parameter set has one associated implicit join (see Fig. 3b).

The mutual exclusion between parameter sets is explicitly expressed by a test on an object node with one token (*choiceS* and *choiceF* in Fig. 3b). The *choiceS* gets its token back from any of the finish fork/join pairs. The Petri net is readily obtained. The only significant transformation results from the duplication of edges to remove the decision and merge nodes.

Reentrant behaviours Fig. 4a shows how to implement a re-entrant behaviour for the implicit activity in Fig. 2a. To allow for orthogonal execution of token flows, from each invocation, one has to implement some way to distinguishing the tokens. This can be achieved by making the object node *re-entrance level* be a deposit of different key values: one for each possible parallel execution. The values are used to stamp all nodes of a given execution: when tokens pass the start join/fork pair one key value is passed to it and attached to all exiting

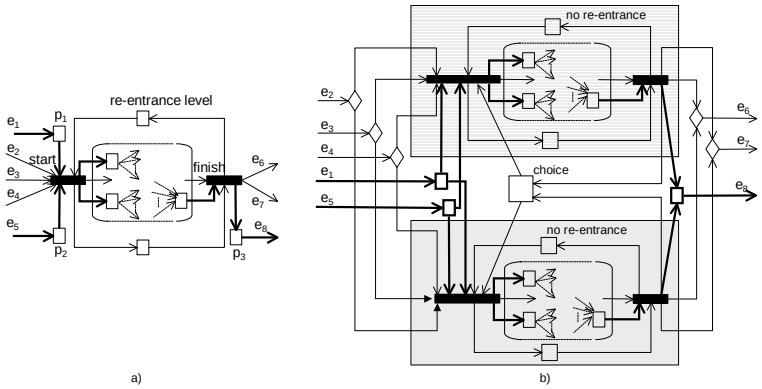


Fig. 4. a) Action calling a re-entrant behaviour with non-stream input and output parameters and b) Activity diagram for an action calling a behaviour with a re-entrance level of two.

tokens; the finish join/fork pair removes the value from the received data values and returns the values to the *re-entrance level* object node. This token stamping allows the called activity to distinguish the tokens of distinct invocations. In the Petri net this implies to the use of high-level tokens, that is, tokens with associated data.

Yet, the token stamp forces the called activity to be defined taking it into account. This can reduce the activity reusability. Also, token stamping is not "natural" for control tokens. In practice all control tokens must be made object tokens so that they are able to include the identifying stamp, and this means additional modification in the called activity. The alternative is the low-level version: to explicitly model the several possible behaviour executions (see Fig. 4b). This flattening of the activity invocations corresponds to the replacement of some textual specification by graphical notation.

Parameter streaming *Complete activities* support parameter streaming. This is specified by the characters *"{stream}"* near an action pin (see Fig. 5a). A stream input parameter does not have to be filled (by a *stream input*) for the action to start (for the implicit activity to be invoked). Symmetrically, a stream output parameter does not have to be filled for an action to finish. Again, this can be made more clear by an activity diagram for the action. Compared to the diagram in Fig. 2b, the stream input (e_5), and output (e_9) parameters are not required to pass-through the respective start join and finish fork (see Fig. 5b). They maintain the same connection with the action environment.

The activity diagram in Fig. 5b is not complete because it does not enforce the additional constraint that "All inputs must arrive for the behaviour to finish (...)" [1, page 353]. To explicitly model this constraint, we must know, for each stream input, if it has arrived as their arrival is a necessary condition to finish the behaviour. This is expressed by an additional fork for each stream input

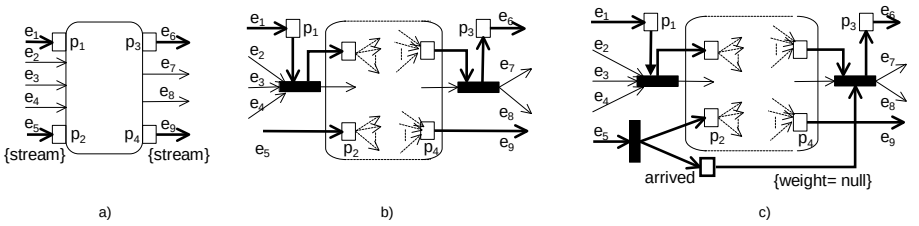


Fig. 5. Action with stream input and output pins and b) the respective, but not complete, activity diagram.

(see Fig. 5c). This fork puts in an additional object node a token signalling the arrival of the respective stream input. The finish join/fork pair has to retrieve at least one token from all *arrived* places. Notice that we had to use an edge with null weight. According to the UML specification "A null weight means that all tokens at the source are offered to the target." [1, page 294]. In Petri net terms, the edge with null weight is a maximal step arc.

Finally, as Petri nets are not able to model abortion they cannot support exception parameters, which "abort all flows in the activity".

3 An example of a *locally behaved* activity

All the classical concurrency problems are potentially good examples of the effectiveness of Petri nets for concurrent systems modelling and analysis. Yet, due to space restrictions, we choose to present an abstract activity diagram (see Fig. 6) illustrating the discussed concepts. The equivalent Petri net is shown in Fig. 7. Notice that three pairs of transitions are enclosed in a rectangle. Each rectangle is mapped to one transition: this means that the transitions inside the rectangle are merged into a single transition. Also notice the edge duplications resulting from the elimination of the decision and merge nodes. Each activity a is translated to a start transition s_a (if it has non-stream parameters or input control edges); an implicit activity ia_a; and a finish transition f_a (if it has stream output parameters or output control edges). The implicit activity is a macroplace; nameless transitions correspond to activity edges in the activity diagram; and nameless places are macroplaces' interface nodes.

4 Conclusions and Future Work

The paper contributes to clarify the relations between the semantics of actions in terms of activities, and between activities and Petri nets. This allows a centralise view, in a succinct and graphical way, of semantic data spread across several different pages in the UML specification. Additionally, the clear distinction between local and non-local behaviours, can also be seen as a guide for a disciplined use of the extremely rich concepts embedded in the activities specification.

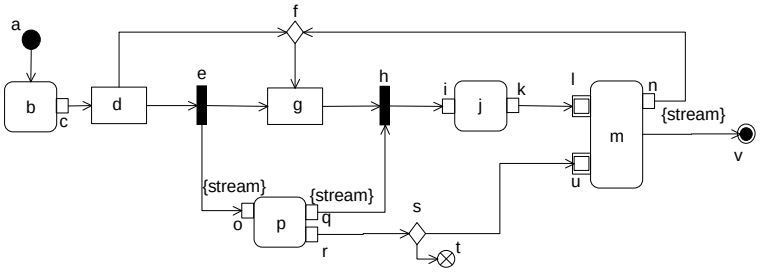


Fig. 6. An activity diagram.

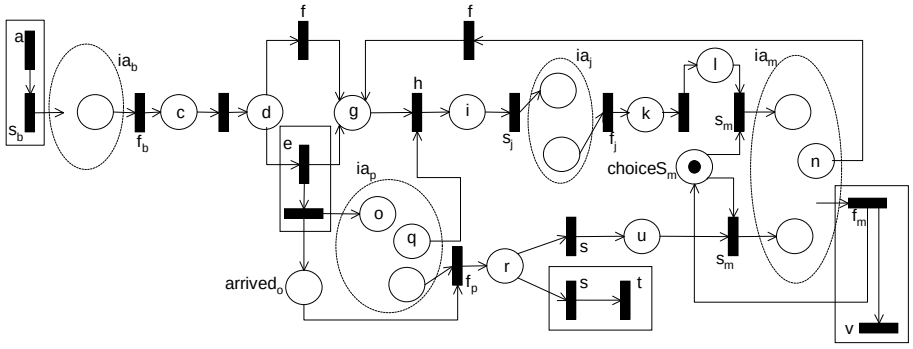


Fig. 7. Petri net equivalent to the activity diagram in Fig. 6, showing the correspondent elements.

Future work will include the validation of the presented Petri nets based equivalences by larger examples where a Petri net tool can be applied. Those examples, and the respective translations to Petri net models, should bring additional insights into the relations between both semantics. Especially important is the formalisation of the now informally presented mapping, as it will allow the construction of a translation tool. The use of a Petri net extension amenable to the specification of non-local behaviours is another interesting aspect deserving further research.

References

1. OMG: Uml 2.0 superstructure specification. `http://www.omg.org/cgi-bin/doc?ptc/03-08-02` (2003) version 2.0. Final Adopted Specification. OMG Adopted Specification ptc/03-08-02.
2. Bock, C.: Post to the u2p-issues mailing list. Available at `http://groups.yahoo.com/group/u2p-issues/message/125` (2003)
3. Jensen, K.: Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use - Volume 1–3. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, Berlin, Germany (1992–1997)