Can Formal Methods Cope with Software-Intensive Systems

27.05. - 01.06.2001

organized by

S. Jähnichen(Technical University, Berlin),
J. Kramer(Imperial College, London),
M. Lemoine(CERT, Tolouse),
M. Wirsing(Ludwig-Maximilians-University, München)

Preface

During the last years practical Software Engineering techniques are used more and more to conduct systematic and rigorous development of large software systems. UML has become the standard notation for guiding and documenting Software Engineering projects. CASE tools of today offer not only the UML notation but also are able to generate code templates and to support round trip engineering between class diagrams and program code. However, used in practice they do not support well the early phases of software development; they still lack analysis and validation methods for requirements and design specifications which are easily connected to the implementation phase.
Formal techniques have undergone a steep development during the last years. Based on formal foundations and deep theoretical results, methods and tools have been developed to support specifications and design of software systems. Model-based and algebraic specifications, abstract state machines, CSP and CCS, temporal logics, rewriting techniques, finite automata, model checking and many other formalisms and verification techniques have been applied to non-trivial examples and are used in practice e.g. for the development of safety critical systems.
Several case studies have been proven to be useful for validating and evaluating formal software development techniques. Case studies tackle the development in the small such as the production cell, the steam boiler and the memory cell. What is missing is a comparison of the development in the large. How do known formal techniques scale up? How do they cope with aspects such as architecture, component ware, distribution, mobility reconfiguration?
The aim of this workshop was to contribute to the field of Experimental System Engineering by proposing a case study for system development which allows one to compare different formal techniques in their abilities to specify, design, analyze and validate large software-intensive systems. The case study addresses the actual problem of controlling autonomous trains and systems and contains features such as local control in a distributed system, synchronous and asynchronous communication, heterogeneous components, and optimization problems.
During the workshop the solutions for the case study was presented and discussed by the participants. Also related work on formal and semi-formal approaches to system development was presented.
Our gratitude goes to the scientific directorate of Schloss Dagstuhl for giving us the possibility of organizing this workshop. However, the workshop would not have been possible without the help of the friendly and efficient staff of Schloss Dagstuhl. Our sincere thanks go to all of them.

The organizers
Martin Wirsing
Michel Lemoine
Stefan Jähnichen
Jeff Kramer

Contents

The BART Case Study
Victor Winter **5**

A Petri net Based study of the Bay Area Rapid Transit District
Fabrice Kordon **5**

BART system revisited: from UML to Z
Michel Lemoine **6**

Desires called for Streetcars
Adriaan de Groot **7**

How Process Formalisms Can Help Formal Methods to Cope with Complex Systems
Leon Osterweil **8**

Integrating Real-Time Requirements into Formal Developments
Werner Stephan **9**

Some formal methods can cope with software-intensive systems, IF ...
Egon Börger **10**

Representing Hierarchical Automata in Interactive Theorem Provers
Steffen Helke **10**

Two applications of LTSA
Jeff Magee **11**

FLAVERS: A finite state verification approach applicable to software systems
Lori A. Clarke **12**

Formalizing Reference Semantics for Object-Z
Florian Kammüller **12**

Constructing Decision Procedures for Domain-Specific Deductive Synthesis
Steve Roach **13**

Generating Specifications from a Family of Formal Requirements
Jan Bredereke **15**

Confidentiality-Preserving Refinement
Thomas Santen **15**

The Formal Semantics of the Integrated Formalism RT-Z
Carsten Sühl **16**

Improving Precision of Use Cases by Formally Specifying Operations
Alfred W. Strohmeier **17**

Formal Methods versus UML's OCL: Is there really any deep difference?
Martin Gogolla **19**

Integration of Object-Oriented Design and Formal Verification
P.H.Schmitt **21**

Requirements Documents as a Means of Communication:
Proposal and Open Questions
Barbara Paech **21**

From Scenarios to Behaviour Models and Back
Sebastian Uchitel **24**

"Xtreme" Specification and Design
C. Choppy **25**

The AutoFocus Development Process –
Some Afterthoughts on the BART Case Study
Bernhard Schätz **26**

Formalizing security properties for mobile systems
Jorge R. Cuellar **26**

A few naive ideas for the stepwise design of train control systems
Michel Sintzoff **27**

Model-Checking UML State Machines and Collaborations
Stephan Merz **28**

Methods and Experiments
Stefan Jähnichen **29**

Report on a Discussion on The BART Case Study and Used Notations
Martin Gogolla **30**

The BART Case Study

Victor Winter
Sandia National Laboratories, Albuquerque

In this talk I will give an overview of the Bart case study. I briefly discuss the relationship between Sandia National Laboratories and the company contracted to develop the BART train control system, a relationship that provided a unique opportunity to create the BART case study.
I then give an overview of some key issues in BART (oscillation, worst-case stopping profile, positional uncertainty, etc.), followed a discussion of some of the consequences resulting from modeling this system (e.g., train behavior) in discrete versus continuous terms.
This is followed by an overview of a refinement-based approach, developed by the High Integrity Software (HIS) team from Sandia National Laboratories, that was used to solve a variation of the problem described in the case study.

A Petri net Based study of the Bay Area Rapid Transit District

Fabrice Kordon
Universit Pierre et Marie Curie, Paris

joint work with:
I. Mounier and E. Paviot-Adet

We extracted from the BART case study the train speed controller system located in the station control computer. To have a progressive modeling and proof strategy, we divided the problem into three steps:

1. Pb1: a train has to go from station to station and stop at stations with a null speed.

   We modeled a circular railway where a train is circulating. The initial state of the system corresponds to a train stopped in a station. It then leaves the station and has to go as fast as possible to the next one. Several transitions in the Petri net model lead to dead states and correspond to unexpected situations: the train misses the station, the train stops before a station. We proved the model is live and therefore, that the strategy is correct.

2. Pb2: two trains do never collide.

   We modeled a section of railway where a crazy train having a random behavior (without backward movement) is followed by a second train. The second train has to behave in order to remain N segments (or distance units) away from the first train. When the crazy train reaches the end of the path, the model stops. As for the first model, an *accident* transition stops the model and is fired if the second train gets into the first one. We show that there is only one terminal state in the model: it corresponds to the exit of the crazy train.

3. Pb3: Pb1 + Pb2, we want to check if the merging of the two strategies does not introduce problems.

   We had no sufficient time to work on this problem but the proof strategy is similar to the one of Pb1. The crazy train is replaced by a "normal" train going from one station to another one. So, we will have to prove that this model is live.

We manually built our Petri nets according to a design/modeling strategy we aim to implement by means of tools. We observed that the problem size (Petri net complexity as well as the underlying reachability graph) remains reasonable since the control problem is deterministic for a given train. We also experimented satisfactory ways to model the complexity of physical mechanisms detailed in the case study by means of "computation tables" located in places.

BART system revisited: from UML to Z

Michel Lemoine
CERT, Tolouse

joint work with:
Gervais Gaudiere

Managing the complexity of Software Intensive Systems needs, before any development, to well understand what are the Requirements. It is unfortunately obvious that the people developing a system are not the people who will use it. Thus, in order to avoid many misunderstandings between end users and developers, it is mandatory to redesign all the requirements document in such a way that any question a developer may ask has an answer in the revisited System Requirements Document (SRD).

In this talk we present a 2 main steps methodology that helps getting the SRQ better structured, and free from removing any ambiguity and inconsistency.
The 1st step aims at:

- developing a semi formal model (using the UML notation) which is completed by a formal description (using the Z notation) of the class diagram
- validating the static model, i.e. the class diagram.

Because during this 1st step many ambiguities, uncertainty are detected and not yet solved, it is necessary to redevelop the SRD. This is the purpose of the 2nd step that suggests:

1. an initial phase in which a stable kernel is designed (according to UML notations) and validated (with respect to the Z method)
2. an iterative phase during which increments are designed and added after they have been validated (design and validation are done in the same way as it is for the stable kernel)
3. a termination phase during which all the semi formal and formal documents are eventually translated and merged into an informal one which is considered as the revisited SRD.

The talk illustrated the way this methodology has been applied on the BART system, and the lessons we have learnt from this experiment.

Desires called for Streetcars

Adriaan de Groot
Katholieke Universiteit Nijmegen

When building a controller for some given system — in this case study we are given a train that displays considerable autonomous behavior — it is essential to know what the system to be controlled *does*. Without that knowledge we cannot know how the system will react to the controller we build, and we may end up building the wrong controller.
To combat this risk we consider it essential to *analyse* and *formalize* the informal specification of the given system. Using a combination of methods and techniques we attempt to formalize as much as possible of our knowledge of the system. We then analyse this formalization, looking for incompleteness and inconsistency. If we find inconsistency or incompleteness in the formalized specification, we must return to the authors of the informal specification to ask questions such as: Have

we understood this correctly? What is the behavior in this case? Was this intended?
We describe here the results of our analysis of the BART / AATC train using a combination of UML, Statecharts and hybrid automaton notation. The analysis brings to light many ambiguities in the informal specification that must be resolved before we can even begin to design a controller that has a serious chance of doing "the right thing."

How Process Formalisms Can Help Formal Methods to Cope with Complex Systems

Leon Osterweil
University of Massachusetts

This talk suggests the importance of formalizing the processes of formal methods. It is most important that formal methods be carried out precisely and completely, and that evidence of their faithful execution be made available. Moreover, it is important that the efficacy of these processes be verified. The talk addresses the problems in devising a process definition formalism that is sufficiently rigorous, precise, clear, and comprehensive. The Little-JIL process language is introduced as an example of such a language. Examples of application of Little-JIL to formalization of software processes are given, and an approach to verification of such processes is sketched.

Integrating Real-Time Requirements into Formal Developments

Werner Stephan
DFKI, Saarbrücken

joint work with:
Georg Rock and Andreas Nonnengart

We present an approach where specifications of real-time systems that are tailored for an efficient requirements analysis can be linked to separately given design specifications which may serve as a starting point for a refinement process.

Requirements specifications capture the behaviour of (real-time) systems in a comprehensive and abstract way. They provide a global view on the entire scenario thereby defining what behaviour has to be enforced by the control system. In general there might be several such views. We use hybrid automata for requirements analysis at this level. Given a hybrid automaton describing, for example, the desired behaviour of trains in the BART case study, one may derive particular properties as, for instance, the property that trains do not get too close to each other.
Design specifications describe how a certain architecture — possibly consisting of several components - solves the problem. The specification of the actual (distributed) system has to be augmented by additional specification units:

- a specification of assumptions about the environment (which is given explicitly in our approach),
- a clock that defines global time,
- a scheduler that defines temporal constraints for the system components and the environment, and
- a unit that filters out the visible states of the entire scenario.

The two sides serve different purposes and they are linked by a formal interpretation that maps visible computation steps to notions used in the requirements specification. The loose coupling - where the formal concepts used in the requirements analysis are not directly associated with technical concepts in the design - allows for flexibility in the (formal) development process.
The whole development takes place in the Verification Support Environment (VSE), a case tool for formal developments based on abstract data types and temporal logic (TLA). Hybrid systems are integrated as they are translated into the temporal logic of VSE such that the validity of properties is preserved. The translation introduces a discrete time scale that is also used for the clocks within design specification.

Some formal methods can cope with software-intensive systems, IF ...

Egon Börger
Universit degli Studi di Pisa

This talk was triggered by the discussions in the Seminar. It surveys a) some of the outstanding problems of so-called Formal Methods, b) some of the properties rigorous design and analysis techniques must satisfy to be practical, and c) a successful use of a rigorous method in a real-life case study, namely the definition, verification and validation of the programming language Java and of its implementation on the Java Virtual Machines, using Abstract State Machines.

References

[1] R. Staerk, J. Schmid, E. Boerger, "Java and the Java Virtual Machine - Definition, Verification, Validation", Springer-Verlag, June 2001

Representing Hierarchical Automata in Interactive Theorem Provers

Steffen Helke
Technical University, Berlin

joint work with:
Florian Kammüller

Model Checking is an algorithm that checks whether a finite state system satisfied a temporal property. One major drawback of most model checking approaches is the restriction to finite state spaces. Particularly data contained state systems are mostly implicitly infinite. Therefore, to use model checkers one has to first simplify the system that has to be verified. This problem can be overcome by embedding a formalism, e.g. statecharts, in a theorem prover and combine it with a model checker.
In this talk we present a formalization of statecharts. Hierarchical Automata represent a structured model of this formalism which we have formalized in Isabelle/HOL. The formalization is on two levels. The first level is the set-based semantics, the second level exploits the tree-like structure of the hierarchical automata to represent them using Isabelle's datatypes and primitive recursive functions. Thereby the proofs about statecharts are simplified. In order to ensure

soundness of this twofold approach we have defined a mapping from the latter to the former representation and proved that it preserves the defining properties of hierarchical automata.

References

[1] Steffen Helke, Florian Kammüller, Representing Hierarchical Automata in Interactive Theorem Provers, TPHOLs, LNCS, Springer-Verlag, 2001

Two applications of LTSA

Jeff Magee
Imperial College, London

LTSA is a labelled transition analysis/model checking tool. The talk described two applications of this tool. The first being modelling and analysis of "OpenFlow" transactional workflow schemas. Openflow schemas can be automatically translated into the input formalism required by LTSA and during the talk, a demonstration was given of how LTSA simulated executions can be animated against the original workflow schema. In addition, counter examples produced by analysis can be directly related to OpenFlow schemas. Scalability is achieved by using compositional analysis that works well in this context since the structure of the compositional analysis derives directly from the hierarchical structure of OpenFlow schemas.

The second application was modelling and analysis of a system consisting of active objects communicating by a reliable causal ordered communication channel. This is modelled as a queue for LTSA. The talk presented results on how scalability is achieved using partial order reduction techniques and presented some pointers as to the use of partial order reduction in a compositional analysis setting.

FLAVERS: A finite state verification approach applicable to software systems

Lori A. Clarke
University of Massachusetts

Finite state verification is an emerging technology for verifying properties about software systems. To date most approaches have been applied to hardware descriptions or manually created abstract models of software systems. FLAVERS, Flow Analysis for VERifying Systems, uses a very concise model of a software system that can be automatically derived and then selectively refined by the analyst as guided by previous analysis results. FLAVERS has been successfully applied to several software systems and shown to grow relatively slowly as the size of the system increases. This talk describes the software model used by FLAVERS, presents some experimental results, and discusses the importance of applying formal methods consistently throughout the software development process, from early architectural design through software maintenance.

Formalizing Reference Semantics for Object-Z

Florian Kammüller
Technical University, Berlin

joint work with:
Graeme Smith, SVRC, University of Queensland
Thomas Santen, Technical University, Berlin

Object-Z [3], the object oriented extension of the specification language Z, used to have a value semantics. However, to enhance the refinement to object oriented programming languages, the designers of Object-Z decided to switch to a reference semantics in which objects are represented not only by their states but rather as unique identities referring to a state.
Formalization of this reference semantics in the generic theorem prover Isabelle has a two-fold effect: it clarifies the semantics and provides a theorem prover for Object-Z. We extend a mechanization of the earlier value semantics in Isabelle/HOL [2] to the new reference semantics. This approach is based on the idea of distinguishing the internal and external effects [1] of class methods of Object-Z.

By an additional translation process it is possible to generate from an Object-Z specification a representation that enables modular reasoning. The translation of the internal effects of methods enables reasoning about the part of a class that does not refer to other classes.
The external effects are encoded as a recursive Isabelle datatype of messages. The notion of classes is built simply by extending the encoding of classes of the value semantics by such messages; the object level is represented by environments – functions from identities to object values. Once a specification is completed, the external effects of a specification represented by the messages can be evaluated by a formulation of the effects as an inductive Isabelle definition over rules that define how the messages affect the object environment.

References

[1] A. Griffiths, A Formal Semantics to Support Modular Reasoning in Object-Z, PhD thesis, University of Queensland, 1997

[2] T. Santen, A Mechanized Logical Model of Z and Object-Oriented Specification, PhD thesis, Technische Universität Berlin, 1999.

[3] G. Smith, The Object-Z Specification Language, Kluwer Academic Publishers, 2000.

Constructing Decision Procedures for Domain-Specific Deductive Synthesis

Steve Roach
University of Texas at El Paso

One problem with domain-specific component-based software development is the cost associated with learning the functionality of the components and the need of the user of the components to be competent in both the domain and in the construction of software.
Amphion is a deductive synthesis system that generates software solutions to specifications given in an abstract, domain-specific language by composing components from a trusted component library. Users, typically domain experts, create declarative specifications in a graphical, domain-specific language. Graphical specifications are transformed into first-order logic and given to the fully-automated theorem prover, Snark. The theorem prover generates a proof and a functional program that satisfies the specification. This program is translated into the target language.

Amphion has a domain-independent architecture and is tailored to a specific domain through the creation of a domain theory. The domain theory describes the abstract specification language, the concrete implementation-level components, and the mappings between transformations at the abstract level and the concrete level. One Amphion application is in the domain of celestial mechanics. Domain experts can generate programs that solve problems related to the motion of planets and spacecraft. For example, a domain expert can specify a program to compute the solar incidence angle at a point on a planet under observation by a spacecraft-borne instrument. Programs generated by Amphion are currently being used by NASA scientists to plan observation sequences for the Cassini mission to Saturn.

Two problems with deductive synthesis systems are the combinatorial explosion in the search space of the theorem prover and the difficulty building and maintaining usable domain theories. While it is fairly straightforward to build a declarative domain theory, in practice, the domain theory leads to exponential growth in the theorem prover's search for a solution. Thus, only small specifications can be solved. The usual practice is to rewrite parts of the domain theory to force the theorem prover to select particular paths. This rewriting of the domain theory is difficult, time consuming, requires great expertise in theorem proving, and makes maintenance of domain theories difficult.

One approach to solving these problems with domain theory operationalization is to replace subsets of the domain theory with decision procedures. The tool, TOPS, is designed to automate the process of decision procedure creation. TOPS takes a domain theory as input and produces a modified domain theory and a set of decision procedures as output. The decision procedures decide formulae over subsets of the domain theory and generate terms corresponding to program fragments. TOPS automatically operationalizes domain theories, improving theorem prover performance and facilitating maintenance. TOPS has been applied to the NAIF domain theory. The performance of the theorem prover after the introduction of decision procedures was slightly better than the performance of the theorem prover after a year of hand-tuning by a theorem proving expert.

Generating Specifications from a Family of Formal Requirements

Jan Bredereke
University, Bremen

We are concerned with the maintenance of formal requirements documents in the application area of telephone switching. We propose a specification methodology that avoids some of the so-called feature interaction problems from the beginning, and that converts some more into type errors. We maintain all the variants and versions of such a system together as one family of formal specifications. For this, we define a formal feature combination mechanism. We present a tool which checks for feature interaction problems, which extracts a desired family member from the family document, and which generates documentation on the structure of the family. We also report on a quite large case study.

Confidentiality-Preserving Refinement

Thomas Santen
Technical University, Berlin

joint work with:
Maritta Heisel, University, Magdeburg
Andreas Pfitzmann, Technical University, Dresden

Dependable IT-systems of relevant size can only be built if all dependability attributes have a clear meaning. This meaning has to be consistent both with the common understanding of the people building and using the IT-system as well as with the tools they use and the development process they adhere to. Therefore, a formal meaning of all dependability attributes is needed which is compatible with the usual refinement process of systems engineering.
Our goal is to establish a formal meaning of security attributes. Confidentiality, integrity, and availability are seen as the generic properties of security. The relationship between the security attributes integrity and availability, and formal notions compatible with refinement is roughly clear. For terminating computations, integrity corresponds to partial correctness and availability corresponds to assured termination combined with sufficient computational resources to fulfill real-time requirements. Integrity and availability together correspond to total correctness and sufficient computational resources. For reactive systems, integrity

means that the defined processes satisfy certain required predicates, and availability corresponds to fairness and liveness combined with sufficient computational resources.
To date, the relationship between confidentiality and refinement is less well understood. This is our motivation to develop a formal notion of a confidentiality-preserving refinement. We develop a condition for confidentiality-preserving refinement which is both necessary and sufficient.
We specify systems by their behavior and a window. For an abstract system, the window specifies what information is *allowed* to be observed by its environment. For a concrete system, the window specifies what information *cannot* be hidden from its environment. A concrete system is a confidentiality-preserving refinement of an abstract system, if it behaviorally refines the abstract system and if the information revealed by the concrete window is allowed to be revealed according to the abstract window.
In describing the refinement relation between the more abstract and the more concrete system, we do not only define relations between abstract and concrete data and behavior, but we also consider probabilistic behavior of the involved systems. We derive a condition on these probabilities, which is both sufficient and necessary for preserving confidentiality, thus the refined system does not reveal more information – in the sense of Shannon – on the abstract data than permitted by the abstract specification.

The Formal Semantics of the Integrated Formalism RT-Z

Carsten Sühl
GMD FIRST, Berlin

I have presented an integration of the model-based specification language Z and the real-time process algebra timed CSP, called RT-Z. In particular, I have focused on the definition of its formal semantics.
After briefly discussing the need to integrate existing, complementary formalisms and possible approaches to achieve this, I have introduced the integrated formalism RT-Z by presenting its basic language features:

- the structuring operators provided by RT-Z to decompose the specification of large and complex systems,
- the internal structure of the basic units of the hierarchical decomposition of an RT-Z specification, called specification units, and

- the distinction between abstract and concrete specification units, which together support different levels of abstraction and therefore different development phases.

The main subject of this talk was the denotational semantics of RT-Z. Its definition consists of two sub-tasks: the definition of the meaning of the structuring operators, most importantly aggregation and extension, and the definition of the semantics of single specification units.
Concerning the definition of the semantics of the structuring operators, a transformational approach was pursued resulting in a mapping from the syntax of RT-Z including aggregation and extension to the syntax of single specification units.
Concerning the semantics of single specification units, I have restricted myself to concrete specification units. The definition of the semantics of concrete specification units consists of three stages associated with two intermediate semantic models and the final timed failures/states model, which successively bring together the semantic models of Z and timed CSP. I have also discussed the major achievement of the semantic model underlying RT-Z, namely its ability to cope with the concurrent execution of operations that work on different subspaces of the data state but which might nevertheless interact. In particular, I have explained under which conditions the concurrent execution of operations leads to a well-defined effect on the data state.

Improving Precision of Use Cases by Formally Specifying Operations

Alfred W. Strohmeier
EPFL, Lausanne

Currently in industry much of what would be loosely classified as system specification is performed with use cases. Use cases are popular because of their informal, easy to use and to understand style which caters to technical as well as non-technical stakeholders of the software under development. Use cases can be decomposed into further use cases, and therefore they are scalable to any system size. Moreover, it is possible to trace between subordinate use cases and the "parent" use case. Also, use cases have a wide spectrum of applicability: they can be used in many different ways, in many different domains, even non-software domains, making them a very versatile tool to have in one's development toolkit. On the other side, because use cases are informal descriptions, almost always

written in natural language, they usually lack rigor and are not an adequate basis for reasoning about system properties.
In our presentation, we look at bringing the benefits of behavioral specification techniques to main-stream software development by proposing the use of operation schemas as a supplement to use cases. An operation schema declaratively describes the effects of a system operation by pre- and postconditions using the Object Constraint Language (OCL), as defined by the Unified Modeling Language (UML). We illustrate the advantages of complementing use cases with operation schemas by an example of a multi-cabin elevator control system. Moreover, we look at how one gets from use cases to operation schemas, and thus propose a mapping from use cases to operation schemas.
More material on the case study dealt with in the presentation is available at: *http://lglwww.epfl.ch/˜sendall/case-st udies/elevator/multi-cabin_without-safety/index.html*

References

[1] Shane Sendall and Alfred Strohmeier, From Use Cases to System Operation Specifications, UML'2000 - The Unified Modeling Language: Advancing the Standard, Stuart Kent and Andy Evans (Ed.), LNCS no. 1939, 2000, pp. 1-15.

Formal Methods versus UML's OCL: Is there really any deep difference?

Martin Gogolla
University, Bremen

The talk explains the connection between the Unified Modeling Language UML and its Object Constraint Language OCL. It shows OCL features by considering the BART case study and giving a small abstract specification for it. This specification is animated by means of the UML Specification Environment USE developed at University of Bremen. Finally the talk discusses to what extent OCL is different from existing formal specification languages.

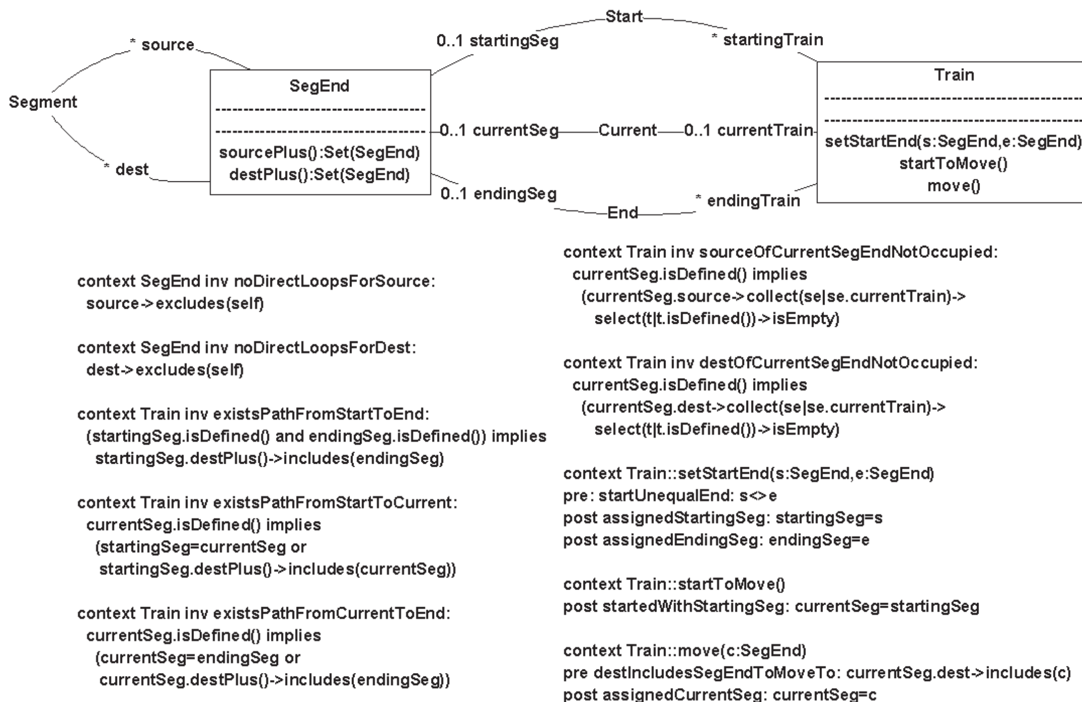


Figure 1: UML Class Diagram for Part of BART Case Study

This approach to the BART case study starts with the class diagram given in Figure 1. In the class diagram, tracks are described by objects of class `SegEnd` and an association `Segment` between `SegEnd` objects. Links in the association `Segment` represent directed edges with a source `SegEnd` and a destination `SegEnd` object. Trains are specified by a class `Train`. A journey for a train is determined by starting and ending `SegEnd` objects and a current `SegEnd` object giving the

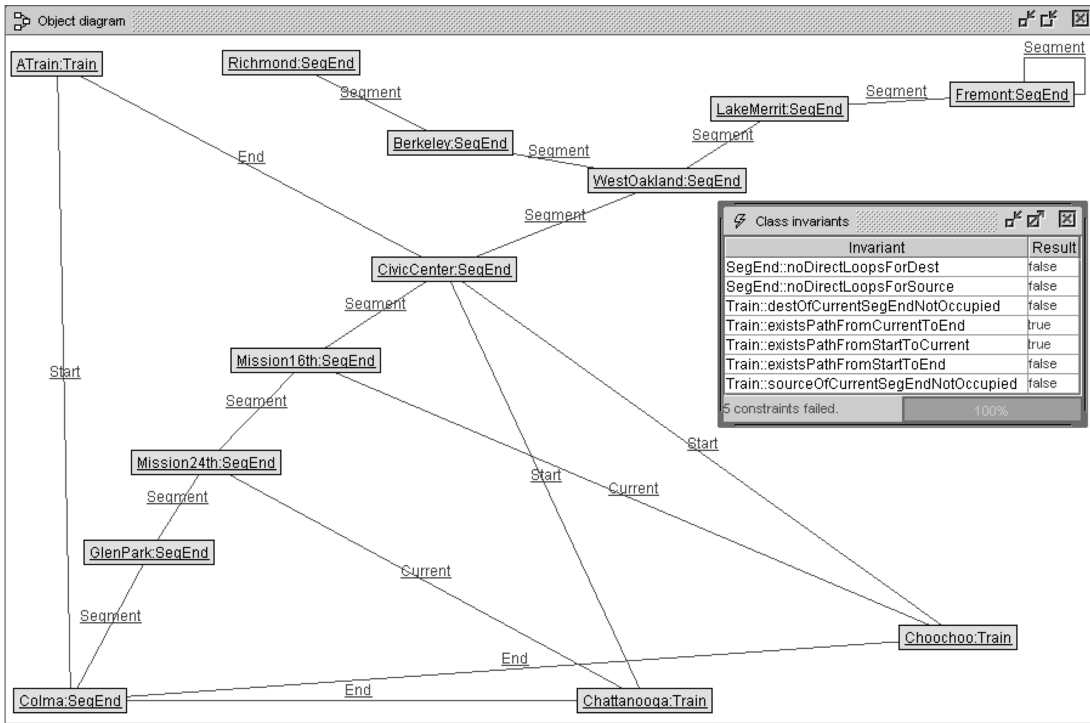


Figure 2: USE Object Diagram

current position of the train. The class diagram is further restricted by (1) invariants expressing operation independent properties which hold in all system states and (2) pre- and postconditions describing the operations in more detail. A snapshot of the USE animation with stations represented as `SegEnd` objects and `Train` objects is pictured in Figure 2. The window entitled “Class invariants” analyses the specified invariants and reveals that some of them do not hold in the shown system state. Double clicking a “false” invariant would open a further window explaining in detail the reason for invariant failure.

Integration of Object-Oriented Design and Formal Verification

P.H.Schmitt
University Karlsruhe

In the introductory part of this talk we present the KeY-Project, its overall goals, central paradigm and architecture. In the rest of the talk we concentrate on two details.
One way to use the KeY tool is to verify that a given Java program satisfies its spezification, given in the form of a UML/OCL diagram. The proof is constructed using an interactive theorem prover for dynamic logic. We thus need a formal semantics of Java. We quickly review the state of the art on this particular issue including our own approach, which provides Dynamic Logic proof rules for each Java language construct.
UML is established, so it seems, as the major OO-modeling language. OCL, on the other hand, though part of the UML standard, is used more reluctantly. Notation is not the problem. It is more the problem to decide what needs to be said and what should be the main concepts. We spent the greatest part of the talk to present our proposal to assist programmers in generating OCL contraints. The idea is to pigyback it onto the concept and use of "design patterns". We augment "design patterns" by constraint schemata which in the process of adapting the design pattern to a current UML model will automatically get translated into syntactically OCL expressions. The syntax of constraint schemata is a slight extention of OCL syntax.

Requirements Documents as a Means of Communication: Proposal and Open Questions

Barbara Paech
Fraunhofer Institute for Experimental Software Engineering

joint work with:
Erik Kamsties and Antje von Knethen

In the talk I present first ideas of the BMBF-funded project QUASAR (joint work with GMD FIRST) which aims at integration of requirements specification and quality assurance within automotive control software development. In particular, an appropriate structure of the requirements documents integrating natural

language, semi-formal and formal models has to be developed. This shall serve as a basis for

- systematic development steps between the documents
- early defect detection through inspection
- early development and use of test specifications
- change management based on traceability, re-inspection and regression testing.

Interviews with car manufacterers have realed the following problems with the current requirements documentation:

- mixture of system and software requirements in one document
- mixture of user (marketing) view of functionality and developer view (e.g. signals to/from sensors/actuators) in one document
- unclear responsibility distribution between manufacterers and suppliers
- inflexible association between functions and electronic control units already in the requirements documents.

To overcome these problems we suggest 5 levels of requirements documents, where each level captures a particular LEVEL OF ABSTRACTION of the problem to be solved AND where each level serves as a MEANS OF COMMUNICATION between different roles. The levels are an extension of the 3 SCR-levels presented by Connie Heitmeyer at Dagstuhl-Workshop 99241.
As a running example we use in the following the control of a seat position in a car which takes into account different user settings.

1. User requirements specification which captures the user view in terms of user input (e.g. determine settings, open door, push handle at the seat) and expected system reactions (e.g. position seat). This document is used for discussion between user / marketing and system engineers.
2. System requirements specification in terms functions between monitored (e.g. user, speed) and controlled variables (seat position) of the system (corresponding to SCR-relation REQ). This is the system engineers view of what the system should do without taking into account how the system can monitor or control the variables.
3. System specification in terms of functions between directly monitored and controlled variables. This reflects e.g. the fact that not the users themselves can be monitored by the system, but only the key typed in by the user.

Similarly, not the seat position as a whole can be controlled, but only components like the different angles of the seat components. This document is used in the discussions between system engineers and software engineers. It also specifies the concrete sensors and actuators being used.

4. Software requirements specification in terms of functions between software input variables and software output variables (namely, the representation of the signals to and from the sensors and actuators) (correspondig to SCR-relation SOFT). This document captures the software engineers view of what the software should do.

5. Software specification which extends the software requirements specification by an explicit distinction between device-dependent software and device-independent controller software (SCR-relation REQ). This document serves as input to the software implementors.

While these distinctions are a first step to describing single functions, there remain a lot of open questions:

- at what levels should functions be grouped
- at what level and how should dependencies between functions be described
- at what level should faults (e.g. battery low) be described
- how to deal with non-functional properties.

It seems that the question of which modeling formalism to use at what level is orthogonal. So e.g. SCR and statecharts can be used on level 2-5. However, we feel that SCR is more suitable for levels 2 and 3, while statecharts are more suitable for level 4 and 5.

From Scenarios to Behaviour Models and Back

Sebastian Uchitel
Imperial College, London

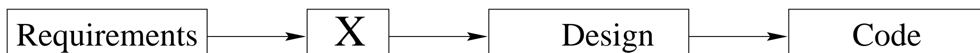
Scenario-based specifications such as Message Sequence Charts (MSCs) are becoming increasingly popular as part of a requirements specification. Scenarios describe how system components, the environment and users working concurrently interact in order to provide system level functionality. Each scenario is a partial story which, when combined with other scenarios, should conform to provide a complete system description. Our objective is to facilitate the development of behaviour models in conjunction with scenarios. Such models are complementary to scenarios. In addition to providing an alternative view, we believe that there is benefit to be gained by experimenting with and replaying analysis results from behaviour models in order to help correct, elaborate and refine scenario-based specifications. We aim to provide tools for synthesising behaviour models from scenario specifications, and for constructing scenario specifications from behaviour models. In addition we aim to provide mechanisms for providing feedback of behaviour analysis results in terms of scenarios. Scenario-based specifications describe system decomposition and system behaviour. However, it is not always possible to build a set of components according to the specified decomposition that provides exactly the described system behaviour. Implied scenarios may appear as a result of unexpected component interaction. Implied scenarios are the result of an inconsistency between system decomposition and system behaviour and are not an artefact of a particular scenario notation. Rather, they are the result of specifying the global behaviours of a system that will be implemented component-wise. Implied scenarios can be detected by building behaviour models and drive users to complete their specifications. Thus, they are an example of the benefit that can be gained from scenarios and behaviour models. In this talk, we discuss synthesis of behaviour models that describe the closest possible implementation (i.e. minimal with respect to trace inclusion) for a scenario specification. The scenario specification language is a subset of the ITU MSC standard Z.120 that includes basic and high-level MSCs. We also discuss a technique (implemented and integrated into LTSA) for detecting and providing feedback on the existence of implied scenarios. In addition, we show how this feedback can be used to obtain more complete scenario specifications through the use of constraints or negative scenarios. Finally, we discuss how reverse engineering of MSC specifications from behaviour models can provide feedback of relevant design issues to stakeholders.

“Xtreme” Specification and Design

C. Choppy
Universit Paris-Nord

joint work with:
Maritta Heisel, University Magdeburg

From requirements to design, this “X” box should provide means for abstraction and structuration so as to help the understanding and the analysis of a problem.



We propose two steps of specification to this end, and address the issue of how to help their obtaining. We suggest to use Michael Jackson’s problem frames [1] to help obtaining the (gross) structure of the specification (we believe that the process of selecting an appropriate problem frame will help starting to analyze the problem). Then the choice of appropriate architectural styles is suggested to provide further structuration in the specification. Both choices should be guided with some heuristics and criteria that may rely both on functional and non functional requirements. We address these through working on case studies as well as the issue of “matching” a problem frame to an architectural style. Future research involves working on other case studies such as the BART system, on the use of design patterns for further structuration, ...

References

[1] Michael Jackson, Software Requirements and Specification: a lexicon of practice, principles and prejudices, Addison-Wesley, 1995

The AutoFocus Development Process — Some Afterthoughts on the BART Case Study

Bernhard Schätz
Technical University, München

To make formal methods accessible for an industrial-strength engineering process, they have to be integrated into the development process as smoothly as possible. AutoFocus supports several aspects of such an integration on different levels: it offers precise view-based description techniques; syntactic, user-defineable consistency rules help to check for syntactically detectable defects like explicitly stated non-determinism; classical refinement steps (structural refinement, behavioural refinement) can be verified using model checking or interactive theorem proving; code as well as test cases for the implementation can be generated from executable specifications.
Considering the BART case study, AutoFocus offers a formal development process supporting a manageable and precise description of the controller, structural refinement steps to incrementally add requirements according to their priority (respecting worst case bounds, dealing with position uncertainly, reducing unnecessary bounds, etc.), and the refinement of states (avoid mode changes). While thus it is possible to support a formal software engineering process two topics of the system engineering process (discretization from a time-continuous to an event-driven model, performing reliability analysis) are not addressed.

Formalizing security properties for mobile systems

Jorge R. Cuellar
Siemens AG, München

The work discusses some work done on formalizing requirement and verifying security and safety properties for GSM, UMTS and mobile IP. For the internet, the given (very poor) trust relationships together with strong restrictions on scalability and on the performance or computational power of the protocols, handhelds and servers prohibit "secure" solutions, that is, solutions that resist with a very high probability attacks with the current abilities and computing power of potential attackers. Even when the protocols are believed to be secure (say, in the UMTS case), the case where an attack is successful is considered as

possible scenario to which the protocol has to react in a "correct" manner. For instance, blocking many thousnds or millions of mobile sets is not the correct behavior even in the case that an attacker is able to get into a highly trusted system for several thousand times. Our interest is how to formalize the security properties of those systems.

A few naive ideas for the stepwise design of train control systems

Michel Sintzoff
Universit catholique de Louvain

We propose to organize the design process in the following phases, in an outward spiralling mode: elaboration of requirements; design of system specifications; choice of an architecture; system reifications and refinements; enrichments and alternatives.
We begin the requirements analysis by investigating the problem domain, as suggested by M. Jackson. This domain concerns transportation systems. Generic properties are expressed, and then specific aspects about trains. In this context, the principal goals, e.g. safety, performance and reliability, are stated. They are systematically refined into more technical objectives, as proposed by A. van Lamsweerde.
The system specifications are deduced from the domain properties and the objectives, so that the latter are entailed by the domain properties and the system specifications. Following J.-R. Abrial, the system specifications are immediately modelled in a simple, abstract, idealized kernel design. This model is organized as an agent system, made of a logical view (properties, constraints), a static view (data, interfaces), and a dynamic view (input-output relations, differential equations). The static and dynamic view guarantee the logical view. This abstract model could be expressed in the style of the abstract machines of the B method, extended with continuous dynamics; related formalisms are abstract state machines, hybrid automata, TLA, Z, VDM and, who knows, UML with OCL.
Since the system is a control system, its architecture is based on classical control theory, as specialized for computer control systems; cfr E. Sontag and D. Parnas among others. The system is thus organized as a game between two players, namely the "plant", viz. the given physical environment, and the "controller", viz. the control system to be designed. The players communicate through ideal interfaces: the measurement and command informations are transmitted contin-

uously and instantaneously. The optimal control policy is to be derived mathematically from the plant dynamics and the desired objectives.
The basic control architecture is refined in steps yielding a discrete-time concurrent system. First, communication delays are introduced by use of local variables and communication channels; additional invariants ensure the validity of this refinement. Secondly, the space-time geometry is discretized by introducing track segments and time slots, viz. half-second periods. Again, auxiliary invariants ensure the approximate data define safe information envelopes. To maximize performance, these envelopes must be as tight as feasible.
Additional refinements must integrated. For instance, mutual exclusion on critical track sections is reified by gate semaphores and sensors. More complete designs are obtained by stepwise enrichment of the above skeleton, for instance by considering goals such as reliability, and by adding components such as stations.
Alternative designs should be studied. For instance, the control system could be made fully distributed: each train would directly monitor the dynamics of trains in the neighborhood, and would determine its own dynamics accordingly. This would yield a cooperating, communicating, distributed control system, and could also be used for future air and road control systems.
Clearly, the distance between the above considerations and a satisfactory system design is huge.

Model-Checking UML State Machines and Collaborations

Stephan Merz
Ludwig Maximilians University, München

joint work with:
Alexander Knapp and Timm Schäfer

The Unified Modelling Language provides different kinds of diagrams to specify the dynamic behavior of object families: interaction diagrams assert that certain executions (sequences of events) are possible, whereas state machines describe the behavior of objects operationally. We report on a tool for checking the consistency of these complementary views. Specifically, we compile state machines into PROMELA models and collaboration diagrams into sets of Büchi automata ("never claims"), and then call upon the model checker SPIN to analyze the resulting model. If the behavior expressed by the collaboration diagram is possible, SPIN generates a "counter example" that is then replayed to the user. Unlike previous work on model checking variants of Statecharts, our compiler is

not based on a translation to hierarchical automata, but on a simple operational model where a state machine is represented by processes that correspond to the individual states, plus a process each for dispatching events and firing transitions. State explosion is avoided by having processes that constitute a single state machine communicate synchronously, and by atomic execution of each run-to-completion step. The compiler also has a backend for generating Java code, based on the same overall structure, that can be used for prototyping and as a basis for implementation. Our tool is freely available at *http://www.pst.informatik.uni-muenchen.de/projekte/hugo/*.

Methods and Experiments

Stefan Jähnichen
Technical University, Berlin

The talk gives some ideas on what a method should be and more important, what a method should contain. Although there are good reasons to apply systematic development techniques in a systematic manner, two examples are given for which even in an academic environment, the processes were not applied systematically and consistently. The reasons for this observation are found in the complexity of the developments and in the difficulty to identify the requirements correctly. The two examples are a complex scheduling problem for a large railway network and the control system for a new satellite. It is expected that the further development of the examples is accompanied by a formal treatment to improve reliability, security, and fault tolerance.

Report on a Discussion on The BART Case Study and Used Notations

Discussion Chair:
Martin Gogolla

Discussion Participants:
Jan Bredereke, Adrian DeGroot, Fabrice Kordon, Michel Lemoine, Stephan Merz, Emanuel Paviot-Adel, Bernhard Schätz, Michel Sintzoff, Werner Stephan, Alfred Strohmeier, Victor Winter

The discussion took place on the second day of the workshop. On the first Workshop day four proposals for handling the BART case study had already been presented. As a starting point we considered the following questions:

- Were the used formalisms appropriate for the BART case study?
- What are the strengths of the solution?
- What are the shortcommings of the solution?
- On the other hand, is the BART case study appropriate for the used formalisms?
- Should we also consider other aspects of the BART scenario like delays of trains?
- What about reconfiguration aspects, for example, reconfiguration of schedules?
- Do we have to modify the BART case study to meet our needs?

After these starting questions, the discussion focused on the following three issues.

Issues concerning the case study text:

- Should we criticize the content of text?
- Is the case study text really a requirement document?
- The case study concerns implementation of an increment of a system. This was seen as a valuable point of the case study task.

- Is the case study good for formal methods? It is not ideal, because the goals are incompletely described.
- The case study text is a real world document.
- Reliability and probability is usually not covered by formal methods. Therefore this case study seems challenging.
- Provided, we modify the case study text, we should not remove information, but only add information to make the text less ambigous. Perhaps we could do so by enriching the question and answer section.
- There were opinions in favour of restructuring the document.
- Most topics from the case study requirement should be described in the solutions.

Issues concerning the presented solutions:

Adrian DeGroot mentioned that his approach treats trains in a more abstract way, but recognized that the part covered is too small.

Victor Winter emphasized that his solution is an algorithmic specification based on finite sets. It seems open how to shift to a more general solution, for example a solution based on continous function.

Fabrice Kordon saw in the case study a good example to apply the theory-methodology-tools idea. In his solution it was possible to prove lemmata. However, protocol like tasks seem easier to handle with his approach.

Michel Lemoine pointed out that the static part could nicely be described in Z. However the real complexity comes into the game because of the use of different notations for different parts.

Issues concerning system goals and further open questions:

We discussed the main goals of the system: Safety, efficiency, smoothness. All three are important, but, if in conflict, the given order should have priority. As an amendment to case study text, a definition of what efficiency means could be added.

Finally we discussed what happens when one is re-starting or starting the system. We also concentrated on the coordination between station computers and argued about the interlocking system.