Rigorous Analysis and Design for Software Intensive Systems

07.11 - 12.11.1999

organized by

S. Jähnichen (Technical University, Berlin),
M. Lemoine (CERT, Tolouse),
T. Maibaum (King's College, London),
M. Wirsing (Ludwig-Maximilians-University, München)

Preface

The seminar was concerned with a challenging problem in current software technology: the use of non-sequential components in heterogeneous systems. Both topics are related and raise many interesting issues, such as concurrency, distribution, reliability, etc. They challenge existing formalisms and methods and were addressed at the workshop by various speakers. Heterogeneity of systems (e.g., hardware vs. software, continuous vs. discrete, etc.) is asking for the assumption that software can be considered in isolation. The methods used for sequential component development are being extended in an attempt to cope with these new requirements. At present, it is not clear whether these methods are in fact extendable. New methods and formalisms are being invented to address the challenges of building such systems.

To tackle the task of rigorous analysis of large systems, the methods will focus on high level specifications. That is, complex heterogeneous systems and the constituent components are described more abstractly, say on the level of system architecture rather than on the level of mere programs. A system architecture reflects interaction and interfaces between the components without specifying all their complex internal functionality. Analysis of such an architecture is a new challenge for methods being applied to ordinary software systems so far.

When discussing about systems in the large, we are also faced with refinement issues. Detailed information about timing or any physical limitation is not known on the abstract level of specification. For supporting the incremental development new strategies for refinement are introduced, i.e. how to develop a system design straightforward from a high level specification.

In practice, semi formal methods like UML are accepted by a broad audience of software engineers in order to describe heterogeneous systems on a high level. Although UML models are primarily used to communicate only a design, the emerging question is how formal notations and languages, which are developed for rigorous analysis already, can support the design phase. A formalization that bridges the gap between semi formal and formal notations is to be developed and investigated.

In order to make technologies available and useful, adequate tool support has to be provided for actual usage in real applications. We aim at environments in which tools and notations are adequately integrated and which support methodological guidance without constraining the users creativity and individual progress.

In addition to the topics dealt with by the speakers, the workshop participants formed three working groups to discuss further questions of interest:

In order to get a comparison in the results of the different formal approaches, the community should benefit from treating a particular case study with the different formalisms. A new case study reflecting the needs of software intensive systems has to be found. In one working group a list of criteria to be characteristic for a suitable case study was worked out. According to these criteria the group agreed upon a rapid transportation management system (including a train control system, data management, etc.) to be a suitable case study.

In a second working group possible integrations of UML with formal methods were discussed. Although UML is known as an uprising formalism it lacks a formal foundation as well as tool support through formal treatment. Several suggestions how to relate the notations of UML with formal notations were discussed.

A standardization of formal methods based on a formal methods web repository was discussed in the third working group. The repository to be set up should collect all current formalisms as well as their corresponding software environments (if available) and case studies treated so far. This is to give a survey to industry or other potential users.

As a result of these discussions and the subsequent presentations, it was decided to apply for another Dagstuhl seminar, with a similar orientation but calling additionally for the presentation of techniques and methods in the framework of a common case study to be distributed with the call for participation. A suggestion for this case study was the development of a train-control system (the result of one of the working groups).

The organizers wish to thank all the workshop participants for their work and interest in the topic. Special thanks go to the organizers of the working groups and especially to those who arranged the accompanying musical programme.

The organizers
Stephan Jähnichen
Michel Lemoine
Tom Maibaum
Martin Wirsing

Some rigorous people at Dagstuhl
considered the rules for a new tool:
        can be used by a fool
        by the pool where it's cool
it should be a present for next Jul

BKB

Contents

Incremental Development of Real-Time Specifications

Graeme Smith
SVRC, University of Queensland

Formally refining a real-time specification to an implementation is only possible when the specification allows for all physical limitations, and timing and signal errors inherent in the implementation. Allowing for such implementation-specific details in a top-level specification can, however, obscure the desired functionality and complicate analysis. Furthermore, such an approach assumes the specifier has an understanding of the physical limitations and errors of the implementation which may not yet have been developed. As an alternative, we propose introducing a notion of realisation into the formal development process. Realisation is an approach to specification development which allows errors and physical limitations to be introduced. It also allows properties of the new specification to be derived from those proved for the original.

A Mechanized Logical Model of Z and Object-Oriented Specification

Thomas Santen
Technische Universität Berlin

We present a mechanized logical theory for specifying and analyzing object-oriented software components. The theory is based on the specification notation Z, higher-order logic (HOL), and its implementation in the tactical theorem prover Isabelle/HOL.
Building proof support for analyzing specifications "in-the-small" with Isabelle/HOL is the first step in the construction of our theory. An investigation of the semantic relation between Z and HOL reveals that the two formalisms are very similar. That observation justifies a "shallow" embedding of Z in HOL. Its implementation $\mathcal{HOL}$-$\mathcal{Z}$ in Isabelle/HOL preserves the structure of a Z-specification, as it is given by Z-schemas. Proof tactics tailored for Z exploit that structure to analyze operation specifications efficiently. It turns out that this *structural* reasoning is essential to make proofs about non-trivial specifications possible in practice.
For specifying "in-the-large", we define the essential concepts of object-oriented extensions of Z in HOL. We develop a theory of classes, objects, and method

invocations within Isabelle/HOL. The theory allows us to specify the components of a class in Z and compose them to make up a class specification in HOL later. This is possible because a HOL-theory of isomorphisms combines the shallow embedding $\mathcal{HOL}$-$\mathcal{Z}$ with the concepts of object-orientation as they are defined in HOL (used as a meta-language).
Behavioral conformance is the logical relationship between classes that guarantees that polymorphism induced by inheritance does not lead to unexpected behavior of objects. Our HOL-theory of object-oriented specification allows us to interactively prove non-trivial, general properties of behavioral conformance in the system Isabelle/HOL. Examples are the transitivity and compositionality of conformance. Theorems about conformance are directly applicable to concrete class specifications. In this way, we can reduce propositions about concrete specifications in-the-large to verification conditions in-the-small in a reliable, uniform, and efficient way. Using $\mathcal{HOL}$-$\mathcal{Z}$ to prove the resulting verification conditions, we achieve a chain of reasoning that is completely checked, and to a large extent automated within Isabelle.
We have validated our approach by case studies that are based on real, existing software products: the Z specification of a radiation therapy machine, and the specification of a part of the Eiffel Base Libraries, which we set up and analyzed for behavioral conformance induced by inheritance relationships.

Specification of Safety-Critical Software with Complex Data Models: A Systematic Approach

Maritta Heisel
University of Magdeburg

joint work with:
Kirsten Winter, GMD FIRST, Berlin
Thomas Santen, Technical University of Berlin

We have developed a method to specify software for a special kind of safety-critical embedded systems, where sensors deliver low-level values that must be abstracted and pre-processed to express functional and safety requirements adequately.
These systems are characterized by a *reference architecture*. The method is expressed as an *agenda*, which is a list of activities to be performed for setting up the software specification, complemented by validation conditions that help detect and correct errors.

The method was developed in the ESPRESS project, a joint project of if academia, research institutes, and industry. It was validated be an industrial case study, the safety-controller of traffic light systems.

Modular, Changeable Requirements for Telephone Switching

Jan Bredereke
University of Bremen

We are concerned with structuring software requirements with regard to future modifications. This is particularly difficult for telephone switching requirements, both due to the inherent=20 dependences among them and due to their current, rapid change. We perceive all variants and revisions as a single requirements family, documented together. We employ the Functional Documentation approach to requirements specification, also known as "Parnas tables". But we structure the requirements in a different, modular way suitable to =20 our application area; and we present a way to compose the partial =20 specifications written in Functional Documentation. We facilitate =20 avoiding undesired interactions introduced by extensions, and we support the detection of remaining interactions by specifying =20 explicitly the dependences among the partial requirements, and by distinguishing explicitly the essential part of each specified =20 feature.

Architectural specifications in CASL

Andrzej Tarlecki
Institute of Informatics, Warsaw University
Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland

joint work with:
M. Bidoit and D. Sannella

One of the most novel features of CASL, the Common Algebraic Specification Language, designed by the CoFI group (Common Framework Initiative) is the provision of so-called architectural specifications for describing the modular structure of software systems. In this talk, a brief presentation of CASL specifications

and discussion of their refinement provides a setting for the rationale behind architectural specifications. I follow with some details of the features provided in CASL for architectural specifications, hints concerning their semantics, and simple results justifying their usefulness in the development process. I conclude with a list of technical issues yet to be resolved to make CASL architectural specifications fully usable, and discuss the possibility of using architectural specifications in this "static" form to design architecture of reactive systems, with dynamic, interacting modules.

The integration of coordinated formalisms

Antonia Lopes
University of Lisbon

joint work with:
Jose Luiz Fiadeiro

In this talk we present a formal account of some of the contributions of coordination, in the sense of recently proposed languages and models, to the integration of different formalisms for software specification and design. We use an extension of Goguen's categorical approach to systems design as a platform for the formalisation of coordination and as a framework for defining the integration. Goguen's categorical approach was extended in order to capture the vertical structuring principles of a design formalism as well as the specific rules that govern the interconnection of components.

Consistent Transformations for Software Architecture Styles of Distributed Systems

Ugo Montanari
Universita' di Pisa

joint work with:
Dan Hirsch, Universidad de Buenos Aires

One major problem for the specification and verification of software architectures and specially with distributed systems, is when system evolution includes dynamic changes and reconfigurations of components and connections.
This paper presents a method for specifying reconfigurations or *transformations* over the topology of the architecture style, being sure that if the transformation

can be specified, then its application over the system will be consistent with respect to the expected architecture style configuration. Styles are described by context-free hyperedge graph grammars. In this context, an instance of an architecture style is determined by a graph generated by the grammar.
The formalization of the method will be introduced in two ways. The first approach is a visual presentation and has the intention of showing that it can be used by software architects in real life in a simple way. For this, as we mentioned, graph grammars represent styles and their generated graphs specific system architectures. A transformation is visualized as a rule that takes a graph generated by the grammar and applies a reconfiguration to part of that graph, obtaining a new graph that is surely another valid graph of the grammar. The second approach presents a formal model for the first one using lambda-terms and tile sequents showing also that it is implementable.

Modeling Software Architecture Styles and Reconfiguration

Dan Hirsch
Universidad de Buenos Aires

joint work with:
Paola Inverardi, Universita' di L'Aquila
Ugo Montanari, Universita' di Pisa

A software architecture style is a class of architectures exhibiting a common pattern. The description of a style must include the structure model of the component types and their interactions or connections (structural topology), the communication pattern (interactions among component types) and the laws governing the dynamic changes in the architecture (reconfiguration and/or mobility). A simple and natural way to describe a system is by using graphs, and as an extension of this, by using grammars to describe styles. So a grammar will generate all possible instances of that style. In our work we represent a system as a graph where hyperedges are components and nodes are ports of communication. The construction and dynamic evolution of the style will be represented as context-free productions and graph rewriting. To model the evolution and reconfiguration of the system we need to choose a way of selecting which components will evolve and communicate. For this we propose to use techniques of constraint solving already applied in the representation of distributed systems. In the case of software architectures we use constraint productions to coordinate communications between

components. To model reconfiguration, we add the possibility of creating new names to identify ports (nodes) of communication and use constraint productions to communicate and share the names among components (hyperedges), i.e., allowing complex reconfigurations of components and connections. This approach is well suited to model architectures of distributed systems obtaining a unique language to describe the style.

References

[1] D. Hirsch and P. Inverardi and U. Montanari, Graph Grammars and Constraint Solving for Software Architecture Styles, Proceedings of the Third International Software Architecture Workshop, Orlando, E.E.U.U., November 1-2, 1998

[2] D. Hirsch and P. Inverardi and U. Montanari, Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving, In Proceedings of the First Working IFIP Conference on Software Architecture, San Antonio, Texas, E.E.U.U., February 22-24, 1999

[3] D. Hirsch and U. Montanari, Consistent Transformations for Software Architecture Styles of Distributed Systems, Proceedings of the Workshop on (formal methods applied to) Distributed Systems, Gheorghe Stefanescu, Ed., Iasi, Rumania, September 2-3. Electronic Notes in Theoretical Computer Science, Vol. 28, pages 23-40, 1999, http://www.elsevier.nl/locate/entcs/volume28.html

Debugging Architectural Designs - An approach based on UML and High-level Petri-nets

Harald Störrle
Ludwig-Maximilians Universität München

Architecture is the basis for the most important tasks in Software Engineering (SE), in particular functional evolution, concurrent engineering and reuse of systems.

However, there is currently very little support for working on this level of abstraction, in particular wrt. (automated) tools. What one would really like to have is a system, where one could not only design an architecture, but also play around with it: making little changes here and there and observe (some of) the implications and repercussions. The following issues would have to be addressed:

1. **Concepts and Syntax**
   There are several approaches (e.g. Wright, Darwin,...) that already provide

most of the concepts for Architectural Designs (ADs). Given that the UML is likely to stay the lingua franca of SE for some time to come, the existing concepts for ADs should be embeded into the framework of the UML. This would also result in syntax (and tools!) to be instantly available, and fairly well-known.

2. **Semantics**
   Working with ADs, one would need tools like simulators, generators and analzsis tools of all kinds. They require formal semantics for the fragment of the UML that will be used. Considering the nature architectural units, Petri-nets seem to be a good choice for a semantical domain, as they allow to easily represent causality, location, true concurrency, state and events. Also, they feature a rich theory and a wealth of tools. All conceivable extensions (time, probability,...) have been studied, too.

3. **Properties**
   Finally, practically interesting properties have to be defined on the basis of the semantics. From the large number of available analysis algorithms, the appropriate ones have to be determined experimentally.

In my talk, I will present the approach in general, along with some of the considerations on possible choice-points. I will present the concept of capsules, their syntax, embedding in the UML metamodel, and the semantics. Together with the semantics of scenarios (in the form of Sequence Diagrams), I give some examples on intuitive consistency conditions, and how they can be checked.

Modelling for mere Mortals

Jeff Kramer
Imperial College London

In the past, attempts to convince practising software engineers to adopt formal methods of software development were generally unsuccessful. The methods were too difficult to learn and use, provided inadequate tool support and did not integrate well into the software development process. In short, they could only be used effectively by the gods who created them! Are we in a better position today? Recent advances in and experience with specification techniques and automated model checking have demonstrated the utility of these techniques. In this talk we outline one such effort which is specifically intended to facilitate modelling as part of the software process. We use the familiar formalism of

state machines (Labelled transition Systems LTS) to specify system processes in a process algebra, the software structure to dictate the composition of these processes, and compositional reachability analysis (CRA) to perform safety and liveness model checking using the analysis tool, LTSA. The intention is to make model specification and model checking accessible to mere mortals.

Analyse This!

Dimitra Giannakopoulou
Imperial College London

To be usable by software engineers, modelling and analysis should be an integral part of software development. To achieve this, our approach performs modelling and analysis based on the software architecture of a system. A user provides models for the primitive components in the architectural hierarchy of a system. Then the architecture is used to drive the process of putting models together, to obtain the overall system model. This is performed automatically, and has been achieved by a mapping that we defined between features of our ADL (architecture description language) Darwin, and operators of our specification notation, FSP. In this context, one can then check both safety and progress properties.
We presented our experience with analysing the "Bounded Retransmission Protocol", a protocol that is use in one of Philip's products. We have explained how we model (discrete) time in our framework, and how we can express both the condition of maximal progress, and the property that a timed system should exhibit neither zeno exexutions, nor time deadlock. In fact, our approach deals very elegantly with these issues, by using a simple action priority operator, and a simple progress property. We have shown how, by checking safety properties with different values of timeouts in the protocol, we managed to establish the minimum values that these timeouts can acquire in order for the protocol to work correctly.
Finally, we have discussed some open problems, and plans for future work. This mainly concerns the issue of compositionality in the presence of time, and the issue of choosing the right unit of time, when the latter is modelled as a discrete entity.

Distance functions for defaults in reactive systems

Sofia Guerra
University College London

Default reasoning has become an important topic in software engineering. In particular, defaults can be used to revise specifications, to enhance reusability of existing systems, and to allow a more economic description of systems. In this talk I present a framework for default specifications of reactive systems.
Non-monotonicity in temporal logic is formalised by defining a pre-order between temporal morphisms. This formalism is based on the notion of default institution, where the semantics of defaults are given by a (generalised) distance between interpretations. In this way, using temporal logic as a specification language, we get a way of handling defaults in specifications of reactive systems. I illustrate the developed formalism with an example in which a specification is reused, but where the new behaviour contradicts the initial specification. In this example, the initial specification is seen as a default to which exceptions are added.

Introducing New Software Engineering Techniques into Practice

Barbara Paech
Fraunhofer Institute for Experimental Software Engineering

This talk describes the IESE approach for putting specification techniques into practice. Two major obstacles for introducing new techniques are to identify which techniques are adequate for which process and to convince the process participants to adopt the new techniques. These obstacles are met by the Quality Improvement Paradigm (QIP) published in 1994 by Vic Basili. The main idea is to characterize the existing process to achieve a common understanding, to set explicit improvement goals and to break them down into metrics. Then data is collected giving evidence on the weaknesses of the existing process, new techniques are chosen and introduced and again data is collected giving evidence on the goal achievement. Experiences with these new techniques are gathered and managed within an experience factory. The Fraunhofer requirements assessment and improvement method (RE-KIT-FRAIME) tailors this paradigm to the improvement of requirements processes. In particular, it emphasizes the importance

of treating requirements as knowledge which is used throughout the software engineering tasks. Thus, the general goal for improving requirements processes is to establish professional knowledge management for requirements within the organisation. RE-KIT-FRAIME also provides a framework for choosing adequate techniques. According to this framework all the stakeholders interested in using requirements documents are interviewed in order to determine the contents to be captured in requirements documents. Then the steps for creating, validating and managing the documents are drafted and only then specific specification techniques are chosen according to the degree of which they support these steps. Experiences show that specfication techniques which

- support light weight application (e.g. in the form of perspective-based inspections),
- integrate well with natural language requirements,
- have methodological support, and
- come with empirical evidence on their value are most easy to introduce.

A Case Study in Statistical Testing of Reusable Concurrent Objects

Helene Waeselynck
LAAS - CNRS, Toulouse

joint work with:
P. Thevenod-Fosse

A test strategy is presented which makes use of the information got from OO analysis and design documents to determine the testing levels (unit, integration) and the associated test objectives. It defines solutions for some of the OO testing issues: here, emphasis is put on applications which consist of concurrent objects linked by client-server relationships. Two major concerns have guided the choice of the proposed techniques: component reusability, and nondeterminism induced by asynchronous communication between objects. The strategy is illustrated on a control program for a production cell. The program was developed using the Fusion method and implemented in Ada 95. We used a probabilistic method for generating test inputs, called statistical testing. Test experiments were conducted from the unit to the system levels, and a few errors were detected.

The UniForM Workbench a Universal Development Environment for Formal Methods

Bernd Krieg-Brückner
Universität Bremen

The UniForM Workbench supports combination of Formal Methods (on a solid logical foundation), provides tools for the development of hybrid, real-time or reactive systems, transformation, verification, validation and testing. Moreover, it comprises a universal framework for the integration of methods and tools in a common development environ-ment. We describe the relation to a repository for Formal Methods structured by Language, Logics, Methods, and Tolls Graphs, resp., driven by process model standards (cf. presentation by Carla Purper). see http://www.tzi.de/ uniform/

The Process Web-Centre

Carla Purper
Universität Bremen

The process web-centre GDPA is presented. It consists in a web-based information system to supply information on process technologies applying formal methods.

Process Technologies:
Meta-models for safety-critical systems process based on existing standards in order to consider:

- Modeling of heterogeneous components
- Interoperability of components from multiple paradigms
- Analysis techniques for non-sequential systems
- Methodology for non-sequential systems are outlined.

Formal Methods:
Taxonomical forms in order to identify formal methods and tools:

- for each phase of the meta-models
- for each system property
- to compare formal methods
- to disseminate the use of formal methods and tools are introduced.

Graphical Animation of Behavior Models

Jeff Magee
Imperial College London

Graphical animation is a way of visualizing the behavior of design models. This visualization is of use in validating a design model against informally specified requirements and in interpreting the meaning and significance of analysis results in relation to the problem domain. We describe how behavior models specified by Labeled Transition Systems (LTS) can drive graphical animations. The semantic framework for the approach is based on Timed Automata. Animations are described by an XML document that is used to generate a set of JavaBeans. The elaborated JavaBeans perform the animation actions as directed by the LTS model.
Keywords:
Labeled Transition System, Graphic Animation, Behavior Analysis

Analysing scheduling with a model checker

Thorsten Gerdsmeier
University of Essex

We are analysing timed and functional behaviour of a concurrent real-time implementation in Ada95. If a set of tasks is running on a uniprocessor or multiprocessor system, there are in general more tasks than processors. Scheduling tasks influences timed behaviour and can influence functional behaviour as well. We want to prove properties about concurrent implementations in Ada95. Functional behaviour in non concurrent systems has been well studied. For analysing timed behaviour traditionally scheduling theory is used. We combine the analysis of functional behaviour and timed behaviour in the framework of timed automata with effective tool support. Analysing scheduling with a model checker turns out to be superiour to traditional scheduling theory. We are illustrating our approach with a simple case study of a mine pump.

Validation of UML designs with Z formal specifications

Michel Lemoine
ONERA, Centre de Toulouse

In this talk we emphasized the fact that the (re)development of a safety critical system should follow a Rigorous and Systematic Life Cycle that embeds semi formal and formal methodologies.
Among others, UML, as a set of standardized notations, is a well suited candidate. But being semi formal, its notations are not rigorous enough to guarantee they met all the non functional and functional requirements of the system to be (re)developed.
We suggested to complement each UML notation by a formal one which will help demonstrating the expected qualities of the final product.
We showed how the Z formal notation can be fruitfully used to formally specify functional requirements. Moreover the Z method help satisfying 3 main properties: satisfiability, consistency and robustness of specifications.
It is as well mandatory to support developers with a right development process integrating both semi formal and formal methodologies in a very homogeneous manner. This can be achieved with a dedicated Development Process we call Evolutionary Process. We described it, insisting on the validation phases
We gave a flavor of 2 successful (re)designs achieved according to the evolutionary process.
We concluded confirming that formal methods can be fruitfully used iff

1. they complement semi formal industrial methods such as UML
2. experts of formal notations / methods are required to realize the formal specifications
3. a dedicated process is set up.

Modelling agent-based system with Graph Transformation and UML

Reiko Heckel
Universität Paderborn

As the concept of autonomous agents becomes increasingly attractive to software industry, the methods for developing agent-based systems have to be incorporated into the common practice of software engineering. Today, the analysis and

design phases of software development largely rely on visual techniques like the Unified Modeling Language (UML). We present an approach which integrates the modeling of object-oriented and agent-based systems. It is based on the formal framework of typed graph transformation systems with temporal logic to formally specify agents goals and state diagrams to represent the plans that they might follow.
Altogether, this provides a visual modeling approach, based on UML notation, which accounts for the main aspects of agency like autonomy, reactivity, and pro-activity in a formal and integrated way. The concepts are justified and explained by a running example of an agent-based extension for a commercial online banking system currently under development.

From Informal Requirements to COOP: a Concurrent Automata Approach

Pascal Poizat
Universite de Nantes

joint work with:
Christine Choppy, Universite de Paris Nord
Jean-Claude Royer, Universite de Nantes

Methods are needed to help using formal specifications in a practical way. We present a method for the development of mixed systems, i.e. systems with both a static and a dynamic part. Our method helps the specifier providing means to structure the system in terms of communicating subcomponents and to give the sequential components using a semi-automatic concurrent automata generation with associated algebraic data types. These components and the whole system may be verified using common set of tools for transition systems or algebraic specifications. Furthermore, our method is equipped with object oriented code generation in Java, to be used for prototyping concerns. Our method is presented on a small example: a transit node component in a communication network.

From CSP-OZ to Java

Clemens Fischer
Universität Oldenburg

Object-Z and CSP are specification languages which offer powerful formal support for the design of distributed, communicating systems. CSP-OZ is a combination of both using CSP to specify dynamic and Object-Z to specify data aspects.
Java is an ideal implementation language for CSP-OZ. It offers build in constructs for programming distributed systems and a synchronized modifier that closely models synchronous CSP communication. But developing provably correct Java implementations from these specifications is notoriously difficult.
To bridge this gap we suggest to use Jass, which extends Java with assertions, as an intermediate language. The idea presented in the talk, is to generate these assertions automatically from CSP-OZ specifications. This does not guarantee a provably correct implementation, but allows an easy way of run-time refinement check. Furthermore, error messages can be linked directly to the formal specification.

Using CASL to Specify the Requirements and the Design: A Problem Specific Approach

Christine Choppy
LIPN, Universit Paris
and
Gianna Reggio
Universit di Genova

In [1] M. Jackson introduces the concept of problem frames to describe specific classes of problems, to help in the specification and design of systems, and also to provide a framework for reusability. He thus identifies some particular frames, such as the translation frame (e.g., a compiler), the information system frame, the control frame (or reactive system frame), ... Each frame is described along three viewpoints that are application domains, requirements, and design.
Our aim is to use CASL (or possibly a sublanguage or an extension of CASL if and when appropriate) to formally specify the requirements and the design of particular classes of problems ("problem frames"). This goal is related to methodology issues for CASL, that are here addressed in a more specific way, having in mind some particular problem frame, i.e. a class of systems.

It is hoped that this will provide both a help in using, in a really effective way, CASL for system specifications, a link with approaches that are currently used in the industry, and a framework for the reusability of CASL specifications.
This approach is illustrated with some case studies, e.g., the information system frame is illustrated with the invoice system case study.

References

[1] M. Jackson, Software Requirements & Specifications: a lexicon of practice, principles and prejudices, Addison-Wesley, 1995

From Development environments to a Conceptualisation of Engineering: revisiting and revitalising the logical empiricists

Armando Haeberer
visiting Ludwig-Maximilians-Universität, Germany
and
Tom Maibaum
King's College London

Attempts at building CASE tools and SW development environments have largely been perceived as failures. If we are to continue building tools (and methods) to support SW development, what criteria should we use to characterise requirements and designs for such tools? To begin to answer this question, we should first understand what engineering is. We do this by borrowing ideas from epistemology, in particular Carnap's two level theory of the language of science and its variants. We see there many ideas, with sophisticated theoretical and operational characterisations, which help us understand what activities, such as specification, testing, validation, refinement, etc., actually are. This aids us in developing the criteria mentioned above, as well as in developing more exact characterisations of what tools should do in order to be useful.

Towards an epistemology-based methodology for verification and validation testing

Maria Victoria Cengarle
Ludwig-Maximilians-Universität München
and
Armando Martin Haeberer
visiting Ludwig-Maximilians-Universität München

How can evidence stated in a language restricted to an observable vocabulary confirm hypotheses stated in a theoretical language that outstrips the first in both vocabulary and expressiveness?
This is a problem that puzzled philosophers of science by decades. It is a central issue in validation testing of design specifications and (software) artifacts, in testing of design against requirements specifications, and in verification testing of (software) artifacts against design specifications.
Based on the seminal paper *Testability and Meaning* by Rudolf Carnap, the so-called bootstrap approach to testing of scientific theories was further developed in a more precise manner by Clark Glymour. The bootstrap testing schemata were born. After suffering sharp criticism and further polishing by the epistemology community, a relatively stable proposal was published in 1983.
The bootstrap testing approach, goes from evidence to the falsifiability of theoretical hypotheses, i.e., in opposite direction to the hypothetico-deductive method, which derives observational consequences from theoretical statements.
We have taken the bootstrap schemata and applied them to our setting of software engineering. In fact, they have proved to be useful in more than one sense. They can be used for testing a program against a design specification (i.e. verification) without taking into account the structure of the former, thus proving suitable for reverse engineering; for testing a design specification against a requirements one (i.e. validation); etc.
Summarising, we have presented a testing strategy that promises to be powerful, is independent of the architecture of the realisation, doesn't force the nature of the evidence, and is suited for distributed realisations as well as for reverse engineering.
We now plan to: clean up the setting by, for instance, applying correspondence rules to enable _-conversion; address the infinitary domain issue by looking back at Carnap's and Hintikka's inductive logics; study which schema is better suited for different scenarios; extend the ideas to other (non-equational) logics, like for instance 1st-order, modal, and temporal; and bring to the surface inherent (internal) issues of different models of computation so that we can analyse the strategies for testing in relation to these models. This setting suggests a starting point for the analysis of the relationship between design and requirements specifications.

Software Architectures and Component Programming

Paola Inverardi
Universita degli Studi di l'Aquila

In recent years the focus of software engineering is continuosly moving towards systems of larger dimensions and complexity. Software production is becoming more and more involved with distributed applications running on heterogeneous networks, while emerging technologies such as commercial off-the-shelf (COTS) products are becoming a market reality. As a result, applications are increasingly being designed as sets of autonomous, decoupled components, promoting faster and cheaper system development. The development of these systems poses new challenges and exacerbates old ones. A critical problem is understanding if system components integrate correctly. To this respect the most relevant issue concerns dynamic integration. Indeed, component integration can result in architectural mismatches when trying to assemble components with incompatible interaction behavior, leading to system deadlocks, livelocks or in general failure to satisfy desired functional and non-funtional system properties. In this context Software Architecture (SA) can play a significant role. SAs have in the last years been considered, both by academia and software industries, as a way to improve the dependability of large complex software products, while reducing development times and costs. SA represents the most promising approach to tackle the problem of scaling up in software engineering applications manageable. The originality of the SA approach is to focus on the overall organization of a large software system (the glue) using abstractions of individual components. This approach makes it possible to design and apply tractable methods for the development, analysis, validation, and maintenance of large software systems. In this talk I will present our research efforts in the area of analysis of SA, system testing at the architectural level and performance analysis of SA.

Developing Critical Software Systems with the SCR Requirements Method

Constance Heitmeyer
Naval Research Laboratory, Washington
(This research is supported by the Office of Naval Research.)

This talk provided an overview of the SCR (Software Cost Reduction) requirements method—a method for developing software systems introduced in the late

1970s, which has since been applied to a wide range of critical software systems. These systems include control systems for nuclear power plants, telephone networks, space systems, and avionics systems. To support the SCR method, our group at the Naval Research Laboratory has developed a set of integrated tools, including a consistency checker, a simulator, a model checker, an invariant checker, and a theorem prover (see, e.g., [2],[3]). In SCR, the required system behavior is represented using a state machine model. The SCR tools are designed to analyze state machine specifications expressed in the SCR tabular notation. To date, the SCR method has been applied successfully to a number of practical applications, including the International Space Station, a commercial flight guidance system, a weapons system [2], and a cryptographic device. In addition, more than 80 organizations in industry, academia, and government are experimenting with the SCR method.

The SCR requirements method was demonstrated by showing its application to a complex system called the Advanced Automatic Train Control (AATC) system [4], which controls the movement of trains traveling on the Bay Area Rapid Transit (BART) system, a subway system in San Francisco. The AATC system receives periodic updates from the trains (i.e., position, velocity, and timing information) and information about the status of gates along the track (i.e., whether a gate is open or closed). It controls the trains' speed and velocity by sending each train periodic velocity and acceleration commands. The design of the BART control system is complex because the system must operate the trains in a manner that satisfies a set of complex functional and safety properties. Examples of these properties are:

- When traveling in a particular track segment, a train should travel close to the maximum speed allowed for that segment.
- A train in a segment must not exceed the maximum speed for the segment.
- A train must not collide with a closed gate.
- A train behind a second train must never get so close that if the train ahead stopped, the trains would collide.
- The change in the speed of a train must be gradual enough to prevent injury to the passengers.

The SCR specification of the required behavior of the AATC system consists of a collection of tables. A number of challenging issues arose in developing this requirements specification. These issues include

- how should we describe the various environmental aspects that constrain the velocity of trains (e.g, the grade of a given segment of track, the maximum speed allowed on a given track segment),

- in what order should various parts of the specification of this complex control system be developed, and
- how should we describe the required train velocity and acceleration, both complex functions of many variables that must simultaneously satisfy the properties listed above.

A way was proposed to resolve some of these issues using the SCR method, and some open problems (e.g., the last issue) were discussed.

References

[1] C. Heitmeyer, J. Kirby, B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Softw. Eng.*, 24(11), November 1998.

[2] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.

[3] Constance Heitmeyer, James Kirby, Jr., Bruce Labaw, and Ramesh Bharadwaj. SCR*: A toolset for specifying and analyzing software requirements. In Moshe Vardi and Alan Hu, editors, *Proc. Computer-Aided Verification, 10th Annual Int'l Conf. (CAV'98), (LNCS 1427)*, pages 526–531, Vancouver, Canada, June/July 1998.

[4] V. Winter, R. Berg, and J. Ringland. Bay Area Rapid Transit District Advanced Automated Train Control System: Case study description, 1999.

Using ASMs for Integrating Different Design And Analysis Methods

Egon Börger
Universita degli Studi di Pisa

We illustrate through a case study the possibilities to use ASM for integrating functional and operational desing and analysis methods. We provide a rigorous framework for language and platform independent design and analysis of modern exception handling mechanisms in object oriented programming languages and their implementations. To illustrate the practicality of the method we develop it for the exception handling mechanism of Java and show that its implementation on the Java Virtual Machine (JVM) is correct. For this purpose we define precise abstract models for exception handling in Java and in the JVM and define a compilation scheme of Java to JVM code which allow us to prove that in corresponding runs, Java and the JVM throw the same exceptions and with

equivalent effect. Thus the compilation scheme can with reasonable confidence be used as a standard reference for Java exception handling compilation. Through this case study we develop a novel combination of Abstract State Machine based run time analysis with structural design and verification methods. (Joint work with Wolfram Schulte (Microsoft Research Redmond), to appear in TSE 2000.)

Methodology for Model Checking ASM: Lessons learned from the FLASH Case Study

Kirsten Winter
GMD First, Berlin

Gurevich's Abstract State Machines (ASM) constitute a high-level specification language for a wide range of applications. The existing tool support for ASM was extended, in a previous work, to support computer-aided verification, in particular by model checking. In this paper we discuss the applicability of the model checking approach in general and describe the steps that are necessary to fit different kinds of ASM models for the model checking process. Along the example of the FLASH cache coherence protocol we show how model checking can support development and debugging of ASM models. We show the necessary refinement for the message passing behavior in the protocol and give examples for errors found by model checking the resulting model. We conclude with some general remarks on the existing transformation algorithm.

Locales
A Sectioning Concept for Isabelle

Florian Kammüller
GMD First, Berlin

Locales are a means to define local scopes for the interactive proving process of the theorem prover Isabelle. They delimit a range in which fixed assumption are made, and theorems are proved that depend on these assumptions. A locale may also contain constants defined locally and associated with pretty printing syntax. Locales can be seen as a simple form of modules. They are similar to sections as in AUTOMATH or Coq. Locales are used to enhance abstract reasoning and similar applications of theorem provers. This talk introduces Isabelle and summarizes

research presented in [1]. It motivates the concept of locales by examples from abstract algebraic reasoning.

References

[1] F. Kammüller and M. Wenzel and L. C. Paulson, Locales – a Sectioning Concept for Isabelle, Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, 1999

Topology in Process Calculus

Mingsheng Ying
Ludwig-Maximilians-Universität München

Various behaviour equivalences between agents are central notions in process calculus. In its applications, specification and implementation are described as two agents. Then correctness of programs is treated as a certain behaviour equivalence between specification and implementation. But in many situations, the implementation can only approximate the specification. Thus a problem naturally raises: What is approximate correctness of programs and how to describe it formally?

In this talk, we provide some suitable and useful mathematical concepts and tools for the understanding and analysis of approximate correctness and infinite evolution of programs in concurrent systems. The main idea of our work is to construct some natural and reasonable topological structures which can reveal suitably mechanism of approximate computation in process calculus. We construct two different classes of topological structures: one is determined by the behaviours of processes and so completely extensional and observable; and the other involves somewhat intensional factors in the sense that a certain topological structure on actions is presumed and it is employed to induce some topological structures on processes. The presumed topology on actions is given according to the concrete applications that we are dealing with. We have two different kinds of methods to construct our intended topological structures: one is a dynamic approach in which we consider the topologies possessed by a sequence (or more general, net) of processes with Moore-Smith theory of convergence in topology; and the other is a static approach in which we introduce some weakened (looser, approximate) versions of behaviour equivalences and they are given in a topological way.

In this talk, we propose the concepts of bisimulation limits, near bisimulations and bisimulation indexes and present some applications of bisimulation indexes in timed CCS and real time ACP to show how they can be used to describe approximate implementations.

Black Box View of State Machines

Max Breitling
Techniche Univeristät München

joint work with:
Jan Philipps, Techniche Univeristät München

The behaviour of reactive systems could be specified in (at least) two ways:

- Using a Black Box View, a system's behaviour is decribed as a relation of input and output streams, not reflecting any internals.
- Using a State Transition System, a behaviour is described in an operational, step-by-step manner, and specifies *how* its behaviour is achieved.

In our work, we propose a way to prove that a State Transition System has indeed the property described by a Black Box View. To achieve this, we introduce an intermediate level that is based on streams of messages, but is also reflecting intermediate states using internal variables of the system. We present schemes to prove safety and progress properties, using invariants and leads-to properties. Based on verification diagrams, all necessary proof tasks can be generated automatically, and in most cases even proved automatically by tools. Closing the gap between the two views, we hope to contribute to the effort to reach an integrated use of development tools from specification down to code generation, including verification.

Software Development with OCL: From Classes to Programs

Martin Wirsing
Ludwig-Maximilians-Universität München

joint work with:
Rolf Hennicker, Ludwig-Maximilians-Universität München

The Object Constraint Language OCL is a new specification language which is used for formalizing semantic constraints of UML diagrams. In this lecture we present a method for applying OCL for the development of class implementations and introduce a new Hoare-Calculus for OCL. For simplicity we restrict our approach to total OCL formulae.
In the first part of the lecture we characterize OCL as a first-order logic with bounded quantification whose basic data types such as containers, sets, multisets

and sequences can be algebraically axiomatized. "Internal" consistency conditions in the spirit of Z and Z++ are given to ensure the semantic well-definedness of class specifications.
In the second part of the lecture a new Hoare-Calculus for OCL is presented. As programming language we choose a (sequential) kernel of Java. The Hoare rules are as usual for while programs and for method calls. Access and update of instance variables is handled by an explicit substitution operator which also takes care of aliasing. The rule for creation of objects uses the "allInstances"-operator of OCL.
Our calculus is inspired by Poetzsch-Heffter and Mueller's Hoare calculus for Java; the main difference is that we do not have any explicit representation of state in our formulas. In this respect we rather follow the ideas of Gries and De Boer for handling arrays and references by explicit substitution.
As main result we prove the soundness of our Hoare-Calculus.