Contents

Preface

The Seminar "Logic for Systems Engineering", held at Schloß Dagstuhl during March 3–7, 1997, was a successor of the Dagstuhl Seminars "Logical Theory for Program Construction" in 1991 and 1994[1] The slightly modified title is intended to stress that program development methods must be integrated in a general software development method, if they are to be applicable in practice.

The importance of formal methods for the development of correct software is now generally recognized. A few "realistic" case studies have meanwhile been carried out. Moreover, the software industry is increasingly interested in these methodologies.

Main subjects treated during the Seminar were:

- formal specification methods
- formal object-oriented specification techniques
- integration of formal and "pragmatic" software development techniques
- practical applications of formal specification methods
- reuseability

The Seminar focussed on problems, the solution on which is a prerequisite for industrial applications.

On behalf of all participants the organizers would like to thank the staff of Schloß Dagstuhl for providing an excellent environment to the conference.

The Organizers

Stefan Jähnichen   Jacques Loeckx   Martin Wirsing

[1]Seminar-Reports Nr. 7 and 84 are still available at Dagstuhl office.

An Algebraic Approach to Mixins and Modularity

Davide Ancona

Universitá di Genova

I will present a kernel modular language, called MIX(FL), supporting the notion of mixin module. A mixin is a module which contains deferred components, i.e. components to be imported from another module; a binary and commutative operator of merge allows to associate deferred components of one mixin with the corresponding definitions (if any) of the other. In this way, mixins allow recursive definitions to span module boundaries. MIX(FL) is a very simple typed module language, supporting separate compilation and providing a set of constants which are mixin definitions (collections of type and function declarations and definitions) and five operators for mixins manipulation (merge, freeze, restrict, hide and rename). Hide and rename are standard operators, whereas freeze and restrict allow to deal with virtual components and redefinition of them by means of overriding. These five operators can be expressed by the combination of three primitive and powerful operations (primitive merge, reduct and primitive freeze). The denotational semantics of the module language is based on an algebraic approach (i.e. module = algebra, interface = algebraic signature). What is more, the semantics is parametric in (the semantics of) the core language; this is suggested by the notation MIX(FL), where FL can be replaced by other languages.

Verification Based on Monadic Second-Order Logic

David Basin

Universität Freiburg

(Joint work with Nils Klarlund, AT&T Bell Labs)

We present show how WS2S (the weak monadic theory of two successors) provides a language for specifying many kinds of problems in hardware and distributed computing. We report on experience with a verification tool based on WS2S, which implements validity checking/counter-model generation based on a reduction of formulas to canonical automata.

The decision problem for WS2S is non-elementary decidable and thus unlikely to be usable in practice. However, we have used our implementation to automatically verify, or find errors in, a number of circuits studied in the literature. Previously published machine proofs of the same circuits are based on deduction and may involve substantial interaction with the user. Moreover, our approach is orders of magnitude faster for the examples considered. We show why the underlying computations are feasible and how the tool we use generalizes standard BDD-based hardware reasoning.

The Jakarta Project

Don Batory

University of Texas, Austin

Jakarta is a tool suite for building GenVoca generators — software generators that synthesize applications through component composition. Jakarta provides an extensible version of Java, which supports metaprogramming, i.e., constructs to create, parameterize, instantiate, and manipulate programs as abstract syntax trees. The core of the suite is Bali, a GenVoca generator of compilers: customized versions of the Java language and its compiler are synthesized through component composition. Each Bali component encapsulates a primitive extension to Java, which might be a new programming construct (e.g., templates) or a sophisticated domain-specific GenVoca generator.

If Refinement is The Key, Where is The Key to Refinement?

Michel Bidoit

LSV, CNRS & ENS de Cachan

(Joint work with Rolf Hennicker, Ludwig–Maximilians–Universität München)

We introduce a concept of behavioural implementation for algebraic specifications which is based on an indistinguishability relation (called behavioural equality). The central objective of this work is the investigation of proof rules that first allow us to establish the correctness of behavioural implementations in a modular (and stepwise) way and, moreover, are practicable enough to induce proof obligations that can be discharged with existing theorem provers. Under certain conditions our proof technique can also be applied for proving the correctness of implementations based on an abstraction equivalence between algebras in the sense of Sannella and Tarlecki. The whole approach is presented in the framework of total algebras and of first-order logic with equality.

Industrial Use of ASMs for System Documentation A Case Study: The Production Cell Control Program

Egon Börger

Università di Pisa

Abstract State Machines (alias evolving algebras as defined by Y. Gurevich) have been used to specify languages (e.g. Prolog, C++, VHDL) and architectures (e.g. PVM, APE100, Transputer, DLX), to validate standard language implementations (e.g. of Prolog, Occam), to verify numerous distributed and real-time protocols, etc.. See [2] for a recent survey and general discussion of the approach.

In this talk, we explain the use the software designer can make of ASMs for a systematic modular development of well documented programs. We exemplify the method by an ASM solution of the production cell control problem (posed in the case study book *Formal Development of Reactive Systems* edited by C. Lewerentz and T. Lindner, Springer LNCS 891, 1995).

As starting point we present the *ground model* which faithfully reflects the production cell as informally specified by C. Lewerentz and T. Lindner op. cit. We explain how the *information hiding* and *abstraction* mechanisms of ASMs allow the designer to formulate the ground model as an appropriate interface to the customer; this provides a possibility to discuss and negotiate with the customer, in rigorous yet simple terms of traditional mathematical language, so that one can come to a common understanding of the contract (i.e. of what the informal description really means) and of the conditions on its being met eventually by the algorithmic design. The modularity capabilities of ASMs are used to reflect in a direct way the distributed nature of the production cell.

Starting from this abstract model of the system we define a series of refined models all the way to executable C^{++} code (which controls successfully the production cell simulation environment, running at FZI Karlsruhe). The ability of ASMs to reflect arbitrary abstraction levels facilitates the modular development of different components and a transparent definition of *precise interfaces* through which these components are put together in a distributed environment. The mathematical relations between the various models allow us to prove the required properties of the system (safety, liveness etc.) at appropriate abstraction levels. These proofs are given in ordinary mathematical terms, as used by engineers; they are nevertheless amenable, where this is considered to be necessary for safety-critical reasons, to mechanization in machine based proof environments like PVS. K. Winter from GMD Berlin is working on a model checking verification of our proofs.

The hierarchy of refined models, together with the proofs relating the different levels, constitute a full documentation of the whole design and make the executable code amenable to rigorous inspection. This approach, which combines modular development with controlled stepwise refinement including optimizations, provides also an economical way to achieve extendability and modifiability for the design of complex systems in a context where cost effective maintenance of the final product is a crucial issue.

The work on the production cell is joint work with my student Luca Mearelli; for a draft version see [3], a revised version which contains also the model checking work on our specification done by K. Winter is in preparation. For another case study which exemplifies this approach to the design of well documented and formally inspectable code see [1].

References

[1] Christoph Beierle, Egon Börger, Igor Ðurđanović, Uwe Glässer, and Elvinia Riccobene, *An Evolving Algebra Solution to the Steam-Boiler Control Specification Problem.* In: Jean Raymond Abrial, Egon Börger, and Hans Langmaack (Eds.), *The Steam-Boiler Case Study Book*, Springer LNCS 1996 (to appear)

[2] E. Börger, *Why use of evolving algebras for hardware and software engineering.* In: M. Bartosek, J. Staudek, J. Wiedermann (Eds), SOFSEM'95 22nd Seminar on Current Trends in Theory and Practice of Informatics, Springer Lecture Notes In Computer Science, vol. 1012, 1995, pp. 236–271.

[3] Luca Mearelli, *An evolving algebra model of the production cell.* Tesi di laurea, Dipartimento di Informatica, Università di Pisa, February 1996.

A Loose Approach to OOA Techniques

Ruth Breu

Technische Universität München

Object oriented analysis techniques like UML, OMT or OOSE integrate description techniques of various application domains that have been successfully applied for years in practice like Entity/Relationship diagrams and state transition diagrams. However, the integration in the context of objects is only ad-hoc. Many concepts lack of an exact interpretation and the interrelations between the different description techniques remain unclear.

The presented formal foundation aims to overcome these deficiencies by providing an integrated system view. The system to be developed is modelled abstractly by a notion of system models describing the static and dynamic properties of objects. In particular, the dynamic evolution of a system of objects is described by the concept of life cycles. The approach is based on the ideas of loose modeling. This means that the abstract view of a system is open for later extensions and modifications and thus is able to cope with incompleteness issues.

In a second step, the different description techniques are mapped onto this semantic domain. The overall system view obtained is the basis for studying interrelations between different description techniques and refinement concepts. The description techniques considered are class diagrams for modeling the static structure of a system, sequence diagrams for modeling exemplary object behaviour and state transition diagrams for modeling complete object behaviour.

Combining Tools for the Verification of a Fault-Tolerant System

Bettina Buth, Rachel Cardell-Oliver, Jan Peleska

Universität Bremen

The formal methods developed over the last decades have been advertised as a means for improving software quality in general and for increasing confidence in safety critical software in particular. Nevertheless, the impact of formal methods on industrial software development is minimal. Many reasons are given for this, including their unsuitability for the development of large systems (scalability), the dependence on trained staff to apply formal methods, and the lack of tool support for their application.

In order to complement these efforts it is necessary to gain experience with the methods by applying them to exemplary but realistic applications in industrial projects. The aim here is not to support the overall development, but rather to identify crucial components and phases where the effort is justifiable.

One of our interest here is the investigation of software development approaches with respect to verification support. The starting point for this research is a case study on the design of a fault-tolerant computer system presented by Peleska. The development in this example is based on an invent-and-verify paradigm, where the design steps are refinements of CSP specifications and processes. Peleska has presented a formal theory to justify the links between the different phases of the development method and provided sketches for the proofs involved. In order to provide verification support for the proof obligations arising we employ a combination of tools:

- HOL for the verification of generic theories, which are used for the refinement of sat-relations,
- PAMELA+PVS for the verification of invariant properties arising for the proofs that concrete processes satisfy given specification,
- FDR for checking the refinement of processes.

Each of the tools is used in a specific phase of the development, but there is no direct interaction between the tools. This allows to choose freely from the tools available.

Reverse Engineering of Fortran Code Using Algebraic Specifications

Sophie Cherki], Christine Choppy]]

]Université Paris-Sud,]]Université Nantes

This talk describes an experience in trying to help maintaining existing Fortran 77 code of a large industrial application. Our approach is to start with reverse engineering this code using algebraic specifications to provide an abstract description of its functionalities. This implies an active reading of the code (together with the comments in the code) which lead us to find out interesting bugs and anomalies.

The resulting algebraic specification consists in a graph of specification modules; the axioms may be first order formulae, but we mainly used conditional equations. The code of this application is structured in such a way that, most of the time, a module implements a few functionalities. The structure of the algebraic specification reflects this code structure.

The key issues are to find out the signatures and the axioms for the specification modules. Fortran 77 exhibits only predefined types and anonymous array types. As a consequence, extracting the signature of a specification associated to a Fortran module raises various difficulties and we present how to overcome them. Extracting the specification axioms is achieved by means of identifying unit actions within the code, composing their associated equations, and simplifying the - rather unreadable - resulting expression in order to obtain axioms that are easier to read (this last step requires the use of theorem proving).

The resulting specification may serve both as a precise documentation, and as a basis for the development of the new version of the code, where both useless code, anomalies and bugs are removed.

Classification and Integration of Formal Specifications for Distributed Systems

Felix Cornelius

Technische Universität Berlin

A semantic unifying framework is presented aiming at providing a target for the main aspects covered by existing formal specification languages for ‘dynamic’ systems. It consists of five levels:

DATAS	I	Languages	Algebras
RELS	II	Data States	Relations
TRANS	III	D.S. Transformations	Relations/Functions
PROC	IV	Processes	Interface Sets/LTS’s
ARCH	V	Architectures	Hyper-graphs

Each level is given by a set, inter-relations are expressed by appropriate mappings. The three most important mappings are:

1. $\mathit{Datas} : \mathit{Rs}(P) \to \mathbb{P}(R(P))$

   For each *reactive state* $rs \in \mathit{Rs}(P)$ (the set of states in the labeled transition system describing P) there is a set $\mathit{Datas}(rs)$ of *data states* in level II.

2. $\mathit{transform} : (\to_{\ominus}) \to \mathit{TRANS}$

   For each *transition* t in the Labeled Transition System of a process there is a corresponding *transformation* on the underlying data state.

3. $\mathit{zoom\text{-}in} : \mathit{Nd}(N) \to (\mathit{ARCH} \cup \mathit{PROC})$

   Each node in the hyper-graph of a distributed process in *ARCH* corresponds to a “lower level” process, be it an atomic process in *PROC*, be it another distributed process in *ARCH*.

This five level semantic framework is intended to provide a *pool*, an additional rôle-oriented semantics for arbitrary specification documents. Hence, it is a structured integration platform for partial specifications.

Combining FOL and ω-Automata Methods — A Large Industrial Application

Jorge R Cuellar

Siemens AG, München

If you were an civil engineer, trying to apply mathematical methods (general math. methods, control theory, integral operators, PDE's, linear algebra) to solve a *real life* problem, perhaps you will start at a high abstraction level and go down the ladder to a very concrete setting where you are left with solving eigenvalue problems for matrices or with other forms of calculations.

On the example of a large industrial example, the author suggests that this is also possible in the world of Formal Methods. You may *safely* combine the methods of Gurevich, Lamport, Pnueli, Cousot-Cousot, Wonham, and many others.

For many reasons (synthesis, controllability, I/O, etc.) it is not too reasonable to restrict yourself to only one method. You may start with Evolving Algebras (ASM's), but at a certain point it is reasonable to restrict yourself to, say, TLA or LTL and then further to FOL and later to an intuitionistic setting or to S1S to compile your implementation.

In our example the idea is first to represent the system as restrictions on the sequences of events (typically as automata) and events as transition relations on (auxiliary) variables. These relations are given as the conjunction of a set of FOL-constraints. Later, ideas of abstract interpretation are used for the purpose of compilation.

Communication of Distributed Objects: A Temporal Logic Approach

Grit Denker

Technische Universität Braunschweig
(Joint work with Hans-Dieter Ehrich)

We present fundamentals of an approach to object oriented specification of distributed systems. In contrast to most of the work existing in this field about network of processors, hardware, etc., our concern is the high-level specification of distributed systems. We do not assume global time for concurrent object systems. For specifying such systems we propose DTL, a distributed temporal logic. The main contribution is that DTL is capable of specifying complex constraints about the behavior of concurrent systems without explicitely talking about concurrency. In particular, we introduce different kinds of synchronous communication in distributed systems such as deferred and immediate call, and deferred and immediate execution. The ideas are illustrated by examples given in TROLL, a formal object oriented specification language.

Origin Tracking and Software Renovation

Arie van Deursen

CWI Amsterdam

Legacy systems are software systems that resist change. Software renovation is an activity aiming at improving legacy systems such that become more adaptable, or at actually carrying out required mass modifications. A typical renovation is the year-2000 problem. Tools for carrying out year-2000 conversions look for initial date infections (seeds, such as suspicious keywords), propagate these through MOVEs and CALLs, try to reduce the number of infections found, and then (semi)-automatically modify the code using a widening or windowing approach.

Of great importance for year-2000 conversions and software renovation is an accurate data flow analysis tool that can be easily connected to all source languages used in the system to be renovated. In the context of the ASF+SDF formalism, the DHAL Data Flow High Abstraction Language has been proposed. Languages are easily mapped to DHAL and on top of DHAL several elementary data flow operations such as goto elimination and alias propagation have been defined.

Origin tracking is a general technique concerned with linking back analysis results, obtained for example from DHAL operations to the original source code. For transformations expressed in a functional style (using, e.g., term rewriting), origin information can be maintained automatically. For each reduction, origin annotations in the reduct are constructed in a way that depends on the form of the rewrite rule applied. We discuss several approaches (syntax-directed, common subterms, collapse-variables, any-to-top, non-linear rules), as well as their use in typical specifications occurring in a renovation setting.

Modularisation, Craig-Robinson Interpolation and the Validation and Construction of Refinements

Theodosis Dimitrakos

Imperial College, London

The encapsulation of specification refinement in terms of *implementation steps* ('canonic-steps') has been matured, within the last 15 years, as a 'logical/proof-theoretic' counterpart of the 'algebraic/semantical' approach to (formal) specification refinement. One contribution of this approach to refinement is establishment of a close relation between the compossibility of refinements (modularisation) and some interpolation properties of the underlying formalism.

In this talk I present a straightforward generalisation of th an implementation step to an arbitrary *Entailment System* (alias: π-Institution) which is then followed by analogous generalisation of the Craig-Robinson Interpolation (CRI) property. CRI can be viewed as a combination of Robinson's Consistency Theorem with Craig Interpolation Lemma and turns out to be similar to what is sometimes called 'Splitting Interpolation'. Modularisation, in this framework, is shown to coincide with the preservation under pushouts of a (closed under composition) class of faithful theory interpretations. The existence of Craig-Robinson Interpolants consequently shown to be a *necessary and sufficient* condition for the preservation of faithful (conservative) theory morphisms under pushouts and, therefore, a necessary and sufficient condition for the general case of Modularisation.

The *Uniform* version of Craig-Robinson Interpolation (UCRI) is also shown to facilitate the computation and mechanical validation of data refinements. Unfortunately, this particular UCRI is absent from most 'expressive' logics that are currently used. (Classical Propositional & Heyting's Intuitionistic Logic have UCRI, but Equational, First Order Classical/Intuitionistic do *not* have UCRI).

To compensate for this inadequacy, I present a generic method to extend *conservatively* a large class of Entailment Systems with which have some version of CRI to corresponding Entailment Systems which have a UCRI. This extension is accompanied by a (mechanisable) method to , *validate* and, in some cases, to *construct* correct refinements.

Refinement on the Entailment System of Classical First Order logic is presented as a case study. Classical first order logic, has CRI but lacks UCRI. The extension method results to a weak Second Order entailment which is conservative over the first order origin and has a adequately strong version of UCRI. The validation and construction of refinements are, then performed on the weak Second Order Entailment System ('development logic') whereas the specification takes place in the First Order subentailment ('specification logic').

Reconciling Real-Time with Asynchronous Message Passing

Radu Grosu

Technische Universität München
(Joint work with Manfred Broy, Cornel Klein)

At first sight, real-time and asynchronous message passing like in SDL and ROOM seem to be incompatible. Indeed these languages fail to model real-time constraints accurately. In this talk, we show how to reconcile real-time with asynchronous message passing, by using an assumption which is supported by every mailing system throughout the world, namely that messages are time-stamped with their sending and arrival time. This assumption allows us to develop a formalism which is adequate to model and to specify real-time constraints. The proposed formalism is shown at work on a small real-time example.

Making Formal Techniques Applicable for Non-Experts: Agendas and Strategies

Maritta Heisel

Technische Universität Berlin

Using formal techniques in software development may lead to a considerable gain in product quality because formal techniques make it possible to express and eventually prove *semantic* properties of the developed products (e.g., specifications, designs, programs, or test cases). A formal *method* should not only consist of a formal language with a rigorously defined semantics, but also include *methodological* support for its application.

I present two concepts designed to support non-experts in using formal techniques. The first concept of an *agenda* provides guidance for the application of formal techniques without the need for machine support. An agenda is a

list of activities to be performed when carrying out some task in the context of software engineering. Each step of an agenda is associated with an expression of the formalism to be used. Steps may also have associated *validation conditions* that provide application independent checks of the developed product. If the context in which a formal technique is to be applied is made sufficiently precise, agendas can be quite detailed. This is demonstrated by an agenda for specifying safety-critical software.

The second concept of a *strategy* aims at a machine supported application of formal techniques and a partial automation of software development activities. Strategies are a generic knowledge representation mechanism for formally representing knowledge about software development activities (that may be informally expressed using agendas). Simpler strategies can be combined to more powerful ones with *strategicals*. Strategies can be represented in a modular way, which makes them implementable in conventional programming languages. An abstract problem solving algorithm and a generic system architecture provide uniform implementation concepts for systems supporting strategy-based software development activities. All in all, strategies provide a comprehensive, formalism-independent and flexible framework for supporting the application of formal techniques by machine.

Agendas and strategies can be used to support various development tasks and formalisms. Examples are the specification acquisition, design of software systems using architectural styles, and program synthesis.

Domain Specific Specification Languages or: Captain Hook Visits the Telephone Company

James Hook

Pacific Software Research Center
Oregon Graduate Institute of Science and Technology

The talk presents a technique for the definition of domain specific languages in the context of a domain engineering process being applied at Lucent Technologies/Bell Labs. The talk is in four parts. Chapter 1, in which Captain

Hook prepares to hunt the illusive domain-specific language, outlines Software Design Automation (SDA), a collection of the techniques for finding potential specification languages in the workplace and capturing them with formal definitions. Chapter 2, in which Captain Hook visits the Telephone Company, outlines Weiss' domain engineering process, Family Abstraction, Specification and Translation (FAST), being applied jointly by researchers and developers at Lucent and discusses experiences to date with the integration of FAST and SDA. Chapter 3, in which Captain Hook shows his creation to Gyro Gearloose, presents technical details of the application of monads in the structure of a software component generator based on a domain specific language developed using the SDA process. The Talk concludes with Chapter 4, in which Captain Hook dreams of his future visit to the telephone company. This chapter discusses general and specific opportunities and challenges for the effective integration of formal methods and engineering practice.

Specware: Support for Formal System Development

Richard Jüllig

Kestrel Institute, Palo Alto

Specware is an attempt to realize the best of formal methods research in a software development environment. It represents a synergy of decades of research in formal specifications—algebraic specification and general logics—and abstract mathematical theories originally invented for dealing with complex structures—category theory and sheaf theory.

Specware is a tool that supports the modular construction of formal specifications and the stepwise and componentwise refinement of such specifications into executable code. Specware may be viewed as a visual interface to an abstract data type providing a suite of composition and transformation operators for building specifications, refinements, code modules, etc. This view has been realized in the system by directly implementing the formal foundations of Specware: category theory, sheaf theory, algebraic specification and general logics. The language of category theory results in a highly parameterized, robust, and extensible architecture that can scale to system-level software construction.

Software development in Specware is characterized by two tenets: description and composition. In Specware, we always deal with descriptions, i.e., a collection of properties, of the artifact that we ultimately wish to build. These descriptions are progressively refined by adding more properties, until we can exhibit a model or witness (usually a program) which satisfies these properties. Descriptions in Specware are written in one of several logics. Specware handles complexity and scale by providing composition operators which allow bigger descriptions to be put together from smaller ones. The colimit operation from category theory is pervasively used for composing structures of various kinds in Specware. Besides composition operators, one needs bookkeeping facilities and information presentation at various abstraction levels. Specware uses category theory for bookkeeping and abstraction.

A Formal Approach to Object-Oriented Software Engineering

Alexander Knapp, Martin Wirsing

Ludwig–Maximilians–Universität München

The goal of this talk is to show how formal specifications can be integrated into one of the current pragmatic object-oriented software development methods. Jacobson's method OOSE ("Object-Oriented Software-Engineering") is combined with object-oriented algebraic specifications by extending object and interaction diagrams with formal annotations. The specifications are based on Meseguer's Rewriting Logic and are written in an extension of the language Maude by process expressions. As a result any such diagram can be associated with a formal specification, proof obligations ensuring invariant properties can be automatically generated, and the refinement relations between documents on different abstraction levels can be formally stated and proved. Finally, we provide a schematic translation of the specification to Java and thus an automatic generation of an object-oriented implementation.

Object-Oriented Specification of Distributed Systems — Specification Style and Method

Ulrike Lechner

Universität Passau

The object-oriented specification language Maude has proven itself to be expressive. However the object model of Maude is very unconventional. We use alternative specification formalisms, algebraic and coalgebraic specifications and the μ-calculus to reason about the object model, the properties of Maude specifications and in particular the properties that are inherited via the set of object-oriented reuse and structuring concepts developed for Maude.

(Semi-)Automatic Test Generation from Rigorous Specifications

Marielle Doche, Michel Lemoine, Christel Seguin

ONERA CERT, Toulouse

One of the main problem the industry is faced with is the accuracy of testing accordingly the specifications that have been built. Indeed it is very common for industry to subcontract software from informal requirements document and informal specifications. Moreover because the confidence in the software it receives back is not high enough, the industry is used to develop some testing specification in a second step, independently of the informal specification it provided to the subcontractor.

The talk is mainly dedicated to the applicability, at an industrial level, of some formal methods in a new way, in order to derive test in a (semi) automatic manner from formal specifications that must be built from some informal requirements.

Starting from an existing design (an Automatic Teller Machine communicating through a connection line with a Central System) we have developed a formal Z specification composed of 3 corresponding (semi) independent state spaces. We then transform each operation (on state space) into a Disjunctive Normal Form of sub-operations, each one containing only AND-operators. Each sub-operation is no more than a "Test Class" from which "Test Data" are classically derived.

Because this 1st step is only able to help the designer to find out independent "Test Data" it is necessary to complement it with a 2nd formal specification allowing to specify the dynamic behavior of the system. For that purpose we have used TRIO, a 1st order logic including some temporal logic capacity. We then specify some temporal constraints we would like the dynamic behavior meets. Because TRIO works with a "temporal window" (from Now to "n" time units in the future or from "n" time units in the past down to "m" time units in the future) it is possible to derive ALL the "models" (sequences of assignments of values) that satisfy the temporal constraints.

Combinatorial explosion is avoided because values are restricted to these ones computed in the 1st step and because the considered "temporal window" is short enough.

The talk emphasizes as well the usefulness of combining two "compatible" formal notations, each one providing some accurate "by product".

Regularity $=\!\!\leftharpoondown$ Parallelism

Christian Lengauer

Universität Passau

We develop parallel versions of two programs for the polynomial product:

1. A *nested loop* program. Here, we are employing a geometrical space-time mapping model, the *polytope model*, in which nested loop programs can be parallelized automatically, optimizing some stated objective function like the number of parallel steps (execution time), the

number of processors, the number of communication channels, etc. The formal techniques employed in this method are linear algebra and linear programming. We present two parallel programs, each with a linear time and space complexity.

2. A *divide-and-conquer* program. Here we are considering a modification of the space-time mapping model for loop parallelization to obtain a parallel loop program for Karatcuba's polynomial product. The reason to choose divide-and-conquer over nested loops is to achieve a sublinear time complexity (at the expense of an increased number of processors).

The objective of the presentation is to discuss the strengths and weaknesses of the space-time mapping approach to the parallelization of loops and recurrences. Papers on this topic can be obtained on the Web under:

`http://www.uni-passau.de/~lengauer/`

Hierarchical Constructive Specifications

Jacques Loeckx

Universität Saarbrücken

Constructive specifications have been described in [1]. While their semantics is defined operationally they may—alternatively—be viewed as initial specifications satisfying a number of constraints. One of these constraints, the "termination constraint", guarantees the existence of a reduction ordering. This constraint is automatically verified when the operatioins are defined in a "primitive recursive way". In other cases it may be difficult to find a suitable reduction ordering.

Hierarchical constructive specifications are constructive specifications satisfying an additional constraint: informally speaking, the different operations have to be defined "successively". The goal of this talk is to present a very simple and intuitive proof method for the verification of the termination constraint of these specifications. The method presents similarities with the proof method described in [2]. As a difference it is model-oriented rather than presentation-oriented. Hence, contrasting with the method presented in [2] it has not been developed as an automatic proof method; on the other hand, it leads to proofs that are intuitively clear and easy to understand.

References

[1] Loeckx J., Ehrich H.-D., Wolf M. Specification of Abstract Data Types, Wiley-Teubner, 1996.

[2] Arts Th., Giesl J.. Termination of Constructor Systems, Internal Report IBN 95/34, Technische Hochschule Darmstadt, 1995.

$\daleth\Box$ (Refinement morphism = Component morphism)

Antonia Lopes

University of Lisbon

(Joint work with Jose Luiz Fiadeiro)

When several components are interconnected together to form a complex system, the overall behaviour of a system depends on the individual behaviour of its components.

In general, components are described in terms of the behaviour they ensure to have in any context and, hence, composition is given by conjunction, i.e. the properties of a system are given by the conjunction of the properties of its components. In this case, the system also acts as a refinement of its components in the sense that all their properties are preserved.

In the action-based approach to systems specification, it is not satisfactory to describe a system only in terms of the behaviour it ensures to have in any context. Since communication is based on the synchronisation of actions, a system cannot guarantee to have a liveness property in any context, unless it makes some assumptions on its environment. An alternative to the assumption commitment style of specification is to consider that systems are also described in terms of the behaviour they are willing to have when working as a component of a larger system. In this case, the willingness properties of a complex system are not given by the conjunction of the willingness properties of its components and, thus, the notion of refinement does not coincide with the notion of component of.

We illustrate this approach using a Modal Action Logic to specify the safety properties that a system ensures to have and the readiness properties (a kind of required nondeterminism) that a system is willing to have.

Shedding New Light in the World of Logical Systems

Alfio Martini

Technische Universität Berlin

The notion of an *Institution* is here taken as the precise formulation for the notion of a logical system. By using elementary tools from the core of category theory, we are able to reveal the underlying mathematical structures lying "behind" the logical formulation of the satisfaction condition, and by doing so, to get a both suitable and deeper understanding of the institution concept. This allows us to systematically approach the problem of describing and analyzing relations between logical systems, as well as to redesign the notion of an institution to a purely categorical level, so that the satisfaction condition becomes a functorial (and natural) transformation from specifications to (subcategories of) models. This systematic procedure is also applied to discuss and give a natural description for the notion of an institution morphism. The last technical discussion is a careful and detailed analysis of two examples, which tries, as an outcome, to outline how the new categorical insights could help in guiding the development of a unifying theory for relations between logical systems.

Accepted for publication in CTCS'97 (Conference on Category Theory in Computer Science)

Probabilistic Temporal Logic

Annabelle McIver

Oxford University

Various attempts have been made to extend temporal logic to include probabilistic reasoning, with mixed success. For example Feldman and Harel [1] present a probabilistic dynamic logic which covers probabilistic choice, but without nondeterminism; and Hart and Sharir [2] construct a probabilistic propositional logic which deals only with properties that hold with probability 1.

Our recent work [5] (following eg Jones [3]) is based on *expectations* rather than explicit probability distributions, and its smooth extension of predicate transformers seems to hold the key to a similar extension of temporal logic, including both nondeterminism (extending [1]) and 'proper' probabilities (lying strictly between 0 and 1, extending [2]).

In particular the theory of probabilistic predicate transformers [5] leads naturally to a reformulation of Morris' weakest-precondition-based temporal logic [4]. By defining appropriate generalisations of the predicate operators (implication, conjunction and disjunction) on the space of expectations, likewise many of the laws of standard temporal logic generalise also, making for a practical tool for the calculation of temporal properties and their associated probabilities.

Finally the interpretation of the 'probabilistic formulae' over (probabilistic) transition systems has lead to some startling insights into the operational interpretation of (even non-probabilistic) temporal operators, in terms of games.

References

[1] Feldman and Harel, A Probabilistic Dynamic Logic, Journal of Computer System Sciences 28, 193–215 (1984).

[2] Hart and Sharir, Probabilistic Propositional Temporal Logics, Information and Control, 70, 97–155 (1986).

[3] Jones, C. Probabilistic Non-determinism, Doctoral Thesis, 1990.

[4] Morris, J. M. Temporal Predicate Transformers and Fair Termination, Acta Informatica 27, 287–313 (1990).

[5] Morgan, McIver and Seidel. Probabilistic Predicate Transformers, ACM Transactions on Programming Languages and Systems, 18, (3) 325–353, (1996).

Specifying the Boiler in Timed Rewriting Logic

Peter Ölveczky$_{\rceil}$, Piotr Kosiuczenko$_{\rceil\rceil}$, Martin Wirsing$_{\rceil\rceil}$

$^{\rceil}$University of Bergen, $^{\rceil\rceil}$Ludwig–Maximilians–Universität München

In this talk crucial parts of our object-oriented algebraic solution of the steam-boiler specification problem are presented. The solution is written in Timed Maude. Timed Maude is a specification language under development where the static parts of the specified system are described by equational specifications, whereas the behaviour of a process is described by timed term rewriting. Timed Maude is based on Meseguer's Maude language, and its underlying logic is timed rewriting logic, an extension of rewriting logic to deal with hard real-time systems.

Industrial Use of ASMs for System Documentation

Peter Päppinghaus], Egon Börger]]

]Siemens AG,]]Università di Pisa

We report on ongoing work using ASMs to document the basic functionality of an existing large software system implemented in C++. The software to be documented is the toolset TRANSIT (TRain ANalysis and SImulation Tool set) developed and used at Siemens for the simulation of railway systems (cf. SIGNAL + DRAHT (88) 3/96).

The system consists of several components communicating with each other by a message passing mechanism. One of the goals of the work is to specify these components on a level of abstraction appropriate to express constraints about this interaction, which currently are not made explicit.

A Graphic Notation for Formal Specification of Dynamic Systems

Gianna Reggio

Università di Genova

Given an already established formal specification method for reactive systems, we develop an alternative graphic notation for its specifications to improve the writing and the understanding of such specifications and, hopefully, the acceptance of the method by industrial users.

Applying Formal Methods to Security Protocols

Peter Y. A. Ryan

DRA Malvern

We present an overview of the aims and key results of a 3 year research programme on the application of "mainstream" formal methods to the analysis and design of security protocols. By "security protocol" we mean a prescribed interaction between nodes of a distributed network that uses cryptographic mechanisms such as encryption, hashing, digital signatures etc to achieve certain security requirements. These can include:

- Authentication
- Confidentiality
- Key-exchange
- Integrity
- Anonymity
- Non-repudiation

The attraction of such protocols from the point of view of formal analysis is that they are very compact yet have to satisfy subtle properties in a hostile environment. They are, in Roger Needham's phrase: "typically 3 lines programmes that people still succeed in getting wrong".

The main thrust of the programme has been the use of CSP to formalise the network, the nodes, and the hostile agent(s). The security properties can also be elegantly formalised in CSP and then refinement checks can be conducted to establish whether or not the system refines the property in question. These checks are conducted in a highly automated fashion using the CSP model-checker FDR. If the check fails, FDR returns a counter-example that in effect details an attack on the protocol. The model-checker operates by exhaustively checking all (essentially distinct) behaviours of the two processes.

Once the specification has been established the approach is thus highly automated. Considerable ingenuity may be required of course to compress the models to keep the state space size small enough for model-checking to work whilst ensuring that no significant behaviours are lost.

The effectiveness of this approach has been demonstrated by, for example, the finding of a number of novel attacks on well-known protocols, most notably Lowe's uncovering of an attack on the Needham-Schroeder Public-key protocol. This protocol had previously been considered to be secure since it's publication some 17 years earlier and indeed had been "proven" secure using the BAN-logic.

The application of the concepts and of the concepts and techniques to other areas such as safety-critical and fault-tolerant systems is also outlined. For example the concept of non-interference used in security to formalise the absence of information flow can be adapted the weaker notion of non-disruption that neatly formalises the concept of fault resilience.

In summary: it has been shown that:

- security protocols are a very fruitful application area for formal methods,
- Model-checking techniques can be highly effective,
- formal methods can yield valuable results at reasonable cost as long as the application is chosen carefully and tools and techniques are suitably matched.

A Formal Object-Oriented Method Inspired by Fusion and Object–Z

Wolfram Schulte

Universität Ulm

We present a new formal OO method, called Fox, which is a synergetic combination of the semi-formal Fusion method and the formal specification language Object–Z. To manage complexity and to foster seperation of concerns, Fox distinguishes between analysis and design. In each phase structure and behaviour specifications are developed step-by-step. The specifications may be graphical or textual. We give proof obligations to guarantee that the developed models are formally consistent and complete, and that the resulting system conforms to the original specification. By walking through a simple example—a graph editor—we illustrate the application of Fox.

Integrating Object–Z and CSP

Graeme Smith

Technische Universität Berlin

While most specification languages can be used to specify entire systems, few, if any, are particularly suited to modelling all aspects of such systems. The formal development of particularly large, or complex, systems can, therefore, often be facilited by the use of more than one formal specification language.

This talk presents a method of formally specifying concurrent systems using the object-oriented state-based specification language Object–Z together with CSP. The rationale is that Object–Z provides a convenient method of modelling the complex data structures needed to define component processes, and CSP enables the concise specification of process interaction. The advantage of Object–Z over more traditional state-based languages such as Z is that its class structure provides a construct easily identifiable with CSP processes. The basis of the integration is a semantics of Object–Z classes identical to

that of CSP processes. This enables classes specified in Object–Z to be used directly within the CSP part of the specification.

The approach uses the exisiting languages without altering their syntax or semantics. This makes it accessible to users who are already familiar with the languages and also enables the use of tools and methods of verification and refinement developed for them.

Using Formal Methods for the Validation of Prolog Programs

Karl Stroetmann

Siemens AG

(Joint work with Thomas Glaß and Martin Müller)

So far, attempts to apply formal methods for the verification of software had only a very limited success. This is mainly due to the excessive costs associated with interactive formal proofs. The situation is different in those areas where automatic proof systems can be applied, e.g. in hardware verification. It is therefore argued that in order to apply formal methods we should shift our attention from verification to validation and apply automated theorem proving to the static analysis of software.

This idea is demonstrated with the programming language Prolog. To this end, Prolog is extended with a type system. Furthermore, a new declarative semantics that describes the effects of the cut operator faithfully is introduced. Using this semantics it is possible to check the completeness and consistency of a Prolog program with the help of an automatic theorem prover. The industrial evaluation of this idea has shown that this approach does increase the quality of software dramatically without increasing its cost.

Another Look at Localities and Failures in the π-Calculus or: Everything May Fail

Markus Wolf

Universität Saarbrücken

In 1994 Amadio and Prasad introduced an extension of the π-calculus for handling located processes, channels and failure of locations, [1]. This seemed to be an interesting setting to study mobile objects. Unfortunately, the calculus assumes one permanent or non-failing location which does not correspond to the intuition that every location may fail. While removing the assumption and trying to model a mobile process it turned out that it is not possible to model mobility that is robust under failure of the original location of a process. However, it is easy to model the situation adequately if one moves to a higher-order version of the located π-calculus. This seems to contradict Sangiorgi's result [2] that every process of the higher-order π-calculus can be compiled to a process of normal π-calculus exhibiting essentially the same behaviour. Using Sangiorgi compilation on a mobile process in the higher-order located setting yields essentially the mobile process in the first-order located setting which is bot robust under failure. Hence a notion of bisimulation, as for example in [1], which does not observe failed locations would distinguish the two processes.

References

[1] R. Amadio, S. Prasad. Localities and Failures. In P. S. Thiagarajan (ed.), Proc. 14th Conf. Foundations of Software Technology and Theoretical Computer Science (LNCS 880). Springer, Berlin, 1994, pp. 205–216.

[2] Davide Sangiorgi. Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms. PhD Thesis, University of Edinburgh, 1992.