Contents

Introduction

The aim of this workshop was to bring together experts to describe and evaluate their approaches to formal methods for program construction, especially with respect to their suitability for computer support. Emphasis should be put on comparisons and synergetical effects between the different approaches.

The workshop was intended as a forum for researchers and developers for gaining awareness of current practical and experimental work across the breadth of the field.

Among the topics encompassed are

- specification languages and methods
- algebraic algorithmic calculi
- program development based on type theory, category theory and logical theorem proving
- formalization of methods in meta-calculi

Many thanks to Debora Weber-Wulff for portraying the speakers and the castle.

The organizers

Jean-Pierre Finance Stefan Jähnichen Jacques Loeckx Douglas Smith Martin Wirsing

Relational Theory of Data Types

Roland Backhouse

University of Eindhoven

Category theoreticians argue that monotonic functions are just functors in a particular category, Galois connections are just adjunctions, closure operators are just monads, and so on. In this talk we argued the opposite: category theory may be viewed as lattice theory with constructive evidence for each instance of an ordering, and such that the constructions obey certain coherence properties. Thus, category theory equals coherently constructive lattice theory. The usefulness of this idea was illustrated by outlining how it is possible to adapt proofs in the algebra of regular languages to the construction of programs establishing natural isomorphisms and simulations between list structures. An example is the star decomposition rule of regular algebra:

$$(a + b)^* = b^*(ab^*)^*$$

The corresponding programming problem is the so-called lines-unlines problem. (Consider a list of symbols drawn from two sets, the set a containing separators and the set b containing ordinary letters. Call a list of b's a "line". Write a "lines" program to split the list into a line followed by a list of (separator, line) pairs. Construct in addition the "unlines" program, the inverse of the lines program.)

Program Synthesis as Higher Order Resolution

David Basin

MPI Saarbrücken

My research concerns the translation and formalization of program development calculi in first and second-order logic. Given a calculus, I formalize program development rules as derived rules in the appropriate theory. I apply these rules using higher-order resolution to build programs from proofs of their correctness. I partially automate such proof construction using techniques from inductive theorem proving.

My work is based on Paulson's Isabelle system. I use Isabelle to formalize the rules of desired synthesis calculi and derive them in the appropriate theory. This provides a way of formally (on a machine) establishing the correctness of the proof rules. This is possible since Isabelle's meta-logic is higher-order so we may use it both to formalize rules (even for weak logics like first-order logic) and reason about them as well. Once the rules of the calculi are derived, I build proofs using higher-order resolution whereby the desired program is left unspecified and built incrementally via unification during resolution steps. Resolution steps are applied interactively in Isabelle, but are partially automated by use of tactics. In particular, I apply tactics from inductive theorem proving to partially automate synthesis proofs. Overall, the framework I propose gives a clean separation of the underlying development logic, the derived rules, and heuristics (tactics) for automating proof construction.

I have reconstructed several synthesis calculi in this framework. One example is logic program synthesis. There Prolog programs are synthesized from specifications in sorted,

first-order theories. Proofs are of the form of an equivalence between a specification S and a program P. The equivalence is proven in first-order logic and P, the program built, is in a restricted subset of the language of first-order predicate calculus that can be directly translated to a Prolog program. The proof rules I use can be used for both verification and synthesis proofs. If the proof begins with P given as a concrete logic program, then the proof can be seen as one of verification: showing P meets the specification S. On the otherhand, as proof rules are applied using higher-order unification P may be left as an unknown and constructed by application of proofs rules. This methodology has also been used to implement a calculus for synthesizing circuits from correctness proofs (based on the Kent "Formal Synthesis Calculus") and a calculus for functional program synthesis with functionality similar to that of Manna and Waldinger's Deductive Tableaux.

Verification Support for Compiler Development

Bettina Buth

University of Kiel

In principle, program verification is the only adequate means to ensure the correctness of software with respect to precise specifications. But since realistic programs such as code generators and other parts of compilers tend to be large and complex, some mechanical support is necessary for the verification of such programs. The verification system PAMELA is intended to be used for the verification of the correctness of sequential programs consisting of sets of mutually recursive function and procedure definitions (called explicit specifications) with respect to implicit specifications in the form of pre- and post-conditions. The approach is modular in that it allows to prove the correctness of each

single function with respect to its implicit specification independent from the other full set of the implicit specifications. The approach is based on a fixpoint induction argument. The induction hypothesis is adopted for recursive calls. The proof for a single functions is performed by a case distinction determined by the various (abstract) results of the function body. These are determined by the path through the body, which are calculated together with their result and the strongest postcondition in the splitting phase of PAMELA. For each of the pairs of result and strongest postcondition (representing one single path through the body) it has to be checked that the pathcondition implies the validity of the postcondition of the function for the respective result. Further proof obligation arise for recursive calls encountered during splitting: for these it must be ensured that the precondition is fulfilled for the arguments of the call. PAMELA tries to discharge all proof obligations automatically by rewriting based on conditional rules provided with the specification and a set of build-in rules. The rewriting process is determined by user-defined strategies.

Growing Programs from Proofs

Debora Weber-Wulff
Technical College of Berlin

It is not always the case that we have a complete specification to verify against an implementation, or a complete implementation that "just" needs to be verified. It is often necessary for the specification, implementation and proof to be "grown" together. In this talk a small proof example was discussed that was done together with Bill Bevier from Computational Logic, Inc. using the Boyer-Moore theorem prover. The problem domain

was the construction of an "indentator function" that converts the absolute indentations found in some programming languages (occam, Miranda, etc.) to relative indentations representing block begin, block end, and statement separators.

The "growth" of the program from a naive implementation and specification was discussed. This involved first separating out certain concerns, for example dealing with empty lines and determining how many blanks constitute one indentation level. The resulting function,

```
indentator (toks) =
  (flatten1 (emit (halve (remove-empty-lines toks))))
```

defines four "passes" over a sequence of tokens. It is possible to combine the passes manually into just one function with an appropriate case structure, and to prove that this function is the same as `indentator`. This function can be said to have been grown together with the proof. A discussion about some of the difficulties of proving non-trivial programs ensued.

Constructive versus axiomatic and initial specifications

Jacques Loeckx (with M. Wolf)
University of Saarbrücken

A classification of specification methods of abstract data types distinguishes between axiomatic, initial and constructive specifications. Axiomatic specifications generally use predicate logic and loose semantics; informally, an axiomatic specification is interpreted as

the class of all models of a set of formulas. As a variant it is possible to consider only models freely generated by a set of "constructors". Initial specifications use equational logic and initial semantics; informally, an initial specification is interpreted as the class of all initial models. There exist different formalisms for constructive specifications; all of them use operational semantics. It is clear that these three semantics are fundamentally different: loose semantics have to do with logic, initial semantics with algebras and homomorphisms and operational semantics with algorithms.

The talk presents a constructive specification method inspired from the theory of recursive functions. It is proved that for these specifications the operational semantics, the loose semantics and the initial semantics coincide.

An Algebraic/Axiomatic Interpretation of Data Flow Diagrams

Friederike Nickl

University of Munich

In this talk a schematic translation of data flow diagrams into stream-based specifications in the algebraic/axiomatic specification language SPECTRUM is presented. The need for such a translation arised during the development of a formal requirements specification for a medical information system in the KORSO case study HDMS-A [CHL 94]. In order to establish a specification which was communicable also to persons without a deep knowledge in formal methods, it was decided to combine semi-formal diagrammatic descriptions with formal specifications. Therefore, in [SNM+93], the dynamic behaviour of the system is described by data flow diagrams extended by axiomatic specifications for the actions associated with the nodes in the diagrams.

Using the translation method presented in [Nic 93] data flow diagrams and axiomatic specifications of actions combine to form a single SPECTRUM-specification. In this way a formal semantics for extended data flow diagrams is provided. The stream-based specification obtained by the translation can serve as a foundation for the formal analysis of the dynamic behaviour of the system. In particular, consistency checks between the specification of the actions and the diagrams can be performed. An aspect of current research is the generation of proof obligations on the specification of actions which guarantee that all the traces described by the diagrams are also admissible with respect to the specification of actions.

References

[CHL 94]
F. Cornelius, H. Hußmann, M. Löwe: The KORSO Case Study for Software Engineering with Formal Methods: A Medical Information System. In: M. Broy, S. Jähnichen (eds.), KORSO - Correct Software by Formal Methods, to appear 1994.

[SNM+93]
O. Slotosch, F. Nickl, S. Merz, H. Hußmann and R. Hettler: Die funktionale Essenz von HDMS-A, Technischer Bericht I9335, Technische Universität München, 1993.

[Nic 93]
F. Nickl: Ablaufspezifikation durch Datenflußmodellierung und stromverarbeitende Funktionen, Technischer Bericht I9334, Technische Universtität München, 1993.

Software Verification with the KIV System

Wolfgang Reif

University of Karlsruhe

This research was partly sponsored by the BMFT project KORSO.

This talk presents a particular approach to the design and verification of large sequential systems. It is based on structured algebraic specifications and stepwise refinement by program modules. The approach is implemented in KIV (*Karlsruhe Interactive Verifier*), and supports the entire design process starting from formal specifications and ending with verified code. Its main characteristics are a strict decompositional design discipline for modular systems, a powerful proof component, and an evolutionary verification model supporting incremental error correction and verification. The talk presents the design methodology for modular systems, a feasible verification method for single modules, and our evolutionary verification technique based on reuse of proofs. We report on the current performance of the system, compare it to others in the field, and discuss future perspectives.

References

[1] Reif W., *The KIV-Approach to Software Verification: State of Affairs and Perspectives*, Broy, Jähnichen (eds.), *KORSO, Correct Software by Formal Methods*, Springer LNCS, 1994 (to appear).

[2] Reif W., Stenzel, K., *Reuse of Proofs in Software Verification*, Conference on Foundations of Software Technology and Theoretical Computer Science, Bombay, India, Shyamasundar (ed.), Springer LNCS 761, 1993.

[3] Reif W., *Verification of Large Software Systems*. Conference on Foundations of Software Technology and Theoretical Computer Science, New Dehli, India, Shyamasundar (ed.), Springer LNCS 1992.

[4] Heisel, M., Reif, W., Stephan, W. : *Formal Software Development in the KIV System*. in Automating Software Design, Lowry McCartney (eds), AAAI press 1991.

[5] Heisel M., Reif W., Stephan W., *Tactical Theorem Proving in Program Verification*. 10th International Conference on Automated Deduction, Kaiserslautern, FRG, Springer LNCS 1990.

Second-Order Translations for Minimal, Quasi-free, Observational and Behavioural Specifications

Pierre-Yves Schobbens

University of Namur

Besides explicit axioms, an algebraic specification language contains model-theoretic constraints such as term-generation or initiality. For proving properties of specifications and refining them, an axiomatization of these constraints is needed; unfortunately,

no effective, sound and complete proof system can be constructed for most constraints of algebraic specification languages.

We propose thus a translation to second-order logic. This logic is powerful enough to provide exact translations for constraints commonly found in specification languages, and simplified forms useful for the universal fragment, that rest on the characterization of term algebras. They are shown to be sound and complete, (but not effective, since the underlying second-order logic is not effective). A good level of machine support is possible, however, using higher-order proof assistants (e.g. Isabelle). set theory.

The talk only presented the basic case of free algebras; but the report contains also a treatment of extensions of free models designed

to deal with disjunctive and existential axioms, namely the minimal models of Bidoit, the quasi-free models of Kaplan, and the surjective models. The report also treats behavioural equality and observational abstraction.

Consistency requirements (such as persistency) can also be expressed and proved in second-order logic.

Logical Frameworks

Jan Smith

University of Gothemburg

This talk gives a background and presentation of type theory as a logical framework. The main idea of a logical framework is that the user of the framework should just have to input the rules and axioms she wants to use and the framework will then give a interactive proof system which will take care of all implementational issues.

A logical framework based on type theory uses the idea of propositions as types; hence proof verification becomes type checking. The first proof checker based on type theoretical ideas was Automath, developed by deBruin in the sixties. During the last five years several truly interactive system have been implemented: Coq at INRIA in France, LEGO at Edinburgh and ALF at Göteborg. Coq and LEGO are based on Coquand and Huet's Calculus of Constructions and ALF is based on Martin-Löf's type theory with inductive definitions.

A number of examples have been developed in these systems, ranging from verification of a data link protocol used in industry to extracting algorithms from constructive proofs in pure mathematics, like Ramsey's theorem.

ALF – An interactive editor for programs and proofs

Bengt Nordström
University of Gothemburg

Alf is an interactive editor for proofs and programs. It is based on the idea that to prove a mathematical theorem is to build a proof object for the theorem. The proof object is directly manipulated on the screen, different manipulations correspond to different steps in the proof. The language we use is Martin-Löf's monomorphic type theory. This is a small functional programming language with dependent types. The language is open in the sense that it is easy to introduce new inductively defined sets. A proof is represented as a mathematical object and a proposition is identified with the set of its proof objects. The basic part of the proof editor can be seen as a proof engine with two basic commands,

one which builds an object by replacing a placeholder in an object by a new object, and another one which deletes a part of an object by replacing a sub-object by a placeholder.

As an example, we present a completely formal proof of the correctness of a simple compiler. The compiler translates an arithmetic expression (with one variable) to a list of instructions for a simple stack machine. We give the operational semantics for the two languages involved and show that the compiled expression is correctly executed on the stack machine.

`Seduct` — A Proof Compiler for Program Verification

Karl Stroetmann

Siemens AG

Up to now there have been two competing paradigms in software verification:

- the automatic paradigm,
- the interactive paradigm.

Due to the complexity of the verification conditions encountered in practice, the automatic paradigm has so far been unsuccessful. On the other hand, purely interactive theorem proving is far too tedious to be taken seriously in industrial applications. Therefore we introduce the concept of a *proof compiler*. The idea behind this approach is the following: In order to prove a piece of software correct you should first sketch a proof of its correctness with pencil and paper. This proof should then be transformed into a formal notation. The

result of this transformation will be called an *abstract proof*. By its nature and origin this proof is in general incomplete. This proof will then be given to a proof compiler. If the the abstract proof is detailed enough the proof compiler will be able to transform it into a complete formal proof.

In my talk I discussed the tool SEDUCT, which has been designed as a proof compiler. To be efficient this tool has a powerful automated theorem prover embedded. The talk concentrated on those aspects of the theorem prover that make this theorem prover especially tailored to the problem of software verification. There are two aspects which are particularly important in the context of software verification:

- the theory of a linear order,
- the freeness axioms.

We have hard–wired these theories into the theorem prover embedded in SEDUCT.

Furthermore, certain aspects of the command language of SEDUCT were described. In particular, the methods for proving inductive theorems were discussed. In practice it has turned out that structural induction and computational inductions are sufficient. In order to automate inductive theorem proving, the introduction of parametrized macros has proved to be very successful.

Finally, a small example demonstrated the "look–and–feel" of SEDUCT. Up to now SEDUCT has been successfully used to verify various sorting algorithms, the correctness of a tautology checker, and a lift control. Currently the verification of a functional BDD–package is being attacked.

References

[1] K. Stroetmann. SEDUCT — A Proof Compiler for First Order Logic. In: M. Broy and S. Jähnichen (eds.): KORSO, *Correct Software by Formal Methods*, Springer LNCS, 1994.

Verifying Functional Programs

Simon Thompson

University of Kent

It has long been an article of faith that functional programs are more amenable to verification than imperative programs, yet there has been little work in the area of larger-scale verification of programs. Our aim is to axiomatise a real (lazy) functional programming language – Miranda – and to build implementations of the logic to support substantial verification work.

Our logic for Miranda covers all the high level features of the language, including how to axiomatise both the types of the language and the definitions given in particular scripts. Novel in the treatment of types is the use of multiple induction principles to characterise the multiplicity of subtypes of lazy structured types such as lists.

Definitions in Miranda appear to be simply equations – a right-hand delivers the value corresponding to a particular left-hand side. This first interpretation has to be modified to deal with pattern matching, guards and local definitions, each of which adds some complication, which is compounded by their combination. We present a full algorithm for translating a definition into a logical assertion.

We conclude with a discussion of implementations of the system built in a combination of Miranda and C – the MTP prover – and one built on top of Paulson's Isabelle theorem prover.

An algebra of mathematical derivations

Michel Sintzoff (with M.Simons and M.Weber)
University of Louvain

Mathematical derivations here include proofs of theses from hypotheses, as well as deductions of programs from specifications. Thus, proofs are related to program derivations rather than to the resulting programs.

The proposed algebra consists of a binary operation, called maplet, of a monoid with a composition operation, and of a complete, distributive lattice with a join operation and a meet operation. The maplet serves to build new derivation rules (viz. inference rules or production rules) from available derivations, including previous rules. The composition operation is used for composing derivations sequentially, as in the transitive composition of rules. Joins allow to define sets of derivations, so as to open relevant search spaces; meets serve to restrict the actual search in these derivation sets, by filtering these on the basis of desired goals (viz. theses to be proven or programs to be constructed).

The definition of the algebra is completed by interface axioms between its components. These additional axioms are inspired by those for relation algebras and for function spaces: in particular, composition distributes over joins, and the LHS and RHS of maplets are respectively contravariant and covariant wrt. to the lattice ordering. This algebra reminds of relation algebras and of Freyd's allegories (a relational variant of categories): the main difference is the maplet operation (viz. rule construction). The application operation is introduced as a Galois adjoint of the maplet operation.

The considered algebra allows to express assumed deduction rules (production rules and inference rules) as well as derivations (linear ones and structured ones) using these rules. Derivations can be contracted by eliminating compositions of rules, similarly to relational composition or to logical syllogism. Maximally contracted derivations correspond to theorems or to new rules. This style of expression resembles the one elaborated in using the typed functional language Deva, after continued experiences. Indeed, the proposed algebra can be seen as an algebraic modelization of such a language. It amounts to a lifting of derivations from the level of functions to the level of propositions (viz. types). This seems more understandable by users, and allows to present a clear semantic model. Another feature of the proposed algebra is the internalization (by way of compositions) of tactics as used in many frameworks.

Modules for Mathematical Derivations

Matthias Weber

Technical University of Berlin

The expression of mathematical derivations takes place at two levels: at a local level of individual deductions and tactics and at a global level of systems of interrelated theories. In this talk, I presented and discussed several concepts for structuring formal mathematical derivations at the global level. The underlying philosophy was to apply successful engineering concepts, such as modularization, parametrization, encapsulation, and inheritance, for structuring very large (formal) systems of software.

At a first level of structuration in-the-large, conceptually related functions, axioms, definitions, tactics, theorems, and their proofs are encapsulated within (possibly parametrized)

theories. Since the concrete structure of the proofs, and their auxiliary lemmas and tactics, is immaterial from an outside point of view, they are presented separately from the other structures of a theory. At a second level of structuration in-the-large, conceptually related theories are encapsulated within *theory-systems*.

All illustrations of these concepts were based on a new algebraic notation for formal derivations (cf. related talk by M.Sintzoff given at the same workshop).

My talk ended by stressing the importance of presentation and communication concerns for satisfactory formal proof tools.

Computation with Proofs in Linear Logic

Didier Galmiche
CRIN-CNRS & INRIA Lorraine, Nancy

In this talk, we present a proof-theoretic foundation for proof construction and transformation in full linear logic. At first, we systematically study the permutability properties of the inference rules in this framework and exploit them to introduce an appropriate notion of movement of an inference in a proof. Then we investigate the possibilities of proof normalization which depend on the proof search strategy and the fragment we consider. Thus, we can define the concept of normal proof forms (that are complete and tractable) that might be the basis of works about automatic proof construction and logic programming languages based on linear logic. Moreover, we can use these results for proof transformations and interactive proof development in some fragments of linear logic.

Reasoning about Local Variables

Kurt Sieber

University of Saarbrücken

Traditional denotational semantics for block structured languages with local variables does not capture the intuition that a global procedure has no access to a locally declared variable. Hence such a semantics cannot be used to validate important reasoning principles for local variables like the concepts of 'representation invariant' and 'representation independence' in object oriented programming languages.

The search for better denotational models of such languages began in the mid 80s, and the two most recently developed models by O'Hearn/Tennent and by myself seemed to capture the desired reasoning principles, at least they worked for all examples of observational congruences which had been presented in the literature. For my own model I now succeeded to prove that it validates *all* observational congruences for program pieces whose free procedures are of order ≤ 2 (and this subsumes all the examples in the literature). In more technical terms: The model is fully abstract for the second order subset of an ALGOL-like language.

It remains an open question whether the O'Hearn/Tennent model satisfies the same property and whether the result can be generalized to types of order > 2.

Formalization of Software Construction in a Type-Theoretic Setting

Friedrich von Henke

University of Ulm

We show how, based on the idea of semantic subtypes, a type theory with strong Sigma-types (dependent sums) und Pi-types (dependent products) provides the bases for an expressive specification mechanism in which all specifications are represented as types. Realizations of specifications are represented by elements of the types, and parameterized specifications as mappings from specifications to specifications. We show how transformational approaches to program derivation like that of D. Smith can be formalized in this setting: schematic algorithms, their basic theories and enabling conditions can be represented in the language, using higher-order functions and predicates, and formally verified in the given framework.

Partial correctness wrt a single assertion

Stefan Sokołowski

Polish Academy of Sciences

As probably generally realized, the classical Hoare's logic of partial correctness makes a rather poor specification system. Its expressiveness is low, therefore it is unavoidable to extend it by additional information (e.g., ghost variables, or lists of variables that may change value, etc.). The reusability of the concept is definitely unsatisfactory.

In my talk, I have shown how the removal of one assertion in the pair of assertions in Hoare's triple results in a remarkable upgrading of the notion's properties. The main idea is to allow a special assertion-valued place holder (denoted by Φ) to occur inside assertions. Then, let p **sat** a, where p is an imperative program and a is an assertion with a possible occurrence of Φ, stand for the collective statement of partial correctness

$$\{a\}p\{\Phi\} \quad \text{for any formula } \Phi.$$

For instance,

p **sat** $\Phi[r^n/x]$	means	"p sets x to r^n and does not change other variables"
p **sat** $\forall_x \Phi$	means	"p may change x and does not change other variables"
p **sat** $\forall_{x'}(x' < x \Rightarrow \Phi[x'/x])$	means	"after p, x is smaller than before and other variables are unchanged"

This gives rise to a formalism, which is as manageable as Hoare's logic; in fact, the respective inference system is very similar to the classical case. It is almost arbitrarily reusable. Its expressiveness is comparable with that of binary postassertions and by large superior to that of Hoare's logic.

The price for this improvement lies mostly in a slight deterioration of the readability of specifications and in the need to get used to dealing with formulas with explicit substitution operators. On the cost side, we also have to put the inevitability of having local variables in the language.

Characterizing Behavioural Semantics and Abstractor Semantics

Michel Bidoit (with Rolf Hennicker and Martin Wirsing)

Ecole Normale Supérieure, Paris

Observability plays an important role in program development. In the literature one can distinguish two main approaches to the definition of observational semantics of algebraic specifications. On one hand, observational semantics is defined using a notion of observational satisfaction for the axioms of a specification and on the other hand one can define observational semantics of a specification by abstraction with respect to an observational equivalence relation between algebras. In this talk we present an analysis and a comparative study of the different approaches in a more general framework which subsumes not only the observational case but also other examples like the bisimulation congruence of concurrent processes. Thereby the distinction between the different concepts of observational semantics is reflected by our notions of behavioural semantics and abstractor

semantics. Our main results show that behavioural semantics can be characterized by an abstractor construction and, vice versa, abstractor semantics can be characterized in terms of behavioural semantics. Hence there exists a duality between both concepts which allows to express each one by each other. As a consequence we obtain a sufficient and necessary condition under which behavioural and abstractor semantics coincide. Moreover, the semantical characterizations lead to proof-theoretic results which show that behavioural theories of specifications can be reduced to standard theories (of some classes of algebras).

Proof Systems for structured specifications with behaviour

Martin Wirsing (with Rolf Hennicker)
University of Munich

In this lecture complete proof systems for proving properties and refinements of structred algebraic specifications are given. The underlying specification language is a version of ASL which contains constructs for forming basic specifications, export interfaces, the combination of specifications and the observational behaviour of specifications. Following a result of Hennicker, Bidioit and Wirsing (ESOP_94), this operator is defined as the observational closure of the fully abstract models of a specification. For technical reasons two other operators are added: the observable quotient and the class of fully abstract models of a specification.

A sound and complete proof system for deriving the validity of first order formulas is given. The proof system is relative to each specification SP and extends first order logic by particular axioms and rules for SP. In particular for specifying the fully abstract

model of the specification an infinitary context induction rule has been added to first order logic. Similarly the observable quotient needs an infinitary rule. The proof system for the observable behaviour is constructed from the one characterizing the fully abstract models of a specification using a generalization of the omitting type theorems the completeness of the system is shown.

A refinement of a specification SP by a specification SP_ is defined as model class inclusion

with SP and SP_ having the same signature. A proof system with exactly one rule for each operator of ASL is given that allows one to derive the validity of a refinement. It is shown that this proof system is sound and complete.

References: R. Hennicker, M. Bidoit, M. Wirsing: Characterizing Behavioural Semantics and Abstractor Semantics. ESOP_94. To appear in Lecture Notes of Computer Science.

R. Hennicker, M. Wirsing: Behavioural Specifications. International Summer School "Logic and Algebra of Specification", Marktoberdorf, 1993.

Behavioural Specifications of Multiagent Systems

Władysław M. Turski

University of Warsaw

For historic reasons, programming (and its theory, as well as methodology) evolved from the *computing paradigm*. Many computer applications in common use do not fit this paradigm well. Here we follow another paradigm, *viz.* that of *system behaviour*, in which finite *actions* are undertaken and accepted in specific circumstances. In this approach

there is no notion of action sequencing, therefore we avoid all problems of synchronization and global time. Instead, we recognize that action executions are not instantaneous and—therefore—the circumstances under which a particular action was deemed desirable may have changed during an execution to ones under which its outcome is no more needed.

The chosen paradigm accommodates an arbitrary degree of parallelism in the sense that an unspecified number of actions may be executed concurrently and—as far as the execution processes are concerned—independently. All actions that *can* be initiated in a given state, *are* initiated, regardless of any possible future conflict; conflicts are resolved in the accepting state.

Observable system behaviour obtains from successful executions of actions. For each system the collection of actions is determined by its specification which associates each action with *two* guards. A *preguard* characterizes the set of system *states* in which an action may be undertaken, a *postguard* characterizes the set of states in which the outcome of the executed action is acceptable *i.e.* may be reflected in the system state. Each action is assumed finite if undertaken in a state satisfying the corresponding preguard.

Actions are performed by *agents* of which there is an unspecified number (often it is essential that the number of agents exceeds one). Each agent is capable of executing any action, but different agents may need different lengths of time to complete the same action. No agent is allowed to idle in a system state in which a preguard is satisfied. Actions are assigned to agents in a fair way: when more than one can be assigned, the actual choice is unbiased.

If, at the instant of action completion the *system* state satisfies the corresponding postguard, the action outcome is instantaneously accepted. In other words, the paradigm aims to capture behaviours exhibited by systems in which actions take time (and may be futile) but all controls and state-updates are instantaneous.

States of the system are represented by (vector) values of a variable $\boldsymbol{z}$, possibly with subscripts. Truth valued functions of the state, known as *predicates*, will be denoted by capital italic letters $P, Q, R, \ldots$, possibly with subscripts. Actions will be denoted by $a, b, \ldots$, possibly with subscripts. Actions may be identified with their programs.

$$(P, Q) \rightarrow [\boldsymbol{x}|a|\boldsymbol{y}]$$

is the notation employed for specification of an action a that may be undertaken in a state satisfying predicate P, and whose outcome is accepted iff at the instant of its termination the system state satisfies predicate Q. Variables in $\boldsymbol{x}$ constitute the inputs to a, variables in $\boldsymbol{y}$, its output. Actions are executed on a *private* state which includes all variables in $\boldsymbol{x}$ and in $\boldsymbol{y}$.

Primary objects of interest are specifications of the form

$$(P_i, Q_i) \rightarrow [\boldsymbol{x}_i|a_i|\boldsymbol{y}_i] \text{ for } 0 \leq i < N \tag{1}$$

which are intended to prescribe total system behaviour.

This paradigm, first described in [1], can be successfully applied to classic problem of concurrency an has some interesting applications to control problems [2]. It may be used

for static analysis of such problems as "is given state stationary" and "which actions may modify a given state".

References

[1] On specification of multiprocessor computing. *Acta Informatica*, 27:685-696, 1990

[2] On doubly guarded multiprocessor control system design. In *Proceedings of the 4th DCCA Conference*, pages 1–11, San Diego, California, 1994

Verification of a Communication Network

Hardi Hungar (with Roland Baumann and Gudula Rünger)

University of Oldenburg

The first author has been funded by the German Ministry for Research and Technology (BMFT) within the project KORSO (Korrekte Software) under grant No. 01 IS 203 P. The responsibility for the contents lies with the authors.

Temporal logic is widely recognized as a useful specification language for reactive systems. Advantages of temporal logics are that both the tautology problem and the model check problem are decidable. Therefore, one can hope for good tool support for the development process. However, weaknesses of the temporal logic approach are lack of expressiveness of the logic and the complexities of the decision procedures which often are too high for completely automatic verification.

In this work, a real life example is studied. We tried the verification of a message passing system. We used a collection of techniques to overcome the difficulties mentioned

above. Data independence reasoning makes up for the inability of unbounded counting in formulae. Modular assumption/commitment specifications and rules for the parallel composition reduce correctness assertions to components of the system. Abstractions allow model checking of components whose models would otherwise be too big. All these techniques are supported by a dedicated verification tool which has been developed at Oldenburg within the project KORSO (funded by the BMFT). In this way, a substantial part of the verification task had been done. It is expected that a complete verification is possible with an improved version of the tool.

A Specification Development Framework

Jeanine Souquières
CNRS—CRIN—INRIA-Lorraine, Nancy

In our talk, we have presented the basis for an interactive specification development environment in which development methodologies can be supported and the specification is a component. Within the environment, design concepts and strategies are captured by the application of development operators which enable the incremental construction and modification of specifications. The operators enable the specifier to develop specifications in an intuitive fashion by separating the use of design concepts from the technical details of how they are captured in the specification language. They also offer flexibility since it is possible to define a library of operators capturing alternative definitions of particular concepts and strategies.

The model is currently being validated with a prototype tool developed in the Esprit2

project ICARUS. This environment consists of a plan editor connected to a product manager. The main activity of the specifier does not consist in writing down the specification but in applying predefined operators in order to make it evolve. This approach reduces the direct intervention and focuses its attention on the reasonning followed during the elaboration of the specification.

The model has two main features. Firstly, it is language-independent. Therefore, it can be used with existing specification languages and the resulting specifications can be verified and refined using existing tools and methodologies (experimentation have been done with Glider and Z). Secondly, it records the design decisions taken during the specification's development. This information represents a deeper understanding of the specified system than the specification alone. It facilitates reading the specification and may also aid the steps of verification and refinement.

How to Manage Formal Specifications?

Maritta Heisel

Technical University of Berlin

The talk presented joint work with Jeanine Souquières, sponsored by PROCOPE.

The starting point of this contribution is the observation that it is taken for granted that the process of developing formal specifications relies heavily on the specification language that is used. However, the specification process should be guided by the *problem* to be solved, not by the language to be used.

We claim that the process of developing formal specifications can be carried out and supported in a *language independent* way. This is illustrated by an example. Using the

framework designed by Jeanine Souquières (see abstract of her talk), a specification of the user's view of the Unix filing system is developed. Performing exactly the same steps on the workplan level, two different specifications of the system are set up simultaneously: one in the algebraic language PLUSS, one in the model-based language Z.

In doing so, we apply different *specification styles* (reuse, algebraic, and state-based). These styles are represented *explicitely* as sets of development operators. This approach allows one to organize the specification process independently of the language used. Additionally, the concept of specification style can be used to structure the library of development operators: once the specifier has chosen a certain style, the implemented support system will only offer the operators belonging to the chosen style.

In the Unix case study, switching between styles occurred very naturally. This is a further argument for the approach of *not* identifying specification styles with languages.

Logical tools for reliable system specification

Egon Börger

University of Pisa

We present the concept of evolving algebras, defined by Yuri Gurevich, and survey what has been achieved until now by using this concept for specification of large systems (Prolog, C, WAM, Occam, Transputer, PVM, APE100 parallel architecture). We explain in which sense evolving algebras offer a logical tool for reliable system specification. We give arguments for "reliability" to be understood as "supported by a mathematical proof of correctness" - a feature which evolving algebras permit to reach even for large systems where traditional specification methodologies face combinatorial explosion of formalism.

As concrete example to support our claims we present an evolving algebra approach to prove correctness of Lamport's Bakery Algorithms: we construct two evolving algebras, capturing lower and higher view of the bakery algorithms, which enables us to give a simple and precise proof of correctness. This can be done for both the atomic and the durative action interpretation, the difference essentially lies in the different notion of state. (This is joint work with Yuri Gurevich (Ann Arbor) and Dean Rosenzweig (Zagreb), to appear with Oxford University Press in the book "Specification and Validation Methods", 1994, ed.E.Börger.)

Classification Approach to Design

Douglas R. Smith
Kestrel Institute, Palo Alto

A brief introduction to KIDS (Kestrel Interactive Development System) and an overview of recent results on the synthesis of high-performance transportation scheduling algorithms was given. The main part of the talk addressed the problem of how to construct refinements of specifications formally and incrementally. The idea is to use a taxonomy of abstract design concepts, each represented by a design theory. An abstract design concept is applied by constructing a specification morphism from its design theory to a requirement specification. Procedures for computing colimits and for constructing specification morphisms provide computational support for this approach. Although the classification approach applies to the incremental application of any of knowledge formally represented in a hierarchy of theories, the talk focused on a hierarchy of algorithm design theories and derivation of

code that invokes a fast network flow library routine to solve a goods distribution problem.

References:

Smith, D.R. and Parra, E.A., Transformational Approach to Transportation Scheduling, in *Proceedings of the Eighth Knowledge-Based Software Engineering Conference*, (Best Paper Award), IEEE Computer Society Press, September 1993, 60-68.

Smith, D.R., Constructing Specification Morphisms, Journal of Symbolic Computation, Special Issue on Automatic Programming, Vol 16, No 5-6, 1993, 571-606.

Smith, D.R., Classification Approach to Design, Technical Report KES.U.93.4, Kestrel Institute, Palo Alto, CA, November 1993, 24 pages.

HOLCF: A conservative extension of HOL with LCF

Franz Regensburger

Technical University of Munich

On the basis of HOLC the well known logic LCF (Logic of Computable Functions: Scott 69, Paulson 87) was developed as a theory in HOLC using the technique of conservative extension. HOLC is a variant of classical higher-order logic (Simple Theory of Types: Church 40, Andrews 63, Gordon 88) that offers the additional concept of type classes which allows one to restrict the degree of polymorphism (see literature about the programming language Haskell for more details about type classes). For HOLC there is an instance in the generic theorem prover Isabelle (Nipkow 90). A definition for syntax and semantics of HOLC will be presented in my doctoral thesis that is supposed to be completed at the TU München in 1994.

The motivation for this work was the fact that in LCF many properties of the semantics cannot be expressed within the syntax of the logic. Although they are central notions in the semantics of LCF it is not possible e.g. to talk about ω-chains, least upper bounds, continuity of functions or admissibility of predicates inside the logic.

A way out of this problem is to formalize LCF completely in higher-order logic as a theory of complete partial orders called HOLCF. With this method one not only has access to the semantic concepts of LCF but also gets the possibility to combine techniques ¿from higher-order logic (description operators, higher-order predicates, various induction principles) with those of LCF (partial functions, fixed-point operator for continuous functions).

The above extension of HOLC with LCF however requires the proper distinction between types without structure (just interpreted as sets) and types with structure as e.g. partial orders or complete partial orders which are interpreted as sets with an ordering on them. This technical problem is solved by using type classes.

Starting with the theory of natural numbers in the Isabelle HOLC system (just called HOL in Isabelle), the theories of partial orders and pointed complete partial orders were developed. The notion of continuity is introduced which is the basis for the new type of continuous functions. The theory of continuous functions over ω-chain-complete spaces is developed. Strict products, strict sums and lifting are introduced. This constitutes the assembly language for recursive type definitions in the style of LCF. The fixed point operator with its unfolding law and fixed point induction is derived. The semantic definition of admissibility permits the derivation of lemmas which are beyond the expressiveness of LCF.

The complete logical contents of LCF with all its axioms and inference rules was derived in HOLCF (Higher-order Logic of Computable Functions). If desired, it is possible to work in an LCF subset of HOLCF that has the look and feel of the LCF logic. If needed, e.g. in a very involved proof about admissibility, one can also use the full expressiveness of higher-order logic.

Some recursive data types, e.g. streams and natural numbers, have been formalized using Peter Freyds method of axiomatizing least solutions for domain equations by stating a continuous isomorphism on a domain and requiring identity to be the least endomorphism on that domain. Besides the usual theorems for such recursive types also a co-induction principle (Andrew Pitts 93) was derived for every such type.

HOLCF is not only a higher-order version of LCF. It also offers deep insight into the nature of domain theory. Variations of domain theory can be studied in a uniform framework. Semantics of weaker formalisms such as hardware description languages or programming logics can be formalized in HOLCF, combining techniques from HOL and LCF. Since HOLCF was developed in the Isabelle system, a powerful theorem prover for HOLCF is already at hand.

Formal Mathematics as Key Component of Deductive Tools

Christoph Kreitz

Technical University of Darmstadt

Formalized mathematics can be a means to implement verifiably correct tools for program construction. If used to express all the semantic knowledge contained in a derivation method it makes their descriptions independent of the peculiarities of underlying logics and environments and supports completely formal reasoning at a comprehensible level. As a consequence various ideas can be represented in a uniform manner. Approaches which currently appear to be entirely different can be integrated. Thus formal mathematics can be viewed as a vehicle supporting collaboration in the field.

The idea is illustrated by a theorem abaut the constructive equivalence between three main paradigms in program synthesis and an example formalization of a strategy deriving global search algorithms from specifications in terms of a verified meta-level theorem.

Knowledge Modelling for Evolutionary Program Synthesis

Jutta Eusterbrock
GMD Darmstadt

For designing realistic program synthesis systems, a coherent methodology is needed, which allows to apply a single notation and a coherent concept for describing various useful forms of knowledge, eg. domain data or software components, and supporting reasoning tasks, eg. knowledge acquisition, software construction or optimisation. Furtheron - from a practical point of view - it is highly desirable to have program synthesis systems which assure the re-usability of components, flexibility and evolution.

Hence, we consider synthesis systems as collections of interacting theories and propose a multi-layer methodology for the design of knowledge-based program synthesis systems within the logic programming paradigm. The approach depends upon the definition of three hierarchically organized abstraction layers, each of them is subdivided into data type and rule specifications in the format of typed Horn Clauses. The interaction between theories and abstract representations is realized by interface definitions.

For modelling domain data, abstract data types and for modelling domain knowledge, typed horn programs are used. The classical two-valued logic frame can be extended with a third value to express lack of information. Metatheories allow for the specification of generic program synthesis knowledge. Generic tactics are derived as metatheorems from metatheories and allow to state generic program synthesis methods in a domain independent way. Our approach borrows concepts from object oriented programming to

structure collection of theories by frame-like data types and to support inheritance. Meta-programming techniques provide language facilities to describe the dynamic change of theories. Generic methods for program synthesis tasks are derived as metatheorems from generic theory types.

Based on this theoretical foundation we implemented a program synthesis system ASK ("Acquiring Synthesis Knowledge"), which integrates means for knowledge specification, knowledge acquisition, program construction and transformation tasks. The system has been applied for the discovery of mathematical algorithms.

Lemma Generation by Anti-Unification

Birgit Heinz

Technical University of Berlin

Automated inductive theorem proving gets often stuck because of the problem to find the appropriate auxiliary lemmata by which the proof could be completed automatically. Usually, if the theorem prover cannot complete a proof automatically, the user has to suggest possible lemmatas.

Our aim is to provide a practical and useful mechanism to automatically generate suggestions of such lemmata by using anti-unification [1] with respect to a certain background knowledge given by equational theories.

In a first approach, we developed an anti-unification-approach restricted to canonical equational theories, akin to narrowing in the unification case. This approach enumer-

[1] the dual operation to unification, see [4]

ates all generalizations of two terms. Through experimentation, this approach was found impractical to generate suggestions for lemmata.

To obtain a more practical approach we concentrated our interest on a restricted class of equational theories E. These theories are characterized by allowing the description of the equivalence classes of terms by regular sorts [2] or regular tree languages [1, 2]. The approach described in [3] will be used to generate the corresponding sort definition or grammar for an equivalence class.

The basic idea of the sort approach is that the E-generalizations of terms will be computed by the pairwise syntactical anti-unification of the elements of the corresponding equivalence classes. The result is a closed representation of all E-generalizations of terms by a new sort or tree grammar, i.e. the set of E-generalizations corresponds to the language generated by the new sort definition or grammar.

Following this approach, we developed an anti-unification-algorithm of regular sorts or tree grammars which recursively computes the pairwise generalizations described by a new sort definition. We proved the soundness and completeness of this algorithm.

The scenario for the generation of lemmata suggestions can be described as follows: The mechanism to generate lemmata will be needed if a prover reaches a situation where it is not possible to continue but the proof is still not finished, i.e. the prover reaches a term and can not further replace the term. The goal then is to find an applicable lemma, i.e. an equation, with this term as left hand side and a right hand side to be "learned" from a sufficiently small number of different ground instances of the term, obtained by using anti-unification with respect to the equational theory to be considered.

Then, given these ground instances and their corresponding normalforms, the equivalence classes, i.e. sorts, will be computed and anti-unification of regular sorts is applied. The resulting sort definition describes all possible lemmata which at least hold for the selected ground instances. So the sort contains also all valid lemmata.

But this final sort may contain many useless or meaningless lemmata. Using external filter criterias it is possible to restrict undesired suggestions of lemmata. Because of the closed representation of E-generalizations by a sort it is possible to connect such filter operations in a flexible and efficient manner yielding a result that is again a closed representation by a (new) sort.

By enumerating all terms of this final filtered sort, i.e. enumerate all sentences of the tree language, each possible suggestion of lemmata is generated.

The interesting question is whether this approach suggests useful applicable lemmata in practical time. We have a prototypical implementation in Prolog and the first few tests have increased our expectations in a positiv manner because they demonstrated that the selected test examples have practicable runtimes.

[2] Regular sorts correspond to regular tree languages of ground constructor terms and have the same nice closure properties like regular word languages.

References

[1] J. Burghardt: Eine feinkörnige Sortendisziplin und ihre Anwendung in der Programmkonstruktion. Dissertation at University of Karlsruhe, 1992

[2] H. Comon: Equational formulas in order-sorted algebras. in: Proc. ICALP, Warwick, Springer-Verlag, Jul 1990

[3] H. Emmelmann: Code Selection by Regularly Controlled Term Rewriting. Proc. of Int. Workshop on Code Generation, 1991

[4] G. Plotkin: A Note on Inductive Generalization. Machine Intelligence 5, 1970, 101-124

Ideal Streams

Bernhard Möller

University of Ausgsburg

Many formal specifications and derivations suffer from the use of lengthy, obscure and unstructured expressions involving formulas from predicate calculus. This makes writing tedious and error-prone and reading difficult. Concerning machine support, the lack of structure leads to large search spaces for supporting tools. Moreover, proofs or derivations produced automatically tend to be incomrehensible to humans, so that audit is impaired.

Modern algebraic approaches aim at more compact and perspicuous specifications and calculations. They attempt to identify frequently occurring patterns and to express them

by operators satisfying strong algebraic properties. This way the formulas involved become smaller and contain less repetition, which also makes their writing safer and speeds up reading. The intention is to raise the level of discourse in formal specification and derivation as closely as possible to that of informal reasoning, so that both formality and understandability are obtained at the same time. In addition, the search spaces for machine assistance become smaller, since the search can be directed by the combinations of occurring operators.

If one succeeds in finding equational laws, proof chains rather than complicated proof trees result as another advantage. Moreover, frequently aggregations of quantifiers can be packaged into operators; an equational law involving them usually combines a series of inference steps in pure predicate calculus into a single one.

The present paper reports on an attempt to carry over these ideas to the area of stream processing. We treat streams within an algebra of formal languages and show its use in reasoning about streams. In particular, we give a description of the alternating bit protocol and an algebraic correctness proof for it.

The basis is the prefix order on finite words. A set of words is directed w.r.t. this order iff it is totally ordered by it. Therefore ideals, i.e., prefix closed directed sets of words, are a suitable representation of finite and infinite streams.

It is well known that the space of streams under the prefix ordering is isomorphic to the ideal completion of the set of finite streams. Since, however, ideals are just particular trace languages, we can use all operations on formal languages for their manipulation. A large extent of this is covered by conventional regular algebra. Moreover, we can apply the tools we have developed for quite different purposes in a number of earlier papers on algebraic calculation of graph, pointer and sorting algorithms.

Using regular expressions rather than automata or transition systems gives considerable gain in conciseness and clarity, both in specification and calculation. While this has long been known in the field of syntax analysis, most approaches to the specification of concurrency stay with the fairly detailed level of automata, thus leading to cumbersome and imperspicous expressions. Other approaches use logical formulas for describing sets of traces; these, too, can become very involved. By extracting a few important concepts and coming up with closed expressions for them one can express things in a more structured and concise form. This is done here using regular and regular-like expressions with their strong algebraic properties.

Another advantage of our approach is that we can do with simple set-theoretic notions thus avoiding the overhead of domain theory and introduction of explicit $\perp$-elements; also we obtain a simple treatment of non-determinacy. Finally, we do not need additional mechanisms for dealing with fairness; rather, fairness is made explicit within the generating expressions for trace languages.

Panel Discussion:
Does Tool Support Enhance Our Intellectual Capabilities?

Transcript by Martin Simons

Panelists:

Roland Backhouse (RCB), Wolfgang Bibel (WB), Wolfgang Reif (WR), Michel Sintzoff (MS)

Moderated by Bettina Buth

To begin with, each of the panelists gave a brief, provocative position statement. RCB started out by saying that the answer to the question in the title is: No, it does not; one shouldn't even strive to build tools that do. A tool should be flexible to that extent that it helps in constructing both beautiful and ugly derivations, correct and incorrect proofs. It should allow a proof to be read and to be revised, but the user should stay in control at all times and not be hindered in his or her creative work.[3] — WB expressed the view that the main reason for the ongoing software crisis is the lack of a programming methodology based on formal approaches. Tools, he believes, are our last hope, and, in

[3]The citations used by RCB to illustrate his claims can be found at the end of this transcript.

the long term, only synthesis tools will prevail. In principle, the necessary technology is available but it has not yet been integrated into a coherent system. Within the next ten years, however, such tools should become available. — WR said that, in his view, the answer is no but he added that he was not really interested in the question. He expatiated on this: there will be no correct software without formal methods; there will be no formal proofs without tools; and, at the same time, there will be no proofs without humans. Tools must therefore integrate automated reasoning and human expertise. The problem is to build tools that reconcile these inherently contradictory requirements. Tools must enable the user's creativity to unfold, and be as flexible as possible, while while at the same timeallowing automated reasoning support. — MS's answer to the question was a hearty "yes and no". Like Roland, he enjoys beautiful proofs, but he wants them to be correct as well. He wants a tool to help him avoid mistakes. He would like a high-level "proof programming language" following computer science principles in general, and software engineering principles in particular. Such a language should allow proofs to be expressed at a similar level of detail as, e.g., the proofs contained in the recent book by Gries and Schneider. Finally, he stressed that the individual's capabilities determine to what extent he or she can utilize a tool. To illustrate this point, he claimed that five nanoseconds of Douglas Smith's time are approximately equivalent to an infinite number of his own lifetimes.

The discussion proper started with Debbora Weber-Wulff's statement: A fool with a tool remains a fool. The subsequent first round centred on the way the question was worded. Wladyslaw Turski felt that the question should be rephrased in the following terms: "Does tool support allow us to make better use of our intellectual capabilities". A compiler is a very effective tool, but it does not enhance intellectual capabilities. Egon Börger agreed, and added that, with the advent of new tools, the number of fields tractable by humans increases, so intellectual capabilities are not enhanced but new problems may arise and can then be tackled. The next round centred on the usefulness of tools. RCB questioned the usefulness of verification tools such as KIDS or KIV, because with these tools, the user is no longer master but slave. WR countered that in the context of formal program development compromises have to be made if we do not wish to give up correctness. Douglas Smith basically agreed, adding that a tool is a good tool if it is economically viable. At present, he said, this is not the case. We have to study and experiment with various systems and approaches in order to improve their prototypical nature. Stefan Jähnichen raised the next point by claiming that tools may also pose a threat to our mental capabilities in that they may entice people to stop thinking about their initial design ideas prematurely turn to a tool instead. Such a tendency can be observed with programmers, for instance: the better their compilers, the worse their code. Maritta Heisel objected to this by citing the case of Douglas Smith deriving, with the help of the KIDS system, a much better scheduling algorithm than anyone before him. Karl Stroetmann agreed, giving a related example from a SIEMENS development where model checkers were used to test the correctness of certain protocols whose complexity greatly exceeded human capabilities. The final round centred on the notion of proof itself, with Wadyslaw Turski wondering why everybody was so excited about proofs in the first place. Proofs, he said, are really of no relevance whatsoever. Egon

Börger objected that proofs in computer science are by their very nature different from those used in ordinary mathematics. For instance, there is much less stability: because small changes in assumptions may occur on a daily basis in formal program development there is a need to try and reuse or adapt old proofs. WB agreed, pointing out that it was computer science that turned proofs into first-class citizens. Thomas Santen added that, in the context of safety critical systems, proofs of correctness are highly relevant, if not to say crucial.

(Martin Simons assumes full responsibility for misinterpretations or miscitations)

Citations used by Roland Backhouse

Kernighan and Plauger (Software Tools): "[A tool] solves a general problem, not a special case; and it's so easy to use that people will use it instead of building their own."

Jan Smith (about Nuprl): "You really had to know the system very well in order to do something with it."

Debora Weber-Wulff (about the Boyer-Moore-Prover): "5 Matt minutes = 10 Yu hours = 20 Debbie days"

Michel Sintzoff: "We aim at beautiful derivations, ones that seduce people."

Bettina Buth: "If formulas become too large we are not able to comprehend them."

Karl Stroetmann: "The formal proof is usually that huge that it doesn't make sense to read it."

D.E. Knuth: "It would be nice if a system like METAFONT were to simplify the task of type design to the point where beautiful new alphabets could be created in a few hours. This, alas, ist impossible; ... METAFONT won't put today's type designers out of work; on the contrary, it will tend to make them heroes and heroines, as more and more people come to appreciate their skills."

Kernighan and Plauger: "Reading and revising are the key words. No program comes out as a work of art on its first draft, regardless of the techniques you use to write it."

Matthias Weber: "Let the user do the creative work."

Panel Discussion:
Are there any good methods and logics for program design?

Transcript by Thomas Santen

Panelists:
Egon Börger (EB), Wlad Turski (WT), Doug Smith (DS), Martin Wirsing (MW)
Moderated by Richard Jüllig

Richard Jüllig opened the discussion by stating that we probably do not have good methods or logics yet, and that only constructive existence proofs of the contrary are admissible. Then he invited each of the panelists to give a short, provocative statement.

EB started by citing mathematical abstraction as a witness for good methods. It is useful to define and understand systems of processes, because it focuses on the content of the concept of reality underlying any particular problem. Logic in the contrary focuses on the form of reality and is therefore not a suitable means to carry out program development. The relation between program design and logic is that of a task to solve a particular class of problems to a universal method that can by definition only produce abstract forms of general principles since it considers all possible worlds and not just the one of the problem

domain. Finally, EB pointed out the danger of formal methods, i.e. that they may be straight-jackets to creativity: "Even though naive set theory is inconsistent, for a working mathematician it is still the best to work in Cantors paradise."

A short discussion clarified that program design for EB means to solve problems of reality and that mathematical abstraction is the best witness for a good method because it is the most precise means we have.

WT compared ancient Greek mathematics to Roman engineering: The Romans achieved very impressing engineering tasks though totally unaware of any mathematical theory which the Greeks had already developed. The situation today is similar: Engineers work without mathematics most of the time. Still they have to learn to use it in some situations. If we view programming as a high-quality engineering task, then three criteria apply: reliability of the product, efficiency of production, and thorough market analysis. The classical engineering technique of testing cannot be applied in programming for complexity reasons. Therefore logics or some means of conducting proofs is certainly needed to ensure reliability. But a good programming method will not help much to enhance efficiency of production since only one third of software production costs are due to programming. Good methods are needed for market analysis, i.e. to find out how to produce the right software. Here, much more communication with the user is needed.

WT disagreed with EB's view of logic: The concentration on formalisms is only a recent trend in logic. For WT logic is about convincing people and winning arguments. He advocated that this kind of logic be teached to users and predicts that the so-called software crisis disappears as soon as they have learned to express their needs. As for the general role of logics: "For each Hilbert there is a Gödel just around the corner."

MW made the point that the methods available today are suitable for programming in the small and not so very suitable for composition of existing components. The big practical problem of modifying software is still not solved at all and can only be tackled if we are able to put existing methods together.

At this point, Stefan Jähnichen brought up the question if there indeed is a "software crisis". He did not see that software today works that bad, e.g. considered the 300.000 LOC that implement the software used in handy-phones. MW disagreed. He sees a crisis because the demand for software is greater that the actual output. Only a very small amount of software is reused, because it lacks documentation and strange effects occur if it is used in unforeseen contexts. This makes everybody invent their own solutions from scratch. Simon Thompson added that this is even true for the software verification community: everybody writes their own verification system even though the techniques used are often very similar. MW added the example of faulty text editors. Text editing is surely one of the most frequent tasks performed on a computer but these programs are still very error prone. He concluded that there is no "artistic" way of writing 300.000 LOC of reliable software. Methods are needed to systematically modify software in a safe manner.

DS stated that while there is a diversity of good ideas, consensus is lacking and a steady accretion of coherent sets of ideas is necessary. He pointed out that while having quite a good understanding of how to synthesize algorithms, there is nearly no method to synthesize data structures and much less to synthesize systems. He compared predicate

logic to Turing machines in the sense that both are simple enough to study basic concepts of mathematics or computing. But as we do not use Turing machines for programming we should have a much more abstract and powerful logical language to reason about programs. EB added that markets and customers provide the criteria of what is a good method or a good logic for software design.

In the following discussion, Peter Pepper came back to the (non-)existence of a software crisis. In his opinion there is none: what were the "crisis problems" ten years ago are standard tasks today. Our ambitions are way ahead of our capabilities, and this is the real problem to him. WT disagreed: not our ambitions are ahead of our capabilities but demand and social responsibility make up the need for higher quality software.

Karl Stroetmann illustrated the existence of a crisis by citing SIEMENS projects that have thrown away their product because they were just not able to make them work. For WT this is not an example of bad programming practice but of errors in management. For him, improper manager decisions, as the percentage of project funding spent on quality control, are the primary source for bad software quality. Debora Weber-Wulff added that most programs evolve from running programs and that most software construction methods do not apply to this situation. WT agreed and pointed out the difference between software construction and classical engineering: "Have you ever heard of a portable tunnel? – But you have surely heard of portable software!"

In his view this also applies to the managerial problems. To date there are no metrics on software products or problems and thus no means to extrapolate from the known to the expected. This is one reason why there are so many management errors in software projects.

Stefan Jähnichen remarked that to talk of a software crisis is dangerous because customers might conclude that they can do better than computer scientists. Still, Peter Pepper saw the need of standard methods as they are used in classical engineering. With few exceptions like database theory he does not see such methods in computer science to date. He sees our role in building such theories and finding languages to communicate them. They surely need a sound foundation such as mathematics and logic, but abstraction is merely a basis for a method and not a method in itself.

Michel Sintzoff considered the comparison to building bridges or tunnels outdated. He would like to speak of system design rather than program design. Thomas Beth suggested to compare software engineering to genetics rather than classical engineering and Debora Weber-Wulff saw good methods for algorithm design but not for program or system design.

Finally, Richard Jüllig reminded of the history of engineering from building the pyramids to the Golden Gate Bridge. He concluded that there "is surely more than one way of slicing a pie" and that methods for architectural and structural design of software systems are lacking to date.

(Thomas Santen assumes full responsibility for misinterpretations or miscitations.)