Beyond ProCoS at Kiel: A Synopsis of Recent Research

Martin Fränzle[1], Wolfgang Goerigk[2],
Burghard von Karger[1], and Markus Müller-Olm[3]

[1] Carl v. Ossietzky Universität, Department of Computer Science,
P.O. Box 2503, D-26111 Oldenburg, Germany
{Martin.Fraenzle|von.Karger}@Informatik.Uni-Oldenburg.DE

[2] Christian-Albrechts-Universität Kiel, Department of Computer Science,
Preußerstr. 1–9, D-24105 Kiel, Germany
wg@informatik.uni-kiel.de

[3] University of Dortmund, Department of Computer Science, FB 4, LS 5,
Baroper Str. 301, D-44221 Dortmund, Germany
Markus.Mueller-Olm@cs.uni-dortmund.de

1 Introduction

After completion of the ProCoS projects, the ProCoS group at Kiel, headed by Prof. Hans Langmaack, continued research on the broad scope of topics that the ProCoS projects had sparked. These topics, involving algebraic models of reactive systems, real-time model-checking and controller synthesis, and compiler design and verification, may seem diverse, yet they are closely linked within the ProCoS approach: Setting up a consistent set of formalisms and methods for the variety of abstraction levels that arise during embedded system design requires a firm grasp of all of them. However, the scientific subjects involved are nevertheless diverse, and no single research group would be able to substantially further all of them without being linked to other, particularly more specialised, groups. Being broader in scope than the scientific contacts that a single site can set up, the ProCoS working group proved to be an excellent basis for the required kind of scientific exchange. Its meetings provided an indispensable forum for presenting and discussing the work in various stages, and finally became an important means for teaching the newly developed techniques to other personnel.

It is just now that it becomes apparent how successful these information dissemination activities were: The compiler verification activities of the ProCoS project led to a dedicated German compiler verification project named "Verifix" [GDG$^+$96,Lan97c,Lan97a] that builds upon the ProCoS techniques [MO96b]. Furthermore, ProCoS researchers from Kiel have been taken over by other universities: Markus Müller-Olm, whose extensive case study of applying the ProCoS compiling verification techniques onto the translation of a prototypic hard-real time programming language to an actual processor (the Inmos Transputer) is documented in the PhD thesis [MO96a] that appeared recently as a monograph in the LNCS series of Springer-Verlag [MO97], moved to the University of Dortmund. Martin Fränzle, who has been concerned with real-time model-checking

ProCoS WG workshop at FM'99, available from FM'99 CD-Rom, pp. 1–17, 1999.

and hardware synthesis from temporal logic and Burghard von Karger, who has investigated algebraic models of reactive systems, are now at the University of Oldenburg. Bettina Buth is currently at the University of Bremen.

In the following, we will summarise the contributions made by former ProCoS researchers from Kiel to the aforementioned topics. We start with compiler verification in Sect. 2, then turn to model-checking and controller synthesis in Sect. 3, and finally review the work on algebraic models of reactive systems in Sect. 4.

2 Compiler Verification and Compiler Implementation Verification

At the bottom end of the ProCoS tower of abstraction levels, at the implementation level, there have been major efforts on compiler verification and on hardware compilation [MO90,BBF$^+$92,BFOR93,HHF$^+$94,HHMO$^+$96,HPB93]. The ProCoS work on compiler verification concentrated, on the one hand, on languages with hard real-time constraints (cf. Sect. 2.5) and, on the other hand, on developing a methodology for modularising code generator correctness proofs that permits to master the complexity involved in verifying code generators for commercially available processors (cf. Sect. 2.2). For a conscientious and mathematically rigorous proof of compiler correctness, however, also an implementation correctness proof for the compilers themselves is needed, in addition to the (semantical) verification of their specification.

In 1995, as a followup of the ProCoS project, three research groups at the universities of Karlsruhe (Prof. Dr. G. Goos), Ulm (Prof. Dr. F. W. von Henke) and Kiel (Prof. Dr. H. Langmaack) in Germany accepted the challenge and started on a six years joint project on *Correct Compilers – Verifix*, funded by the Deutsche Forschungsgemeinschaft (DFG), focusing on the full verification of compilers and compiler generators for sequential imperative and object-oriented languages on real machines. In 1997, a supplementary DFG-funded project on "techniques for compiler implementation verification (*VerComp*)" started at Kiel for two years, focusing on practically usable proof techniques for the binary compiler implementation correctness.

2.1 The Verifix and VerComp Projects

The major goals of the *Verifix* and *VerComp* projects [Lan97b,GDG$^+$96,Lan97d] are to develop methods and techniques for correct realistic compiler construction for practically relevant source languages and concrete target machines, and to completely verify compilers down to their binary machine code implementation. *Verifix* concentrates on

- the construction of correct compilers for realistic imperative and object-oriented languages,
- on real target and host processors with their finite resource limitations,

– correctly implemented down to their binary executable code,
– generating efficient code, comparable to that of unverified compilers,
– using mechanical proof support and classical compiler construction methods as far as possible.

In that, we assume that application programs are correct w.r.t. their specification, and that hardware works as defined in the instruction manuals. The work closes the gap between high level program and compiler verification on the one hand and the correctness of machine executables on the other hand. It turns out, that a rigorous correctness requirement nevertheless allows for the use of standard and approved compiler construction techniques and classical compiler architectures [GGH+97,GZG+98]. Program checking and checker-based program verification [GGZ98] enable us to even use unverified tools like e.g. code generator generators [GZG99] or parser generators [HGG+99] for compiler implementation, without weakening the rigorous correctness property established for the final compiler machine program. Implementation correctness (refinement) has to capture the intuitive requirement for transformational programs: If a machine program successfully returns a result, then that result is correct (preservation of partial correctness, cf. Sect. 2.3).

The crucial work in compiler verification has been for over 30 years and will remain the semantical correctness of the transformation. The Verifix project seriously addresses the rigorous mathematical question what we additionally have to and have to be able to prove for the complete correctness proof of compiler executables, in order to improve on the present situation which is best characterised by the moral of Ken Thompson's Turing Award lecture in 1984 [Tho84]: "You can't trust code that you did not totally create yourself. (Especially code from companies that employ people like me.) No amount of source-level verification or scrutiny will protect you from using untrusted code." We thus head for an implementation correctness proof for an initial fully verified compiler executable (cf. Sect. 2.4), which is needed as a sound basis for further compiler bootstrapping and system program development [Goe99] (cf. Sect. 2.4). Although in principle cross-platform bootstrapping is possible, we nevertheless aim, for many reasons, at an independently repeatable technique for such proofs from the scratch.

2.2 A Methodology for Verified Design of Code-Generators

One major difficulty when verifying code generators for actual processors (in contrast to idealised toy processors) is the large number of more or less unrelated questions that has to be addressed in a precise manner: addressing modes, side effects of instructions, allocation of memory and registers, representation of data, no separation of program and data memory, to mention just a few. The construction of a verified code generator does not start from scratch but from a rough intuitive understanding gained from prior compiler building experience and tradition. A useful methodology for the correct construction of code generators must allow to capture these informal ideas precisely in a stepwise and incremental fashion.

We developed a methodology that involves the stepwise derivation of increasingly *abstract views* to the target processor's behaviour from a *base model* of its execution cycle. The abstraction steps allow to treat particular aspects of translation or machine program execution in isolation and are performed by algebraic calculations employing a program-like notation [Hoa91]. Then *compiling-correctness relations* are defined that specify the intended semantic relationship between source and target code, which is largely simplified by the availability of the abstract views. Afterwards, concrete code patterns are studied by means of theorems about the compiling-correctness relations. From these *translation theorems* a code generator can be implemented without further semantic considerations.

These ideas are elaborated in a dissertation thesis, which appeared as a monograph in Springer's LNCS series [MO97]. As a case study, the verified design of a code generator that translates a simple prototypic real-time programming language to the Inmos Transputer is presented. The structuring ideas are also discussed briefly in [HHMO+96].

2.3 Towards an Adequate Notion of Correct Compilation

At first thought, the question what is to be expected from the code generated by a correct compiler is simple to answer: it should behave precisely as the source code from which it was generated. At further scrutiny, however, the question turns out to be amazingly subtle and one is faced with a large number of options. Should one expect that the implementation of a diverging program diverges or that the implementation of a regularly terminating program terminates regularly? (Often one shouldn't, as resource constraints of the executing machinery might in both cases lead to irregular termination.) Should one expect that irregular outcomes of programs are preserved? (Often one shouldn't: common optimising program transformations can blur the distinction between different irregular outcomes.) Further difficulties arise from the fact that source and target program work on different state spaces and that different styles can be used to capture semantics of source and target language (operational, denotational, etc.).

Different application areas call for different correctness properties. In a controller of an embedded system, e.g., error outcomes like 'stack-overflow' or 'out-of-memory' are totally unacceptable. For such applications a compiler essentially has to preserve total correctness [MO97,MOW99]. This is different for strictly transformational programs. The crucial requirement for this program class is that a regular results of an implementation is always a possible results of the source program because this allows to rely on results without further scrutiny. This property essentially amounts to preservation of partial correctness (PPC) [MO95a,MO96b,GMO96,MOW99] and is vital in bootstraps of verified compilers, as compilers are transformational programs.

In [MO96b] we characterise PPC (in the situation where the state sets of source and target program may differ) in various semantic styles and prove that these characterisations are equivalent. Immediate motivation is that in the Verifix project different semantic styles are employed. While the Karlsruhe group

– based on Gurevich's Abstract State Machines [Gur91] – uses an operational simulation technique [ZG97,HL98], the Kiel group advocates more abstract kinds of semantics as well as reasoning within a refinement algebra, building on the experiences gained during the ProCoS project [MO97,HJS93].

The classical concepts of partial and total correctness identify all types of runtime errors and divergence. In [MOW99] we argue that the associated notions of translation correctness cannot cope adequately with practical questions like optimisations and finiteness of machines. As a step towards a solution we propose more fine-grained correctness notions, which are parameterised in sets of acceptable failure outcomes, and study a corresponding family of predicate transformers that generalise Dijkstra's wp- and wlp-transformers. It is planned to apply the resulting setup for proving the correctness of the translation of nested parameterless procedures to machines with bounded stacks. Steps in this directions are documented in [Wol99a,Wol99b]. The problem of acceptable failures has also been addressed in the setting of relational semantics [GMO96].

2.4 An Initial Fully Verified Compiler Implementation

The Verifix work at Kiel focuses on the construction and verification of an initial fully verified binary compiler executable from a subset ComLisp [GH96] of Common Lisp to the Transputer machine code. In order to achieve a conscientious and mathematically rigorous correctness proof, we modularise the compiler correctness problem into the following four steps:

1. define an appropriate notion of *correct compilation* for sequential imperative languages, which guarantees sufficient correctness properties for the target program on concrete target processors with finite resource limitations (preservation of partial correctness or L-simulation, cf. Sect. 2.3)
2. define a compiling specification $\mathcal{C}_{\mathbf{SL},\mathbf{TL}}$ relating source to target programs, and prove semantically, that $\mathcal{C}_{\mathbf{SL},\mathbf{TL}}$ preserves partial correctness (compiling verification, [GH98c]),
3. construct a compiler program $\pi_{\mathcal{C}}$ in the source language and prove, that $\pi_{\mathcal{C}}$ is a *refinement* (correct implementation) of $\mathcal{C}_{\mathbf{SL},\mathbf{TL}}$ in the sense of preserving partial correctness (correct compiler construction, [GH98b]), and finally,
4. use an existing (unverified) implementation of the source language to execute $\pi_{\mathcal{C}}$, apply $\pi_{\mathcal{C}}$ to itself and bootstrap a compiler executable $m_{\mathcal{C}}$. Check syntactically, that $m_{\mathcal{C}}$ actually has been generated according to $\mathcal{C}_{\mathbf{SL},\mathbf{TL}}$ (compiler implementation verification, [GH98b,Hof98]).

As an integral part of the *Verifix* project, *VerComp* concentrates on compiler implementation verification (steps 3 and 4), in particular on the final step, proposing a practically usable proof technique of *a posteriori code inspection based on syntactical code comparison* [GH98b,Hof98].

Specification and correct compiler construction In [GH98c] we define an explicit specification of a four-phase compilation transforming ComLisp–programs to binary machine code executables on Transputer T400 processors. The compilation is modularised to four steps using three intermediate languages, a stack language, a C-like abstract machine oriented language, and an assembly language. Compiling specifications between each pair of source and target languages are given as inductively defined relations. They can easily be refined to a system of first order mutually recursive ComLisp–functions, as documented in [GH98a], thus defining a high level compiler implementation as a ComLisp-program which compiles ComLisp to Transputer machine code, enabling compiler bootstrapping and thus the proposed implementation correctness proof (see below).

Compiling verification uses denotational techniques like in [MO90] for the front end. For the back end (from abstract machine code to binary code) we use predicate transformers and algebraic-denotational techniques [MO97,MOW99], extended and enhanced to work for preservation of partial correctness (cf. Sects. 2.2 and 2.3). The compiler has been bootstrapped successfully on a Transputer T400 single board computer with 1 MB of memory. The complete code checking documents (see below) have been generated. The compiler runs sufficiently fast. There are (unverified) code-generators for i386, DEC α, and MC 68000 processors and for C and Forth target code available as well.

Binary compiler implementation verification Adopting the scenario from [Tho84] of a well-known attack to Unix operating system programs due to intruded Trojan Horses in compiler executables, we show in [Goe99] in detail how to construct a provably correct compiler source program $\pi_{\mathcal{C}}$ and an incorrect machine implementation $\overline{m_{\mathcal{C}}}$ of it which reproduces itself when applied to the correct source code $\pi_{\mathcal{C}}$. Moreover, it generates incorrect code in one more (catastrophic) case. Such a compiler will pass nearly every test, it will pass state of the art compiler validation, the compiler bootstrap test, any amount of source code inspection and verification, but for all that, it nevertheless might eventually cause a catastrophe. Obviously, transformation verification and source level verification of the compiler implementation are not sufficient in order to avoid such hidden Trojan Horses.

There must be a (syntactical) mismatch between the incorrect implementation $\overline{m_{\mathcal{C}}}$ and what we would expect as the correct result $m_{\mathcal{C}}$. Assuming the correctness (preservation of partial correctness) of $\mathcal{C}_{\mathbf{SL},\mathbf{TL}}$ and $\pi_{\mathcal{C}}$, we can prove by transitivity (or compositionality) of the refinement relation, that if the unsafe bootstrapping succeeded in generating $m_{\mathcal{C}}$, and $(\pi_{\mathcal{C}}, m_{\mathcal{C}}) \in \mathcal{C}_{\mathbf{SL},\mathbf{TL}}$, then $m_{\mathcal{C}}$ is a correct implementation of $\mathcal{C}_{\mathbf{SL},\mathbf{TL}}$ as well [Hof98]. Thus, for the binary implementation we can reduce the semantical question of correct compilation to a syntactical *a posteriori code inspection based on code comparison* between $\pi_{\mathcal{C}}$ and $m_{\mathcal{C}}$, testing (checking the result of) compiler bootstrapping in a stronger sense. $m_{\mathcal{C}}$ might be generated by any initial unsafe implementation of $\pi_{\mathcal{C}}$.

It turns out that there is a technique for such proofs, which exploits modularisation into adequate intermediate layers, checking sufficient syntactical criteria which together imply $(\pi_{\mathcal{C}}, m_{\mathcal{C}}) \in \mathcal{C}_{\mathbf{SL},\mathbf{TL}}$. The two programs are annotated with compile time information, printed side by side and checked module by module for correct annotation and correct transformation. A *diagonal argument* allows for trusted machine support to generate large and in particular low level parts without need for checking at all [Hof98]. This can be seen as an application of the work of Goodenough and Gerhart [GG75] on software testing [Lan97a]. We also use result-checking techniques [WB97], for verification [GGZ98], but also for further reduction of the code inspection work load [Hof98,GH98b]. It turns out that the complete proof documentation compares to what is usual in certification processes. So we are able to prove the correctness of compiler machine executables rigorously, and to give a complete proof documentation.

2.5 Timing in High-Level Programs

Traditionally, sufficiently high performance of hard real-time code is ensured by counting machine cycles in assembler code constructed manually or generated by a compiler. The objective of our work on hard real-time constraints is to identify adequate constructs that allow to describe the timing requirements in the source program. The proposed constructs must, on the one hand, be convenient for the program design, which calls for constructs that permit specification of rather global requirements and for an idealised view of time consumption of basic statements. On the other hand, the timing specifications must be soundly decidable by a compiler.

We propose a solution [FMO94,MO97], in which the program designer can specify upper-bounds for the execution delays of basic blocks and can apply the idealisation that only input/output statements cause execution delays but internal activity is immediate, i.e. takes no time to execute. This idealisation leads to smooth algebraic laws (program transformation rules) and largely simplifies the task of program construction. By exploiting that only communications are externally observable, the idealisation can be justified by obliging the compiler to shift the delays caused by internal activity to subsequent communications and to settle it with the communication's latency. These ideas have been applied in an experimental compiler implemented in Standard ML that translates a prototypical hard-real time language into Transputer code [MO97,MO95b]; for designing the code generator we applied the methodology described in Sect. 2.2.

3 Model-checking and Controller Synthesis

Among the good reasons industry has for being reluctant against introduction of formal methods into their development processes, the need for extraordinarily skilled workers and the cost of those particular activities that call for them is a predominant one. One of the most demanding tasks in this respect certainly is formal verification. Therefore, it has become a popular (though not-at-all universally agreed) belief in the formal methods community that availability of “key-press” verification techniques, freeing the designer from the burden of verification, or even automatic synthesis from specifications as means of “getting it right first time” could be a crucial factor for industrial takeover of a certain method. Consequently, ProCoS researchers have extensively addressed this issue.

3.1 Finite-state Verification and Characteristic Formulas

In the last two decades model-checking [CES96,QS82,MOSS99] has emerged as a promising and powerful approach to automatic verification of finite-state systems. Roughly speaking, a *model checker* is a procedure that decides whether a given finite structure M is a model of a logical formula ϕ. The modal mu-calculus [Koz83], a small, yet expressive branching-time temporal logic, has found particular interest as it provides a clean core logics for the construction of model-checkers.

In [MO98] we show how modal mu-calculus formulas characterising finite-state processes up to bisimulation can be derived directly from the greatest fixpoint characterisations of the bisimulation relations. Our derivation simplifies earlier proofs for the strong bisimulation case [GS86,SI94] and, by virtue of derivation, immediately generalises to various other bisimulation-like relations, in particular weak bisimulation and many behavioural preorders. Characteristic formulas allow to apply model-checkers for automatically checking equivalence or refinement between finite-state processes w.r.t. the considered process relations. In [MOSC99] we show how a certain bisimulation-like refinement relation could be of use in ensuring downward compatibility of components of reactive process libraries.

Imperative programming languages typically offer a sequential composition operator which allows the straightforward specification of phased behaviour. The *chop*-operator in interval temporal logics like Moszkowski's ITL [Mos85] or the Duration Calculus DC [ZHR91] serves a similar purpose. In [MO99] we introduce a logic FLC (Fixpoint Logic with Chop) that extends the modal mu-calculus by a chop-operator and investigate its expressiveness and decidability properties. As far as we know FLC is the first such branching-time logic. To enable an elegant semantic treatment, formulas are interpreted by predicate transformers instead of predicates. FLC is strictly more expressive than the modal mu-calculus but remains decidable for finite-state processes and can thus be model-checked effectively. We also show that characteristic FLC-formulas (w.r.t. simulation or bisimulation) can be constructed for context-free processes, a certain type of finitely generated infinite-state processes. Thus an FLC-based model-checker could be used to automatically check simulation or bisimulation between a finite-state and a context-free process.

3.2 Deciding and Model-Checking Dense-Time Duration Calculus

Verification in general and particularly automatic verification becomes much more intricate when the systems are no longer finite state. The latter applies to e.g. embedded real-time systems interacting with their environment in dense real-time. Therefore, the project “Models of Real-Time Systems II”, funded by the German National Research Foundation DFG under contract no. La 426/13-2, was dedicated towards development of semantic models for embedded real-time systems that soundly describe realistic hardware without overburdening verification efforts with extraneous detail. While we share this objective with most other projects devoted to formal methods for embedded system design, our line of research distinguishes itself by a re-investigation of the consequences of some simplifications that have been adopted in most other models. The key observation guiding our research has been that some of the formal models can encode more information in a timed behaviour than can actually pass the sensor/actuator-barrier between embedded systems and their controlled environment, and that furthermore quite some undecidability results do crucially rely on these extra coding capabilities. Obviously, such undecidability results do not provide an indication of the inherent complexity of the design task; rather, they are artifacts of a particular formalisation of embedded system dynamics.

To avoid this problem, we have analysed more restricted models of the temporal behaviour of embedded systems which do not feature these extra coding capabilities. The particular restrictions adopted took the form of constraints on the maximal density of state changes over time and have been deduced from obvious physical properties of embedded controllers. In order to provide an indication that these restrictions can indeed simplify the design of embedded controllers, decidability of the Duration Calculi — a major group of formal logics for design and analysis of embedded real-time control systems [Zho93] — has been investigated. While the Duration Calculi are highly undecidable with respect to the standard model [ZHS93], a bunch of positive decidability results has been obtained on the more restricted model classes [Frä96a,Frä97].

Through a bottom-up approach, deriving abstract models of embedded controller dynamics from circuit-level behavioural models of asynchronous VLSI as well as from transfer-level models of synchronous circuits, it has been proven that the restricted model classes are fully sufficient for modelling circuit behaviour [Frä95,Frä96c,Frä97]. Based on this correspondence between certain circuit classes and the restricted model classes, new automatic model-checking techniques for asynchronous circuits [Frä97,Frä98,Frä99b] as well as controller synthesis techniques for synchronous circuits [Frä96b,Frä96c,Frä97] have been derived from the decision procedures. To the best of our knowledge, these are the first effective procedures available for a dense-time Duration Calculus with metric time, the chop modality, and unrestricted negation. Table 1 provides an overview over the decidability results for model-checking timed automata against various fragments of dense-time Duration Calculus.

A particular implication of these findings is that even extremely abstract real-time formalisms can be integrated into the design process of embedded controllers through key-press techniques. This claim is substantiated in [Frä97, Chapter 9], were an embedded controller for the ProCoS gas-burner [RRH93], the major case study of the ProCoS projects, is automatically synthesised from its requirements specification, thereby obtaining a device of comparable engineering quality as the manually developed controller of [RRH93]. However, complexity of such an automatic controller synthesis procedure may easily become prohibitive for practical, larger-scale applications such that compositional variants are urgently needed, which is the theme of [FL98]. Beyond these contributions to formal methods for embedded system design, a bunch of theoretical results, including an automata-theoretic characterisation of the expressiveness of Duration Calculus [Frä97, App. A], have been obtained.

Table 1. Decidability of the model property for timed automata with behavioural restrictions wrt. specifications from different fragments of Duration Calculus. The fragments are named after the shapes of allowed atomic formulae.

		Behaviour class of timed automaton		
		finitely variable	n-bounded	time-wise discrete
Subset of DC	$\{\lceil P \rceil\}$	Decidable *[ZHS93, Theo. 10]*	Decidable *[Frä99b, Theo. 5.1]*	Decidable *[Frä99b, Theo. 6.2]*
	$\{\lceil P \rceil, l < k, l = k, l > k\}$ with exactly one, outermost, negation*	Decidable *[Frä99b, Theo. 4.2]*	Decidable *[Frä99b, Theo. 5.1]*	Decidable *[Frä99b, Theo. 6.2]*
	$\{\lceil P \rceil, l = k\}$	Undecidable *[ZHS93, Theo. 11]*	Decidable *[Frä99b, Theo. 5.1]*	Decidable *[Frä99b, Theo. 6.2]*
	$\{\int P = k\}$	Undecidable *[ZHS93, Theo. 11]*	Undecidable *[Frä99b, Theo. 5.2]*	Decidable *[Frä99b, Theo. 6.2]*
	$\{\int P = \int Q\}$	Undecidable *[ZHS93, Theo. 12]*	Undecidable **	Undecidable **
			asynchronous circuits	synchronous circuits
		Corresponding device classes		

* I.e., there are no inner negations in formulae, yet one at the outermost level. The connectives available are conjunction, disjunction, and chop. Note that this subset covers the so-called DC-implementables of [Rav95].

** The proof of theorem 12 of [ZHS93] does not depend on finite variability.

3.3 Key-Press Verification of Hybrid Systems

By the techniques discussed in the previous section, we have been able to show that by suitable restriction of the model class used in behavioural descriptions of system dynamics, model-checking for large subsets of Duration Calculus becomes feasible. The particular restrictions adopted are motivated by physical properties of practical verification problems, namely band-limitedness of reactive systems and synchronicity of clocked systems. These promising results have motivated us to investigate the impact of physically realistic modelling on even more expressive formalisms than Duration Calculus. In [Frä99a], we attack the problem of automatically verifying hybrid automata.

Hybrid automata have been introduced in both control engineering and computer science as a formal model for the dynamics of hybrid discrete-continuous systems. In the case of so-called linear hybrid automata this formalisation supports semi-decision procedures for state reachability, yet no decision procedures due to inherent undecidability [HKPV95]. Thus, unlike finite or timed automata, already linear hybrid automata are out-of-scope of fully automatic verification.

However, it is illustrating to take a closer look at the proof technique used in [HKPV95] for showing undecidability of state reachability in linear hybrid automata. The core machinery is an instantiation of the following proof pattern:

> *Effectively encode two-counter machines by hybrid automata, representing the counter values by two continuous variables of bounded range. E.g., by variables of range* $[0, 1]$ *through the embedding* ε *defined as* $\varepsilon(k) = 2^{-k}$.

Although the results thus obtained are formally correct and absolutely well-done, their relevance to the practical design problems hybrid automata are intended to cover is questionable. The encodings used (e.g. ε) encode infinite information, namely the set of natural numbers, within a compact interval of continuous states, whereas the ubiquity of noise limits the information content encodable by any bounded continuous variable encountered in real hybrid systems to a finite value. Hence, on simple information-theoretic grounds, the undecidability results thus obtained can be said to be artefacts of an overly idealized formalization.

Based on this insight, in [Frä99a] we devise a new semi-decision method for safety of linear and polynomial hybrid systems which may only fail on pathological, practically uninteresting cases. These remaining cases are such that their safety depends on the *complete* absence of noise, a situation unlikely to occur in real hybrid systems. Furthermore, we show that if low probability effects of noise are ignored akin to the way they are suppressed in digital modelling then safety becomes fully decidable.

4 Algebraic models of reactive systems

Work in the ProCoS project included formal proofs at many different levels of program abstraction: For reasoning at these various levels a number of models of computations were used, including relations, time diagrams, and CSP style traces. Inspired by the desire for a theory that would provide a common framework of laws for this spectrum of models Tony Hoare and Burghard von Karger invented the Sequential Calculus [HvK95]. The Sequential Calculus is a slight generalisation of Tarski's calculus of relations. Most algebraic laws that hold for binary relations have analogues in sequential calculus, but relations are only one model among many to which the sequential calculus applies. Later work concentrated on combining the sequential calculus with the theory of Galois connections to produce a new, calculational approach to temporal logics. The definitive reference at this point of time is [vK97] which is available from the author's homepage. Various excerpts from this thesis have been presented at conferences [vK95,BvK95] or published in journals [BvK96,vKB98,vK98].

References

[BBF+92] B. Buth, K.-H. Buth, M. Fränzle, B. v. Karger, Y. Lakhneche, H. Langmaack, and M. Müller-Olm. Provably correct compiler development and implementation. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, volume 641 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[BFOR93] Jonathan Bowen, Martin Fränzle, Ernst-Rüdiger Olderog, and Anders P. Ravn. Developing correct systems. In *Fifth Euromicro Workshop on Real-Time Systems*, pages 176–187. IEEE Computer Society Press, June 1993.

[BvK95] Rudolf Berghammer and Burghard von Karger. Formal derivation of CSP programs from temporal specifications. In Bernhard Möller, editor, *Mathematics of Program Construction*, LNCS 947, pages 180–196. Springer-Verlag, 1995.

[BvK96] Rudolf Berghammer and Burghard von Karger. Towards a design calculus for CSP. *Science of Computer Programming*, 26:99–115, 1996.

[CES96] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1996.

[FL98] Martin Fränzle and Karsten Lüth. Compiling graphical real-time specifications into silicon. In A. P. Ravn and H. Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, volume 1486 of *Lecture Notes in Computer Science*, pages 272–281. Springer-Verlag, 1998.

[FMO94] Martin Fränzle and Markus Müller-Olm. Towards provably correct code generation for a hard real-time programming language. In Peter A. Fritzson, editor, *Compiler Construction '94, 5th International Conference Edinburgh U.K.*, volume 786 of *LNCS*, pages 294–308. Springer, April 1994.

[Frä95] Martin Fränzle. A discrete model of VLSI dynamics in hybrid control applications. ProCoS Technical Report Kiel MF 17/3, Christian-Albrechts-Universität Kiel, Germany, April 1995.

[Frä96a] Martin Fränzle. Decidability of duration calculi on restricted model classes. ProCoS Technical Report Kiel MF 21/1, Christian-Albrechts-Universität Kiel, Germany, July 1996.

[Frä96b] Martin Fränzle. Hardware synthesis from temporal logic: Undecidability need not matter. Position paper, Hardware Synthesis and Verification Workshop, Cornell University, Ithaca, USA, August 1996.

[Frä96c] Martin Fränzle. Synthesizing controllers from duration calculus. In Bengt Jonsson and Joachim Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT '96)*, volume 1135 of *Lecture Notes in Computer Science*, pages 168–187. Springer-Verlag, 1996.

[Frä97] Martin Fränzle. *Controller Design from Temporal Logic: Undecidability need not matter.* Dissertation, Technische Fakultät der Universität Kiel, Germany, 1997. Available as Bericht Nr. 9710, Institut für Informatik und Praktische Mathematik, Christian-Albrechts-Universität Kiel, June 1997, and via WWW under URL `http://ca.informatik.uni-oldenburg.de/~fraenzle/diss.ps.gz`.

[Frä98] Martin Fränzle. Model-checking dense-time duration calculus. In M. R. Hansen, editor, *Proceedings of the Duration Calculus Track of the 33rd European Summer School on Logic, Language and Information.* Universitä Saarbrücken, 1998.

[Frä99a] Martin Fränzle. Analysis of hybrid systems: An ounce of realism can save an infinity of states. In *CSL'99*, Lecture Notes in Computer Science. Springer-Verlag, to appear September 1999.

[Frä99b] Martin Fränzle. Model-checking dense-time duration calculus. *To appear in a special issue on Duration Calculus of Formal Aspects of Computing*, 1999.

[GDG+96] Wolfgang Goerigk, Axel Dold, Thilo Gaul, Gerhard Goos, Andreas Heberle, Friedrich W. von Henke, Ulrich Hoffmann, Hans Langmaack, Holger Pfeifer, Harald Ruess, and Wolf Zimmermann. Compiler Correctness and Implementation Verification: The *Verifix* Approach. In P. Fritzson, editor, *Proceedings of the Poster Session of CC '96 – International Conference on Compiler Construction*, pages 65 – 73, IDA Technical Report LiTH-IDA-R-96-12, Linkøping, Sweden, 1996.

[GG75] J.B. Goodenough and S.L. Gerhart. Toward a Theory of Test Data Selection. *SIGPLAN Notices*, 10(6):493–510, June 1975.

[GGH+97] Th. Gaul, G. Goos, A. Heberle, W. Zimmermann, and W. Goerigk. An Architecture for Verified Compiler Construction. In *Joint Modular Languages Conference JMLC'97*, Linz, Austria, March 1997.

[GGZ98] Wolfgang Goerigk, Thilo Gaul, and Wolf Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, Malente, 1998. Springer Verlag.

[GH96] Wolfgang Goerigk and Ulrich Hoffmann. The Compiler Implementation Language ComLisp. Technical Report Verifix/CAU/1.7, CAU Kiel, June 1996.

[GH98a] Wolfgang Goerigk and Ulrich Hoffmann. Compiling ComLisp to Executable Machine Code: Compiler Construction. Technical Report Nr. 9812, Institut für Informatik, CAU, October 1998.

[GH98b] Wolfgang Goerigk and Ulrich Hoffmann. Rigorous Compiler Implementation Correctness: How to Prove the Real Thing Correct. In *Proceedings FM-TRENDS'98 International Workshop on Current Trends in Applied Formal Methods*, Lecture Notes in Computer Science, Boppard, 1998. To appear.

[GH98c] Wolfgang Goerigk and Ulrich Hoffmann. The Compiling Specification from ComLisp to Executable Machine Code. Technical Report Nr. 9713, Institut für Informatik, CAU, Kiel, December 1998.

[GMO96] Wolfgang Goerigk and Markus Müller-Olm. Erhaltung partieller Korrektheit bei beschränkten Maschinenressourcen. – Eine Beweisskizze –. Technical Report Verifix/CAU/2.5, CAU Kiel, 1996.

[Goe99] Wolfgang Goerigk. On Trojan Horses in Compiler Implementations. In F. Saglietti and W. Goerigk, editors, *Proc. des Workshops* Sicherheit und Zuverlässigkeit softwarebasierter Systeme, ISTec-Berichte, Garching, 1999. To appear.

[GS86] Susanne Graf and Joseph Sifakis. A modal characterization of observational congruence on finite terms of CCS. *Information and Control*, 68:125–145, 1986.

[Gur91] Y. Gurevich. Evolving Algebras; A Tutorial Introduction. *Bulletin EATCS*, 43:264–284, 1991.

[GZG+98] Wolfgang Goerigk, Wolf Zimmermann, Thilo Gaul, Andreas Heberle, and Ulrich Hoffmann. Praktikable Konstruktion korrekter Übersetzer. In *Proceedings Softwaretechnik ST'98*, volume 18(3) of *Softwaretechnik-Trends*, pages 26 – 34, Paderborn, 1998.

[GZG99] Thilo Gaul, Wolf Zimmermann, and Wolfgang Goerigk. Construction of Verified Software Systems with Program-Checking: An Application To Compiler Back-Ends. International Workshop on "Runtime Result Verification", Trento, Italy, 1999.

[HGG+99] A. Heberle, T. Gaul, W. Goerigk, G. Goos, and W. Zimmermann. Construction of Verified Compiler Front-Ends with Program-Checking. In *Proceedings of PSI '99: Andrei Ershov Third International Conference on Perspectives Of System Informatics*, Lecture Notes in Computer Science, Novosibirsk, Russia, 1999. Springer Verlag. To appear.

[HHF+94] Jifeng He, C. A. R. Hoare, Martin Fränzle, Markus Müller-Olm, Ernst-Rüdiger Olderog, Michael Schenke, Michael R. Hansen, Anders P. Ravn, and Hans Rischel. Provably correct systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'94)*, volume 863 of *Lecture Notes in Computer Science*, pages 288–335. Springer, 1994.

[HHMO+96] J. He, C. A. R. Hoare, M. Müller-Olm, E.-R. Olderog, M. Schenke, M. R. Hansen, A. P. Ravn, and H. Rischel. The ProCoS approach to the design of real-time systems: Linking different formalisms. Tutorial Material for FME'96 (Formal Methods Europe '96), March 1996.

[HJS93] C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.

[HKPV95] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 373–382. ACM, 1995.

[HL98] Andreas Heberle and Welf Löwe. On ASM-Based Specification of Programming Language Semantics and Reusable Correct Compilations. In Uwe Glässer and Peter H. Schmitt, editors, *Proceedings of the 5th International Workshop on Abstract State Machines*, pages 68–90, 1998.

[Hoa91] C. A. R. Hoare. Refinement algebra proves correctness of compiling specifications. In C.C. Morgan and J.C.P. Woodcock, editors, *3rd Refinement Workshop*, pages 33–48. Springer-Verlag, 1991.

[Hof98] Ulrich Hoffmann. *Compiler Implementation Verification through Rigorous Syntactical Code Inspection*. PhD thesis, Technische Fakultät der Christian-Albrechts-Universität zu Kiel, Kiel, 1998.

[HPB93] He Jifeng, I. Page, and J. P. Bowen. Towards a provably correct hardware implementation of Occam. In G. J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods (CHARME'93)*, volume 683 of *Lecture Notes in Computer Science*, pages 214–225. Springer-Verlag, 1993.

[HvK95] C.A.R. Hoare and Burghard von Karger. Sequential calculus. *Information Processing Letters*, 53:123–130, 1995.

[Koz83] D. Kozen. Results on the propositional mu-calculus. *TCS*, 27:333–354, 1983.

[Lan97a] H. Langmaack. Contribution to Goodenough's and Gerhart's Theory of Software Testing and Verification: Relation between Strong Compiler Test and Compiler Implementation Verification. *Foundations of Computer Science: Potential-Theory-Cognition. LNCS*, 1337:321–335, 1997.

[Lan97b] H. Langmaack. The ProCoS Approach to Correct Systems. *Real Time Systems*, 13:253–275, 1997.

[Lan97c] Hans Langmaack. Softwareengineering zur Zertifizierung von Systemen. *it+ti — Informationstechnik und Technische Informatik*, 39(3):41–47, 1997.

[Lan97d] Hans Langmaack. Theoretische Informatik ist Grundlage für das sichere Beherrschen realistischer Software und Systeme. *25 Jahre Informatik an der Universität Hamburg. Informatik: Stand, Trends, Visionen*, pages 47–62, 1997.

[MO90] Markus Müller-Olm. Korrektheit einer Übersetzung der Sprache rekursiver Funktionsdefinitionen erster Ordnung in eine einfache imperative Sprache. Master's thesis, CAU Kiel, 1990.

[MO95a] Markus Müller-Olm. An Exercise in Compiler Verification. Internal report, CS Department, University of Kiel, 1995.

[MO95b] Markus Müller-Olm. A short description of the prototype compiler. ProCoS Technical Report [Kiel MMO 14/1], Christian-Albrechts-Universität Kiel, Germany, August 1995.

[MO96a] Markus Müller-Olm. *Modular Compiler Verification*. Dissertation, Technische Fakultät der Universität Kiel, Germany, 1996.

[MO96b] Markus Müller-Olm. Three Views on Preservation of Partial Correctness. Technical Report Verifix/CAU/5.1, CAU Kiel, October 1996.

[MO97] Markus Müller-Olm. *Modular Compiler Verification: A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1997. PhD Thesis.

[MO98] Markus Müller-Olm. Derivation of characteristic formulae. *Electronic Notes in Theoretical Computer Science*, 18:12, August 1998. URL: `http://www.elsevier.nl/locate/entcs/volume18.html`.

[MO99] Markus Müller-Olm. A modal fixpoint logic with chop. In Christoph Meinel and Sophie Tison, editors, *STACS'99*, volume 1563 of *Lecture Notes in Computer Science*, pages 510–520. Springer, 1999.

[Mos85] Ben Moszkowski. A temporal logic for multi-level reasoning about hardware. *IEEE Computer*, 18(2):10–19, 1985.

[MOSC99] Markus Müller-Olm, Bernhard Steffen, and Rance Cleaveland. On the evolution of reactive components: A process-algebraic approach. In *FASE'99*, volume 1577 of *Lecture Notes in Computer Science*, pages 161–175. Springer, 1999.

[MOSS99] Markus Müller-Olm, David Schmidt, and Bernhard Steffen. Model-checking: A tutorial introduction. In *SAS'99*, Lecture Notes in Computer Science. Springer, 1999. to appear.

[MOW99] Markus Müller-Olm and Andreas Wolf. On excusable and inexcusable failures: Towards an adequate notion of translation correctness. In *FM'99*, Lecture Notes in Computer Science. Springer, 1999. to appear.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In M. Dezani-Ciancaglini and U. Montanari, editors, *Proc. 5th Internat. Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[Rav95] Anders P. Ravn. *Design of Embedded Real-Time Computing Systems*. Doctoral dissertation, Department of Computer Science, Danish Technical University, Lyngby, DK, October 1995. Available as technical report ID-TR: 1995-170.

[RRH93] Anders P. Ravn, Hans Rischel, and Kirsten M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1):41–55, January 1993.

[SI94] Bernhard Steffen and Anna Ingólfsdóttir. Characteristic formulae for processes with divergence. *Information and Computation*, 110(1):149–163, 1994.

[Tho84] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–763, 1984. Also in ACM Turing Award Lectures: The First Twenty Years 1965-1985, ACM Press, 1987, and in Computers Under Attack: Intruders, Worms, and Viruses Copyright, ACM Press 1990.

[vK95] Burghard von Karger. An algebraic approach to temporal logic. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *Proceedings of the Sixth International Joint Conference on Theory and Practice of Software Development*, LNCS 915, pages 232–246. Springer-Verlag, 1995.

[vK97] Burghard von Karger. *Temporal Algebra*. Habilitationsschrift, Christian-Albrechts-Univ. Kiel, 1997. Available from www.informatik.uni-kiel.de/ bvk/.

[vK98] Burghard von Karger. Temporal algebra. *Mathematical Structures in Computer Science*, 8(3):277–320, June 1998.

[vKB98] B. von Karger and R. Berghammer. A relational model for temporal logic. *Logic Journal of the IGPL*, 6(2):157–173, 1998. Available from www.oup.co.uk/igpl/Volume_06/Issue_02.

[WB97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.

[Wol99a] Andreas Wolf. An Exercise in Compiler Verification Revisited – Preserving Partial Correctness . Technical Report Verifix/CAU/6.1, CAU Kiel, February 1999.

[Wol99b] Andreas Wolf. The Adequacy of a Loop's Definition. Technical Report Verifix/CAU/6.2, CAU Kiel, February 1999.

[ZG97] W. Zimmermann and T. Gaul. On the Construction of Correct Compiler Back-Ends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.

[Zho93] Zhou Chaochen. Duration calculi: An overview. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *Lecture Notes in Computer Science*, pages 256–266. Springer-Verlag, 1993.

[ZHR91] Zhou Chaochen, C. A. R. Hoare, and Anders P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.

[ZHS93] Zhou Chaochen, Michael R. Hansen, and Peter Sestoft. Decidability and undecidability results for duration calculus. In P. Enjalbert, A. Finkel, and K. W. Wagner, editors, *Symposium on Theoretical Aspects of Computer Science (STACS 93)*, volume 665 of *Lecture Notes in Computer Science*, pages 58–68. Springer-Verlag, 1993.