A Framework for Verifying and Implementing Embedded Systems

Klaus Schneider and Tobias Schuele

Reactive Systems Group
Department of Computer Science, University of Kaiserslautern
P.O. Box 3049, 67653 Kaiserslautern, Germany
{Tobias.Schuele,Klaus.Schneider}@informatik.uni-kl.de
http://rsg.informatik.uni-kl.de

Abstract

We present a framework for the development of embedded systems called Averest. It includes a compiler for synchronous programs, a symbolic model checker, and a tool for hardware and/or software synthesis. Averest can be used for modeling and verifying finite as well as infinite state systems. Thus, Averest is not only well–suited for hardware design, but also for the development of embedded software. In particular, our input language Quartz allows the description of concurrent programs at a high level of abstraction.

1 Introduction

In Averest [13], a system is described using our Esterel–like synchronous programming language Quartz, and specifications can be given in temporal logics such as LTL, CTL, subsets of CTL*, and the μ–calculus. Currently, Averest consists of the following components that cover large parts of typical design flows:

- Ruby: a compiler for translating Quartz programs to transition systems
- Beryl: a symbolic model checker for finite and infinite state transition systems
- Topaz: a code generator for hardware and/or software

Figure 1 shows the dependencies between these components: A given Quartz program is first translated to a symbolically represented transition system in Averest's Interchange Format AIF which is based on XML [21]. Temporal logic specifications are thereby translated to alternating ω–automata and/or μ–calculus formulas. Depending on the used data types (integers with limited or unlimited bitwidth), the transition system has finitely or infinitely many states. The AIF description can then be used for verification and code generation. Moreover, third–party tools, e.g. other model checkers such as SMV, are integrated via translations of AIF descriptions to the input formats of these tools.

2 Compiling Synchronous Programs

The basic paradigm of synchronous languages [1, 6] is the distinction between micro and macro steps in a program. From a programmer's view, micro steps do not take time, while macro steps

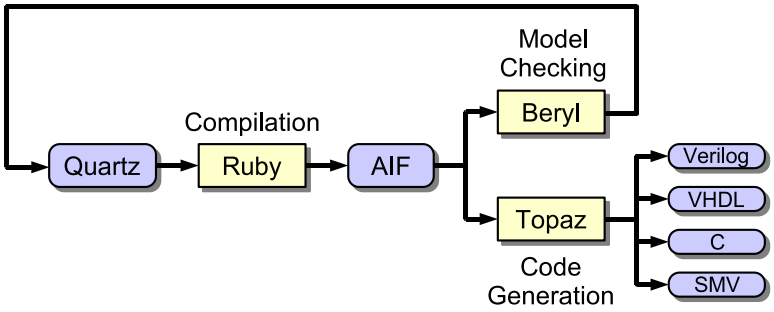

Figure 1: Averest Design Flow

take one unit of time. Thus, consumption of time is explicitly programmed by partitioning the program into macro steps. This programming model, referred to as perfect synchrony [6], together with concurrency allows the compilation of multi–threaded synchronous programs to deterministic single–threaded code. As a result, synchronous programs can be executed on small microcontrollers without the need to run complex operating systems.

A distinct feature of synchronous languages is their detailed formal semantics that is usually given by means of transition rules in structural operational semantics. This makes synchronous languages attractive for safety–critical applications where formal verification is mandatory. Moreover, they allow direct translations to synchronous hardware circuits. On the other hand, the synchronous programming model challenges the compilers: Intrinsic problems like *causality and schizophrenia problems* must be solved [14, 15].

An introductory example is the program shown in Figure 2. This program contains four main ingredients which are characteristic for synchronous programs: synchronization (**await**), signal emission (**emit**), sequencing (;), and concurrency (||). The behavior of the program is described as follows: There are two processes that concurrently wait for the input signals a and b. As soon as both signals have occurred, the program immediately emits the output signal o. Finally, this behavior is repeated each time the reset signal r occurs.

```
module ABRO:
input
  a,b,r: event;
output
  o:event;
  loop
    [await a || await b];
    emit o
  each r
spec
  Spec1: A G (o -> a | b)
end
```

Figure 2: Example for Synchronous Programs

Our compiler Ruby translates given Quartz programs to symbolic descriptions of transition systems. The correctness of Ruby has been formally proved by means of an interactive theorem prover [9–11, 15]. In addition to most Esterel statements, Quartz offers the following features:

- generic programs (parameters must be compile time constants)
- (synchronous, asynchronous, interleaved) concurrency
- explicit nondeterministic choice

- fixed bitwidth integers with a complete set of binary arithmetic operations
- infinite integers (with a restriction to decidable fragments of arithmetic)
- temporal logic specifications (LTL and certain subsets of CTL*)

Generic statements are a powerful means for describing parameterized systems. For example, the above program can be generalized to an arbitrary but fixed number N of processes using the **syncpar** statement that implements synchronous parallelism (`ubits(N)` is thereby the number of bits required to represent the compile–time constant `N` as an unsigned binary number):

```
syncpar for i:bitvec[ubits(N)] := 0 to N-1 do
  await a[i]
end;
emit o
```

Figure 3: Example for a Generic Statement

Besides the translation of Quartz programs to transition systems, Ruby translates temporal logic specifications to alternating ω–automata and/or μ–calculus formulas, where the classification into safety, liveness, and fairness [12] is taken into account. This classification not only makes the verification process more efficient, it is also crucial for bounded model checking (see next section).

3 Verifying Finite and Infinite Transition Systems

Beryl is a symbolic model checker for the verification of finite and infinite state systems. As usual in symbolic model checking, finite sets are encoded by their characteristic functions in propositional logic using binary decision diagrams (BDDs) [4]. For the representation of infinite sets, however, more powerful logics are required, since propositional logic is naturally limited to the representation of finite sets. For that purpose, we use Presburger arithmetic, a decidable first–order predicate logic over the integers with addition as the basic operation [7, 8]. In the same way, we lift the specification language from propositional logic to Presburger arithmetic. An important aspect concerning efficient implementations is that every Presburger formula can be translated to a finite automaton that encodes its models. As there exists for every finite automaton an equivalent minimal one, automata can serve as a canonical normal form for Presburger formulas. This is very much in the same spirit as BDDs are used as a canonical normal form for propositional logic. Thus, automata can be viewed as a generalization of BDDs for the representation of infinite sets.

The ability to deal with data types over infinite domains makes Beryl attractive for the verification of datapaths without considering implementation specific details such as the bitwidth. In particular, there is no need to break a design down to Boolean expressions, as it is required for BDD based symbolic model checkers. For instance, large parts of microprocessors and assembler programs can be easily modeled in Presburger arithmetic. This is also an important step towards the verification in early design phases, where higher data types are used to model systems at an abstract level.

Beryl's specification language is the modal μ–calculus that subsumes popular temporal logics such as CTL and LTL. To check μ–calculus formulas, Beryl contains algorithms for global model checking [5, 12, 19], local model checking [3, 12, 19], and bounded variants thereof [2, 18, 20]. There are many differences between these algorithms, in particular, global and local model checking algorithms follow radically different paradigms: while global model checking algorithms use fixpoint iteration to check a specification, local model checking algorithms are based on the construction of proof trees (tableaux) using syntax directed decomposition rules.

The idea of bounded model checking is to reduce the verification problem to a satisfiability problem of the base logic, so that sophisticated SAT solvers can be employed to check the resulting formula.

For finite state systems, it is beneficial to be able to choose between different algorithms, since their runtimes may vary significantly. For example, safety properties can often be proved most efficiently using local model checking that incorporates induction–like reasoning. In contrast, bounded model checking is often superior when disproving a safety property by efficiently searching for a counter–example. For infinite state systems, however, the choice of the best algorithm is not primarily a matter of efficiency, but rather a matter of termination [19]. For instance, due to the different nature of global and local model checking algorithms, it may happen that one algorithm terminates for a given specification while the other does not, and vice versa. For this reason, Beryl contains different model checking algorithms and heuristics to achieve termination for a large class of specifications. Additionally, the implemented algorithms can even be combined, either manually or automatically. As an example, given a nested fixpoint formula, the outer fixpoint can be solved using local model checking and the inner one using global model checking. In this way, the class of specifications that can be checked automatically is extended even further [19, 20].

Beryl can also be used to determine tight bounds on the worst case execution time (WCET) of Quartz programs and other transition systems at an architecture independent level [16, 17]. WCET analysis is a crucial concern in the design of embedded systems, e.g. safety critical applications often require that certain tasks are completed before a strict deadline. One of the main problems of WCET analysis is that the number of executed instructions may heavily depend on the input data, in particular, the number of loop iterations may vary significantly. Using symbolic state space exploration techniques, however, it is possible to obtain exact information on the number of executed macro steps.

4 Generating Code for Synthesis

Topaz is a tool for translating transition systems generated by Ruby to hardware description languages (VHDL, Verilog) for hardware synthesis and to conventional programming languages (C) for software synthesis. For Verilog and VHDL code generation, finite state machines are generated that can be used as input for other tools like FPGA compilers to build hardware. The C code generation can be used for simulation or for a direct implementation on a microcontroller. For that purpose, interface functions are added to set the inputs and to get the outputs of a program. Topaz has been successfully used by our students for programming LEGO Mindstorm robots in practical courses.

5 Integrated Development Environment

The Averest tools have been integrated in the Eclipse development environment. Eclipse is an open source project dedicated to providing an extensible and full–featured platform for building software. As a major advantage, Eclipse allows the integration of third–party tools by means of plug–ins. In this way, the developer can benefit from standard components such as project and version management, while having access to specialized tools via comfortable graphical user interfaces. Figure 4 shows a screenshot of an Averest session in Eclipse. Key features of the Averest plug–in for Eclipse are an own perspective for Averest projects, syntax highlighting of Quartz programs, user–friendly dialog boxes for Ruby, Beryl, and Topaz, and the possibility to execute other tools (e.g. C compilers).

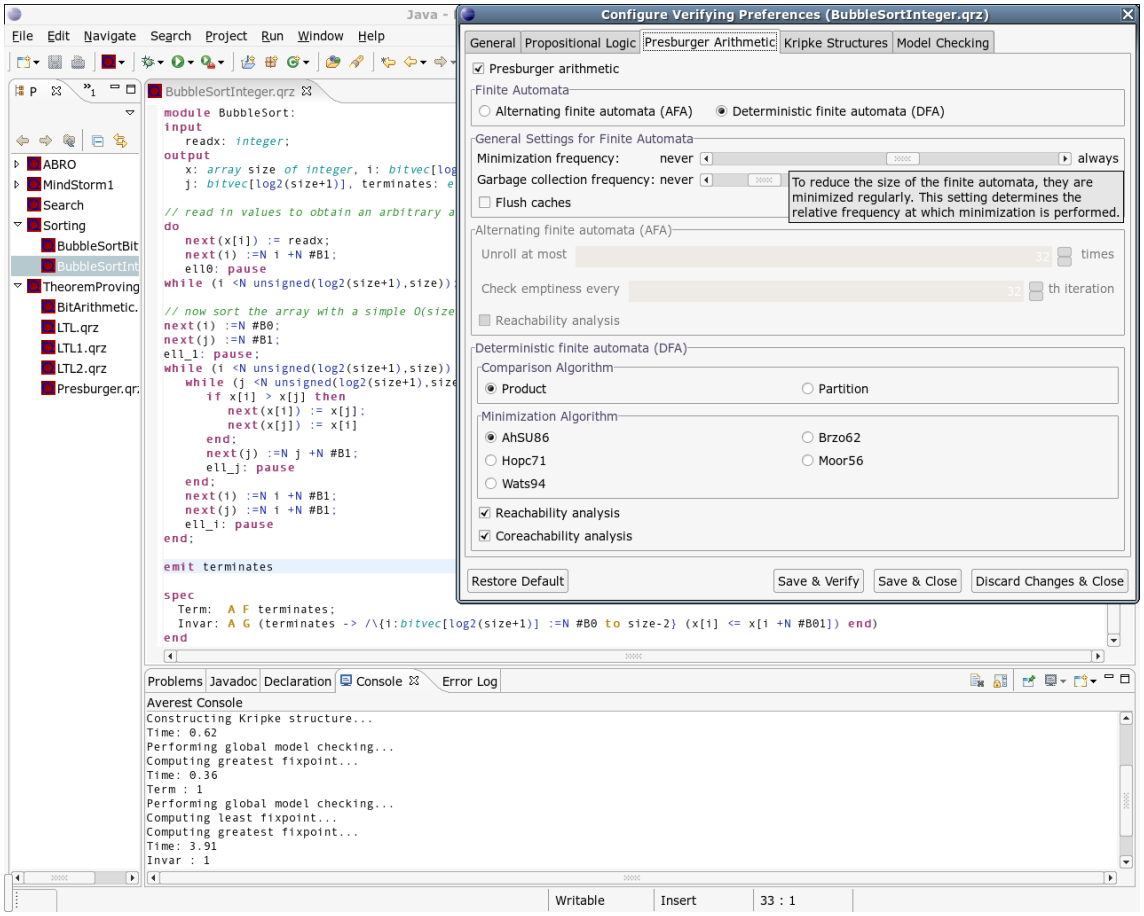

Figure 4: Screenshot of Averest Session in Eclipse IDE

6 Future Work

The Averest project aims at providing a complete set of tools for the specification, verification, and implementation of embedded systems. Currently, it consists of a compiler for our synchronous language Quartz, a symbolic model checker, and a tool for hardware and/or software synthesis. Even though some of the features are also available in other tools, Averest is unique in many respects: First, it contains the only non–commercial compiler for synchronous languages that is able to generate hardware and software from programs with delayed actions. Second, it contains efficient algorithms for symbolically translating temporal logic specifications to ω–automata that often outperform traditional approaches. Third, our model checker can be used for the verification of finite and infinite state systems using different model checking algorithms. In particular, it can benefit from local model checking, whereas most available symbolic model checkers are restricted to global model checking.

In the future, we plan to develop interfaces to other tools such as the HOL theorem prover. Clearly, also Ruby, Beryl, and Topaz will be improved. In particular, we are currently developing better algorithms for causality analysis in Ruby to eliminate constructive cyclic dependencies [14]. On the specification side, we plan to support Accellera's property specification language PSL. Moreover, we are working on additional heuristics to achieve termination for infinite state systems that make use of invariants and well–founded orderings. Regarding software synthesis, there will be a direct translation to C code that maintains the structure of the given Quartz program. In contrast to the currently generated cycle–based code, this will produce much faster code of less size. Finally, we are extending Averest by tools for supervisory control.

References

[1] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[2] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*, pp. 193–207, Amsterdam, The Netherlands, 1999. Springer.

[3] J.C. Bradfield. *Verifying Temporal Properties of Systems*. Progress in Theoretical Computer Science. Birkhäuser, Boston, Basel, Berlin, 1992.

[4] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[5] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT, London, England, 1999.

[6] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[7] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. *Sprawozdanie z I Kongresu Matematyków Krajów Słowiańskich, Warszawa 1929*, pp. 92–101, Warszawa, 1930.

[8] M. Presburger. On the completeness of a certain system of arithmetic of whole numbers in which addition occurs as the only operation. *History and Philosophy of Logic*, 12:225–233, 1991. Translation and commentary by D. Jacquette.

[9] K. Schneider. A verified hardware synthesis for Esterel. In *Workshop on Distributed and Parallel Embedded Systems (DIPES)*, pp. 205–214, Paderborn, Germany, 2000. Kluwer.

[10] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)*, pp. 143–156, Newcastle upon Tyne, UK, June 2001. IEEE Computer Society.

[11] K. Schneider. Proving the equivalence of microstep and macrostep semantics. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Higher Order Logic Theorem Proving and its Applications (TPHOL)*, volume 2410 of *LNCS*, pp. 314–331, Hampton, VA, USA, 2002. Springer.

[12] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.

[13] The Averest Framework, 2006. `http://www.averest.org`.

[14] K. Schneider, J. Brandt, and T. Schuele. Causality analysis of synchronous programs with delayed actions. In *Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 179–189, Washington D.C., USA, September 2004. ACM.

[15] K. Schneider, J. Brandt, and T. Schuele. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 2006.

[16] T. Schuele and K. Schneider. Exact runtime analysis using automata-based symbolic simulation. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 153–162, Mont Saint-Michel, France, 2003. IEEE Computer Society.

[17] T. Schuele and K. Schneider. Abstraction of assembler programs for symbolic worst case execution time analysis. In *International Design Automation Conference (DAC)*, pages 107–112, San Diego, California, USA, 2004. ACM.

[18] T. Schuele and K. Schneider. Bounded model checking of infinite state systems: Exploiting the automata hierarchy. In *Formal Methods and Models for Codesign (MEMOCODE)*, pages 17–26, San Diego, CA, June 2004. IEEE.

[19] T. Schuele and K. Schneider. Global vs. local model checking: A comparison of verification techniques for infinite state systems. In *International Conference on Software Engineering and Formal Methods (SEFM)*, pp. 67–76, Beijing, China, September 2004. IEEE.

[20] T. Schuele and K. Schneider. Three-valued logic in bounded model checking. In *Formal Methods and Models for Codesign (MEMOCODE)*, Verona, Italy, 2005. IEEE.

[21] XML. *Extensible Markup Language 1.1*. World Wide Web Consortium (W3C), February 2004. `http://www.w3.org/TR/xml11`.