Averest: Specification, Verification, and Implementation of Reactive Systems

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

1. Introduction

The Averest framework[1] provides a set of tools for the specification, verification, and implementation of reactive systems. Currently, it consists of a compiler for our Esterel-like synchronous programming language Quartz, a symbolic model checker for finite and infinite state transition systems, and a code generator for hardware/software synthesis.

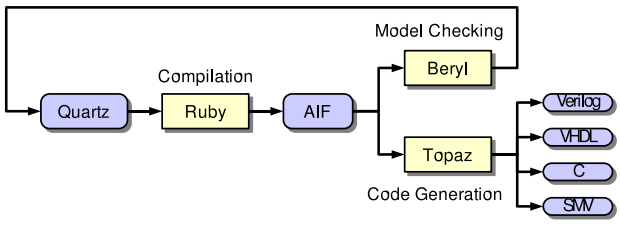

Figure 1. Averest Design Flow

Figure 1 shows the Averest design flow: A given Quartz program is first translated to a symbolically represented transition system in Averest's Interchange Format AIF that is based on XML. 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 resulting transition systems have finitely or infinitely many states. The AIF description can then be used for verification and for code generation. In the following, we briefly describe the main tools of Averest, namely Ruby, Beryl, and Topaz.

2. Ruby: Compiling Synchronous Programs

A common paradigm of synchronous languages [1, 5] is the distinction between micro and macro steps of a program. From a programmer's view, micro steps do not take time, while all macro steps take the same amount of logical time. This programming model together with synchronous concurrency allows the compilation of multi-threaded synchronous programs to deterministic single-threaded code. A distinct feature of synchronous languages is their formal semantics that is usually given by means of SOS transition rules. Moreover, synchronous programs can be directly translated to hardware circuits. On the other hand, the synchronous programming model challenges the compilers: Intrinsic problems like *causality and schizophrenia problems* must be solved.

Our compiler Ruby is able to translate Quartz programs to symbolic descriptions of transition systems. The correctness of these translations have been formally proved by means of an interactive theorem prover [7, 8, 9, 11]. In addition to most Esterel statements, Quartz offers the following features: generic programs (compile time parameters), different forms of concurrency (synchronous, asynchronous, interleaved), explicit nondeterministic choice, fixed bitwidth integers with a complete set of arithmetic operations, arrays, infinitely large integers (with a restriction to Presburger arithmetic, see Section 3), and temporal logic specifications (LTL, CTL, and subsets of CTL*). The specifications are translated to alternating ω-automata and/or μ-calculus formulas, where the classification into safety, liveness, and fairness [10] is taken into account.

3. Beryl: Symbolic Model Checking

Beryl is a symbolic model checker for the verification of finite and infinite state systems. It contains algorithms for global, local, and bounded model checking of μ-calculus formulas [3, 10, 16, 17]. The ability to deal with infinite state systems makes Beryl particularly attractive for the verification at high abstraction levels, where implementation specific details such as the bitwidth can be neglected. Additionally, Beryl can also be used to determine the worst case execution time of Quartz programs and other transition systems at an architecture-independent level [14, 15].

[1] http://www.averest.org

For finite state systems, it is beneficial to choose between different model checking 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. For infinite state systems, however, the choice of the best algorithm is not primarily a matter of efficiency, but rather a matter of termination [17]. For this reason, Beryl is not restricted to a particular approach, but offers different algorithms to achieve termination for a large class of specifications.

As usual in symbolic model checking, *finite* sets are represented by means of propositional logic. Currently, Beryl supports the BDD packages CUDD [4] and BuDDy [2] as well as the SAT solver zChaff [6]. For the representation of *infinite* sets, we employ Presburger arithmetic, a decidable predicate logic over the integers. Presburger arithmetic formulas are internally represented by means of finite automata (there are decision procedures based on deterministic and alternating finite automata).

4. Topaz: Generating Code for Synthesis

Topaz is a tool for translating transition systems generated by Ruby to hardware description languages (Verilog, VHDL) for hardware synthesis or to conventional programming languages such as C for software synthesis. For Verilog and VHDL, 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 provided to set the inputs and to get the outputs of a program. For finite state systems, there is also the possibility to generate code for other model checkers such as SMV.

5. Future Work

We are currently developing better algorithms for causality analysis in Ruby to eliminate constructive cyclic dependencies [11, 12, 13]. Moreover, the estimation of reachable states, as already available, will be improved and will be used by Topaz for code generation. Furthermore, we plan to support Accellera's property specification language PSL. Although Beryl already contains different model checking algorithms and heuristics to achieve termination for infinite state systems, we are working on additional techniques that make use of invariants and well-founded orderings. Topaz will be improved to generate event based code that proved to be much faster for certain applications than the currently generated cycle-based code. Additionally, we are working on tools for automatic program synthesis that use algorithms developed in supervisory control theory.

References

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

[2] BuDDy: Binary Decision Diagram Package, Release 2.4. http://buddy.sourceforge.net.

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

[4] CUDD: CU Decision Diagram Package, Release 2.4.0. http://vlsi.colorado.edu.

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

[6] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Design Automation Conference (DAC)*, pp. 530–535, Las Vegas, USA, 2001. ACM.

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

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

[9] K. Schneider. Proving the equivalence of microstep and macrostep semantics. In *Higher Order Logic Theorem Proving and its Applications (TPHOL)*, LNCS 2410, pp. 314–331, Hampton, USA, 2002. Springer.

[10] K. Schneider. *Verification of Reactive Systems – Formal Methods and Algorithms*. Springer, 2003.

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

[12] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Improving constructiveness in code generators. In *Synchronous Languages, Applications, and Programming (SLAP)*, Edinburgh, Scotland, UK, 2005.

[13] K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Conference on Application of Concurrency to System Design (ACSD)*, St. Malo, France, June 2005. IEEE.

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

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

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

[17] 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.