Model Checking: Back and Forth Between Hardware and Software

Edmund Clarke[1], Anubhav Gupta[1], Himanshu Jain[1], and Helmut Veith[2]

[1] School of Computer Science, Carnegie Mellon University
{emc, anubhav, hjain}@cs.cmu.edu
[2] Institut für Informatik (I-7), Technische Universität München, Germany
veith@in.tum.de

Abstract. The interplay back and forth between software model checking and hardware model checking has been fruitful for both. Originally intended for the analysis of concurrent software, model checking was first used in hardware verification. The abstraction methods developed for hardware verification however have been a stepping stone for the new generation of software verification tools including SLAM, BLAST, and MAGIC which focus on control-intensive software in C. Most recently, the experience with software verification is providing new leverage for verifying hardware designs in high level languages.

1 From Software Verification to Hardware Verification

The origins of model checking date back to the early 1980s, when Clarke and Emerson [5] and, independently, Queille and Sifakis [19] introduced a new algorithmic approach for the verification of computer systems. Their approach amounts to checking the satisfaction of a logical specification over a system model which is represented by an annotated directed graph; hence, the term *model checking*. Prior to that, the use of temporal logic for the analysis and specification of computer systems had been advocated by Pnueli [18], and model checking has in fact been employing variants of temporal logic as the predominant specification language ever since.

Experiments with early model checkers however quickly made clear that the size of the model represents the crucial technical barrier for realizing the full potential of model checking. In fact, the progress on the state explosion problem is the key to appreciating the technical achievements in model checking during the last decades. The development of *symbolic model checking* [3, 17] was arguably a turning point in the formal methods field. Employing a combination of binary decision diagrams and fixed-point algorithms, the symbolic model verifier (SMV) became the first model checker to verify models with hundreds of Boolean variables and a tool to benchmark new ideas for more than a decade.

Model checking was originally designed for the verification of finite state systems. Although the first practically useful applications of model checking were oriented towards hardware verification, where the finite state restriction comes naturally, the method was originally conceived of as an approach to software verification. The early papers on model checking clearly drew their motivations from the software area, focusing in particular on concurrency properties to be verified over the synchronization skeleton of a program, i.e., a finite abstract model which preserves the relevant behavior for interprocess communication [6]. This setting reflects three major principles in successful model checking applications:

1. The separation of control flow and data flow in the system.
2. Abstraction techniques which enable us to remove significant parts of the data flow from the system.
3. Efficient tools to check properties on the abstracted systems.

Since hardware designs typically have a relatively clear separation of data and control, and are finite state, it was very natural to apply model checking to verify hardware systems [11]. In combination with symbolic model checking, this resulted in making hardware model checking a success.

Starting with [8], there has been a lot of work in devising systematic approaches to abstraction. Abstraction techniques reduce the program state space by mapping the set of states of the actual system to an abstract, and smaller, set of states in a way that preserves the actual behaviors of the system. In systems where there is no clear distinction between the control flow and data flow, it may be necessary to refine the abstraction. This abstraction refinement process has been automated by the counterexample-guided abstraction refinement paradigm [16, 7, 1], or CEGAR for short. In the CEGAR approach, the model checker initially works on a coarse abstraction. If the model checker finds a "spurious" abstract error trace which does not occur in the original program, it analyzes the error trace to refine the abstraction. This process is repeated until the property is either verified or disproved. Note that model checking and abstract interpretation [12] share many common techniques which deserve further exploration.

2 From Hardware Verification to Software Verification

The last several years have seen the development of a new generation of software model checkers such as SLAM, BLAST and MAGIC [1, 14, 4] which are based on predicate abstraction [13] and counterexample-guided abstraction refinement (CEGAR). Note that the CEGAR approach, first proposed for hardware verification [16, 7], lends itself equally naturally for software [1]. Figure 1 illustrates the application of CEGAR to software. This new generation of software model checkers has been quite successful for specific classes of software, most notably device drivers, embedded software, and system software whose specifications are closely related to the control flow. While predicate abstraction is highly versatile in expressing the control flow conditions of a program, it is apparently much harder to reason about data, in particular dynamic data structures.

An alternative approach is taken by the CBMC model checker [10]. CBMC is based on bounded model checking [2], i.e., counterexample search using a SAT procedure. CBMC exploits the relatively simple semantics of the C language by describing a SAT formula which amounts to a symbolic unwinding of the C program. While this approach in principle allows to account for complicated dynamic data structures, current SAT solvers enable us only to perform a relatively small number of unwindings.

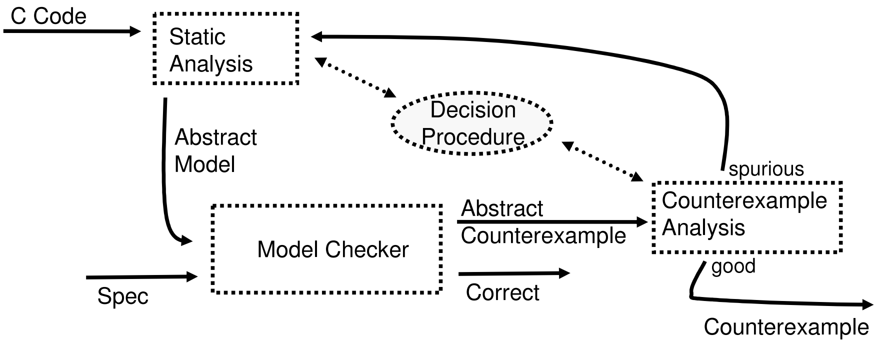

Fig. 1. Counterexample-Guided Abstraction Refinement for Software.

3 Back to Hardware

Most model-checkers used in the hardware industry use a very low level design, usually a netlist, but time-to-market requirements have rushed the Electronic Design and Automation (EDA) industry towards design paradigms that offer a very high level of abstraction. This high level can shorten the design time by hiding implementation details and by merging design layers. As part of this process, an abundance of C-like system design languages like SystemC, SpecC has emerged. They promise to allow joint modelling of both the hardware and software component of a system using a language that is well-known to engineers.

Some fragments of these languages are synthesizable, and thus allow the application of netlist or RTL-based formal verification tools. However, the higher abstraction levels offered by most of these languages are not yet amenable to rigorous, formal verification. This is caused by the high degree of asynchronous concurrency used by the models, which requires thread interleaving semantics. Since languages like SystemC are closer to concurrent software than to a traditional hardware description, one needs techniques from software verification to verify programs written in these languages [15].

4 Conclusion

Control-Intensive versus Data-Intensive Systems

We have argued that the success of model checking in hardware is closely related to the relatively clear separation between the control flow and the data flow. As illustrated in Figure 2, this phenomenon occurs for both hardware and software, and explains why model checking is particularly useful for control-intensive software e.g. in embedded systems, device drivers etc.

Perspectives

Going back and forth between hardware and software, the research in model checking is gradually pushing the limits of the method by means of automated or manually

	Hardware	Software
Data Intensive	Digital Signal Processors Floating Point Units Graphical Processors	Verifying Compiler Financial Software
Control Intensive	Cache Coherence Protocols Bus Controllers	Embedded Software Device Drivers

Fig. 2. Control-intensive versus data-intensive systems.

assisted abstraction, and has extended the reach of model checking quite significantly. As expressed in Rushby's notion of "disappearing formal methods", our goal is for model checking to finally become a push-button technology for certain classes of software such that the trade-off between the preciseness and the computational cost of the correctness analysis can be controlled by a few simple parameters. Generally though, the principal undecidability of virtually all questions in software verification makes clear that there is no silver bullet for verification, and there will always be a need to design model checking methods specific to problem classes.

References

1. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proc. Model Checking Software, 8th International SPIN Workshop*, volume 2057 of LNCS, pages 103–122, 2001.
2. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th Conference on Design Automation (DAC)*, pages 317–320, 1999.
3. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *In Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.
4. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *Proc. 25th Int. Conference on Software Engineering (ICSE)*, pages 385–395, 2003. Extended version in IEEE Transactions on Software Engineering, 2004.
5. E. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs: Workshop*, volume 131 of *LNCS*, pages 52–71. 1981.
6. E. Clarke, E. A. Emerson, A. P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. *Proc. POPL*, pages 117-126, 1983.
7. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 154–169, 2000. Extended version in *J. ACM* 50(5): 752–794, 2003.
8. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

9. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
10. E. Clarke, D. Kroening and F. Lerda. A Tool for Checking ANSI-C Programs. In *Proc. TACAS*, pages 168–176, 2004.
11. E. Clarke and B. Mishra. Automatic Verification of Asynchronous Circuits. In *Proc. Logic of Programs*, pages 101-115, 1983.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
13. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proc. Computer Aided Verification (CAV)*, volume 1254 of LNCS, pages 72–83, 1997.
14. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proc. ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 58–70, 2002.
15. H. Jain, D. Kroening, and E. Clarke. Verification of SpecC using predicate abstraction. In *MEMOCODE 04*, pages 7–16. IEEE, 2004.
16. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.
17. K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, Dordrecht, 1993.
18. A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science (FOCS)*, pages 46–67, 1977.
19. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Symposium in Programming*, volume 137 of *LNCS*, pages 337–351, 1982.