Formal Analysis for Embedded Real-Time Systems

John Rushby

Formal Analysis for Embedded Real-Time Systems

John Rushby
Computer Science Laboratory, SRI International, Menlo Park CA 94025 USA

Abstract: Timed systems are notoriously hard to debug and to verify because the continuous nature of time allows vast numbers of different behaviors; embedded systems must often deal with faults, and these introduce another dimension of complexity. Simulation and testing provide little assurance in these domains because they can visit only a small fraction of the possible behaviors. Formal methods of analysis have some promise, but until recently they could deal only with one dimension at a time: classical model checking could cope with faults but could not model continuous time; model checkers for timed automata could deal with continuous time but not the "case explosion" due to faults.
Recently, a new class of "infinite bounded" model checkers has been developed; these show promise that they can cope simultaneously with both continuous time and discrete faults.

Keywords: formal methods, model checking, automated verification

1. Introduction to Model Checking

Computer systems are hard to debug and to verify because the presence of decision points in the flow of control introduces discontinuous behavior: we cannot assume that "nearby" inputs and states should produce "nearby" outputs—and hence there is no justification for extrapolating from tested cases to untested ones. In the absence of continuity, the only way to fully analyze the behavior of a computer system is explicitly to consider every case (either by testing or by reasoning).
This is infeasible in general, but when the state and input variables range over fairly small discrete sets of values there is a technology called "model checking" that *can* consider every case. The simplest form of this technology is an *explicit state* model checker: this is like a simulator for the programming or modeling notation employed, except that it records every state (i.e., assignment of values to state variables) encountered, explores all the successors to each state (by considering all possible assignments to input variables and by resolving nondeterminism in all possible ways), and thereby systematically explores the entire space of reachable states.
A model checker can verify that a given predicate is satisfied in every reachable state (e.g., "at most one process is in its critical region"), and it can verify more complex properties (e.g., "every request is eventually followed by a grant") specified in a property language that is usually based on some form of temporal logic. When a property is violated, the model checker can construct a *counterexample*, which is an explicit scenario leading to the violation.
A modern explicit state model checker such as SPIN [12] typically can explore tens of millions of reachable states in an hour or so. On the other hand, *symbolic* model checkers based on BDDs (reduced, ordered Binary Decision Diagrams) can often explore systems having 10^{200} states in a few hours. As the name suggests, this class of model checkers represents states and programs symbolically: this is generally more compact and efficient than explicit representations (for example, the set of explicit states $\{(0,1),(0,2),\ldots,(1,2),(1,3),\ldots(2,3),(2,4),\ldots\}$ can be represented symbolically as $\{(x,y)\,|\,x < y\}$). NuSMV [4] and SAL [5] are examples of tool suites that provide symbolic model checkers.
Bounded model checkers also use a symbolic representation, and originally were specialized for finding counterexamples: given an explicit bound (hence the name) a bounded model checker decides whether there is a counterexample to the specified property that is no longer than the bound. Bounded model checkers use SAT solvers (tools that solve satisfiability problems for formulas in propositional logic) as their underlying engines; although SAT is the quintessential NP-complete problem, modern SAT solvers are very effective and bounded model checkers can often handle larger systems than symbolic model checkers. There are a several methods for extending bounded model checking from refutation (i.e., looking for counterexamples) to verification: one such method is k-induction (where the bound k is a small integer), which is a generalization of ordinary induction (which is 1-induction in this framework). NuSMV and SAL both support bounded model checking and k-induction.
Model checkers have their own idiosyncratic system specification and property languages. Given a system description in some other notation, it is first necessary to translate it into the language of the model checker concerned: there are several translators from model-based design notations into the languages of various model checkers. Next, the requirement must be formulated in the property language of the model checker; alternatively, the system description can be augmented with a *monitor* that raises an error sig-

nal when a requirement is violated and the model checker is then given the property "always not error." Finally, the chosen model checker can be invoked; these usually have several command line options (e.g., the bound to be used in bounded model checking) but their actual operation is automatic. The possible outcomes are that the property is verified, or it is refuted and a counterexample is provided (which may need to be translated back to the notation of the original system description), or the model checker exhausts memory or time (or the patience of its user). In the latter case, it may be necessarily to downscale the system description (i.e., to chop down the size of its data structures or other variable parameters, or to simplify it in other ways), or to abstract it (a more principled form of simplification). Excessive simplification may lead to false counterexamples, which can then suggest how the simplification can be made more precise: this process is automated as "counterexample guided abstraction refinement" (CEGAR) in software model checkers such as BLAST [11].

There are many examples of successful industrial application of model checking to embedded and other systems, for example [16]. Experience indicates that we can learn more (and find more bugs) using model checking to examine *all* the behaviors of a possibly simplified model than we can by testing just *some* of the behaviors of the real system.

2. Discrete Time

It is easy to extend model checking to timed systems when activities are synchronized to a global clock that provides discrete "ticks," as in hardware circuits. We simply add the clock to the system description and specify properties in terms of elapsed clock ticks: e.g., "every request is followed by a grant after at least 3 but no more than 7 ticks."

In the simplest models, the clock advances by one tick in every step and other components sit idle until it is time for them to do something. A model checker may then need to go infeasibly many steps deep to verify or refute a property. An alternative approach has each component indicate the number (which may be nondeterministically determined) of ticks into the future when it will next do something. The clock can then advance time all the way to the earliest such "timeout"; that component then performs its action and sets its next timeout. The modeled behavior thus alternates between component actions (there may be several components waiting on the same timeout) and the clock advancing time to the next timeout. This modeling approach using *timeout automata* was developed by Dutertre and Sorea [9] for continuous time, but Lamport uses a similar approach for discrete time [13].

In some applications, the regular ticks of a discrete clock can be replaced by irregular events: rather than the clock generating ticks that are received by other components, the components generate events that are perceived by some observer and we count how many events arrive until some interesting state is achieved. If events are related to clocks local to each component, then we may be able to verify timed properties of asynchronous systems. This is done, for example, in analysis of the startup protocol for the Time Triggered Architecture (TTA) [21], where we are able to verify that a 4-node TTA cluster will always start up within 23 TDMA slots; by varying the bound, we are able to show that 23 is indeed the worst case.

3. Continuous Time

While discrete time is appropriate in some circumstances, accurate modeling of most real-time systems requires that time is treated as a continuous variable. This takes us out of the realm of classical model checking because continuous variables can take an infinite number of values and therefore lead to an infinite state space, whereas classical model checking depends on explicit or symbolic enumeration of a finite state space. Of course, the continuous nature of time also makes real-time systems very hard to design and to test, and therefore makes some kind of automated analysis all the more desirable.

Fortunately, it is possible to combine finite automata and continuous time variables is a way that renders their properties decidable. These are called *timed automata* [1] and methods for model checking their properties are based on efficient ways for solving systems of linear inequalities. Of course, there are standard methods such as the simplex algorithm for analyzing systems of linear inequalities, but timed automata lead to problems that are both simpler and more difficult than those considered in classical linear programming: they are simpler because the inequalities are of restricted forms, and they are more difficult because different (discrete) states may have different behavior, thereby requiring much case analysis. Model checkers for timed automata, such as Uppaal [14] and Kronos [3] use special representations and data structures that are optimized to the particular character of the problems generated by timed automata.

There are several cases of successful application of model checking for real-time systems using timed automata, for example [10]. However, there are few examples where real-time behavior is combined with fault tolerance. For example, a real-time startup protocol is verified in [15], but the analysis excludes the complex fault scenarios considered in the discrete-time analysis of [21]. We conjecture that the timed automata approach is unable to handle the massive case analysis required by the large number of different fault scenarios; conversely, the treatment using discrete time can handle the case analysis, but the treatment of time is less realistic. We would like to be able to

handle both dimensions—continuous time and faults—simultaneously.

4 Infinite Bounded Model Checking and Real-Time Systems

A bounded model checker for finite state systems works by translating the system description and property specification into a propositional SAT problem. The propositional translation encompasses both the decision and control structure of the system description (i.e., the case analysis that needs to be performed) and the operations performed on its state variables. For example, to perform the operation $x + y$, the translator will generate the propositional formulas that describe a binary ripple-carry adder (in effect, the translator compiles the system description into a boolean circuit). Suppose, instead, that we kept $+$ as a mathematical operation in some decidable theory (e.g., linear arithmetic) and somehow made the SAT solver work in combination with a decision procedure for the mathematical theory. There would then be no need to restrict the state variables x and y to finite integer ranges (i.e., fixed-width bitvectors): they could instead range over the reals.

The combination just hypothesized of a SAT solver with decision procedures is known as an SMT (Satisfiability Modulo Theories) solver, and the construction of such solvers is a very active area of research. There is an annual competition for SMT solvers [2] and their performance is already impressive and is improving all the time. A bounded model checker that generates formulas for an SMT solver is called an *infinite bounded model checker* (i.e., a bounded model checker for infinite-state systems) [6]; just like an ordinary bounded model checker, it can be used for refutation and, via k-induction, for verification [7]. The SAL tool suite provides an infinite bounded model checker that is capable of k-induction.

If we could pose analysis of real-time systems as infinite bounded model checking problems, we might be able to solve some challenging problems: the SAT solver would take care of the case analysis while the decision procedures handle the properties of continuous time. A suitable modeling approach is provided by the *timeout automata* of Dutertre and Sorea mentioned earlier. They present some simple real-time systems and show how these can be represented using timeout automata and analyzed using the infinite bounded model checker of SAL [9]. The analysis is not so fully automated as in timed automata: the user must generally pose and verify a series of lemmas that are combined to verify the property of interest, and that property must be formulated in a way that is appropriate for k-induction. However, the development of suitable lemmas and formulations is highly systematic (using *disjunctive invariants* [20]), and is assisted by the counterexamples produced for inadequate formulations. The method yields very good performance; for example, it can verify Fischer's real-time mutual exclusion protocol (a simple benchmark in this area) with more than 50 processes, whereas most tools for timed automata cannot get into double digits.

This approach has been applied and extended by others. Pike [17] shows how the complexity of the verification problem can be reduced by using *synchronous* timeout automata, and by making the clock value implicit. With Johnson, he uses this method to verify a reintegration protocol [19]. The decision procedures of an SMT solver allow uninterpreted constants, thus protocols can be verified with respect to parameterized quantities: instead of fixed delays such as 2 and 5 seconds, we can verify a protocol with respect to delays of `Tmin` and `Tmax` and some suitable relation between these (e.g., $2 \times$ `Tmin` $<$ `Tmax`). Pike and Brown use this capability to verify parameterized versions of the Biphase Mark and 8N1 decoders [18].

Timeout automata must be extended when components interact through the exchange of messages or by synchronizing on actions, as well as by the passage of time. Dutertre and Sorea refer to the extension as *calendar automata*, after the event calendars that are used in discrete event simulation systems. Using this approach, they are able to verify real-time properties of a simplified, but still fault-tolerant version of the TTA startup protocol [8].

5. Prospects

I am very optimistic about the prospects for analysis of embedded real-time systems using timeout and calendar automata, SMT solvers, and k-induction. SMT solvers are a general-purpose technology that already delivers high performance and is undergoing rapid development. The SAT solving component of an SMT solver is able to cope with large amounts of case analysis; thus fault tolerance and other sources of "case explosion" can be handled satisfactorily. Their decision procedures for linear arithmetic allow SMT solvers to handle continuous time, and other numerical quantities. Additional decision procedures (e.g., for queues) will allow automated verification for other kinds of infinite state systems. I look forward to the transfer of this technology from research laboratories into industrial practice in the near future.

6. Acknowledgments

The techniques and tools outlined here are the work of my colleagues Leonardo de Moura, Bruno Dutertre, Harald Rueß, Maria Sorea, and Natarajan Shankar.

7. References

[1] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235,

25 Apr. 1994.

[2] C. Barrett, L. de Moura, and A. Stump. SMT-COMP: Satisfiability modulo theories competition. In K. Etessami and S. K. Rajamani, editors, *Computer-Aided Verification, CAV '2005*, volume 3576 of *Lecture Notes in Computer Science*, pages 20–23, Edinburgh, Scotland, July 2005. Springer-Verlag.

[3] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A model-checking tool for real-time systems. In A. J. Hu and M. Y. Vardi, editors, *Proc. 10th International Conference on Computer Aided Verification, Vancouver, Canada*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550. Springer-Verlag, 1998. Kronos home page: http://www-verimag.imag.fr/TEMPORISE/kronos/.

[4] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, pages 495–499, Trento, Italy, July 1999. Springer-Verlag. NuSMV home page: http://nusmv.irst.itc.it/.

[5] L. de Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. In R. Alur and D. Peled, editors, *Computer-Aided Verification, CAV '2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag. SAL home page: http://sal.csl.sri.com/.

[6] L. de Moura, H. Rueß, and M. Sorea. Lazy theorem proving for bounded model checking over infinite domains. In A. Voronkov, editor, *18th International Conference on Automated Deduction (CADE)*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455, Copenhagen, Denmark, July 2002. Springer-Verlag.

[7] L. de Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. In W. A. Hunt, Jr. and F. Somenzi, editors, *Computer-Aided Verification, CAV '2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 14–26, Boulder, CO, July 2003. Springer-Verlag.

[8] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 3253 of *Lecture Notes in Computer Science*, Grenoble, France, Sept. 2004. Springer-Verlag.

[9] B. Dutertre and M. Sorea. Timed systems in SAL. Technical Report SRI-SDL-04-03, Computer Science Laboratory, SRI International, Menlo Park, CA, July 2004.

[10] K. Havelund, K. G. Larsen, K. Lund, and A. Skou. Formal modelling and analysis of an audio/video protocol: An industrial case study using uppaal. In *Real Time Systems Symposium*, pages 2–13, San Francisco, CA, Dec. 1997. IEEE Computer Society.

[11] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, May 2003. BLAST home page: http://embedded.eecs.berkeley.edu/blast/.

[12] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003. SPIN home page: http://spinroot.com/.

[13] L. Lamport. Real-time model checking is really simple. In D. Borrione and W. Paul, editors, *Correct Hardware Design and Verification Methods: 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175, Saarbrücken, Germany, Oct. 2005. Springer-Verlag.

[14] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997. Uppaal home page: http://www.uppaal.com.

[15] H. Lönn and P. Pettersson. Formal verification of a TDMA protocol start-up mechanism. In *Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS '97)*, pages 235–242, Taipei, Taiwan, Dec. 1997. IEEE Computer Society.

[16] S. P. Miller, A. C. Tribble, and M. P. E. Heimdahl. Proving the shalls. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *International Symposium of Formal Methods Europe, FME 2003*, volume 2805 of *Lecture Notes in Computer Science*, pages 75–93, Pisa, Italy, Mar. 2001. Springer-Verlag.

[17] L. Pike. Real-time system verification by k-induction. NASA Technical Memorandum TM-2005-213751, NASA Langley Research Center, Hampton, VA, May 2005.

[18] L. Pike and G. M. Brown. Easy parameterized verification of biphase mark and 8N1 decoders. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '06)*, Lecture Notes in Computer Science, Vienna, Austria, Apr. 2006. Springer-Verlag. To appear.

[19] L. Pike and S. D. Johnson. The formal verification of a reintegration protocol. In *EMSOFT 2005:*

Proceedings of the Fifth ACM Workshop on Embedded Software, pages 286–289, Jersey City, NJ, 2005. Association for Computing Machinery.

[20] J. Rushby. Verification diagrams revisited: Disjunctive invariants for easy verification. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification, CAV '2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 508–520, Chicago, IL, July 2000. Springer-Verlag.

[21] W. Steiner, J. Rushby, M. Sorea, and H. Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *The International Conference on Dependable Systems and Networks*, pages 189–198, Florence, Italy, June 2004. IEEE Computer Society.