Formal Verification of Real-Time Embedded Software for Multicore Platforms

Sagar Chaki Arie Gurfinkel
Carnegie Mellon Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA, USA
{*chaki,arie*}*@sei.cmu.edu*

***Abstract*—Real-time embedded software (RTES) plays an increasingly critical role in all aspects of our lives. Ensuring that RTES behave in a predictable, safe and secure manner is an open challenge. The emergence of multicore hardware has introduced an additional level of complexity to this arena. In this paper, we take the position that formal verification is a very promising approach to find concurrency-related problems in multicore RTES. We argue that multicore RTES present unique domain-specific restrictions (and new challenge problems) that can be leveraged (and targeted) by formal verification to yield solutions that are precise, scalable, automated, and applicable to source code. We also believe that this effort will increase synergy between formal verification and real-time scheduling.**

***Keywords*-formal verification; multicore; schedulability**

I. Introduction

Real-time embedded software (RTES) permeate our technology-driven existence, controlling a wide variety of systems ranging from nuclear power plants and energy grids to cars and cell phones. Ensuring that RTES operate correctly is therefore of paramount important to the preservation of our modern way of life.

Despite a wide body of research and development effort, ensuring the safe and secure operation of RTES remains an open challenge. Instances of RTES failure, causing varying degrees of damage and destruction, continue with unfailing regularity. The technological and economic compulsion for migrating to multicore platforms further exacerbates concurrency-related issues for RTES. Clearly, the status quo is unsatisfactory, and new insights and techniques are needed in order to advance the state-of-the art in solving this crucial problem.

In this paper, we argue for a sustained effort in applying formal verification in a precise and targeted manner to find errors in RTES deployed on multicore hardware. In particular, we focus on concurrency-related bugs, such as race conditions and deadlocks, that are notorious to detect and eliminate via traditional validation methods like testing. We argue that multicore RTES present unique domain-specific restrictions and novel challenge problems. Further, formal verification that leverages these restrictions, and targets these challenge problems, will yield precise, scalable, and automated solutions that are applicable to source code. Finally, this effort will strengthen the interplay between real-time scheduling and formal verification.

We structure our position in the following steps: (i) background, (ii) appropriateness of formal verification; (iii) existing approaches; (iv) targeted formal verification; and (v) conclusion.

II. Background

Ensuring the correct behavior of programs is a foundational challenge in the computational sciences. One answer to this challenge is formal verification. The main focus of formal verification is ensuring the "functional" correctness of programs – e.g., that a program properly sorts a list of numbers, does not deadlock, interacts with another program via a specific sequence of actions, etc.

Research in formal verification spans several decades, and has made tremendous progress in terms of the degree of automation and applicability to realistic programs. For example, model checking [1] is a fully automated approach to verifying temporal logic specifications over finite Kripke structure models. More recently, the SLAM project [2] has pioneered the application of abstraction, model checking, and refinement techniques to enable automated verification of C programs. In addition, there have been several instances of applying abstract interpretation [3] to the static analysis of industrial software. We believe that formal software verification – by which we mean the gamut of formal techniques for analyzing software statically – is uniquely suited and sufficiently mature for reasoning about multicore RTES.

III. Appropriateness of Formal Verification

The road to practical adoption of formal verification has been long and hard. In theory, techniques like model checking and abstract interpretation are "push-button". In practice, however, they are often implemented in tools that must be used by an expert in order to yield useful results. This make them expensive, and less desirable than cheaper alternatives like testing. We believe, however, that several factors render formal verification to be the superior alternative in the context of finding concurrency-related bugs in RTES:

1) **Cost of failure.** RTES are often deployed in safety-critical situations where the cost of failure is very high or catastrophic. A typical example is a nuclear power plant, a medical device, or an airplane where a failure could lead to loss of human lives. Other situations,

where failures cause high economic damages, are cell phone towers and electric grids. High cost of failures justify the non-trivial up-front cost of applying formal verification.

2) **Non-determinism.** Concurrency-related bugs are extremely difficult to unearth via conventional code review and testing. There are known cases where bugs have remained uncovered despite years of testing. The main reason behind this is the large number of execution paths that a concurrent program is able to exercise due to non-deterministic choices between different thread interleavings. Even the best testing efforts cover only a small fraction of a program's statespace. Formal verification is exhaustive, and covers all possible program executions and thread interleavings.
3) **Multicore Platforms.** The increased concurrency (and indeed the emergence of real-concurrency where multiple threads are able to execute simultaneously) in multicore platforms amplifies the non-determinism of software, and the relative advantage of formal verification over non-exhaustive methods. The migration to multicore platforms is driven by technological and economic incentives that are unlikely to be reversed.
4) **Certification.** Finally, software verification is an important aid in certification. For example, the DO-178C standard explicitly requires the application of formal verification. Other standards, such as the ECSS-E-ST-40C [4], suggest formal verification when alternatives (such as testing) are arguably inadequate. Based on the points above, we believe that such an argument is indeed plausible in the case of multicore RTES.

IV. Existing Approaches

It is worth recalling that our goal is to develop analysis tools that target concurrency-related issues and are precise, scalable, automated, and applicable to source code. We believe that existing formal verification tools (based on static analysis, software model checking, or a combination of the two) fall short on one or more of these accounts.

There is a large body of work on formal verification of models spanning several decades. Of special relevance to us is the verification of timed automata [5] and hybrid automata [6], timed process calculi – such as RTSL [7], ACSR [8] and PARS [9] – and timed Petri nets [10]. They represent foundational ideas and results that guides the verification of multicore RTES. However the focus of this paper is the analysis and verification of source code, which is not only complementary to that of models, but brings in its own set of challenges as well.

The problems in analyzing source code have been the main focus of commercial static static analysis tools (such as Coverity, Klocwork, Grammatech and Fortify). These tools are targeted toward finding sequential problems (such as `NULL` pointer dereferences and buffer overflows) over a large corpus of code. They are automated and scalable, but challenged by a lack of precision (i.e., many false warnings). A main reason behind their imprecision is that they strive for generality in terms of the programs they are able to handle. This means that they are only able to assume – and thus leverage – a limited set of domain-specific restrictions.

In the context of sequential software, there has been several success stories of applying formal verification to develop analysis that is precise, scalable, automated, and applicable to source code. Two notable examples are the SLAM [2] and the ASTREE [11] projects. We believe that the success of both projects stems in large part on their focus on specific problems, and their use of domain-specific restrictions, which enabled them to develop and refine very specific solutions.

For example, the SLAM project focuses on ensuring that Windows device drivers interact with the kernel only in prescribed ways. These properties are expressible as finite state machines and verifiable over Boolean abstractions of the source code. These restrictions enable SLAM to effectively use predicate abstraction and control-flow reachability over Boolean programs as the underlying verification engine.

The ASTREE project focuses on detecting numeric errors and overflows in avionics software. The restriction to specific programs and properties enables the researchers to develop special-purpose abstract domains, such as ellipsoid [11], that are very effective for computing the invariants needed to prove (or disprove) the target properties.

In the same spirit, we believe that novel approaches that target specific concurrency-related problems, and leverage domain-specific restrictions, must be developed for multicore RTES.

V. Targeted Formal Verification

The development of targeted formal verification tools for multicore RTES is aided by unique restrictions afforded by this domain:

1) RTES have deterministic scheduling, which restricts the amount of concurrency and possible interleaving between threads. Leveraging this restriction provides a way to ameliorate the statespace explosion problem, a major obstacle to scaling formal verification.
2) The scheduling in RTES is governed by a precise priority-based mechanism. In the special case of fixed priority scheduling, thread-interleaving is even more restricted. For every thread T, the set of other threads that might preempt T is statically known. In more specific cases, e.g., for RMA-scheduled [12] RTES, even the number of times a thread may preempt another is bounded and known statically. These restrictions offer further opportunities for improving scalability and automation.
3) RTES involve restricted use of complex programming constructs. In many cases, language features like dy-

namic memory allocation and recursion are disallowed or severely restricted. While these restrictions are driven primarily to make the runtime behavior of the system more predictable, they also enable the automated extraction of precise models for the purposes of formal verification.

4) Multicore RTES also open up new challenge problems for formal verification. One source of problems is the emergence of real concurrency. In a multicore platform, two threads are able to run simultaneously on different cores, and thus access shared resources (like memory and cache) leading to problems like race conditions, deadlocks, and bus overload. Techniques for ensuring mutual exclusion, such as the priority ceiling protocol, break down in a multicore environment. Another source of problems is the added dimension of cores in terms of resource allocation. Allocating threads to cores (statically or dynamically) so as to optimize any desired utility measure is a new challenge.

We believe that these restrictions and challenge problems will aid in the development of targeted and effective formal verification for multicore RTES. At the same time, we believe that this endeavor will further strengthen the synergy between formal verification and real-time scheduling.

VI. Conclusion

Ensuring the predictability, safety, and security of RTES remains a fundamental and open challenge. The emergence of multicore hardware only increases the complexity of the problem. However, multicore RTES present unique domain-specific restrictions and new challenge problems. We argue that these restrictions and challenge problems aid the potential of formal verification to lead to precise, scalable, and automated techniques for finding concurrency-related issues in source code. Additionally, we believe that research in this direction effort will increase synergy between formal verification and real-time scheduling.

Acknowledgment

Several people have participated in discussions that led to the ideas and arguments in this paper. In particular, we would like to thank Russell Kegley and Ben Watson from Lockheed Martin, USA, and Mark Klein from the SEI.

References

[1] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA: MIT Press, 2000.

[2] T. Ball and S. K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," in *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN '01)*, ser. Lecture Notes in Computer Science, M. B. Dwyer, Ed., vol. 2057. Toronto, Canada, May 19–20, 2001. New York, NY: Springer-Verlag, May 2001, pp. 103–122.

[3] P. Cousot and R. Cousot, "Abstract Interpretation: A unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints," in *Proceedings of the 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '77)*. Los Angeles: Association for Computing Machinery, January 1977, pp. 238–252.

[4] D. Lesens, "Using Static Analysis in Space: Why Doing so?" in *Proceedings of the 17th International Static Analysis Symposium (SAS '10)*, ser. Lecture Notes in Computer Science, R. Cousot and M. Martel, Eds., vol. 6337. Perpignan, France, September 14–16, 2010. New York, NY: Springer-Verlag, September 2010, pp. 51–70.

[5] R. Alur and D. L. Dill, "A Theory of Timed Automata," *Theoretical Computer Science (TCS)*, vol. 126, no. 2, pp. 183–235, April 1994.

[6] T. A. Henzinger, "The Theory of Hybrid Automata," in *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*. New Brunswick, NJ, July 27–30, 1996. Los Alamitos, CA: IEEE Computer Society, July 1996, pp. 278–292.

[7] A. N. Fredette and R. Cleaveland, "RTSL: a language for real-time schedulability analysis," in *Proceedings of the Real-Time Systems Symposium (RTSS '93)*. Raleigh-Durham, NC, USA: IEEE Computer Society, December 1993, pp. 274–283.

[8] P. Brémond-Grégoire, I. Lee, and R. Gerber, "ACSR: An Algebra of Communicating Shared Resources with Dense Time and Priorities," in *Proceedings of the 4th International Conference on Concurrency Theory (CONCUR '93)*, ser. Lecture Notes in Computer Science, E. Best, Ed., vol. 715. Hildesheim, Germany, August 23–26, 1993. New York, NY: Springer-Verlag, August 1993, pp. 417–431.

[9] M. R. Mousavi, M. A. Reniers, T. Basten, and M. R. V. Chaudron, "PARS: A Process Algebra with Resources and Schedulers," in *Proceedings of the 1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS '03)*, ser. Lecture Notes in Computer Science, K. G. Larsen and P. Niebert, Eds., vol. 2791. Marseille, France, September 6–7. 2003.: Springer-Verlag, September 2003, pp. 134–150.

[10] A. Cerone and A. Maggiolo-Schettini, "Time-Based Expressivity of Time Petri Nets for System Specification," *Theoretical Computer Science (TCS)*, vol. 216, no. 1-2, pp. 1–53, December 1999.

[11] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "A Static Analyzer for Large Safety-Critical Software," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*. San Diego, CA, June 9–11, 2003. New York, NY: Association for Computing Machinery, June 2003, pp. 196–207.

[12] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, January 1973.