Formal software verification for the migration of embedded code from single- to multicore systems

Thorsten Ehlers[a], Dirk Nowotka[a,*], Philipp Sieweck[a] and Johannes Traub[b]

[a]Dependable Systems Group
Department of Computer Science
Kiel University
{the, dn, psi}@informatik.uni-kiel.de
[b]Powertrain Electronics, Daimler AG
johannes.traub@daimler.com

Abstract: The introduction of multicore hardware to the field of embedded and safety-critical systems implies great challenges. An important issue in this context is the migration of legacy code to multicore systems. Starting from the field of formal verification, we aim to improve the usability of our methods for software engineers. In this paper, we present approaches to support the migration process, mainly in the domain of safety-critical, embedded systems. The main contribution is a verification process which is inspired by the waterfall model. Especially, backtracking is considered carefully.

This work is part of the ARAMiS[1] project.

1 Introduction

Multicore systems are becoming more and more common, also in the domain of safety-critical embedded systems. Multiple challenges arise from this, ranging from the design of appropriate hardware to the development of software engineering techniques. One of these challenges is the avoidance of race conditions. Race conditions are accesses of at least two tasks to the same memory location, with at least one task writing. This kind of bugs can cause inconsistent data states, and thus unpredictable system behaviour. Although race conditions are not restricted to multicore systems, the probability of their occurrence is much higher on systems using several CPUs.

As the software used in the automotive domain has become more and more complex - nowadays, it has reached a magnitude of more than 10,000,000 lines of code[BKPS07] in one car - tool support is urgently needed. Our focus lies on the formal verification of software, especially in the automotive domain. In [NT12, NT13] we presented MEMICS. This tool supports the verification of OSEK/AUTOSAR-code [ose, Con], as it is used in

*This work has been supported by the BMBF grant 01IS110355.

[1]Automotive, Railway and Avionics Multicore Systems `http://www.projekt-aramis.de/`

automotive systems. The main focus lies on the detection of race conditions. Additionally, other runtime errors like e.g. “DivisionByZero” or “NullDereference” can be detected.

Nevertheless, there are more requirements than “only” formal correctness. In order to provide remarkable impacts in practical issues, they need to be incorporated into the software engineering processes. In this paper, we show how to involve MEMICS in the process of migrating legacy software from single- to multicore systems. Therefore, we suggest to organize the migration process according to a waterfall model. This model contains the possibility of backtracking in well-defined cases. Furthermore, we suggest to combine testing and model checking. This approach can be used to check whether the behaviour of a systems remains deterministic with respect to scheduling decisions after migrating to a multicore environment.

2 Technical Background

Operating systems compatible to OSEK/AUTOSAR implement tasks and interrupt service routines, which have a fixed priority, and are statically assigned to a core. In a single-core setting, these strict priorities allow only a small number of interleavings. When migrating to a multicore system, there are basically two possibilities. The software engineer can chose one core for every task, or split the functionality of a task into two or more new tasks, and distribute them to different cores. Obviously, the first alternative is the easier one, whereas the second possibility can provide more benefits. Nevertheless, both approaches significantly increase the number of possible interleavings between tasks. Hence, safety properties of the software have to be checked carefully after the migration. If these checks show possible race conditions, they can be avoided by changing the distribution of tasks to cores, or otherwise by adding synchronization via semaphors.

3 Related Work

In the last decade, formal software verification became usable for practical applications. Microsoft reports several cases [BCLR04]. One of them is SLAM [BLR], which is used to check whether device drivers use the Windows Driver Model (WDM) according to its specifications. Only drivers that pass this check become digitally signed, indicating that their usage will not compromise the stability of the system. Furthermore, Microsoft uses VCC [CDH$^+$] in order to formally prove the absence of errors in their hypervisor. As this is a more complex task, it requires the programmers to annotate their code with formal pre- and postconditions. In the avionic domain, formal methods are seen as a supplement of testing [MLD$^+$13]. The reason is not only the completeness of their results, but also the possibility to decrease the costs for testing [CKM12].

Tools like Bauhaus [RVP06] and Polyspace [pol] are able to find runtime-errors in OSEK/-AUTOSAR code. Yet, they work with a large overapproximation of the program behaviour. This yields a lot of false-positive results, making it difficult to use their results

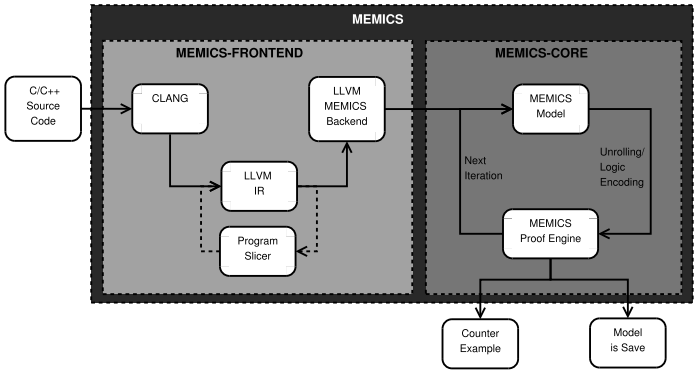

Figure 1: Using the LLVM-IR to generate a MEMICS model

in an iterative migration approach. The high number of false-positives forces the software engineers to perform a manual code review for each of them, taking lots of time. As part of the ARAMiS project, the model checker MEMICS was developed to deal with this problem [NT12, NT13]. It can be used to find software errors itself, or to check error candidates given by tools like Bauhaus or Polyspace.

ESBMC [RBCN12] or Threader [GPR11] are able to handle multithreaded code. They have two major drawbacks: They do not support scheduling according to OSEK and AUTOSAR, and their memory model is not precise enough to significantly reduce the number of false-positive errors reported.

4 MEMICS

Mainly designed to find race conditions, MEMICS is also able to find other runtime errors like DivByZero, NullDereference, and others. It supports priority-based scheduling as used in OSEK/AUTOSAR-based operation systems. The input is either C/C++-code or LLVM IR code [LA04, Lat], which makes it compatible to all languages supported by Clang [Fan10]. This input is transformed to an internal model which is based on the MIPS assembly language, see figure 1. There are basically two modi operandi. Either MEMICS is used to verify an error candidate, given e.g. by Polyspace, or it is used to find and verify error candidates itself. In the first case, MEMICS tries to find a program trace to the error location. The actions on such traces are translated into a first order logic formula, which is solved with regard to a flat memory model that allows tracking arbitrary data structures including pointers and function pointers. Therefore, the question whether a trace is feasible in the original program or not can be answered with high confidence, which is a big advantage in comparison to other tools like Threader [GPR11] or ESBMC [RBCN12]. If a feasible trace is found, it is reported as an error. Otherwise, other traces are searched until either a feasible one has been found, or safety is proven w.r.t. this error candidate. Without given error candidates, MEMICS can unroll the program itself,

following only feasible traces and looking for runtime errors. Although this works for benchmarks, this approach suffers from state space explosion, thus it should be used only if the preprocessing fails, or to compute pre/postconditions for small parts of a program, as suggested in [CKM12].

5 Supporting the Migration Process

We show how different tools can be used to combine their strengths, and find some of the hardest bugs faced when using parallel hardware: race conditions.

5.1 Model Checking

As implied before, there exist tools like Polyspace and Bauhaus which are able to deal with OSEK/AUTOSAR code. Though, they perform an abstract interpretation, yielding an overapproximation of the set of errors. Checking each of these error candidates manually is far too time consuming, and hence too expensive. We use MEMICS in order to find a trace to an error candidate. If there exists a feasible trace, this indicates the existence of a real error. Otherwise, we have found a false positive which can be removed. On the one hand, this massively shrinks the search space and hence accelerates the model checking. On the other hand, we remove false positive results, which makes the results usable for software engineers.

5.2 Combining Testing and Model Checking

In this section, we describe how to combine model checking with testing. We may assume that there exist test cases for the software under consideration, which run in the single-core setting without faults. Furthermore, we assume that model checking as described in section 5.1 has proven the absence of race conditions like Lost Updates. Such race conditions are only a subclass of problems that may occur when dealing with parallel programs [NM92]. Another type of errors which are hard to detect are data races, i.e. situations in which the behaviour of the program depends on the scheduling. Please not that data races may occur even if every access to global memory is synchronized. Although such errors can be found by model checking, they are practically intractable for large programs. Hence, we propose to use test cases from the single-core setting. For each of the test cases we perform model checking, restricting the behaviour of the program according to the test inputs. If we can prove that the results are unique, this implies that they do not depend on the scheduling. Otherwise, we can provide traces which lead to different results.

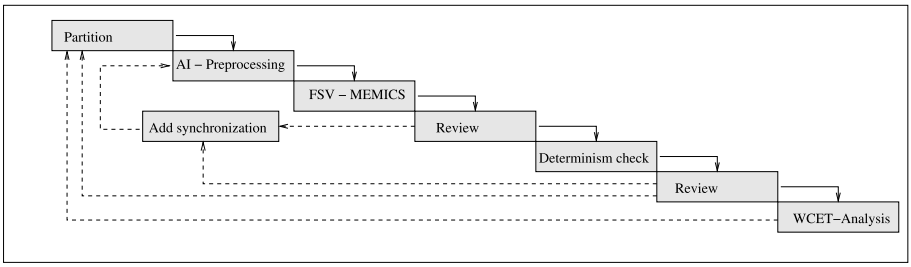

Figure 2: Iterative workflow

5.3 Combined Workflow

Putting this together, we derive an iterative workflow for the migration process, as shown in figure 2. Firstly, a software engineer decides how to distribute the tasks onto the cores, or splits tasks into subtasks, as depicted in section 2. Tools based on abstract interpretation like Polyspace or Bauhaus are used to find error candidates, which are checked using more precise tools, e.g. MEMICS. As we do not require annotations in the code, it is possible to retain false-positive results. This may happen e.g. if interfaces have restrictions that are not visible in the code itself. Thus, the results must be checked by a software engineer. If there are possible race conditions, either another partition can be chosen, or they can be fixed by introducing semaphors.

In the next step, test cases from the single threaded setting can be used to check whether the program is deterministic or not. Again, the importance of nondeterministic behaviour must be evaluated. If no non-determinism occurs, or if it is irrelevant, the process continues with the analysis of the worst case execution time or worst case reaction time. Otherwise, the overall process is iterated, beginning with a new partition.

6 Conclusion & Future Work

We proposed an iterative process to migrate OSEK/AUTOSAR-code from single- to multicore. Furthermore, we suggested how to support this by using testing, abstract interpretation and model checking. Given an appropriate exchange format, parts of this process can be run automatically. Nevertheless, this process cannot be fully automated. Hence, we hope to offer helpful support. This requires sufficiently precise results, as too many false positive error reports slow down the process.

Further research will be necessary to improve the efficiency of this process. This does not only concern better running times and increased precision of the verification processes, but also a deeper understanding of the information required by the software engineer, and his interaction with the tool box.

References

[BCLR04] T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In EerkeA. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods*, volume 2999 of *LNCS*, pages 1–20. Springer Berlin Heidelberg, 2004.

[BKPS07] M. Broy, I.H. Kruger, A. Pretschner, and C. Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95(2):356–373, 2007.

[BLR] T. Ball, V. Levin, and S. K. Rajamani. A Decade of Software Model Checking with SLAM.

[CDH+] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A practical system for verifying concurrent C. In *IN CONF. THEOREM PROVING IN HIGHER ORDER LOGICS (TPHOLS), VOLUME 5674 OF LNCS*.

[CKM12] C. Comar, J. Kanig, and Y. Moy. Integration von Formaler Verifikation und Test. In *Automotive - Safety & Security*, pages 133–148, 2012.

[Con] Autosar Consortium. *AUTOSAR - Specification of Operating System*. http://autosar.org.

[Fan10] D. Fandrey. Clang/LLVM Maturity Report. June 2010. *See* `http://www.iwi.hs-karlsruhe.de`.

[GPR11] A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A Constraint-Based Verifier for Multi-threaded Programs. In *CAV*, pages 412–417, 2011.

[LA04] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[Lat] C. Lattner. *LLVM Language Reference Manual*. http://llvm.org/docs/LangRef.html.

[MLD+13] Y. Moy, E. Ledinot, H. Delseny, V. Wiels, and B. Monate. Testing or Formal Verification: DO-178C Alternatives and Industrial Experience. *IEEE Software*, 30(3):50–57, 2013.

[NM92] R. H. B. Netzer and B. P. Miller. What are race conditions?: Some issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.

[NT12] D. Nowotka and J. Traub. MEMICS - Memory Interval Constraint Solving of (concurrent) Machine Code. In *Automotive - Safety & Security 2012*, Lecture Notes in Informatics, pages 69–84, Bonn, 2012. Gesellschaft für Informatik.

[NT13] D. Nowotka and J. Traub. Formal Verification of Concurrent Embedded Software. In *IESS*, pages 218–227, 2013.

[ose] *OSEK*. http://portal.osek-vdx.org.

[pol] *Polyspace*. http://www.mathworks.com/products/polyspace.

[RBCN12] H. Rocha, R. Barreto, L. Cordeiro, and A. Neto. Understanding Programming Bugs in ANSI-C Software Using Bounded Model Checking Counter-Examples. In John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne, editors, *Integrated Formal Methods*, volume 7321 of *LNCS*, pages 128–142. Springer Berlin Heidelberg, 2012.

[RVP06] A. Raza, G. Vogel, and E. Plödereder. Bauhaus - A Tool Suite for Program Analysis and Reverse Engineering. In *Reliable Software Technologies - Ada-Europe 2006*, volume 4006 of *LNCS*, pages 71–82. Springer Berlin Heidelberg, 2006.