Formal Verification of Concurrent Embedded Software

Dirk Nowotka[1]* and Johannes Traub[2]

[1]Department of Computer Science, Kiel University,
dn@informatik.uni-kiel.de
[2]E/E- and Software-Technologies, Daimler AG,
johannes.traub@daimler.com

Abstract: With the introduction of multicore hardware to embedded systems their vulnerability to race conditions has been drastically increased. Therefore, sufficient methods and techniques have to be developed in order to identify this kind of runtime errors. In this paper, we demonstrate an approach employing a formal technique in the verification process. We use MEMICS, which is a specialized constraint solver able to identify general runtime errors as well as race conditions. We show how this tool can be embedded into an existing software analysis tool chain. In particular, we describe the process of deriving the formal input model for the solver from C code. The advantage of using constraint solving techniques is that we can offer an entire trace leading to a race condition. The ongoing development of MEMICS is part of our work inside the ARAMiS project.

1 Introduction

One of the main goals of the ARAMiS project — "Automotive, Railway and Avionics Multicore Systems" — [BS] is to enhance on safety issues for multicore embedded technologies in vehicles. In terms of embedded systems a safety aspect is the assurance that the software running on them is free of any kind of runtime error, which they may suffer and fault from. Software can suffer from a lot of different runtime errors, like an arithmetic overflow, a division by zero, an index out of bound access, a null dereference, a race conditions and a stack overflow. A detailed list of runtime errors can be found in Table 1 in Section 3. The nastiest of these runtime errors are the race conditions, as they might only occur sporadically and are therefore very hard to detect or trace. With the current introduction of multicore hardware to embedded systems, their vulnerability to race conditions has increased drastically. To get this problem under control new tools and techniques are required.

In [NT12] we introduced the static software analysis tool MEMICS, which is able to detect race conditions as well as common runtime errors in C/C++ source code. Common static analysis tools like Astrée [CCF+05], Polyspace [pol], and Bauhaus [RVP06] are able to analyse large code fragments but do suffer from potential false positives which requires an

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

extensive manual postprocessing of their results. MEMICS is based on constraint solving techniques which eliminate the problem of false positives. However, the complexity of constraint solving algorithms is very high which means that the code fragments MEMICS can analyse are not too large. We believe that a combination of both approaches, approximative and precise techniques, together in one tool chain lead to a significant improvement of the analysis of concurrent code. In this paper we describe how MEMICS fits into a static analysis workflow. Moreover, we give a detailed description of the conversion of C code to the MEMICS input model.

Within the ARAMiS project there are two possible scenarios discussed, in which the MEMICS tool can be used to provide safety:

1. Migration to multicore hardware, and
2. Development for multicore hardware.

Both scenarios have the same origin. Lets assume an OEM has decided to replace the hardware of one of its ECU's — e.g. due to new features, optimized power consumption, or need for more performance — and the replacement hardware contains a multicore CPU, whereas the old one was a singlecore system. In this case the OEM has to decide, either to port the current software version to match all the new features of the multicore hardware or to entirely restart and build a new software from scratch. Still, no matter which of the two choices are picked, it is clear that the possibility of potential races has increased with the new hardware. Therefore MEMICS can be used to determine and eliminate races during the development process.

The MEMICS tool is described in Section 2, where we mainly focus on the MEMICS frontend. Section 3 provides current results of the MEMICS tool. In Section 4 we discuss the role and possible use cases of MEMICS inside the ARAMiS project. Finally we conclude our paper in Section 5 and give a perspective for the future.

2 The MEMICS Tool

In [NT12] we introduced MEMICS, while mainly focusing on the overall tool and the proof engine. The current paper is dedicated to the preprocessing engine in MEMICS, the MEMICS frontend, which is introduced in detail in Section 2.1. Figure 1 shows the architectural overview of MEMICS. The input to MEMICS is C/C++ source code, which in the first step is preprocessed in the MEMICS frontend and results in the MEMICS model. This model is then passed to the core of MEMICS, the actual proof engine, which checks if the model suffers from any runtime error.

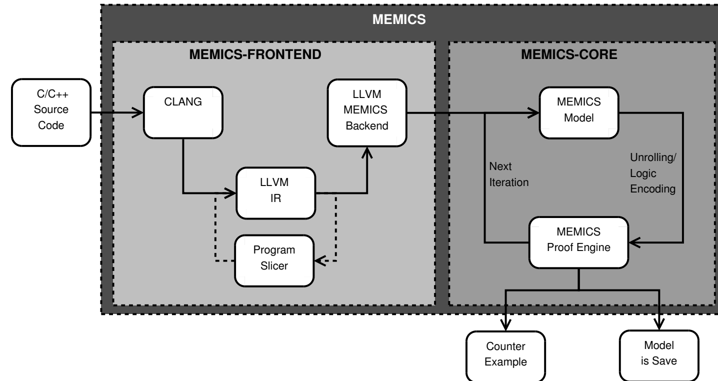

Figure 1: An Overview of the MEMICS Architecture.

2.1 The MEMICS Frontend

The MEMICS frontend describes the interface between the source input, which is C/C++ source code, and the MEMICS model. We decided to use the Low Level Virtual Machine (LLVM) [LA04] infrastructure as a base for this frontend, as it is currently one the most advanced and user friendly compiler framework. In the first step, the C/C++ sources get compiled using the CLANG [Fan10] compiler and are linked together using llvm-ld. The result is one bitcode file, which resides in the LLVM intermediate representation (IR) [Lat]. The LLVM IR is a combination of the LLVM language, which is based on the MIPS [Swe06] instruction set, and an unlimited set of virtual registers. In order to simplify and reduce the input problem, we can optionally run a Program Slicer [Wei81] directly on the LLVM IR. Due to the fact that this slice must not modify the overall behaviour of the program, we can only apply specific slicing techniques. The IR still features function- and variable-pointers as well as other specific types, which are not straight forward dealable by common verification techniques. So, instead of having to lower all the special features on our own, we decided to take advantage of the LLVM backend, which is generating plain machine code. Therefore, we derived the LLVM MEMICS backend from the MIPS backend and added some minor modifications to the instruction lowering. But instead of printing plain MIPS assembly code, the LLVM MEMICS backend creates the MEMICS intermediate representation, which is introduced in Section 2.2. Every machine instruction can be mapped one-by-one to a MEMICS instruction and every global variable is on the one hand applied to the MEMICS RAM and on the other hand assigned to the model.
Like almost any compiler infrastructure the LLVM MIPS backend supports three different relocation types [Lev99]: dynamic-no-pic, pic and static. Pic is short for “position independent code” and even allows the temporal storage of jump destinations into registers. Both, pic and dynamic-no-pic allow libraries to be fetched dynamically, which results in a smaller linked binary. Whereas in static relocation type all libraries are statically linked into the binary, which is therefore bigger. In the current development state our MEMICS intermediate representation requires absolute jump destinations, which forces us to either

use dynamic-no-pic or static relocation type.

2.2 The MEMICS Intermediate Representation

The MEMICS intermediate representation (IR) or the MEMICS model is based on a combination of a finite state machine definition and the MIPS instruction set. An instruction inside the IR is defined as the 4-tuple:

$$< s_i, c, a, s_i' >, \text{ where:}$$

s_i is the current program counter (PC), c is an optional condition (e.g. in a branch instruction), a is the actual MIPS instruction, and s_i' is the successor PC.

Figure 2 shows a small example of the conversion from C source code to the MEMICS IR. The source code shown in the first box is a simple function, which computes the division of the operands `a` and `b`. Compiling this code using CLANG results in the LLVM IR, which is shown in the second box of the figure. It is observable that the IR itself is already more like a machine language, compared to the actual source code. First of all local memory for the operands is allocated, which is afterwards assigned with the actual values of them. In the next step the values are read from the memory into the two virtual registers `%0` and `%1`. Next the division itself takes place and finally the result is returned. The MEMICS IR, which is shown in the last box of Figure 2, is retrieved from the LLVM IR via the LLVM MEMICS backend. The result is even closer to the MIPS assembly language then the LLVM IR. The actual instruction has been embedded between the current program counter and the following program counter, which are both required in order to properly process the model. First of all in line 1 the local stack pointer gets allocated. In line 2 and 3 the operands - respectively the registers 4 and 5 - are stored in the local memory. Now, the actual division takes part in line 4, where the result is store in register `lo` and the remainder in register `hi`. In the next two instructions the result is assigned to the return value register 2 and the stack pointer gets freed. Finally the function returns to its caller, which is stored in the `ra` (return address) register.

2.3 The MEMICS Core

The MEMICS Core is the actual verification engine of the MEMICS tool, which checks if the MEMICS IR and its underlying C/C++ source code suffers from any runtime error or not. The verification process is based on Bounded Model Checking (BMC) [BCC$^+$03]. Therefore, the MEMICS IR is unrolled step by step into a logic formula in Static Single Assignment (SSA) form [AWZ88, RWZ88] and then passed to the MEMICS Proof Engine. This proof engine is a self developed Interval Constraint Solver (ICS), based on the ideas from HySAT and its successor ISAT [FHT$^+$07]. The main difference between an ICS and common SAT-/SMT-Solvers [MMZ$^+$01, dMB09] - e.g. MiniSAT [ES03], Boolector [BB09], Z3 [dMB08] and many other - is, instead of dealing with fix-point variable deci-

C Code

```
int divide(int a, int b) {
    return (a / b);
}
```

LLVM Intermediate Representation

```
define i32 @divide(i32 %a, i32 %b) nounwind {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32* %a.addr, align 4
  %1 = load i32* %b.addr, align 4
  %div = sdiv i32 %0, %1
  ret i32 %div
}
```

MEMICS Intermediate Representation

```
1: PC = 1 -> malloc(sp_reg' , 8) AND PC' = 2;
2: PC = 2 -> sw(4_reg, (memadr(sp_reg, 4) AND __clk__))
           AND PC' = 3;
3: PC = 3 -> sw(5_reg, (memadr(sp_reg, 0) AND __clk__))
           AND PC' = 4;
5: PC = 4 -> (lo_reg' = 4_reg / 5_reg)
           AND (hi_reg' = 4_reg % 5_reg)
           AND PC' = 5;
6: PC = 5 -> (2_reg'  = lo_reg)  AND PC' = 6;
7: PC = 6 -> free(sp_reg) AND PC' = 7;
8: PC = 7 -> PC' = ra_reg;
```

Figure 2: From C Source Code via the LLVM IR to the MEMICS IR

sions during the internal search procedure, variable ranges are deduced. Since the main purpose of our tool is software verification, it contains many special features regarding the analysis of software. For details on these features please refer to [NT12].

3 Results

In [NT12] we have tested MEMICS on an internal benchmark set, which contains different types of runtime errors, based on errors observed in real life. We used the Common Weakness Enumeration (CWE) [cwe] database to define the base classes for these errors. As the CWE gathers almost any kind of error, which is observable in a computer based environment, we do by far not match all error classes, but only show the most relevant ones for static software analysis. The result of these tests is shown in Table 1, where we have compared MEMICS with two analysis tools, CBMC [CKL04] and LLBMC [SFM10], which

are also operating based on BMC.

Class	Benchmark	CWE-ID	MEMICS	CBMC	LLBMC
Arithmetic	DivByZeroFloat	369	✓	✓	○
	DivByZeroInt	369	✓	✓	✓
	IntOver	190	✓	✓	✓
Memory	DoubleFree	415	✓	✓	✓
	InvalidFree	590	✓	✓	✓
	NullDereference	476	✓	✓	✓
	PointertToStack	465	✓	-	✓
	SizeOfOnPointers	467	✓	-	✓
	UseAfterFree	416	✓	-	✓
Pointer Arithmetic	Scaling	468	✓	-	✓
	Subtraction	469	✓	-	✓
Race Condition	LostUpdate	567[1]	✓	○	○
	MissingSynchronisation	820	✓	○	○
Synchronization	DeadLock	833	✓	○	○
	DoubleLock	667	✓	○	○

Table 1: Results of MEMICS compared to CBMC and LLBMC, where a ✓represents a correct verification result, - a false one and ○ signals that the tool does not support the class of testcases

With this results we have shown that our tool is already able to identify a lot of runtime errors, as well common sequential as difficult concurrent ones.

4 MEMICS and the ARAMiS Multicore Platform

As in the introduction already mentioned the main goal of ARAMiS is to provide a platform for multicore development. This platform should feature a seamless integration of the development tools along the development process. For this purpose one current development process is the creation of a global exchange format. This format should help all tools along the development process to intercommunicate with each other and pass on usable information or already computed results.

The MEMICS tool can intercommunicate and share information with common static analysis tools like Astrée, Polyspace, and others as well as race detection tools like Bauhaus [RVP06] and others. Figure 3 illustrates the information sharing between those tools alongside the ARAMiS exchange format. The main idea behind the combination of these tools is to provide the best overall performance for all of them. Whereas tools like Astrée and Polyspace have the ability to handle large amounts of source code, they are based on abstract interpretation [CC77] and may therefore suffer from imprecision in the results. Bauhaus can also handle a lot of input in terms of source code, but it still suffers from false positives in the results, since it is working based on approximative techniques. On

[1] We did not find a straight forward ID for a lost update, but the example in this entry describes one

the other hand BMC tools like MEMICS are limited due to the state explosion problem, while offering enormous precision. In our case we even provide a direct counterexample leading to an error. In Section 4.1 and 4.2 we describe three different scenarios of possible tool intercommunication.

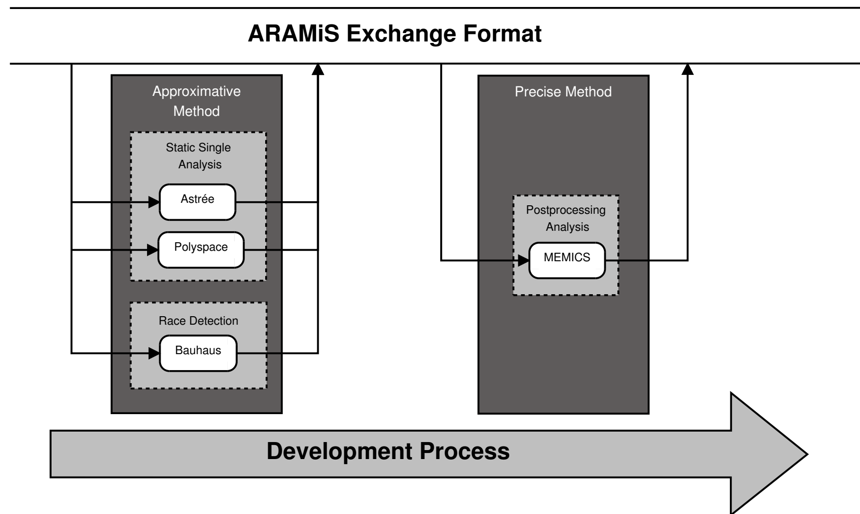

Figure 3: ARAMiS Exchange Format: Intercommunication between Software Analysis Tools

4.1 Combination: MEMICS $\leftrightarrow$ Polyspace

The output of Polyspace is divided in three different groups: the green, orange and red results. A green result states the given property is free of faults, whereas a red one is an actual finding. All of the orange ones are not determinable and must therefore be manually reviewed. One can use MEMICS to check if the error is "real" or not. The definition of the check is acutally quite simple. Let us assume the indeterminable error is a potential division by zero occurring in the example function "divide" of Figure 2. In that case using the definition of the according MEMICS IR from Figure 2, the target-question MEMICS has to determine is:

$$\texttt{PC == 4} \wedge \texttt{5_reg == 0}$$

4.2 Combination: Bauhaus $\leftrightarrow$ MEMICS

In case of the Bauhaus race detector, two different scenarios can be considered. In the first case Bauhaus can just pass its common output as well as the system description - including

the task definitions, their priorities and so on - to MEMICS in order to determine, which of the detected race pairs can really occur in the system. Such a race pair can either be a read operation from task `A` in conflict with a write operation from task `B` on the same shared resource or a write-write conflict between task `A` and `B`. So e.g. for a read/write conflict, given the read access occurs at `PC = x`, the write conflict occurs at `PC = y` and the resource is located at address `z` in the memory, the target-question for MEMICS is:

```
clk(load, z, A, PC = x) > clk(store, z, B, PC = y)
```

In the second case Bauhaus can use MEMICS to gather more information on the scheduling of tasks. With this help Bauhaus can reduce the set of potential race conditions. Let us assume that the initial program counter of task `A` is `PC_taskA = x` and for task `B` `PC_taskB = y`. The target-question for MEMICS, if e.g. the two tasks can start synchronously, is:

```
clk(PC_taskA = x) == clk(PC_taskB = y)
```

The MEMICS tool benefits from the first two scenarios described above, because adding a target-question to input of the MEMICS IR has almost the same impact as Program Slicing. It does not actually reduce the MEMICS IR, but reduces to search space only to the required behaviour, which is shown in Figure 4. This reduction can have a large impact on the overall time MEMICS requires to solve the input problem.

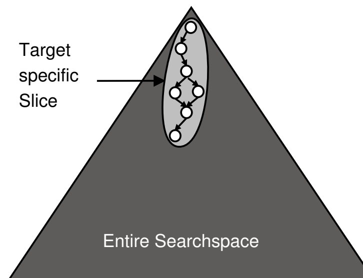

Figure 4: MEMICS IR Slice: Searchspace Reduction to a specific Target

5 Conclusions and Future Work

In this paper we have described, how the software verification tool MEMICS maps C code to its input model. We have shown the advantages of using LLVM and that especially the LLVM Backend is the most suitable solution for our purpose. Moreover, we described the role of MEMICS inside a software analysis tool chain, in particular within the ARAMiS project. This gives our perspective in which cases MEMICS can enhance the development process.

Currently, we are running scalability tests of the MEMICS tool to test the limits of our approach and push those. Another ongoing work is to embed techniques like counterexample guided abstraction refinement (CEGAR) [CGJ+00] in order to improve on MEMICS efficiency. In terms of the ARAMiS project, we will use the exchange format, once it is available, for tying MEMICS into the tool chain. This will help us a lot in case of direct knowledge sharing with other tools like e.g. Bauhaus and Polyspace. The information we can retrieve from these tools is supposed to drastically reduce the size of the input in most cases.

References

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–11, 1988.

[BB09] Robert Brummayer and Armin Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 174–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[BCC+03] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. volume 58 of *Advances in Computers*, pages 117 – 148. Elsevier, 2003.

[BS] Jürgen Becker and Oliver Sander. *Automotive, Railway and Avionics Multicore Systems - ARAMiS*. http://www.projekt-aramis.de/.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[CCF+05] Patrick Cousot, Radhia Cousot, Jerme Feret, Laurent Mauborgne, Antoine Min, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *Programming Languages and Systems, Proceedings of the 14th European Symposium on Programming, volume 3444 of Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CGJ+00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-Guided Abstraction Refinement. In E.Allen Emerson and AravindaPrasad Sistla, editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin Heidelberg, 2000.

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[cwe] *Common Weakness Enumeration*. http://cwe.mitre.org.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[dMB09] Leonardo de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: An Appetizer. In Marcel V. Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications*, volume 5902, chapter 3, pages 23–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[ES03] Niklas En and Niklas Srensson. An Extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

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

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *Journal on Satisfiability, Boolean Modeling and Computation*, 1:209–236, 2007.

[LA04] Chris Lattner and Vikram 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] Chris Lattner. *LLVM Language Reference Manual*. http://llvm.org/docs/LangRef.html.

[Lev99] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[NT12] Dirk Nowotka and Johannes Traub. MEMICS - Memory Interval Constrain Solving of (concurrent) Machine Code. In Erhard Plödereder, Peter Dencker, Herbert Klenk, Hubert B. Keller, and Silke Spitzer, editors, *Automotive - Safety & Security 2012: Sicherheit und Zuverlässigkeit für automobile Informationstechnik*, volume 210 of *Lecture Notes in Informatics*, pages 69 – 83. Springer, 2012.

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

[RVP06] Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus - A Tool Suite for Program Analysis and Reverse Engineering. In *Reliable Software Technologies - Ada-Europe 2006*, volume 4006 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin Heidelberg, 2006.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 12–27, 1988.

[SFM10] Carsten Sinz, Stephan Falke, and Florian Merz. A Precise Memory Model for Low-Level Bounded Model Checking. In *Proceedings of the 5th International Workshop on Systems Software Verification (SSV '10)*, Vancouver, Canada, 2010.

[Swe06] Dominic Sweetman. *See MIPS Run, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[Wei81] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.