MEMICS - Memory Interval Constraint Solving of (concurrent) Machine Code

Dirk Nowotka[1], Johannes Traub[2]

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

Abstract: Runtime errors occurring sporadically in automotive control units are often hard to detect. A common reason for such errors are critical race conditions. The introduction of multicore hardware enables software to be run in parallel, and hence, drastically increases the vulnerability to such errors. Race conditions are difficult to discover by testing or monitoring, only. Hence, a static analysis of code is required to effectively reduce the occurrence of such errors. In this paper we introduce a new Bounded Model Checking tool, which in its core is an Interval Constraint Solver, operating on a machine code based model and is able to handle memory instructions directly. As control units are usually running on task-based operating systems like AUTOSAR or OSEK, our tool features a task model, which is able to handle sequential and concurrent task scheduling.

1 Introduction

Every control unit used in the automotive environment, e.g. driver assistance systems, is running on software systems and it is obvious that the reliability, safety, and security of such systems is of utmost importance. Possible runtime errors software can suffer from are e.g. arithmetic overflows, division by zero, NULL pointer dereferences, race conditions, stack overflows. A detailed list of runtime errors is shown in Section 3 in Table 1. Especially race conditions are hard to detect even in software running on singlecore hardware, however when it comes to multicore hardware it is far more difficult. A common task in the verification of software is the search for such runtime errors. The complexity of this problem is quickly growing with the complexity of the software itself and made much worse when parallel hardware is involved.

In contrast to common static analysis tools like Astrée [CCF$^+$05] and Polyspace [pol], which analyse the source code via Abstract Interpretation [CC77a] and may suffer from potential false positives, this paper is focused on the formal verification of software. One of the techniques used in formal verification of software is Bounded Model Checking (BMC) [BCC$^+$03]. In BMC a model, which in this case is the structural representation of the software, is unrolled up to some fixed depth and passed to a standard proof engine. Such a proof engine is e.g. a SAT-/SMT-Solver, which is used to solve the satisfiability (SAT)

[Coo71] problem or the satisfiability modulo theories (SMT) [dMB09] problem, like in CBMC [CKL04], F-Soft [IYG+04], and LLBMC [SFM10]. The main challenges that we address with the proposed tool are

efficient model unrolling which does directly affect the size of the formula,

internal handling of memory access of the code to be verified, and

implementation of a task model which provides the possibility to check software implementing a task-based operating system like AUTOSAR [Con] and OSEK [ose].

In the following, we elaborate on the aforementioned points and after that conclude the introduction by pointing out where our tool MEMICS advances the state of the art.

1.1 Model Unrolling

In Bounded Model Checking there exist various ways of unrolling the model. The two major approaches of unrolling are: external and internal. In the external variant the model is first unrolled into a logic formula and then passed to an independent proof engine. In each run the engine is started with an entirely new formula. Hence, it cannot reuse assignments already learnt from previous runs. In case of the internal variant the proof engine is directly embedded into the Model Checking tool. This allows for reusing already learnt clauses from previous runs achieved by conflict analysis during the search process. A Model Checker supporting internal unrolling is HySAT [FHT+07].

In [Tra10] we came up with the approach to use HySAT as proof engine in order to take advantage of the Bounded Model Checker and its underlying Interval Constraint Solver. Our approach was working fine for small problems. However, with the problem size exceeding around a hundred lines of code, HySAT was not able to handle the resulting model any more. We located the bottleneck inside the internal unrolling procedure of HySAT. In each iteration every variable of the HySAT model is newly initialized with its defined interval, unless it is not forced to keep its value of the current iteration. Due to that reason each variable representing a memory location must be encoded to store its value as long as the location is not part of a write operation in the current iteration. Therefore, the model and respectively the resulting formula are quickly growing to a prohibitively large size.

1.2 Interval Constraint Solving

Interval Constraint Solving (ICS) is a technique used to find a solution for a SMT input problem, but compared to a SMT-Solver ICS is operating on variable ranges. The search procedure of ICS, which is used in HySAT, is very similar to the common search algorithm used in standard SAT-/SMT-Solvers. An example of such a search algorithm based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DLL62] is given in the Fig. 1. The main difference between ICS and this algorithm resides within the two functions `decide()` and `bcp()`. Instead of computing a new variable decision in `decide()` the ICS computes new interval bounds for the variable to decide. A default splitting strategy for the interval $[min, max]$ can be $[min, \frac{max+min}{2}]$ and $(\frac{max+min}{2}, max]$. The function `bcp()` - abbreviating "boolean constraint propagation" - serves as an in place substitute for "interval constraint propagation" (`icp`) in ICS. In `icp` the current interval decision gets propagated through all the clauses containing the affected variable. If there exists a clause, which is false under the current interval decision, `icp` returns this clause as the conflict clause. If this conflict can not be resolved the input formula is unsatisfiable. If no conflicting clause exists the search procedure continues.

```
while (true) {
  if (!decide())
    return SAT;
  while (!bcp()) {
    if (!resolveConflict())
      return UNSAT;
  }
}
```

Figure 1: Basic Search Algorithm of a SAT-Solver from Chaff [MMZ$^+$01].

1.3 Memory Access

Memory Access is a big challenge in BMC of software, because it has to be encoded into a logical formula. The memory access instruction for a n-byte integer with a common memory representation where an address has a size of one byte may have the following logic encoding:

$$\texttt{load:} \quad result_0 = memory[address] \wedge \Big(\bigwedge_{i=1}^{n} result_i = result_{i-1} \mid (memory[address+i] << (i*8)) \Big)$$

$$\texttt{store:} \quad \bigwedge_{i=0}^{n} memory[address+i] = \big((value >> (i*8))\&0\text{xff}\big)$$

Every memory access, which depends on a variable with a type-size of greater than one byte, results in at least two clauses in the logic formula. If the variable itself is a pointer, it can point to m different memory addresses, which results in m times bytesize(variable) clauses. Hence, the logic encoding of a function with memory access and even containing some pointers can quickly reach an enormous size.

1.4 Task Model

The introduction of a task model requires a proper scheduling mechanism. Scheduling in task-based operating systems like AUTOSAR and OSEK is based on different priorities per task and whether a task is preemptive or not. Also, interrupt service routines (ISRs) have to be considered, as an ISR is able to interrupt every task. To provide a proper logic encoding of such a task model and its scheduler is a big challenge. Especially the unrolling procedure has to be efficient, since all possible interleavings between tasks and ISRs have to be considered, which might quickly lead to a state explosion in the logic encoding.

In the present paper we introduce a new BMC tool consisting of an interval constraint solver, which

- handles memory access internally,
- supports floating point variables,
- is capable of handling sequential and concurrent task-based systems, and
- can reuse already learnt clauses from conflict analysis of previous iterations.

The introduction of this tool is located in Section 2. Section 3 shows some results of our implementation from several test runs on a set of testcases. Finally, we conclude our paper in Section 4 and also discuss potential future work.

2 The MEMICS tool

The MEMICS Model Checker operates based on the architecture shown in Fig. 2. First, the C/C++ source code is compiled into the LLVM Intermediate Representation (IR) [Lat] using the Clang [Fan10] compiler front-end of LLVM [LA04]. Second, the MEMICS model is created by invoking the LLVM code generator on the LLVM IR. The actual MEMICS process starts with unrolling the model into a logic encoding in Static Single Assignment [CC77b] form, which results in a set of clauses. If this set of clauses is empty, the model is safe. If not, the solve process is started. In case the solver does not find a solution for the current state, the next iteration step is computed. If the current state is satisfiable, the solver encountered a runtime error and has a valid counterexample.

The LLVM IR still features data structures like arrays and pointers. In addition the LLVM environment allocates virtual memory for each local variable in a function. Instead of providing a new lowering mechanism, when it comes to the transformation into a proper logic encoding, we decided to reuse the one already provided inside the LLVM Code Generator. As the LLVM IR is based on a MIPS like instruction set, we chose the MIPS Assembly language as a base for the MEMICS model. Hence using the LLVM Backend and the resulting MIPS Assembly Language to generate our model has the following advantages:

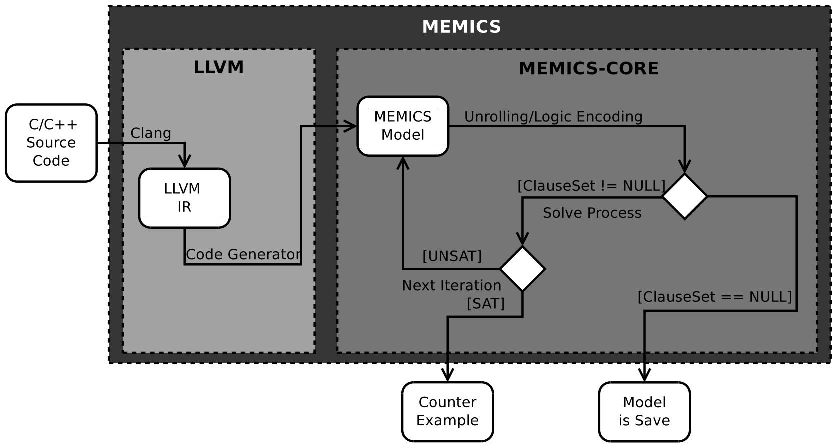

Figure 2: An Overview of the MEMICS Architecture.

- no dereferenciation of pointers,
- no derivation of Phi functions, and
- no assurance of memory allocation for local variables.

The only memory instructions, which we need to deduce, are those for heap based memory management: malloc and free.

2.1 The MEMICS Model

The model, the ICS operates on, is the state transition system M, consisting of a set of states $S = \{s_1, \ldots, s_n\}$ (the instructions) and a set of transitions $T = \{t_1, \ldots, t_m\}$. A transition t_i is defined as the 4-tuple: $< s_{i'}, a, c, s_{i''} >$. Where $s_{i'}$ is the current state or instruction, a the actual action to execute, c an obligatory condition to take the transition, and finally $s_{i''}$ the successor state. The instruction set our model is operating on, is derived from the MIPS Assembly Language [Swe06]. In order to properly handle this language our model also features a representation of a MIPS-CPU. Therefore the model has 32 variables representing the 32-bit registers, a stack pointer register and as well a program counter register in order to traverse over the transition system. Our model supports as many processor cores as required, where each core has the same amount of internal registers including a separate program counter register as well as a separate stack pointer register. The model also contains a vector of byte sized intervals representing the memory. The access to the memory, which is part of the solver itself, is described in Section 2.2.

2.1.1 Task Model

The task model is a set of tasks $R = \{r_1, \ldots, r_p\}$, where each task r_j consists of a subset of states $S' = \{s_{j_1}, \ldots, s_{j_k}\} \subseteq S$ of the MEMICS model M and a subset of transitions $T' = \{t_{j'_1}, \ldots, t_{j'_\ell}\} \subseteq T$. A transition t_i of a task is defined as the following 11-tuple:

$$< s_{i'}, a, c, s_{i''}, idle, run, start, wait, preempt, prio, freq >,$$

where $s_{i'}$, a, c, and $s_{i''}$ are defined as in the MEMICS model. The other components are required to provide a proper scheduling mechanism, where $idle$, run, $start$, and $wait$ represent the current status of the task, $preempt$ states whether it is preemptive or not, $prio$ defines it priority and $freq$ is the frequency, which the task is started with. The MEMICS model has also been equipped with a clock clk, in order to define the scheduling. The logic encoding of a non-conditional instruction is for example defined as:

$$(s_{i'} \wedge run \rightarrow a \wedge s_{i''}) \wedge (s_{i'} \wedge !run \wedge wait \rightarrow s_{i'})$$

The first implication is used to execute the current instruction of a task, if the task is running, otherwise the second implication takes place, which keeps the tasks waiting. In Fig. 3 the scheduling of a task is illustrated. The task can only be in one of the states: idle, start, run, wait at once.If a task is idle and the rule $clk \mod (freq - 2) == 0$ holds the start flag of the task is enabled. From start the task will either be set to run or wait, according to its priority. Finally if a task finished it will be set to idle again. Please note, that the state idle is not implemented into the logic, hence it can be represented as: $!run \wedge !wait \wedge !start$.

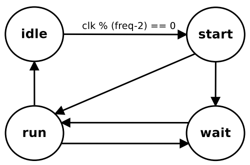

Figure 3: Scheduling of task

For a proper scheduling of all tasks on each core the following rules must hold, where the priority of t_i is higher than the priority of t_k:

1. $\sum\limits_{j=1}^{n} run_j \leq 1$

2. $start_i \wedge \bigwedge\limits_{f=1}^{i-1} start_f + run_f = 0 \wedge \bigwedge\limits_{k=i+1}^{n} run_k + preempt_k \leq 1$
   $\rightarrow run_i \wedge wait_k$ [1]

[1]Please note, $wait_k$ is enabled if an according task was running run_k

3. $start_i \wedge \bigwedge_{f=1}^{i-1} start_f + run_f > 0 \vee \bigwedge_{k=i+1}^{n} run_k + preempt_k = 2$
   $\rightarrow wait_i$

4. $wait_i \wedge \bigwedge_{f=1}^{i-1} wait_f + run_f = 0 \wedge \bigwedge_{k=i+1}^{n} run_k + preempt_k \leq 1$
   $\rightarrow run_i \wedge wait_k$ [1]

5. $(clk)\ modulo\ (freq_i - 2) = 0 \rightarrow start_i$

The first rule ensures that only one task is running in a cycle per core. With the second rule, we assure that from a set of tasks to be started only that task with the highest priority is enabled, unless there is no task running with a higher priority or a non-preemptive one. All other tasks also having the $start$ flag set are set to $wait$ according to the third rule. The fourth rule ensures only that waiting task with the highest priority becomes active unless no task with higher priority or a non-preemptive one is running. The last rule guarantees that each task is correctly started according to its frequency. Note, since run is only valid one cycle after $start$ and $start$ also needs one cycle to become active, $start$ has to be set two cycles in advance.

In our model we handle an interrupt service routine (ISR) as a task with highest priority. The only difference compared to a task is that an ISR can occur in every cycle. With this task model we are also able to handle POSIX threads. Each thread can be treated as a task as well, except it only gets started once after its creation, but can still be interrupted.

2.1.2 Efficient Model Unwinding

In this BMC approach the model is not entirely unrolled in each iteration, instead the model is only unwound instruction per instruction. For all instruction types the unwinding process is self-evident, except the branch instructions have to be considered separately. Fig. 4 shows an example of an instruction list containing a branch and a jump instruction. The branch instruction (s_i) has two possible successor instructions. If the condition `cond`

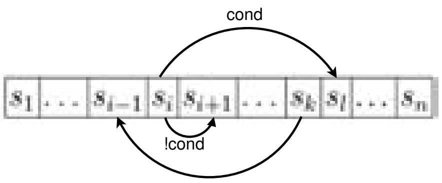

Figure 4: Linked Instruction List containing a Branch Instruction (s_i) and a Jump Instruction (s_k).

is fulfilled the successor instruction is s_l, otherwise s_{i+1}. In SAT-/SMT-Solver a decision made on the root-level (0) is irreversible. All other decisions are hypothetical and can be cancelled. Hence, to unroll a branch instruction based on the condition `cond` two cases have to be taken in account:

1. If `cond` has a pending decision, which was made on decision level (DL) 0:
   ⇒ Either if `cond` is true **only** instruction s_l is unrolled,
   ⇒ or if `cond` is false **only** instruction s_{i+1} is unrolled.
2. If `cond` has a pending decision, which was made on any decision level greater than 0 or has no pending decision:
   ⇒ **Both** instructions s_{i+1} and s_l are unrolled.

If the second case occurs the list of next instructions to unroll increases, but is still a lot smaller than as all instructions were to be unrolled in each iteration.

The described unrolling procedure can also be applied to the task model. Like a branch instruction a task instruction has two possible successor instructions. If it is running the successor instruction is the next instruction in line and if it is waiting it is the instruction itself. In case the variables run and $wait$ of a task have an assignment on DL zero, only the corresponding instruction is going to be unrolled.

2.2 The MEMICS Solver

The solver of MEMICS is an Interval Constraint Solver, for which the overall solve process is shown in Fig. 5.

```
01: bool solve() {
02:   if (!init())
03:     return false;
04:   do {
05:     if (!update())
06:       result = false;
07:     else
08:       if (endOfFile)
09:         return false;
10:       result = search()
11:   } while (result != true);
12:   return true;
13: }
```

Figure 5: The Solve Process of MEMICS.

First, the solver gets initialized in function `init()` (Fig.5 line 02) with the registers of the model and the initial instructions. Then the actual procedure begins, which runs until either a runtime error has occurred or no further instruction is reachable. The unrolling process of the model is represented by the function `update()` (Fig.5 line 06). In this procedure the successor instruction(s), which are reachable from the current state(s) of the model, are transformed into a set of clauses in conjunctive normal form (CNF). The variables occurring in these clauses are in SSA form. If this set of clauses is empty no further instructions are reachable, indicating the safety of the model. Otherwise the search process is invoked. In case the search also did not find any runtime error, the procedure continues with the next unrolling step. If the process has finished without finding any runtime error, the model and the corresponding source code is free of runtime errors. Please note, this result is only valid for those runtime errors our solver can currently detect. All discovered errors are architecture specific. There might exist architectures on which errors, that we find, do not hold. We also have to state that there is currently no support for passing specific bounds for the unrolling of internal loops, but the user can set an overall model unrolling bound.

```
01: bool search() {
02:   while (true) {
03:     conflict = icp();
04:     if (conflict) {
05:       if (DL == 0)
06:         return false;
07:       level = analyze(conflict, learnt);
08:       backtrackAndAddLearnt(level, learnt);
10:     } else {
11:       if (!decide())
12:         return true;
13:     }
14:   }
15: }
```

Figure 6: The Search Process of MEMICS.

Fig. 6 illustrates the overall search procedure. Initially, interval constraint propagation is used in function `icp()` (Fig.6 line 03) to apply the new interval deductions to the current system, which occurred while adding the new clauses in `update()` (Fig. 5 line 05). If this propagation leads to a conflict on decision level (DL) zero, a conflict on the root level has been found. This indicates that the formula of the current iteration is unsatisfiable. If the conflict does not occur on the root level, the reasons for this conflict are computed in `analyze()` (Fig.6 line 07). The result of this process is a clause eliminating this conflict for the succeeding runs, which is added to the system. On the other hand if no conflict occurred, there also exist two possible cases. If there exists no assignment for all variables leading to a run time error, which means that the model is "safe" (in the current iteration) and the solve process continues. If not, the next variable is picked from the variable queue, a new interval decision is computed for it and the process continues in line 3.

2.2.1 Memory Access Handling

In order to handle memory access the instruction set of the solver contains load and store instructions, as well as malloc and free instructions.

The load instruction (32-bit value) is defined as:

$$lw \ \texttt{<target-register, base-address} + \texttt{offset>}$$

The `target-register` is a common 32-bit variable and the `base-address + offset` the address inside the memory vector of the model. The `base-address` is either the current value of the stackpointer register *sp*, or it can also be a value stored in a common register. If the resulting address from `base-address + offset` is outside the allocated range of the `base-address` an index out of bound error is thrown. When interpreting the instruction, the solver reads the 4 bytes starting at the `base-address` and assigns the corresponding value to the `target-register`. The load instructions *lb* (8-bit value) and *lh* (16-bit value) operate likewise.

The syntax of the store instruction (32-bit value) is defined as:

$$sw <\texttt{source-register, base-address} + \texttt{offset}>$$

This instruction as well as *sb* (8-bit value) and *sh* (16-bit value) are working analogously to the load instructions.

2.2.2 Memory Allocation

In order to support dynamic memory management, we have equipped the solver with the table $MT = \{m_1, \ldots, m_n\}$, where an entry m_i is defined as the tuple $m_i =$<`address`, `size`>. The instruction set has also been extended with the instructions `malloc` and `free`. The new instructions are defined as:

$$malloc < \texttt{target-register, size} >$$
$$free < \texttt{base-address} >$$

Whenever a malloc instruction occurs, the solver consults the table, returns the first available address by considering the required size and marks that area occupied by inserting an entry into the table. In case of a free instruction, the solver searches for the `base-address` inside the table. This search process can lead to three different results:

1. The `base-address` is found and the memory addresses according to the size entry in the table get cleared.
2. In case the `base-address` is found but at least one of the memory addresses is already freed, which results in a double free error.
3. If the `base-address` is not found, it results in an invalid free error.

2.3 Race Condition

A race condition [NM92] is a type of defect, where access to shared data among different tasks/threads is leading to undefined behaviour of the software itself. One kind of race condition is a lost update, which is described in the following example.

Lost update example Assumed there are two tasks, task A and task B, which are concurrently executed on different cores. Task A is reading a global variable $ext1$ twice. But in between the to read operations task B has written an new value to $ext1$. Unless that behaviour is claimed, task B continues with a corrupt value. Using the instruction set of the MEMICS model a load and store transition can have the logic encoding:

$$s_{i'} \wedge run\_task_i \rightarrow lw(dest, ext1) \wedge lw\_task_i\_ext1_k = clk_x \wedge s_{i''}$$
$$s_{j'} \wedge run\_task_j \rightarrow sw(value, ext1) \wedge sw\_task_j\_ext1_l = clk_y \wedge s_{j''}.$$

Using this encoding, the following rule has to hold for $o > m$ in order to find a lost update error as described above:

$$lw\_task_i\_ext1_o - lw\_task_i\_ext1_m < freq\_task_i \wedge$$
$$lw\_task_i\_ext1_o > sw\_task_j\_ext1_p \wedge sw\_task_j\_ext1_p > lw\_task_i\_ext1_m$$

3 Results

The implementation of the MEMICS tool was tested on a test set containing different types of runtime errors based on the error classes from the Common Weakness Enumeration (CWE) [cwe] database. In Table 1 the results of a comparison between our tool and CBMC and LLBMC are shown.

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[2]	✓		
	MissingSynchronisation	820	✓		
Synchronization	DeadLock	833	✓		
	DoubleLock	667	✓		

Table 1: Results of MEMICS compared to CBMC and LLBMC

The first column of the table shows the overall class the test cases reside in and column two shows their subclasses. In the third column the official CWE-ID of each subclass is given.

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

With that id, one can look up the detailed description of the runtime error on the CWE website. Column three shows the results of our MEMICS tool, compared to CBMC in column four and LLBMC in column five, where ✓represents a correct verification result, - a false one and a whitespace signals that the tool does not support the class of the test cases.

The results provided in Table 1 show that our MEMICS tool is already able to handle a wide range of test cases and in contrast to BMC tools like CBMC and LLBMC is able to identify concurrency issues like race conditions and synchronization problems. In order to give an example on how our tool is dealing with a task based system, Fig. 7 shows a portion of C code implementing a global variable `signal`, a global event `trigger` and two tasks, a $1ms$ and a $100ms$ one. The $1ms$ task is first reading from the global variable

```
01: int signal;
02: event trigger;
03:
04: void task_1ms()
05: {
06:   int x = 2*signal;
07:   signal = 1;
08:   ...
09: }
```

```
10: void task_100ms()
11: {
12:   int x = signal;
13:   wait_on(trigger);
14:
15:   if (signal < 0)
16:     x = -x;
17:   ...
18: }
```

Figure 7: Simple Task Based System: task.c

`signal` (line: 06) and then assigns 1 to it (line: 07). The $100ms$ task assigns the value from `signal` to its local variable `x` (line: 12) and is, in the following, waiting for the event `trigger` (line: 13). Once the event occurred, the $100ms$ task checks if the value contained in `signal` is negative (line: 15). If it is negative the previous local variable is assigned with its absolute value (line: 16). This scenario describes a lost update error, as described in the example in Section 2.3. The configuration for the system of the task

```
BASE {
  NUMOFCORES = 2,
  FREQUENCY = 100,
  MEMSIZE = 256
};
TASK {
  NAME = task_1ms,
  FREQUENCY = 1,
  PRIORITY = 1,
  CORE = 0,
};
```

```
TASK {
  NAME = task_100ms,
  FREQUENCY = 100,
  PRIORITY = 1,
  CORE = 1,
};
EVENT {
  NAME = trigger,
  FREQUENCY = 0.1
;
```

Figure 8: System Configuration File to the Task Example: example.conf

example (Fig. 7) is shown in Fig. 8. The system features a processor with two cores running at $100MHz$ and $256MB$ of main memory, assigns the $1ms$ task to the first and the $100ms$ task to the second core, and defines the event `trigger` to be raised every $100\mu s$.

The resulting output of our MEMICS tool applied to the task example from Fig. 7 with the configuration from Fig. 8 is shown in Fig. 9. It shows that our tool finds a lost update error as described above. On top of that our tool is able to print the entire trace leading to the error by adding the option `-print-counterexample`. Currently this output is only provided in a MIPS like assembly style and is therefore not shown here, but a mapping back to the C/C++ code is work in progress.

```
user@memics:~/$ memics -task-conf=example.conf task.c

Model is CORRUPTED
Run time error: lost update
  occurred in file: task.c @ line: 15
        if (signal < 0)
  in conflict with statement @ line: 9
        signal = 1;
  triggered from line: 12
        int x = signal;
```

Figure 9: MEMICS called on the Task Example Code.

4 Conclusions and Future Work

In this paper we have introduced the new MEMICS tool for software verification. Its core is a new Interval Constraint Solver, that is directly operating on a machine code based model and is capable of handling memory access internally. In addition, the model, our tool is working on, features an efficient model unrolling strategy. The results from Section 3 have shown that our tool can already verify a large number of error classes. In comparison to tools like CBMC and LLBMC, our tool is able to handle software implementing a task-based operating system like AUTOSAR or OSEK. On top of that it is already capable of handling concurrent software. The last point is essential, as the growing application of multicore processors especially in the field of embedded devices, leads to the need of a method for the verification of concurrent software.

The main goal we want to achieve in the near future is to test our tool on industry software in order to check its scalability. Another huge aspect is to find a suitable partitioning strategy for our model, to run the verification process itself in parallel. This task is of large relevance as the complexity of concurrent software is, compared to sequential software, growing very fast, especially with increasing size.

References

[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.

[CC77a] 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.

[CC77b] 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 ASTRE 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.

[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.

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

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

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

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5:394–397, July 1962.

[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.

[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.

[IYG$^+$04] F. Ivancic, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based bounded model checking for software verification. In *in Symposium on Leveraging Formal Methods in Applications*, 2004.

[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.

[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.

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

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

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

[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.

[Tra10] Johannes Traub. Program Analysis and Probabilistic SAT-Solving. Master's thesis, Universität Stuttgart, 2010.