RWTH Aachen

Department of Computer Science
Technical Report

Doctoral Symposium on Systems Software Verification (DS SSV'09) Real Software, Real Problems, Real Solutions

Proceedings

Ralf Huuck, Gerwin Klein, and Bastian Schlich (eds.)

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2009-14

RWTH Aachen · Department of Computer Science · June 2009

Preface

This volume contains the proceedings of the Doctoral Symposium on Systems Software Verification (DS SSV'09), which was held during the 4th International Workshop on Systems Software Verification (SSV'09) in Aachen, Germany, June 22–24, 2009.

The program of the doctoral symposium was interleaved with the program of the workshop to stimulate exchange between senior researchers and young researchers and to give young researchers an opportunity to receive feedback about their research from senior researchers.

The purpose of SSV is to bring together researchers and developers from both academia and industry, who are facing real software and real problems to find real, applicable solutions. By "real" we mean problems such as time-to-market or reliability that the industry is facing and is trying to fix in software that is deployed in the market place. A real solution is one that is applicable to the problem in industry and not one that only applies to an abstract, academic toy version of it. SSV discusses software analysis/development techniques and tools, and serves as a platform to discuss open problems and future challenges in dealing with existing and upcoming system code.

This volume comprises 9 extended abstracts that were selected by the program chairs. The focus of the selection process was on originality of the research topics.

We would like to thank the authors and the workshop organizers for their contribution to the success of this Doctoral Symposium on Systems Software Verification. Finally, we are grateful for the generous support we received from the European Microsoft Innovation Center (EMIC), Germany.

June 2009 Ralf Huuck, Gerwin Klein, and Bastian Schlich

Organization

Program Chair

Ralf Huuck	NICTA, Australia
Gerwin Klein	NICTA, Australia
Bastian Schlich	RWTH Aachen University, Germany

Local Organizers

Jörg Brauer	RWTH Aachen University, Germany
Stefan Kowalewski	RWTH Aachen University, Germany
Bastian Schlich	RWTH Aachen University, Germany

Contents

Temporal Fairness of a Microkernel Scheduler

Matthias Daum

Computer Science Dept., Saarland University
Saarbrücken, Germany
md11@wjpserver.cs.uni-saarland.de

1 Introduction

We report on the formal verification of a microkernel's temporal property, namely that its multi-priority process scheduler guarantees progress, i. e., strong fairness. This paper summarizes parts of a journal article [2].

Our microkernel VAMOS provides an infrastructure for a dynamically changing number of processes, for memory virtualization, for inter-process communication (IPC), and for user-level device-driver support. The kernel establishes process switches according to IPCs and timer events. The process scheduling, however, follows a hierarchy of priorities, favoring, e. g., system processes over application processes over maintenance processes. Processes may control the kernel tasks and access kernel services via so-called *kernel calls*. A minimal access control is built into the kernel to restrict the kernel calls according to a user-defined policy.

For space restrictions, we cannot present the microkernel or our formal foundation at large. We refer the interested reader to a previous publication [1]. In this paper, we confine ourselves to a short summary on the scheduler functionality as well as the formal kernel model that serves as basis for our verification efforts. In Section 4, we reveal insights into our fairness theorem. Finally, we conclude in Section 5.

2 Scheduler

The basic policy underlying the scheduler in VAMOS is round-robin process selection. This basic policy, however, can be adjusted by two regulators: priorities and timeslices.

Our scheduler supports three different priority levels. Only processes in the highest, non-empty priority class will be scheduled. Processes in a lower class wait until no processes are ready to compute in any higher priority class. This policy is also known as *static-priority scheduling* though technically, priorities might be adjusted through kernel calls at any time during run-time.

Within one priority class, the timeslice determines how long a certain process may compute until it is preempted in favor of another process of the same priority class. Thus, timeslices determine the relative weight of process run-times while priorities lead to the preemption of lower process classes.

Inside the kernel, time measurements are based on a timer device, which periodically raises its interrupt line. In particular, we measure the run-time of processes and the waiting time for IPC in clock ticks. Note that the kernel implementation maintains a solely relative notion of time.

3 Underlying Kernel Model

We base our temporal proof on an operational kernel model, or more precisely, the formal specification of the kernel, which is used for the verification of functional implementation correctness. This model is a state automaton. The state space contains the state of the currently running processes as well as information about the scheduling internals, access control rights, and pending device communication. A transition of the VAMOS specification depends on a device input and has up to three phases:

(a) At first, the current process is advanced. This operation is either process local or it handles a kernel call. (b) If the device input indicates that the timer-interrupt line is raised, the scheduler increases the consumed time of the current process and a global counter, which measures the waiting time for IPC. Afterwards, the kernel wakes up all processes with elapsed IPC timeouts. (c) Finally, VAMOS delivers interrupts to waiting driver processes and saves the remaining interrupts for later delivery.

4 Scheduler Fairness

According to the taxonomy of Francez [3], we formalize the temporal property of strong fairness for our scheduler. In order to state this property in reasonable succinctness, we establish several abstraction layers on top of the operational kernel model. At first, the operational model only formalizes states and transitions. For temporal reasoning, however, we need execution traces. We represent a trace by two functions *states* and *inputs* that map the step number to the current state and to the device input for the next step, respectively.

Definition 1 (Trace). Two functions *states* and *inputs* describe a trace iff (a) The first state is in the set of initial states and (b) for each step number, the transition relation holds for the current state and input and the successor state.

Furthermore, we need an invariant inv_{V} over the state sequences for reasoning about the behavior of a trace. This invariant is quite elaborate such that space restrictions prohibit its presentation. We show that this predicate indeed is an invariant over the traces:

Lemma 1 (Invariant). *Predicate inv_{V} is an invariant over traces.*

Proof. We prove this statement by induction. At first, we establish inv_{V} for all initial states. At second, we show that inv_{V} is maintained by the transition function. Finally, we derive our claim by the definition of traces. Note that despite this short sketch, the formal proof is quite elaborate because of the complexity of the kernel specification and the invariant. Its verification took more than two person months. □

With these prerequisites in place, we can formulate our scheduler's fairness property. Recall that the process scheduling follows a hierarchy of priorities. We describe this phenomenon by the notion of *prioritized fairness.* In our system, the scheduler can only guarantee fairness under several assumptions:

1. Any temporal fairness statement certainly excludes terminating processes. Note that processes might still terminate in our system, we can just not say that the scheduler would treat terminating processes fairly.
2. Changing the priority class of a process contradicts the notion of *static*-priority scheduling. The scheduler does not treat a process fair if its priority is changed infinitely often.
3. If a process chooses to wait infinitely long for an IPC operation, it might starve. This problem is certainly not the fault of the scheduler but a programming or protocol error.

4. Our scheduler relies on a live timer device. At this time, the next process is selected from the highest, non-empty priority class. In other words, a process is only scheduled at all if it has the maximal priority when the timer is on.

With the current state of affairs, however, a formal statement requires explicit quantification on step numbers over the *states* and *inputs* functions. We have proven the theorem on this layer for methodical reasons: The larger context of this work is pervasive systems verification and the long-term goal is a coherent theory in a single theorem prover. Consequently, a specialized theorem prover for temporal logic is insufficient in this context. Hence, we combined the best of both worlds and embedded future-time linear temporal logic (LTL) with *action Kripke structures* into Isabelle/HOL [4]. Thus, we can integrate our main theorem into the overall context while presenting it in LTL:

Theorem 1 (Prioritized Fairness). *The* VAMOS *scheduler is fair with respect to priority classes, i. e., if a process pid (1) finally never terminates, (2) finally remains forever at the same priority class, (3) does infinitely often not pend in an IPC operation with an infinite timeout, and (4) has infinitely often the currently maximal priority when the timer interrupt is raised, it will always eventually progress. Formally:*

$$\begin{aligned}\langle \mathcal{S}_{\mathsf{V}}^0, \Sigma_{\mathsf{V}}, \delta_{\mathsf{V}}\rangle_{\mathsf{A}} \models\ & \Diamond\Box(\lambda(i,s,n) : \mathit{process\_exists}(s, \mathit{pid})) \implies \\ & \Diamond\Box(\lambda(i,s,n) : n.\mathit{priority}(\mathit{pid}) = s.\mathit{priority}(\mathit{pid})) \implies \\ & \Box\Diamond(\lambda(i,s,n) : \neg \mathit{pending\_infinite\_ipc}(s, \mathit{pid})) \implies \\ & \Box\Diamond(\lambda(i,s,n) : \mathit{has\_maxprio}(s, \mathit{pid}) \wedge \mathit{is\_timer\_on}(i) \implies \\ & \Box\Diamond(\lambda(i,s,n) : \mathit{progress}(s.\mathit{procs}(\mathit{pid}), n.\mathit{procs}(\mathit{pid})))\end{aligned}$$

For lack of space, we can neither present the detailed formalization of the above theorem nor a proof sketch. The formal proof based on the quantification over the trace functions took about eight person months and is available at the public Verisoft repository.[1] Though the LTL layer permits a considerably more succinct statement, its formalization and the abstraction of the original statement could be accomplished in just about a person week.

5 Conclusion

During verification, we found a bug in the temporal behavior of our implementation, which cannot be found in a step-wise refinement proof. This finding demonstrates the immediate practical usefulness of our proof. Furthermore, our proof can be integrated with a refinement proof that links our underlying model to the C implementation. To our knowledge, this is the first verification result that establishes a temporal property on a C implementation.

References

1. Matthias Daum, Jan Dörrenbächer, and Sebastian Bogan. Model stack for the pervasive verification of a microkernel-based operating system. In *VERIFY Workshop*, pages 56–70. CEUR-WS.org, 2008.
2. Matthias Daum, Jan Dörrenbächer, and Burkhart Wolff. Proving fairness and implementation correctness of a microkernel scheduler. *Journal of Automated Reasoning, Special Issue on Operating System Verification*, 42(2-4):349–388, 2009. DOI: 10.1007/s10817-009-9119-8.
3. Nissim Francez. *Fairness.* Springer, September 1986.
4. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Springer, 2002.

[1] The public Verisoft Repository is available at `http://www.verisoft.de/VerisoftRepository.html`.

WCET Analysis of ARM Processors using Real-Time Model Checking

Andreas Engelbredt Dalsgaard, Mads Christian Olesen, Martin Toft, René Rydhof Hansen and Kim Guldstrand Larsen

Department of Computer Science
Aalborg University, Denmark
{andrease,mchro,mt,rrh,kgl}@cs.aau.dk

Abstract. This paper presents a flexible method that utilises real-time model checking to determine safe and sharp WCETs for processes running on hardware platforms featuring pipelining and caching.

1 Introduction

In order to produce a reliable and efficient execution schedule for a real-time system (RTS), scheduling algorithms need safe and sharp worst-case execution times (WCETs) for the processes in the system [6]. The developed method utilises real-time model checking performed by the model checker UPPAAL [5] to determine these WCETs. The method is able to analyse program code found in real systems, and an extensive evaluation has been conducted using the benchmark programs published by Mälerdalen WCET Research Group [2].

The WCET for a process depends on the hardware platform that the process is executing on, thus the method is designed to allow a high degree of flexibility. For example, support for new processors only requires a model of the new processor.

Modern processors utilise techniques such as caching and pipelining, which increase the average number of calculations that can be executed per time unit [11]. Since these techniques are also found in many processors intended for embedded devices, such as members of the widely deployed ARM7 and ARM9 families [1], a modern WCET analysis method must take them into account to be useful. The presented method models these techniques in a modular and independent fashion.

It is important to be aware that caching and pipelining in certain instances can lead to timing anomalies [10], which complicate WCET analysis to a great extent. By introducing more non-determinism in the applied models, the presented method can be extended to handle the presence of timing anomalies.

2 Platform

Because the hardware platform must be taken into account when WCETs are determined, we have selected the ARM9TDMI processor core as the basis for an implementation of the method. This processor core is found in e.g. the ARM920T processor [8]. The execution time on a hardware platform generally depends on main memory, caches and pipeline, thus these elements must be modelled carefully.

Instructions and data are stored in main memory, which is very slow compared to the internal registers of a processor. Fortunately, this bottleneck can be mitigated by using one or more caches, which are small, fast memories for storing instructions or data temporarily. Caches increase a systems' overall execution speed by taking advantage of the principles of locality and by being faster than main memory. A cache is characterised on: speed, size, replacement policy and write policy.

The data path in a processor is the circuitry that executes instructions. In non-pipelined processors, an instruction must flow through the entire data path before the next instruction can enter. Since this takes time, a long data path forces the processor to run at a relatively low frequency. Conversely, a pipelined processor has its data path divided into a number of almost independent stages. Since the stages run in parallel, the cycle time becomes shorter and the processor may therefore run faster. For example, the ARM9TDMI processor core has a five stage pipeline [4], where the stages' names are fetch, decode, execute, memory and writeback. Sometimes the stages have inter-dependencies, which give rise to stalls and other situations, where the stages must interact. A stall occurs when an instruction uses the result of the previous instruction, which is delayed due to e.g. memory access. Some events, like branching, also require special handling.

3 WCET Analysis

The presented method is mainly inspired by [3,7,9]. In this work, WCET analysis is split into four analyses: cache, pipeline, path and value analysis. The first three analyses are modelled using model checking of a network of timed automata (NTA), whereas value analysis is performed separately on the disassembled process after reconstructing its control flow.

Besides an automaton for each function in the process' control flow graph (CFG), the NTA contains automata for the hardware platform's pipeline, caches and main memory. Each transition in the function automata simulates an abstract execution of an instruction. The simulation of an instruction is done by synchronising with the automata modeling the pipeline, which in several places synchronise with the automata modeling the caches. Finally, the caches synchronise with the automaton modeling the main memory.

To determine the WCET, we perform real-time model checking on the NTA using the model checker UPPAAL. There are, however, other steps that must precede the UPPAAL verification process. Figure 1 provides an overview of the method and the applied tools.

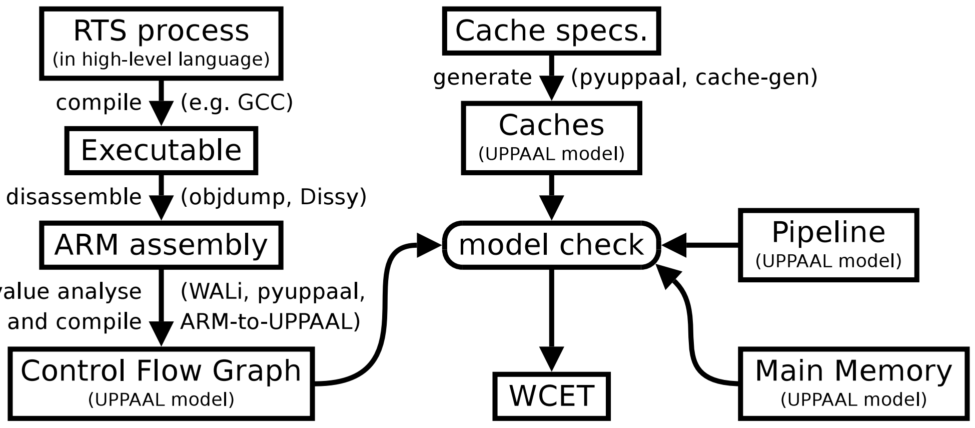


Fig. 1. Overview of the method and the applied tools.

The method is flexible, as it is easy to add support for new processors — the models for pipelines and caches can be re-used with little or no work. The modularisation of the platform model facilitates re-use and easy testing of components independently.

4 Toolchain

WCET analysis must be done at machine code level, as this is the only level with enough information [7]. By taking as input a process written in its original, high-level language and subsequently compiling and disassembling it, we are able to extract required memory addresses and take compiler optimisation into account.

We utilise the existing tools GCC, objdump, Dissy, WALi and UPPAAL, which provide compilation, disassembly, disassembly front-end, value analysis and model checking, respectively. In addition, a number of tools have been written partly or completely for the presented method. This includes pyuppaal for importing, exporting and layouting UPPAAL models, an ARM-to-UPPAAL compiler for parsing ARM assembly and generating CFG automata, "cache-gen" for generating cache automata, and "combine" for putting together all automata in an NTA. UPPAAL identify the WCET by finding the highest value of a global cycle counter clock during a full exploration of the final NTA's state space.

5 Conclusion

This paper offers a highly flexible WCET analysis method based on real-time model checking, taking caching and pipelining into account. Currently, the implementation is able to analyse 14 of the 25 working, non-floating point WCET benchmarks from Mälerdalen, which demonstrates the applicability of the method. The biggest challenge is recovering enough control-flow information, as a lack thereof is handled as non-determinism, leading to state-space explosion. Support for floating point is planned.

References

1. ARM9 - ARM Processor Family. `http://www.arm.com/products/CPUs/families/ARM9Family.html`.
2. WCET Challenge 2006. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.
3. Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66, London, UK, 1996. Springer-Verlag.
4. ARM. *ARM9TDMI Technical Reference Manual*, 2000.
5. Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.
6. Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages.* Pearson Education Limited, third edition, 2001.
7. Reinhold Hechmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 2003.
8. ARM Limited. ARM920T Technical Reference. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0151c/ARM920T_TRM1_S.pdf`, 2001.
9. Alexander Metzner. Why Model Checking Can Improve WCET Analysis. In *Proceedings of Computer Aided Verification*, pages 334–347. Springer Berlin / Heidelberg, 2004.
10. Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.
11. Andrew S. Tanenbaum. *Structured Computer Organization.* Pearson Education, fourth edition, 4 Nov 1998.

Retargeting a Hardware-Dependent Model Checker by Using Architecture Description Languages

Dominique Gückel

RWTH Aachen University
Embedded Software Laboratory
Aachen, Germany
`gueckel@embedded.rwth-aachen.de` [**]

Abstract. Model checking is a promising technique for verifying software for embedded systems. An assembly code model checker such as [mc]square can facilitate the often complicated construction of the system model by using the program under test itself as the basis for an automated construction of the system model. However, this approach is inherently hardware-dependent because it requires a simulator for each embedded system platform to be supported. In the past, such simulators used to be hand-coded. This led to considerable effort prior to any model-checking of software. In order to reduce this effort, we propose a technique for synthesizing simulators from an architecture description language.

1 Introduction

Over the past few years, model checking [1] has evolved from being just a theoretical approach into a feasible technique for detecting errors in software systems. This is especially true for the embedded area. There are two major reasons for this: first, errors in embedded system software can be very expensive or even safety-relevant. Unlike in general purpose computing, a user in general cannot apply any patches provided by the software developer after the system has been shipped. Hence, all important errors have to be detected while the system is still in development. Second, system designs for embedded systems tend to be far less complex than, for instance, an office application. For this reason, the state explosion problem from which model checking suffers is not necessarily an obstacle that cannot be overcome in the case of embedded systems.

The rest of this paper is structured as follows. Section 2 provides a brief introduction into the [mc]square model checker [2]. After this, section 3 describes our ongoing work which aims at reducing the hardware-dependency of [mc]square by means of an architecture description language. The fourth and last section concludes this paper.

2 The [mc]square Model Checker

[mc]square is an assembly code model checker developed at RWTH Aachen University. It is a discrete, on-the-fly CTL model checker with a mostly explicit state representation. The term "'mostly"' means that [mc]square supports hybrid states which are partially symbolic. The symbolic parts represent unresolved nondeterminism, which has to be resolved when the model checking algorithm needs to determine the value of that part of the state.

[mc]square aims at checking software for microcontrollers and Programmable Logic Controllers (PLCs). Given a program as an ELF file, an optional C source file and a CTL formula, it can automatically create a system model and conduct the model checking on it. No model of the environment of the microcontroller is needed, though the user can provide

[**] This work was supported by a PhD scholarship from the German Research Foundation (DFG).

one if it suits his purposes [3]. The process does not depend on an environment because any input to the system is in general considered to be nondeterministic, unless the specific state of the hardware in the simulator allows to rule out this nondeterminism.

The model checking process in [mc]square is supported by various abstraction techniques. Apart from the already mentioned environment model, [mc]square features several variants of a base technique called *delayed nondeterminism* as well as *static analyses*. The goal of the delayed nondeterminism technique is to prevent early resolution of nondeterminism, thus preventing a split-up of the computation path based on different values in memory locations of the simulated hardware. For more details on this technique, refer to [4]. The other major technique present in [mc]square are static analyses. Static analyses extract information about the examined program without actually executing it. This information is then used by the simulator during model checking to reduce the number of states that have to be created and, subsequently, stored and examined. For instance, dead variable reduction can be used to collapse states which differ only in irrelevant memory locations, and path reduction can reduce entire non-branching computation paths into single states.

The basis of state space creation in [mc]square is the interpreted execution of the input program in the simulator component. While this is an advantage with regard to the construction of the system model, and also allows the creation of very precise counterexamples, it proves to be disadvantageous regarding *portability*. Experience values for hand-coding new simulators for microcontrollers and PLCs indicate that this can be performed by a single developer within a span of six months, though there are platforms which are too complex to be modeled in all their finest details within that span [5,6]. In general, what can be achieved in that time is the creation of a so-called *instruction set simulator*. More details on how to distinguish different classes of simulators can be found in [7].

3 Architecture Description Languages and tool synthesis

3.1 General idea

The idea of automatically synthesizing software tools from hardware descriptions is by no means new. A good survey of existing approaches can be found in [7] and in [8]. The general approach is that a system developer uses a so-called Architecture Description Language (ADL) to describe the properties of his platform. This description can then be used by synthesis tools to automatically generate tools such as compilers, (dis)assemblers, profilers and simulators. Such tools are called *retargetable* because they can be changed to support a new platform with little or no effort. Using ADLs it is possible to start software development for new devices before any silicon has been manufactured.

3.2 Retargeting the [mc]square Model Checker

The goal of our ongoing research is to make [mc]square retargetable. In order to achieve this, the most important obstacle to overcome is the high effort in creating the platform simulator. The aforementioned six months for hand-coding it are simply infeasible. The next obstacle is to create appropriate static analyses for the new platform. Without these, many realistic programs become unmanageable due to the state explosion problem.

A solution for both of these major problems is the use of an appropriate ADL. Not every ADL is useful because many focus on compiler construction and do not provide means for describing those properties of a platform that are needed for simulator synthesis. However, we cannot neglect the compiler synthesis part, as precisely the kind of information needed by a compiler is also necessary for static analysis (resp. the according synthesis programs generating those tools). In our opinion, this can only be achieved by a behavior-centric ADL which provides a strict separation of semantics from behavior.

We are currently examining two promising candidate ADLs. The first one is the Isildur language used in the AVRora simulator project [9]. The AVRora simulator design influenced the design of the later [mc]square simulators, hence this approach is probably the easiest with regard to a later integration of delayed nondeterminism and static analysis techniques. However, Isildur apparently provides only very limited support for describing resources accessed by instructions. The second ADL we consider is the LISA language, originally developed at RWTH Aachen. LISA is a very flexible ADL which allows describing processors at variable levels of abstraction. However, integrating or accessing LISA is extremely complicated due to legal constraints imposed by the current owner, CoWare. For this reason, we also do consider a third option, which would be the development of a new ADL providing all the necessary capabilities with regard to expressiveness. This new language would incorporate many of the research results on other ADLs but be limited to its specific goal, that is, synthesis of fast instruction set simulators.

4 Future Work

Currently, we are creating building blocks for synthesized simulators. Such blocks can be composed to form new simulators. The principal blocks are binary file processors and a generic disassembler, which need to be present in any synthetic simulator. A binary file processor extracts the instruction stream from compiler output formats such as ELF, HEX or S-file. The instruction stream then needs to be sent through a generic disassembler which turns it into a stream of instruction objects for a generic and parameterizable machine model.

After creating a suitable generic machine model, we will extend it by more sophisticated techniques such as support for delayed nondeterminism and static analyses. A long-term goal might be the synthesis of simulators that no longer aim at maintainability but speed.

References

1. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* The MIT Press, 1999.
2. Bastian Schlich. *Model Checking of Software for Microcontrollers.* Dissertation Thesis. AIB-2008-14, RWTH Aachen University, 2008. ISSN 0935-3232.
3. Bastian Schlich, Dominique Gückel, and Stefan Kowalewski. *Modeling the Environment of Microcontrollers to Tackle the State-Explosion Problem in Model Checking.* In Proceedings of Symposium Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2008), ISBN 978 963 236 138 3.
4. Volker Kamin. *Extending the symbolic representation of states in [mc]square.* Diploma thesis, RWTH Aachen University, 2008.
5. Florian Scheuer. *Extending the model checking tool [mc]square to handle the Infineon XC167 microcontroller.* Diploma thesis, RWTH Aachen University, 2007.
6. Jörg Wernerus. *Model Checking of Instruction List Programs for Programmable Logic Controllers using [mc]square.* Diploma thesis, RWTH Aachen University, 2008.
7. Ashok Halambi, Peter Grun and Hiroyuki Tomiyama. *Automatic Software Toolkit Generation for Embedded Systems-on-Chip.* 6th International Conference on VLSI and CAD, 1999, ISBN: 0-7803-5727-2.
8. Wei Qin, Subramanian Rajagopalan and Sharad Malik. *A Formal Concurrency Model Based Architecture Description Language for Synthesis Of Software Development Tools.* ACM SIGPLAN Notices, Volume 39, Issue 7 (July 2004) (LCTES '04)
9. AVRrora. *http://compilers.cs.ucla.edu/avrora/*

Test Automation and Static Analysis – Memory Models and Abstraction Methods

Helge Löding

GESy Graduate School of Embedded Systems
University of Bremen
and Verified Systems International GmbH
Germany
`hloeding@informatik.uni-bremen.de`

Abstract. This paper introduces methods for static analysis and automated test data generation for C/C++ programs using symbolic interpretation of an intermediate program representation. It lists corresponding challenges that are subject of the author's current research.

1 Overview

In order to perform static analysis on and to generate test data for C/C++ programs we use symbolic interpretation of semantically equivalent GIMPLE programs as a basis. Rather than dealing with the syntactic complexities and semantic ambiguities of C/C++, we use GIMPLE, an intermediate transformation result from source to assembler generated by the gcc compiler family.

Section 2 outlines the structure of GIMPLE programs. Section 3 describes our approach on symbolic interpretation. Sections 4 and 5 show its usage for test data generation, static analysis and their synergies. Section 6 lists the challenges the author's doctoral dissertation has to meet in order to make these approaches viable. Section 7 gives a conclusion.

2 GIMPLE programs

We use the gcc compiler to transform a given C/C++ program into GIMPLE code. As described in [4,5], this semantically equivalent representation of a program constitutes an intermediate transformation result from source to assembler, where all expressions appearing in statements contain at most one operator and (with the exception of function invocations) at most two operands. Operands may only be variable names or nested structure and array accesses (called selectors) as well as constant values.

By introducing auxiliary variables, all original statements will be transformed in order to adhere to the above requirement. Statements may therefore only be assignments from expressions to variables (or atomic selectors in the above sense). Casting and referencing/dereferencing of variables (or selectors) form expressions themselves, and therefore may not be used as operands, but have to be executed as separate assignments to auxiliary variables. GIMPLE programs contain no loop constructs. Instead, all loops from the original source are transformed into conditional jumps to preceding labelled statements. The concrete semantics for GIMPLE programs is outlined in [7].

3 Symbolic interpretation of GIMPLE programs

Symbolic interpretation of GIMPLE programs is used to capture the effects which assignment statements may have on memory in terms of earlier memory states. Within the symbolic interpretation of a given GIMPLE program, these effects are represented as a set of so-called *memory items*. Memory items reference (1) a range of program counter values for which they are valid, (2) the affected memory segment, (3) a symbolic base address within that segment, (4) expressions over program symbols to capture offsets from base address, (5) the size of affected memory portions, (6) symbolic values written to that memory portion, (7) typedata to specify the encoding within memory, and finally (8) validity constraints that must hold during a concrete execution whenever the corresponding assignment has the effects specified within the memory item.

Each assignment statement within a program will result in one or more memory items being created, and possibly some existing memory items being modified. The state space under consideration therefore consists of program locations specified as a sequence of GIMPLE statements remaining to be interpreted and sets of memory items corresponding to each program location. The semantics of the symbolic interpretation can then be given as a transition relation mapping a given symbolic interpretation state to its successor state [7].

4 Automatic test data generation

For a given trace through a GIMPLE program, symbolic interpretation may be used to capture the symbolic memory state associated with each program location within that trace simply by symbolically interpreting the corresponding sequence of assignment statements within the trace and initially neglecting the applicable guard conditions arising from looping and branching constructs.

In order to decide the feasibility of a trace, one must now consider the guard conditions appearing within that trace. Since these are evaluated at locations with specific program counter values, the memory items corresponding to those program counter values and memory locations referenced in the guard conditions can be identified as relevant. Since their valuations reference earlier (in terms of program counter values) memory items themselves, one can eventually collect a conjunction of constraints on input data that must hold in order for the trace to be feasible.

By collecting these constraints on inputs, the task of finding suitable input values to realise the selected trace is reduced to an SMT instance and passed on to a suitable solver. See [9], [3].

5 Static analysis

We employ abstract interpretation in order to conduct static analysis, again using symbolic interpretation as a basis for this. However, as opposed to automatic test data generation, we now need to consider all possible traces within a given GIMPLE program. Within the symbolic interpretation, this is achieved by unwinding looping constructs for a finite maximum amount of times and calculating fixpoints, if possible. This implies that program counter values must now also be represented as expressions over program symbols.

Depending on the analysis goals, we can now select suitable lattices as valuation domains for different data types and evaluate the symbolic memory state using canonic operator liftings. See [7] for an example.

False alarms may be detected by attempting to construct test data that forces their occurrence. If it can be shown that such test data does not exist, a false alarm has been detected [8]. However, it is generally not possible to detect all false alarms since proof of the inexistence of such test data may be too complex.

6 Challenges

In order for the methods mentioned above to be applicable, several properties must hold:

The symbolic interpretation of all traces of a GIMPLE program must be an overapproximation compared to all possible concrete executions. The lattice-based interpretation of a symbolic memory state again must be an overapproximation. This ensures that static analysis captures all possible defects detectable using an appropriate choice of abstraction lattices.

The symbolic interpretation of a single trace of a GIMPLE program must not be an overapproximation. This ensures that automatic test data generation for a given trace only yields input data that realises that trace during a concrete execution. Also, the symbolic interpretation of a single trace of a GIMPLE program must not be an underapproximation. This ensures that for a given trace, unsatisfiability of the corresponding SMT instance implies unreachability of that trace.

Within the author's doctoral dissertation, it will hence be shown that

- symbolic interpretation of an entire GIMPLE program is a **sound** abstract interpretation,
- lattice-based interpretation of a symbolic memory state is a **sound** abstract interpretation, and
- symbolic interpretation of a single GIMPLE trace is a **sound** and **complete** abstract interpretation.

This requires formal definitions for

- concrete semantics for GIMPLE programs,
- abstraction semantics for symbolic interpretation of GIMPLE programs, and
- abstraction semantics for lattice-based interpretation of symbolic memory states.

Techniques to produce these proofs are discussed in [6].

7 Conclusion

We use symbolic interpretation of GIMPLE programs as a basis for static analysis using lattice-based abstract interpretation. False alarms may be detected by showing that construction of test data which forces their occurrence is not possible. For generating test data for a given trace (or proving its inexistence) symbolic interpretation is again used as basis. The soundness and, in the case of single traces being interpreted, completeness of the involved abstract interpretations will be shown in the author's doctoral dissertation.

References

1. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the ASTRÉE static analyzer. In M. Okada and I. Satoh, editors, *Eleventh Annual Asian Computing Science Conference (ASIAN'06)*, pages 1–24, Tokyo, Japan, LNCS, December 6–8 2006. Springer, Berlin. (to appear).
2. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 2002.
3. 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*, 2007.

4. GCC, the GNU Compiler Collection. The GIMPLE family of intermediate representations. See `http://gcc.gnu.org/wiki/GIMPLE` `http://gcc.gnu.org/wiki/GIMPLE`.
5. Helge Löding. Behandlung komplexer Datentypen in der automatischen Testdatengenerierung. Master's thesis, University of Bremen, May 2007.
6. Jan Peleska and Helge Löding. *Static Analysis By Abstract Interpretation.* University of Bremen, Centre of Information Technology, 2008. available under `http://www.informatik.uni-bremen.de/agbs/lehre/ws0708/ai/saai\_script.pdf` `http://www.informatik.uni-bremen.de/agbs/lehre/ws0708/ai/saai_script.pdf`.
7. Jan Peleska and Helge Löding. Symbolic and Abstract Interpretation for C/C++ Programs. In *Proceedings of the 3rd International Workshop on Systems Software Verification (SSV 2008)*, Sydney, Australia, February 2008
8. Jan Peleska, Helge Löding, and Tatiana Kotas. Test automation meets static analysis. In Rainer Koschke, Karl-Heinz Rödiger Otthein Herzog, and Marc Ronthaler, editors, *Proceedings of the INFORMATIK 2007, Band 2, 24. - 27. September, Bremen (Germany)*, pages 280–286.
9. S. Ranise and C. Tinelli. Satisfiability modulo theories. *TRENDS and CONTROVERSIES–IEEE Magazine on Intelligent Systems*, 21(6):71–81, 2006.

Toward an Efficient Methodology for the Analysis of Fault-Tolerant Systems

Chih-Hong Cheng[1], Christian Buckl[3], Javier Esparza[2], and Alois Knoll[1]

[1] Unit 6: Robotics and Embedded Systems
Institute of Informatics, TU Munich, Germany
[2] Unit 7: Theoretical Computer Science
Institute of Informatics, TU Munich, Germany
[3] Fortiss GmbH, Germany
{chengch,buckl,esparza,knoll}@in.tum.de

Abstract. In this report, we first outline on the feature of the tool FTOS, which is designed for model-based development of fault-tolerant systems, and describe our approach to tackle the analysis problem. Regarding formal verification, a two-phase approach is performed to avoid extreme complexity caused by verification. The result is FTOS-Verify - a toolkit for applying formal verification during development with FTOS.

1 Introduction

In this paper, we report our adaptive scheme for the analysis of FTOS systems. FTOS [2] is a research tool developed by Technical University of Munich, focusing on the ease of modeling and code-generation of embedded control systems with a focus on the automatic generation of fault-tolerance mechanism. In the system development cycle, the designer first constructs models with four abstract views, namely the *hardware model*, the *software model*, the *fault model*, and the *fault-tolerance model*. During code-generation process, predefined templates (platform-dependent) related to fault-tolerance are selected, adopted, and combined into platform specific code.

Although the idea to perform model construction and code-generation is simple, it actually relies on the assumption regarding the existence of a powerful analysis tool which checks the validity of fault-tolerant mechanisms. Since mechanisms and corresponding templates are predefined, the set of selected mechanisms may not be sufficient to resist the faults defined in the model. In fault-tolerant systems this claim of resistance is extremely important, while it requires detailed analysis of system behavior. Note that in general the system behavior might easily become too complicated because of time and infinite value domain.

Since in FTOS, the underlying model-of-computation (MoC) is an extension of Giotto [4] which utilizes the concept of logical execution time, the simplicity and the analyzability of this MoC enable us to perform formal analysis to increase users' confidence about the correctness of the implementation related to fault-tolerance. This leads to the FTOS-Verify project, which tries to formally prove that a system equipped with fault-tolerant mechanisms can resist faults defined by the fault model.

2 FTOS: Execution Semantics and Determinacy

The execution semantics of FTOS follows similar concepts as that of Giotto - during one major-tick, micro-ticks are defined. However, FTOS uses more micro-ticks for the introduction of fault-tolerance mechanisms. The conceptual execution semantics is listed in fig. 1(a). During a major-tick, fault-tolerance mechanisms are executed between major functional steps (input, task, output) if they are specified in the model.

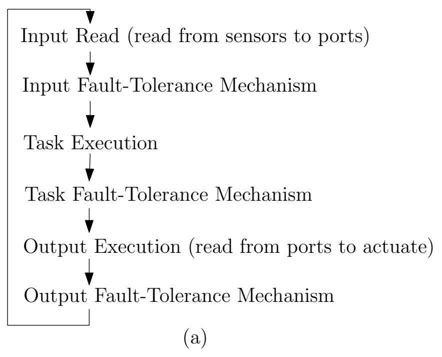
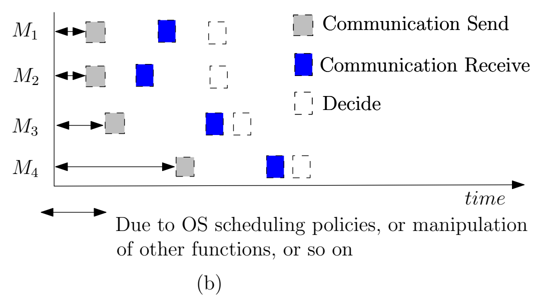

Input Read (read from sensors to ports)
Input Fault-Tolerance Mechanism
Task Execution
Task Fault-Tolerance Mechanism
Output Execution (read from ports to actuate)
Output Fault-Tolerance Mechanism
(a)

M_1
M_2
M_3
M_4
Communication Send
Communication Receive
Decide
time
Due to OS scheduling policies, or manipulation of other functions, or so on
(b)

Fig. 1. Conceptual execution semantics of FTOS (a) and internal nondeterminism due to scheduling differences (b).

2.1 Determinacy

In ordinary Giotto systems, internal determinism is guaranteed, meaning that two deployments having the same relative ordering in the action level will behave the same, irreverent of the absolute timing. Unfortunately, internal determinism will not be maintained if no constraints are added additionally on FTOS functions. Consider fig. 1(b), where M_1, M_2 and M_3 are three deployments. The send action will broadcast messages to other machines regarding its liveness. Ideally when no error happens, then for each machine, it should conceive the fact that all machines are live. Nevertheless, when the scheduling of M_3 changes to that of M_4, even with zero time transmission M_1 and M_2 will not conceive M_4 live because they have finished the process of receiving. This brings semantic incompatibility between different deployments.

To solve this problem, we thus propose the concept called *deterministic assumption*. Intuitively, the goal is to assume that when no error happens, the fault tolerance mechanisms deployed always behave the same. In practice, this might place constraints regarding the earliest and latest arrival time between messages sent.

3 A two-phase verification methodology

In FTOS-Verify we adapt a two-phase verification process.

- **(Phase 1: Verification on the platform independent layer)** We first assume that the deterministic assumption holds in all deployments. Based on this assumption, we construct a verification model. The model is an abstract machine (closed model) where injection of faults is regulated based on the fault model. The model offers precision by revealing detailed mechanisms of fault-tolerance. Our theoretical foundation enables us to construct a concise model with huge benefits[1]. For this phase, the mathematical formulation and the proof of theorems have been stated in [3].
- **(Phase 2: Validity checking of the behavior-architectural mappings)** In this phase, our focus is on two parts. We first have to check whether the deterministic assumption holds in the platform. Secondly, we have to check if a deadline violation is possible, and if this violation exceeds the constraint specified (regulated) in the fault

[1] Our theorem states that we can construct a synchronous verification model (exponentially smaller reachable state space) provided that (1) deterministic assumption holds and (2) properties are in-machine LTL properties without using temporal operator **X**. This makes formal verification of large systems practicable.

model. Note that since the correctness of the data/mechanisms is checked in the first phase, only the protocol checking (timing) is required. The verification model is generated from timing information of the deployment, e.g., network delay, CPU processing time, scheduling algorithms, and so on.

General remark Our process requires the extraction of existing templates related to fault-tolerance to new templates described in formats acceptable by back-end verification engines. This extraction process is done only once and can be continuously reused. Similar approaches can be found for work in STL verification [1].

4 FTOS-Verify

We have implemented our first prototype tool *FTOS-Verify*[2], and deploy it as an Eclipse add-on for FTOS. Currently, it enables an automatic translation from FTOS models into two separate verification models, namely the functional model and the timing model. Also a set of predefined specifications are annotated accordingly. More interesting formal specifications can be added manually.

Regarding execution time, preliminary experiments shows that our separation reduces the total amount of time into reasonable amount in academic projects[3]. Note that this can vary due to the differences of models, e.g., how many mechanisms are introduced, or what is the scheduling policy used in the architecture.

5 Conclusion

We seek for an efficient methodology for designing fault-tolerant systems while correctness can be proven formally. For verification, our principle relies on a two-phase methodology. An experimental tool is constructed with extensibility.

We list some challenges for future investigations. The first is how we can scale our approach to real-embedded systems, where computation units and platforms are much complicated than academic projects. The second issue is how timing parameters can be retrieved to have a faithful verification result.

References

1. N. Blanc, A. Groce, and D. Kroening, "Verifying C++ with STL containers via predicate abstraction," in *22nd IEEE International Conference on Automated Software Engineering (ASE)*. IEEE, 2007, pp. 521–524.
2. C. Buckl, M. Regensburger, A. Knoll, and G. Schrott, "Generic Fault-Tolerance Mechanisms Using the Concept of Logical Execution Time," in *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC'07)*. IEEE, 2007, pp. 3–10.
3. C. Cheng, C. Buckl, J. Esparza, and A. Knoll, "FTOS-Verify: Analysis and Verification of Non-Functional Properties for Fault-Tolerant Systems," TU Munich, Tech. Rep. (arXiv:0905.3946), 2009.
4. T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *Proceedings of the First International Workshop on Embedded Software (EMSOFT'01)*. Springer-Verlag, 2001, pp. 166–184.

[2] The software can be retrieved on demand.
[3] Currently for functional analysis, the required execution time (using Cadence SMV) is less than 15 minutes; for timing analysis, execution time (using UPPAAL) is less than 30 minutes.

Model-based Analysis of Embedded Java Programs

Thomas Bøgholm, Anders P. Ravn and Bent Thomsen

Aalborg University, Denmark
`{boegholm,apr,bt}@cs.aau.dk`

Abstract. The research topic presented here is model based analysis of Java programs for embedded applications. The aim is to combine model-checking and program analysis techniques, such that model checking takes care of dynamic features e.g. caching and program analysis provide suitable abstractions of program blocks and control flow. In this paper we present initial results, a case, study and future directions.

1 Introduction

In the area of embedded systems, Java can be considered a suitable replacement for traditional programming languages, such as C and assembler, since the complexity of such system continiue to increase. Java provides abstractions, relieving the programmer from worries about the low-level and error prone details, such as memory management e.g. (de)allocation, pointers, etc., and hardware specific details.

The use of Java, however, complicates program analyses, such as the required *schedulability analysis* and analysis of *memory usage*. This is partly due to the layered architecture: Java Virtual Machine (JVM), operating system and underlying hardware, and partly the use of garbage collection for memory management. Additionally, applying such analyses on systems for multi-core architectures still is an open research area [6].

The use of Java for embedded systems thus calls for development in two interdependent areas:

Programming Language Profiles: Since standard Java is unsuitable for dependable systems development because of the dynamic nature and unpredictable execution times, caused by e.g. garbage collection, dynamic class-loading, and unspecified scheduling scheme, a safe subset of the language should be provided. Though much work has been done in this area, room for improvement still exist. Part of this research is focused on developing a predictable Java-profile, which smoothly interacts with analysis tools.

Platform Dependent Analyses: As with programming languages, much work has been done in the development of program analysis techniques, some especially suited for real-time applications. Standard schedulability techniques require the computation of Worst Case Execution Time (WCET) for each task in the system, and provide pessimistic, but safe, answers [2]. Abstract interpretation, static analysis and model-checking techniques provide general approaches to program analyses for ensuring program correctness with regards to certain properties, such as *null-pointers*, *array index errors* etc. The goal in this project is to develop new techniques combining these approaches in developing a single tool aimed at the aforementioned Java-profile, using model-checking, aided by suitable analysis techniques.

2 Initial Experiment

In an initial effort in this project [1], a tool named SARTS, Schedulability Analyzer for Real-Time Systems, was developed.

The SARTS tool analyzes Java programs and constructs control flow graphs decorated with timing information in the form of timed automata for the model checker UPPAAL. UPPAAL can then be used to perform schedulability analysis, by verifying the absence of deadlocks in the timed automata model.

For Java programs to be analyzed using SARTS, these programs must be written in the Java profile, SCJ [5], a profile providing a simple analyzable framework for safety-critical embedded Java systems, intended to be executed on the time-predictable Java processor Java Optimized Processor (JOP) [4]. In this framework, a real-time system is a three phase system, *initialization*, *mission*, and *shutdown*, with appertaining sporadic and periodic tasks. The initialization phase is a non-critical phase setting up the system before execution, a phase not considered in the schedulability analysis. The mission phase is the safety-critical execution phase where periodic and sporadic tasks are released. The shutdown phase performs a controlled shutdown of a running system.

The two task classes, periodic and sporadic task, each contains task specific parameters used in the analysis: for periodic tasks, these are offset, period, deadline, and for sporadic tasks: minimum inter-arrival time and deadline. A task implementation contains implementation of a Java method, *run*, invoked when the task is released.

This framework structure is reflected in the generated UPPAAL model, in which standard templates are created for periodic and sporadic tasks, and a template for the scheduler; a deadline monotonic scheduling strategy using a priority ceiling protocol.

For each method in the Java program, including the task run-methods, a template is generated, representing control flow at the byte-code level, decorated with execution time for each byte-code instruction, information about blocking, preemption and method invocation. These templates act as building blocks in the final UPPAAL model and linked together using synchronization channels. The resulting model follows this predefined pattern:

- One scheduler instance
- Task controllers corresponding each task in the Java program, linked with their corresponding *run* methods
- N method models for each Java method, where N is the total number of tasks in the Java program; this is to allow concurrent execution of each method from different tasks

This results in a single model with $1 + T + (T * M)$ timed automata run in parallel, where T is the number of tasks and M is the number of methods in the Java program. Method invocation is performed through channel synchronization, blocking time is modeled by preventing preemption in the scheduler, preemption is performed using *stopwatches*, and the execution time for each task is tracked using *clocks*. In the case of a task deadline overrun, the failing task will cause a deadlock in the model; hence, the schedulability property of the Java program is verified in UPPAAL by verifying the absence of deadlocks.

SARTS has been applied successfully on a case study, a sorting machine built in lego, controlled by the JOP processor and a Java control program consisting of two periodic and two sporadic tasks, amounting to an approximate 300 lines of code. Using *convex hull* approximation, a safe over approximation, this system has been verified in approximately 37 seconds using UPPAAL on standard hardware.

An outline of future work includes further development of the approach using static analysis to improve analysis result and performance, the development of further analyses such as memory consumption, cache behavior, multi-core scheduling and finally the integration of these techniques in a single tool and possibly integration in an IDE, e.g. Eclipse, in order to make the techniques available to developers.

Using static analysis techniques to reduce the non-determinism in the model applies not only to the schedulability analysis technique, but could also further improve analyses such as memory consumption. More precise modeling of the platform should include e.g. caching behavior, such that the analysis result become more accurate.

The approach taken in SARTS, i.e. the translation of Java programs to Uppaal models is believed to be applicable to memory consumption analysis, by modeling the memory allocation scheme of the Java profile. Tracking object creation in the model is deemed a promising approach in being able to answer questions about worst case memory consumption. This technique applied to the analysis of schedulability in a multi-core setting is also considered an interesting future development.

Further development of Java profiles includes defining annotations to aid the analyses. As of now, loop bounds must be specified by the developer in the form of simple comments containing a constant iteration count for a each loop. Since supplying correct loop bounds is the responsibility of the developer, this is a potential source of errors. Improvements in this area includes automatic loop bound verification inspired by the techniques presented in [3]. Additionally, more expressive loop bound notations could be introduced, since cases exists where an upper bound cannot be determined accurately local to the loop e.g. a list sort function contain loop bounds which cannot be determined locally, since they depend on the size of the list in question.

Finally, in integrating these techniques in an IDE, it may be possible to visualize the analysis results. In the case schedulability, code locations with heavy execution times may be highlighted, or in the case of loop bound detection, complicated loop expressions may be identified.

3 Acknowledgements

We would like to thank Henrik Kragh-Hansen, Kim Guldstrand Larsen and Petur Olsen for their efforts during the initial work of this research project [1].

References

1. Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 106–114, New York, NY, USA, 2008. ACM.
2. Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages.* Addison Wesley Longmain, 2001.
3. James J. Hunt, Fridtjof B. Siebert, Peter H. Schmitt, and Isabel Tonin. Provably correct loops bounds for realtime java programs. In *JTRES '06: Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems*, pages 162–169, New York, NY, USA, 2006. ACM.
4. Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems.* PhD thesis, Vienna University of Technology, 2005.
5. Martin Schoeberl, Hans Søndergaard, Bent Thomsen, and Anders P. Ravn. A profile for safety critical java. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 94–101, Washington, DC, USA, 2007. IEEE Computer Society.
6. Fridtjof Siebert. Jeopard: Java environment for parallel real-time development. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 87–93, New York, NY, USA, 2008. ACM.

Verifying the PikeOS Microkernel: First Results in the Verisoft XT Avionics Project

Christoph Baumann[1] and Thorsten Bormer[2]

[1] Saarland University, Dept. of Computer Science, Saarbrücken, Germany
`baumann@wjpserver.cs.uni-sb.de`
[2] University of Koblenz, Dept. of Computer Science, Koblenz, Germany
`tbormer@uni-koblenz.de`

Abstract. In this paper, we are giving an overview of the ongoing Verisoft XT Avionics project reporting on the progress of the project, and presenting first results in the verification of the system calls of the microkernel.

The goal of Verisoft XT Avionics is to formally verify an existing operating system which has not been designed for deductive code verification. The system under consideration is PikeOS, a state-of-the-art microkernel developed by SYSGO AG, which is used in a variety of embedded applications. For automated formal verification we deploy Microsoft's Verifying C Compiler (VCC).

Introduction Functional correctness of the built-in operating system is a crucial requirement for the reliability of safety- and security-critical systems. Hence, operating system kernels are a worthwhile target for formal verification. The goal of the Verisoft XT Avionics sub-project is to prove functional correctness of the microkernel in PikeOS, a commercial operating system for embedded systems [1].

Unlike the predecessor project Verisoft I, we do not target pervasive verification of the whole system stack consisting of – amongst others – the hardware, compilers and the microkernel. We rather pick one layer, the PikeOS microkernel, to show that verification of a real world implementation of a considerable size is a feasible task, taking advantage of the high degree of automation when using verification tools like VCC.

The PikeOS System PikeOS (see `http://www.pikeos.com/`) consists of a microkernel acting as paravirtualizing hypervisor and a system software component. The verification target we have chosen for our project is the PikeOS version for the PowerPC processor family, the OEA architecture, and the MPC5200 platform [6,7].

The roots of PikeOS can be seen in the L4 family of microkernels, although the PikeOS kernel is particularly tailored to the context of embedded systems, featuring real-time functionality and resource partitioning.

Along with another component on top of the microkernel layers, the so-called system software, the PikeOS kernel provides partitioning features. This allows to virtualize several applications on one CPU, where each application runs in a secure environment with configurable access to other partitions, if desired.

In order to provide real-time functionality, there are many regions within the kernel code where execution may be preempted – we thus have a concurrent kernel. Moreover, the kernel is multi-threaded.

To allow for easy adaptation to other platforms and CPU families, the kernel is structured into three layers. There are two layers close to the hardware providing processor abstraction as well as platform-dependent functions. These layers are partly written in PowerPC assembly, which makes about a quarter of the overall code. On top of these abstraction layers, the generic microkernel provides features like tasks and threads with real-time scheduling and memory management and is written entirely in C. This layer is approx. the size of both lower layers combined. In total, the code size of PikeOS is smaller than that of, e.g. Linux by several orders of magnitude.

Verification Methodology and Toolchain To show correctness of the implementation of PikeOS, we use the Verifying C Compiler (VCC) developed by Microsoft Research. The VCC toolchain allows for modular verification of C programs using method contracts and invariants over data structures. These contracts and invariants are stored as annotations within the source code, similar to the approach and syntax used in, e.g. Caduceus [5] or SPEC# (see `http://research.microsoft.com/specsharp`).

As most verifying compilers today, VCC works using an internal two-stage process. Firstly, the annotated C code is translated into first-order logic via an intermediate language called BoogiePL [4]. BoogiePL is a simple imperative language with embedded assertions. This representation is further transformed into a set of first-order logic formulas (called verification conditions), which state that the program satisfies the embedded assertions. In the second stage, the resulting formulas are given to an automatic theorem prover (TP) resp. SMT solver (in our case Z3 [3]) together with a background theory capturing the semantics of C's built-in operators, etc. The prover checks whether the verification conditions are entailed by the background theory. Entailment implies that the original program is correct w.r.t. its specification.

One distinguishing feature of this verification tool, especially important in our setup, is the support for concurrency (see [2] for details).

Verification of the PikeOS Microkernel As already mentioned, we are verifying a real-world implementation of a microkernel that is deployed and used in industry and has not been written with verification in mind. For now, we only consider functional properties of the microkernel – the specifications for this are in parts derived from the informal descriptions of the PikeOS kernel (including the reference manual). In addition, code analysis and inspection provides further insight into implementation details.

One of the first steps to verify PikeOS have been helper functions with limited dependencies on other parts of the system. The next verification target are system calls, both to demonstrate pervasive verification through the different layers of the microkernel and to introduce externally visible contracts of the kernel.

Verification of System Calls As a first target for verification, we have chosen a simple system call which changes the priority of a thread (named `p4syscall_fast_set_prio`). This call has a rather simple functionality, but it serves very well as an example because its execution spans all levels of the PikeOS microkernel, from high-level kernel functionality to hardware-related levels and the user-level interface (system calls are invoked via user interrupts).

Handling Assembly Code On the hardware-related level of the system call to be verified, (inline) assembly code is used to directly access the underlying hardware. We model the relevant parts of the hardware as a C struct in the ghost-state of the program, to be able to verify code including PowerPC assembly. The effects of assembly instructions can then be expressed using this ghost model.

Henceforth, each assembly instruction is modeled by a corresponding specification function that operates on the machine model C struct – this allows us to replace assembly code by a sequence of specification functions being able to be annotated and verified using VCC like normal C code. The specifications of these assembly code parts are then used as contracts on the upper layers of the system call implementation.

The Abstract Layer On the upper layers of the kernel, the functional verification of PikeOS has to establish the contracts that enable users of the microkernel to depend on the specifications made for the outer boundaries of the kernel as given in its documentation.

Because system calls are at the user's interface to the kernel and the PikeOS system is multi-platform, the kernel's specification has to hide any PowerPC implementation details to ensure proper encapsulation. Further, the specification has to reflect the effects of the system calls as given in the documentation of PikeOS and thus the entities defined in this documentation have to be available on the VCC specification level.

Therefore, an abstract model of the kernel's state is introduced that captures the essential properties of the kernels state, including the current hardware configuration. As with the model of the PowerPC hardware, this abstract model is kept in the ghost state of the program.

Some of the fields of the abstract kernel model are related to the underlying machine state and thus to the hardware model mentioned above. These relations are explicitly specified with object invariants depending on both the abstract model and the hardware model. The VCC methodology then ensures that the relation between the abstract model and concrete machine model is obeyed by the implementation.

First Results Based on the model of the underlying PowerPC hardware, we were able to verify low-level functions of the PikeOS kernel. This then enabled us to verify first system calls of the kernel – for the time being under the restriction that no preemption takes place during the call. In the verification of a system call that crossed the boundary between C and assembly, a first version of an abstract model of the kernel's state was defined. The elements of this abstract state that are visible to other components directly correspond to the entities described in the PikeOS documentation.

In the next steps, we will extend the verification effort to further, more complex, system calls. In addition, annotations dealing with concurrency and ownership are ongoing work.

References

1. Christoph Baumann, Bernhard Beckert, Holger Blasum, and Thorsten Bormer. Better avionics software reliability by code verification. In *Proceedings, embedded world Conference, Nuremberg, Germany*, 2009.
2. Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. A practical verification methodology for concurrent programs, 2008.
3. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, Proceedings of the 14th International Conference, Budapest, Hungary*, LNCS 4963, pages 337–340. Springer, 2008.
4. Rob DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
5. Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.
6. Freescale Semiconductor. *Programming Environments Manual for 32-Bit Implementations of the PowerPC^{TM} Architecture*, 3rd edition, September 2005.
7. Freescale Semiconductor. *MPC5200B User's Manual, Rev. 1.3*, Sep 2006.

Testing in the industrial development process for embedded systems

Ralf Mitsching

Embedded Software Laboratory
RWTH Aachen University, Germany
mitsching@embedded.rwth-aachen.de

Abstract. Model-based timed testing is a suitable method for the test of embedded systems which allows the specification of timing behavior in the model as well as in the test cases. As such, it is well suited for the testing of embedded systems, which usually puts a much higher demand on the amount of test cases and quality of the test process than other software systems do. While a thoroughly developed theory of model-based timed testing exists, there are only few examples of industrial applications.

In this presentation, we describe our experiences from an ongoing industrial case study employing model-based timed testing. The goal of this cooperation is to define an industrial strength process for model-based timed testing based on practical experience gained within the cooperation. We will present the adaptions necessary to apply model-based timed testing within an industrial environment. Using a very simplified sub system of the tested product, we were able to successfully apply timed testing to the real system of the industrial partner. We will describe typical problems arising during the adaptations and present suitable solutions. Based on this, we will present a set of lessons learned, that should be of interest for practitioners as well as for people working on the theory of model-based methods.

1 Motivation

Due to the high manual effort needed for testing, testing costs can be up to 70% of the overall system development costs. Automated techniques for test case generation, execution and evaluation can be used to reduce these costs.

A model-based approach forms a good foundation for test automation. In a model-based approach, the development process starts with the definition of a model that unambiguously describes an abstract view of the system's behavior, derived from the system requirements specification. Algorithms exist which can derive test cases from such a model, which then can be executed and evaluated.

One method of model-based testing is specification-based testing: the behavior models, test-generation, -execution and -evaluation are described using a formal semantics in order to be able to prove the correctness of the approach. [1] presents one of the first specification-based test theories (a similar approach is described in [2]).

For safety-critical systems and many embedded systems the proper timing has to be taken into account. Timed testing is an approach to incorporate timing into model-based testing. In the last years, several specification based approaches for timed testing have been developed ([3] - [5]). At least two academic tools support timed testing. One is based on TorX [6] and the other, called UPPAAL Tron [7], on the model checker UPPAAL.

Complexity, configurability as well as the demand on quality of embedded systems are increasing and inducing higher demands on testing embedded applications. Thus, industrial companies are looking for cost-efficient solutions that can cope with these changes.

2 Test process

A simplified industrial test process typical includes the following steps: test planning and control, test analysis and design, test implementation and execution, evaluating exit criteria and reporting as well as test closure activities and the corresponding responsible roles (cf. [8]). Within the test planning step, the requirements engineer identifies and structures the requirements of the customer. He defines the objectives of testing and specifies test activities. The control part of this step focuses on ensuring success of the test project.

There are numerous methods to analyze and specify the customers requirements. In a MBT approach, requirements are documented in a formal language. The industrial process of our industrial partner is only lightly based on formal methods, so most of the requirements documentation is in natural language.

Test analysis and design consists of the creation of test specifications describing test purposes and test goals. An MBT approach formalizes these test specifications as a model of the system.

In the next step, the test harness is defined by the domain engineer. Additionally, test cases are written based on the test spec resp. the system's model. Ideally, these test cases can be automatically derived from the model.

In the following step, implementation and execution of the previously specified test cases is done. Within the limited scope of our current case study, we did not implement test cases, but instead used on-line testing as provided by UPPAAL Tron.

After test execution, test logs are checked against the exit criteria of the test planning, and decisions for continuation of test activities are taken. For each concluded test, a test report is written.

During the test project closure activities, it is checked if all deliverables have been delivered, and a final report including a list of open points is written. The test environment is shut down and archived for later re-use. In a last step, lessons learned are analysed and presented.

Our first case study was meant as a proof-of-concept, and therefore aimed more at timeliness than at completeness. Therefore we did not run a full blown industrial strength test project, but in fact just joined ongoing test activities at our industrial partner. We focused on a sub-system of the complete system, namely the signal controller, and ran some representative test cases there.

3 Case Study

In practice, activities within the test project are not done by the book, but deviate from the defined processes for various reasons.

In our test project, there was no single point of reference for the system specification. Instead, information is found in different places: in the domain engineers's experience, in the documented system specification and in the implementation of the system itself. All this information is neither complete nor consistent – in the end the domain engineers are needed to clarify all open questions.

Thus we used an incremental approach in the modeling process: the model was enhanced in small steps, and then tested against the real implementations or using the model checking capabilities of UPPAAL TRON. Running the model against the implementation is tedious, as the target hardware of the embedded application is often not readily accessible. In our case, the target hardware can only be accessed in the company's own test channel. Availability of the test channel is low, access is costly, and domain engineers are needed for operating the test channel. Note that this problem is typical of embedded system application, while for pure software solutions one can package the whole test harness in a virtual machine.

Instead of using the real target system, we have been able to use simulators of the target system. Some were pure software solutions, where others needed extra hardware (e.g. specific bus adapters). In both cases access to these simulators was effortless. We have been able to run the simulators within our own academic development environment instead of relying on the company's facilities. This increased the availability and also allows for test scenarios which are too difficult or even impossible in target environment, e.g disturbance tests.

In the end, we have been able to run a simple model against the target hardware. The model used for testing was restricted to a small part of functionality, which was basically sending commands to set certain signal patterns and receiving the appropriate responses. The timing aspects in the model were delays and upper bounds on message responses. No bugs were found. This is not surprising, since the signal controller and the signals have already reached a high level of maturity due to extensive use in the field. So, in order to check our test set, we injected some timimg and behavior errors into the SUT. All injected errors were found.

4 Future Work

Our current study was a proof-of-concept only. In further studies, we want to model larger parts of the system. The final goal would be a complete model of a railway station, but currently it is hard to say if this will be feasible within the given time frame. The next step is to take in more complex timing behavior of the system.

The goal is to be able to define a realistic process for model-based timed testing. A focus in our activities is the investigation of methods for defining re-usable models. This could lead to general, configurable test models similar to those found in software product line approaches. In the ideal case, it would be possible to provide domain specific model libraries.

References

1. J. Tretmans, "Test generation with inputs, outputs and repetitive quiescence," *Software - Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996.
2. J.-C. Fernandez, C. Jard, T. Jeron, and C. Viho, "Using on-the-fly verification techniques for the generation of test suites," in *Proc. CAV '96*, ser. LNCS, vol. 1102. Springer-Verlag, 1996, pp. 348–359.
3. E. Petitjean and H. Fouchal, "A realistic architecture for timed testing," in *ICECCS '99: Proceedings of the 5th International Conference on Engineering of Complex Computer Systems.* Washington, DC, USA: IEEE Computer Society, 1999, p. 109.
4. B. Nielsen and A. Skou, "Automated test generation from timed automata," in *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems.* London, UK: Springer-Verlag, 2001, pp. 343–357.
5. K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using uppaal-tron: an industrial case study," in *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software.* New York, NY, USA: ACM, 2005, pp. 299–306.
6. H. Bohnenkamp and A. Belinfante, "Timed testing with TorX," in *Proc. FME 2005*, ser. LNCS, vol. 3582. Springer-Verlag, 2005, pp. 173–188.
7. A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," in *Formal Methods and Testing*, 2008, pp. 77–117.
8. T. Müller, D. Graham, D. Friedenberg, and E. van Veedendal, "ISTQB certified tester - foundation level syllabus," pp. 15–16, 2007. [Online]. Available: `http://www.istqb.org/downloads/syllabi/SyllabusFoundation.pdf`

Present and Absent Sets: Abstraction for Data Intensive Systems Suited for Testing

Petur Olsen, Kim G. Larsen, Marius Mikučionis, and Arne Skou

Department of Computer Science
Aalborg University, Denmark
`{petur,kgl,marius,ask}@cs.aau.dk`

Abstract. This paper presents an abstraction of reactive systems interacting with relational databases. This abstraction is used as a modeling technique, enabling model-based testing of systems utilizing relational databases. The abstraction is developed to work on timed-automata models, in conjunction with Uppaal TRON. The abstraction substitutes each table in the database with two sets: *present* and *absent*. The present set provides an under-approximation of the set of relations which are present in the database, and the absent set is an under-approximation of the set of relations being absent from the database. It is expected that the approach makes data-intensive models more manageable in model checking and testing.

1 Introduction

Safety critical systems are becoming more and more complex. Often using databases to store huge amounts of data influencing the systems. It is therefore important to be able to model these databases in an efficient way. In this paper we propose a novel approach to modeling reactive systems interacting with databases with the purpose of doing model checking and model-based testing.

We consider systems interacting with a database in a shallow manner, meaning the system does not perform complex manipulation of the data. Rather, the system can only insert or remove relations from the database, and the control flow of the system can depend on the presence or absence of relations.

We propose an abstraction of the database using two sets: *present* and *absent*. These sets provide under-approximations of present and absent data respectively. This way we can abstract away from the actual content of the database and only be concerned with a relatively small subset of data.

2 Model-Based Testing

Model-based testing has it's roots in the formal approaches developed by Tretmans [4,5], and implemented in the tool TorX [6]. These approaches have been extended to include real-time by Hessel et al. [2], and implemented in Uppaal TRON [3].

Although the method in this paper does not concern real-time systems directly, we assume the same black-box conformance testing setup, harvest the inspiration from our previous experience in this area and aim at extending this framework to handle real-time systems that are also data intensive.

Uppaal TRON is an online conformance testing tool for real-time systems which assumes timed automata model as a test specification. The specifications unavoidably tend to be non-deterministic in timing and behavior. Non-deterministic specifications lead to lack of control over IUT during testing, where IUT is free to choose more than one option of how to respond and proceed. This poses a challenge to tester, which has to check the behavior against multiple response options, foresee and choose from multiple options for input actions. The problem of multiple choices is even more extreme in real-time cases where the number of timing choices is uncountably large.

The solution to non-deterministic requirements is adaptive testing, where the test is shaped like a tree with condition checks in nodes and input stimuli actions on edges. Such trees can be stored and executed offline or executed at the same time while being derived online (focusing only on relevant part of specification at a time.) In both online and offline model-based testing cases, the model state space is being explored and tests are generated based on current estimate of possible states that IUT can be in.

Uppaal TRON uses difference bound matrix (DBM [1]) structures and Uppaal algorithms to encode and operate on timed automata state estimates. In a similar spirit, we propose simple bit-vectors to abstract and encode the state estimate of relational database-like data in Uppaal timed automata models.

3 Abstracting Data Intensive Systems

We define the following sets:

$\mathbb{D}$ is a set of *elements* (e.g. records, relations, tuples etc.) The complete set of values that can be entered into the database.
$D_n \subset \mathbb{D}$ is the concrete *state* of a database. The database used by the real system and can contain huge amounts of data.
$\mathbb{C} \subseteq \mathbb{D}$ is a set of representative elements. These can be chosen intuitively or by some heuristic, e.g. a few from each table.
$P_n \subseteq \mathbb{C}$ is the *present set*, containing the elements known to be present in database D_n.
$A_n \subseteq \mathbb{C}$ is the *absent set*, containing the elements know to be absent from database D_n.
$d \in \mathbb{D}$ is an element in the actual system.
$c \in \mathbb{C}$ is an element in the abstract system.

Given these sets some interesting observations follow:

$P_n = \emptyset \wedge A_n = \emptyset$ means no knowledge about the contents of database D_n.
$P_n \cup A_n = \mathbb{C}$ means everything is known about database D_n, given the current $\mathbb{C}$.
$P_n \cap A_n = \emptyset$ must always hold. The same element can never be present in and absent from the same database at the same time.

The system can perform three operations: *Insert*, *remove*, and *query* for presence. Below we explain how operations are performed.

Insert:	$D_n = D_n \cup \{c\}$	$\Rightarrow$	$P_n = P_n \cup \{c\}$	$\wedge$	$A_n = A_n \setminus \{c\}$
Remove:	$D_n = D_n \setminus \{c\}$	$\Rightarrow$	$P_n = P_n \setminus \{c\}$	$\wedge$	$A_n = A_n \cup \{c\}$
Query with positive outcome:	$c \in D_n$	$\Rightarrow$	$P_n = P_n \cup \{c\}$	$\wedge$	*assert* $c \notin A_n$
Query with negative outcome:	$c \notin D_n$	$\Rightarrow$	$A_n = A_n \cup \{c\}$	$\wedge$	*assert* $c \notin P_n$

Theorem 1 *Operations on P_n and A_n are consistent and sound with respect to an actual D_n in the following sense:*

1. *The P_n and A_n captured info does not contradict.*
   (a) $\forall c \in D_n \Rightarrow c \notin A_n$
   (b) $\forall c \notin D_n \Rightarrow c \notin P_n$

2. *P_n and A_n capture part of the D_n state.*
   (a) $\forall c \in P_n \Rightarrow c \in D_n$
   (b) $\forall c \in A_n \Rightarrow c \notin D_n$

Sketch of a proof by induction: 1) take $P_n = A_n = \emptyset$ as the initial state – the properties hold trivially, 2) start with arbitrary sets $P_n \cap A_n = \emptyset$, 3) apply the operations with arbitrary element and show that the properties are preserved.

Example: Figure 1 illustrates a simple example of the usage of present/absent sets. The user performs a `login` action. The IUT has to respond with `OK` or `Error` depending on user status. The environment has cases for all combinations of present/absent. When the IUT sends an `OK` or `Error` the environment checks it's guards and performs updates. The `Add` and `Remove` methods in the model are used in case no prior knowledge is available. These will be used in a similar way if the system has to insert values into the database. A trace from the example can be used to populate the database to ensure a certain part of the model is reached, e.g. by setting a user to absent and not present will ensure the left-most edge will be taken.

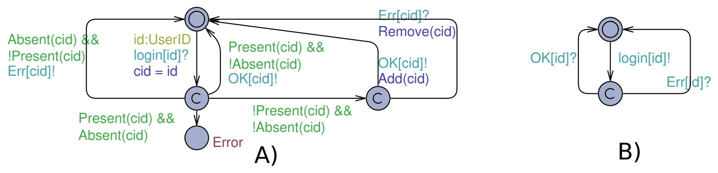


Fig. 1. Example. IUT is the database with present/absent lists (A), and the tester plays the role of environment/user (B).

4 Expected Results

It is expected that using this approach the data-intensive timed-automata models become more manageable during model checking as well as testing for several reasons. 1) The state-space of the model can be significantly reduced using this abstraction. This is accomplished by reducing the, potentially infinitely large, database into a set of relatively small lists. 2) in offline test generation, the proposed sets directly reveal what data items have to be present/absent in the database in order to fulfill the purpose of the test. 3) During online test, the state estimate size does not grow due to data uncertainty anymore and the precision is increasing as the test progresses.

References

1. Alexandre David. Uppaal dbm library. http://www.cs.auc.dk/~adavid/UDBM/, December 2006.
2. Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In *Formal Methods and Testing*, pages 77–117, 2008.
3. K.G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In *Formal Approaches to Testing of Software*, Linz, Austria, September 21 2004. Lecture Notes in Computer Science.
4. Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR*, pages 46–65, 1999.
5. Jan Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing*, pages 1–38, 2008.
6. Jan Tretmans and Ed Brinksma. Torx: Automated model based testing. In *Proceedings of the First European Conference on Model-Driven Software Engineering*, page 12 pp., 2003.

Aachener Informatik-Berichte

This list contains all technical reports published during the past five years. A complete list of reports dating back to 1987 is available from `http://aib.informatik.rwth-aachen.de/`. To obtain copies consult the above URL or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2004-01 *	Fachgruppe Informatik: Jahresbericht 2003
2004-02	Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic
2004-03	Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting
2004-04	Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming
2004-05	Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming
2004-06	Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming
2004-07	Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination
2004-08	Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information
2004-09	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity
2004-10	Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules
2005-01 *	Fachgruppe Informatik: Jahresbericht 2004
2005-02	Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: “Aachen Summer School Applied IT Security”
2005-03	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions
2005-04	Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem
2005-05	Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots
2005-06	Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information
2005-07	Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks
2005-08	Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut
2005-09	Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures
2005-10	Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11	Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture
2005-12	Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems
2005-13	Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments
2005-14	Felix C. Freiling, Sukumar Ghosh: Code Stabilization
2005-15	Uwe Naumann: The Complexity of Derivative Computation
2005-16	Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)
2005-17	Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)
2005-18	Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"
2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers
2005-20	Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21	Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited
2005-22	Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins
2005-23	Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves
2005-24	Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks
2006-01 *	Fachgruppe Informatik: Jahresbericht 2005
2006-02	Michael Weber: Parallel Algorithms for Verification of Large Systems
2006-03	Michael Maier, Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler
2006-04	Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulation
2006-05	Uwe Naumann, Jean Utke, Patrick Heimbach, Chris Hill, Derya Ozyurt, Carl Wunsch, Mike Fagan, Nathan Tallent, Michelle Strout: Adjoint Code by Source Transformation with OpenAD/F
2006-06	Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color
2006-07	Thomas Colcombet, Christof Löding: Transforming structures by set interpretations
2006-08	Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-09	Tingting Han, Joost-Pieter Katoen: Counterexamples in Probabilistic Model Checking
2006-10	Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed
2006-11	Bastian Schlich, Michael Rohrbach, Michael Weber, Stefan Kowalewski: Model Checking Software for Microcontrollers
2006-12	Benedikt Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker: Replaying Play in and Play out: Synthesis of Design Models from Scenarios by Learning
2006-13	Wong Karianto, Christof Löding: Unranked Tree Automata with Sibling Equalities and Disequalities
2006-14	Danilo Beuche, Andreas Birk, Heinrich Dreier, Andreas Fleischmann, Heidi Galle, Gerald Heller, Dirk Janzen, Isabel John, Ramin Tavakoli Kolagari, Thomas von der Maßen, Andreas Wolfram: Report of the GI Work Group "Requirements Management Tools for Product Line Engineering"
2006-15	Sebastian Ullrich, Jakob T. Valvoda, Torsten Kuhlen: Utilizing optical sensors from mice for new input devices
2006-16	Rafael Ballagas, Jan Borchers: Selexels: a Conceptual Framework for Pointing Devices with Low Expressiveness
2006-17	Eric Lee, Henning Kiel, Jan Borchers: Scrolling Through Time: Improving Interfaces for Searching and Navigating Continuous Audio Timelines
2007-01 *	Fachgruppe Informatik: Jahresbericht 2006
2007-02	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl: SAT Solving for Termination Analysis with Polynomial Interpretations
2007-03	Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp: Proving Termination by Bounded Increase
2007-04	Jan Buchholz, Eric Lee, Jonathan Klein, and Jan Borchers: coJIVE: A System to Support Collaborative Jazz Improvisation
2007-05	Uwe Naumann: On Optimal DAG Reversal
2007-06	Joost-Pieter Katoen, Thomas Noll, and Stefan Rieger: Verifying Concurrent List-Manipulating Programs by LTL Model Checking
2007-07	Alexander Nyßen, Horst Lichter: MeDUSA - MethoD for UML2-based Design of Embedded Software Applications
2007-08	Falk Salewski and Stefan Kowalewski: Achieving Highly Reliable Embedded Software: An empirical evaluation of different approaches
2007-09	Tina Kraußer, Heiko Mantel, and Henning Sudbrock: A Probabilistic Justification of the Combining Calculus under the Uniform Scheduler Assumption
2007-10	Martin Neuhäußer, Joost-Pieter Katoen: Bisimulation and Logical Preservation for Continuous-Time Markov Decision Processes
2007-11	Klaus Wehrle (editor): 6. Fachgespräch Sensornetzwerke
2007-12	Uwe Naumann: An L-Attributed Grammar for Adjoint Code

2007-13	Uwe Naumann, Michael Maier, Jan Riehme, and Bruce Christianson: Second-Order Adjoints by Source Code Manipulation of Numerical Programs
2007-14	Jean Utke, Uwe Naumann, Mike Fagan, Nathan Tallent, Michelle Strout, Patrick Heimbach, Chris Hill, and Carl Wunsch: OpenAD/F: A Modular, Open-Source Tool for Automatic Differentiation of Fortran Codes
2007-15	Volker Stolz: Temporal assertions for sequential and concurrent programs
2007-16	Sadeq Ali Makram, Mesut Güneç, Martin Wenig, Alexander Zimmermann: Adaptive Channel Assignment to Support QoS and Load Balancing for Wireless Mesh Networks
2007-17	René Thiemann: The DP Framework for Proving Termination of Term Rewriting
2007-18	Uwe Naumann: Call Tree Reversal is NP-Complete
2007-19	Jan Riehme, Andrea Walther, Jörg Stiller, Uwe Naumann: Adjoints for Time-Dependent Optimal Control
2007-20	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, and Verena Wolf: Three-Valued Abstraction for Probabilistic Systems
2007-21	Tingting Han, Joost-Pieter Katoen, and Alexandru Mereacre: Compositional Modeling and Minimization of Time-Inhomogeneous Markov Chains
2007-22	Heiner Ackermann, Paul W. Goldberg, Vahab S. Mirrokni, Heiko Röglin, and Berthold Vöcking: Uncoordinated Two-Sided Markets
2008-01 *	Fachgruppe Informatik: Jahresbericht 2007
2008-02	Henrik Bohnenkamp, Marielle Stoelinga: Quantitative Testing
2008-03	Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, Harald Zankl: Maximal Termination
2008-04	Uwe Naumann, Jan Riehme: Sensitivity Analysis in Sisyphe with the AD-Enabled NAGWare Fortran Compiler
2008-05	Frank G. Radmacher: An Automata Theoretic Approach to the Theory of Rational Tree Relations
2008-06	Uwe Naumann, Laurent Hascoet, Chris Hill, Paul Hovland, Jan Riehme, Jean Utke: A Framework for Proving Correctness of Adjoint Message Passing Programs
2008-07	Alexander Nyßen, Horst Lichter: The MeDUSA Reference Manual, Second Edition
2008-08	George B. Mertzios, Stavros D. Nikolopoulos: The λ-cluster Problem on Parameterized Interval Graphs
2008-09	George B. Mertzios, Walter Unger: An optimal algorithm for the k-fixed-endpoint path cover on proper interval graphs
2008-10	George B. Mertzios, Walter Unger: Preemptive Scheduling of Equal-Length Jobs in Polynomial Time
2008-11	George B. Mertzios: Fast Convergence of Routing Games with Splittable Flows
2008-12	Joost-Pieter Katoen, Daniel Klink, Martin Leucker, Verena Wolf: Abstraction for stochastic systems by Erlang's method of stages

2008-13	Beatriz Alarcón, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Raúl Gutiérrez, Salvador Lucas, Peter Schneider-Kamp, René Thiemann: Improving Context-Sensitive Dependency Pairs
2008-14	Bastian Schlich: Model Checking of Software for Microcontrollers
2008-15	Joachim Kneis, Alexander Langer, Peter Rossmanith: A New Algorithm for Finding Trees with Many Leaves
2008-16	Hendrik vom Lehn, Elias Weingärtner and Klaus Wehrle: Comparing recent network simulators: A performance evaluation study
2008-17	Peter Schneider-Kamp: Static Termination Analysis for Prolog using Term Rewriting and SAT Solving
2008-18	Falk Salewski: Empirical Evaluations of Safety-Critical Embedded Systems
2009-03	Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-04	Daniel Klünder: Entwurf eingebetteter Software mit abstrakten Zustandsmaschinen und Business Object Notation
2009-05	George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06	George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07	Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08	Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09	Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10	Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11	Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12	Martin Neuhäußer, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes
2009-13	Martin Zimmermann: Time-optimal Winning Strategies for Poset Games

[*] These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.