Pre-proceedings of the
14th International Workshop on
Automated Verification of Critical Systems

AVoCS'14
http://www.utwente.nl/avocs2014/

Marieke Huisman, Jaco van de Pol (Editors)

CTIT – Institute for ICT Research in Context
University of Twente
# Table of Contents

## Foreword

### Invited Talks

- **The GPUVerify Method: a Tutorial Overview**
  
  Alastair F. Donaldson

- **The highs and lows of deploying Formal Methods in Industry**
  
  Guy H. Broadfoot

- **Symbol Elimination for Automated Generation of Program Properties**
  
  Laura Kovács

## Regular Papers

- **Conformance Testing of Cyber-Physical Systems: A Comparative Study**
  
  Morteza Mohaqeqi, Mohammad Reza Mousavi, Walid Taha

- **Analyzing Conflict Freedom For Multi-threaded Programs With Time Annotations**
  
  Jingshu Chen, Marie Duflot, Stephan Merz

- **Verification of Information Flow Properties under Rational Observation**
  
  Béatrice Bérard and John Mullins

- **Using SMT for dealing with nondeterminism in ASM-based runtime verification**
  
  Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene

- **QBF with Soft Variables**
  
  Sven Reimer, Matthias Sauer, Paolo Marin, Bernd Becker

- **Exact and Approximate Abstraction for Classes of Stochastic Hybrid Systems**
  
  Jeremy Sproston

- **Reachability and Reward Checking for Stochastic Timed Automata**
  
  Ernst Moritz Hahn, Arnd Hartmanns and Holger Hermanns

- **On the Random Structure of Behavioural Transition Systems**
  
  Jan Friso Groote, Remco van der Hofstad, and Matthias Raffelsieper

- **A Formal Co-Simulation Approach for Wireless Sensor Network Development**
  
  Adisak Intana, Michael R. Poppleton, Geoff V. Merrett

- **Adaptive Task Automata with Earliest-Deadline-First Scheduling**
  
  Leo Hatvani, Alexandre David, Cristina Seceleanu, Paul Pettersson
Performance Analysis of Distributed and Asynchronous Systems using Probabilistic Timed Actors
Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, Holger Hermanns

Model Checking C++ with Exceptions
Petr Ročkai, Jiří Barnat, Luboš Brim

A Constraint-Solving Approach for Achieving Minimal-Reset Transition Coverage of Smartcard Behaviour
Renaud De Landtsheer, Christophe Ponsard, Nicolas Devos

Research Ideas

Towards the Verification of Bidirectional Railway Models in CSP∥B
Phillip James, Faron Moller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, Helen Treharne

Efficacy Measurement of Early Intervention Techniques
Dave Donaghy, Tom Crick

No-Test Classes in C through Restricted Types
Dave Donaghy, Tom Crick

Physical Type Tracking through Minimal Source-Code Annotation
Dave Donaghy, Tom Crick

Towards a General Approach for Symbolic Model-Checker Prototyping
Edmundo López Bóbeda, Maximilien Colange, Didier Buchs

Formal Verification of the Danish Railway Interlocking Systems
Linh H. Vu, Anne E. Haxthausen, and Jan Peleska

Verification of Fix Protocol Session Layer
Kamil Kuboň

Delta-Oriented Testing for Finite State Machines
Mahsa Varshosaz and Harsh Beohar

Interactive Verification of Hybrid Systems
Xian Li, Klaus Schneider
Foreword

AVoCS 2014, the 14th International Workshop on Automated Verification of Critical Systems has been hosted by the University of Twente, and has taken place in Enschede, Netherlands, on 24th–26th September 2014.

The workshop programme consisted of contributed papers and three keynote speakers. We are grateful to Alastair Donaldson, Imperial College London, Guy Broadfoot, and Laura Kovács, Chalmers University of Technology, for accepting our invitation to address the workshop. Out of 25 submitted papers, 13 papers have been accepted (i.e., an acceptance rate of 50%) as full papers to be presented during the workshop, and to be published in the AVoCS post-proceedings as part of EASST series. In addition, also 9 research ideas will be presented during the workshop. These research ideas are part of these pre-proceedings.

We would like to thank the Steering Committee for the opportunity to host the workshop at the University of Twente. We also would like to thank the Programme Committee and external reviewers for their efforts in reviewing the contributed papers in order to ensure a high quality workshop.

Further, we also thank Holger Schlingloff and Markus Roggenbach, for organising the Model-based design and analysis of cyber-physical systems SPES XT Summer School, co-located with AVoCS 2014.

We are grateful to Formal Methods Europe for providing us the financial means to waive the registration fee for several students participating in the workshop. We are also grateful to the Aspasia Fund of the University of Twente for financially supporting the travel and stay of Laura Kovács. Finally, we are grateful to CTIT for providing a guarantee subsidy.

Finally, special thanks go to Joke Lammerink and Wojciech Mostowski for their efforts in helping to make the workshop a success.

Workshop Programme Chairs

September 2014

University of Twente

AVoCS 2014
Programme Committee

- John Derrick, University of Sheffield, U.K.
- Mike Dodds, University of York, U.K.
- Wan Fokkink, Vrije Universiteit Amsterdam, Netherlands
- Michael Goldsmith, University of Oxford, U.K.
- Jan Friso Groote, Eindhoven University of Technology, Netherlands
- Anne Haxthausen, Technical University of Denmark, Denmark
- Keijo Heljanko, Aalto University, Finland
- Marieke Huisman, University of Twente, Netherlands (co-chair)
- Temesghen Kahsai, NASA Ames Research Center, U.S.A.
- Gerald Luettgen, University of Bamberg, Germany
- Radu Mateescu, INRIA Grenoble – Rhône-Alpes, France
- Stephan Merz, INRIA Nancy & LORIA, France
- Markus Roggenbach, Swansea University, U.K.
- Marco Roveri, FBK-irst, Italy
- Thomas Santen, Microsoft Research, Germany
- Bernard Steffen, Technical University Dortmund, Germany
- Jan Strejček, Masaryk University, Czech Republic
- Jun Sun, Singapore University of Technology and Design, Singapore
- Tayssir Touili, LIAFA, CNRS & University Paris Diderot, France
- Helen Treharne, University of Surrey, U.K.
- Jaco van de Pol, University of Twente, Netherlands (co-chair)
- Laurent Voisin, Systerel, France
- Kirsten Winter, University of Queensland, Australia

Additional Reviewers

- František Blahoudek
- Pieter Cuijpers
- Sascha Fendrich
- Alberto Griggio
- Matthias Güdemann
- Alexander Heuñner
- Falk Howar
- Jeremy Jacob
- Phillip James
- Fatma Jebali
- Sergio Mover
- Johannes Neubauer
- Christopher M. Poskitt
- Olli Saarikivi
- Wendelin Serwe
- Antti Tapani Siirtola
- Marek Trtík
- Wieger Wesselink
- David White
- Stephan Windmüller

Steering Committee

- Michael Goldsmith, University of Oxford, U.K.
- Stephan Merz, INRIA Nancy & LORIA, France
- Markus Roggenbach, Swansea University, U.K.

Local Organisers

- Marieke Huisman
- Wojciech Mostowski (publicity chair)
- Jaco van de Pol
- Joke Lammerink
Invited Talks
The GPUVerify Method: a Tutorial Overview

Alastair F. Donaldson

Imperial College London

Abstract: I present a tutorial overview demonstrating the key technique used by GPUVerify, a static verification tool for graphics processing unit (GPU) kernels. The technique is a method for translating a massively parallel GPU kernel into a sequential program such that correctness of the sequential program implies data race-freedom of the parallel kernel.

Keywords: Formal verification, graphics processing units, predicated execution, concurrency

1 Introduction

GPUVerify is a formal verification technique and tool for the analysis of GPU kernels—programs designed to be executed in parallel on graphics processing units—with respect to two types of defects: data races and barrier divergence [BCD+12, BBC+14]. In this invited paper I present a tutorial overview of the key step undertaken by GPUVerify to allow scalable verification: the translation of a parallel GPU kernel into a sequential program such that analysis of the sequential program yields results about the original parallel kernel. The material presented here has been developed in the process of giving a number of seminars on GPUVerify, teaching about GPUVerify on the Software Reliability course at Imperial College London, and recording videos presenting the project. My hope is that this exposition provides a clearer overview of how the method works compared with the technical presentation in the original GPUVerify paper [BCD+12].

After providing some background on GPU programming (Section 2), the bulk of the tutorial focuses on the key steps involved in the GPUVerify verification method (Section 3). This is followed by a brief overview of the GPUVerify tool chain (Section 4). The paper concludes with a survey of some related work in GPU kernel analysis and verification (Section 5).

2 Background on GPU programming

Originally designed to accelerate graphics processing, a graphics processing unit (GPU) has many parallel processing elements: graphics operations are inherently parallel. Early GPUs had limited functionality, tailored specifically towards graphics computations. Recently GPU designs have become more powerful and general purpose, and are now widely used in parallel programming to accelerate tasks including (among many others, and citing only a small subset of works): medical imaging [CLW04], computer vision [SNS+13], computational fluid dynamics [Har04] and DNA sequence alignment [LD09].

1 GPUVerify can be accessed at http://multicore.doc.ic.ac.uk/tools/gpuverify.
2 Introductory video on GPUVerify: https://www.youtube.com/watch?v=l8ysBPV80vA
Figure 1: Overview of the structure of a typical GPU architecture.

Figure 1 shows the structure of a typical GPU architecture (akin in essence to state-of-the-art architectures from NVIDIA and AMD). The chip consists of a number of processing elements (PEs) each equipped with a small amount of private memory. PEs are organised into groups such that PEs within a group share memory (called local memory in this paper). The set of groups of PEs is sometimes referred to as a grid. The GPU is also equipped with global memory shared among all PEs. In some implementations this global memory is physically separate from main memory; in other implementations global memory is part of the same physical memory.

Data races  A data race occurs in a GPU kernel if:

- Two distinct threads access the same memory location
- At least one access is a write
- The accesses are not separated by a barrier synchronisation operation

Races in GPU kernels can be classified as inter-group or intra-group. Inter-group races are between threads executing on PEs in distinct groups. Necessarily, such races must occur with respect to global memory since this is the only memory that threads on distinct PEs can share. Intra-group races are between threads executing on PEs in the same group, and can be with respect to either global memory or the group’s local memory.

Data races in GPU kernels are usually indicative of programmer errors, and lead to nondeterministic bugs that can be hard to reproduce and fix. A problem which is specific to data races in GPU kernels is that individual GPU architectures may be relatively deterministic: as there is no operating system running on the GPU, GPU threads are not preempted at unpredictable moments as is the case in concurrent CPU applications. This means that a data race may resolve deterministically and in an apparently bug-free manner on a particular GPU architecture, thus evading detection through testing, and then resolve differently, causing a crash or erroneous results, when an application is deployed on another platform, perhaps in a customer device. Unlike with data races in system-level CPU applications, which are often deliberate or regarded as be-
void add_neighbour(local int * A, int offset) {
}

Figure 2: A simple GPU kernel that exhibits a data race.

Example GPU kernel

GPUs are typically programmed by writing a kernel—a function to be executed simultaneously by many threads running across the processing elements of the chip—using a low-level programming model such as the industry standard Open Computing Language (OpenCL) [Khr12], or the NVIDIA-specific Compute Unified Device Architecture (CUDA) [NV11]. Figure 2 shows a simple GPU kernel written in OpenCL C (a superset of a subset of the C99 language). The kernel keyword indicates that the add_neighbour function is a kernel entry point. The kernel takes two arguments: $A$, which is a pointer to an array of integers residing in local memory (indicated by the local keyword), and $offset$, an integer. Every thread executing the kernel runs the add_neighbour function, receiving identical values for the $A$ and $offset$ parameters.

In order to access distinct data values, a thread can use a built-in variable, $tid$, providing access to the thread’s unique identifier. A thread executing this kernel reads from $A$ at offsets $tid$ and $tid + offset$, sums these values and writes the result to $A$ at offset $tid$. This example illustrates a read-write data race. For example, if $offset$ is 1 then thread 0 will read from $A$ at offset 1, thread 1 will write to $A$ at offset 1, and there is no barrier synchronisation operation separating these accesses.

Barrier synchronisation and barrier divergence

A barrier statement is used to synchronise threads in the same group: when a thread reaches a barrier it waits until all threads reach the barrier. When all threads have reached the barrier, the threads can proceed past the barrier, with the guarantee that reads and writes issued before the barrier have completed.

Barriers only allow synchronisation between threads in the same group; inter-group synchronisation within a kernel invocation is not possible.

The OpenCL 1.2 specification [Khr12] requires that threads should synchronise at syntactically identical barriers, making the following illegal:

```c
if (tid == 0) barrier();
else barrier();
```

3 In OpenCL, the `get_local_id` and `get_global_id` functions are in fact used to retrieve a thread’s id within its work group and across all work groups, respectively. Because groups and grids of groups can be multi-dimensional these functions take arguments specifying in which dimension the id is required. For ease of presentation, in this paper multi-dimensional kernels are not considered, and we restrict attention to the case where there is a single group of threads, using $tid$ to denote the id of a thread within this group. The GPUVerify tool supports reasoning about multi-dimensional kernels executed by multiple groups of threads.
The GPUVerify Method: a Tutorial Overview

kernel void add_neighbour(local int * A, int offset) {
    int temp = A[tid + offset];
    barrier();
}

Figure 3: Using a barrier to avoid the data race exhibited by the example of Figure 2.

Furthermore, if a barrier is inside a loop then threads should hit the barrier with the same loop trip count. See [BCD+12] for discussion of an example which violates this requirement.

3 The GPUVerify kernel transformation method

The verification method employed by GPUVerify exploits the OpenCL programming model to transform a massively parallel kernel $K$ into a sequential program $P$ such that:

$$P \text{ is correct (i.e. free from assertion failures)} \Rightarrow K \text{ is free from data races and barrier divergence.}$$

At the conceptual level, the approach has four main ingredients:

1. Race analysis focusses on barrier intervals
2. Analysis is restricted to consider a single, canonical thread schedule, avoiding the need to reason about a large number of schedules
3. Analysis is further restricted to consider an arbitrary pair of threads executing the kernel, using abstraction to model the effects of further threads; this avoids the need to reason simultaneously about a large number of thread schedules
4. Predicated execution is applied to handle loops and conditionals, and to precisely capture the conditions for barrier divergence

Ingredients similar to 1–3 have been employed in other work on GPU kernel analysis [LG10, CCK14, LLS+12], while ingredient 4 is a novel contribution of the GPUVerify project [BCD+12]. Collectively, they allow kernel verification to be explicitly reduced to the analysis of a sequential program, allowing existing technology for sequential verification to be re-used. Careful re-use of sequential verification technology is a distinguishing characteristic of the GPUVerify method in comparison to related work (see Section 5).

Sections 3.1–3.3 elaborate on ingredients 1–3, respectively. Section 3.4 shows how these ideas can be used to perform transformation into a sequential program for kernels that do not exhibit conditional or looping code. Section 3.5 explains how conditionals and loops are handled using predicated execution.

Throughout, for ease of presentation, attention is restricted to the detection of intra-group data races; the implementation in GPUVerify (Section 4) implements inter-group race checking in full.
Figure 4: Illustration of a barrier interval. It is sufficient to restrict intra-group race analysis to pairs of instructions that lie within a common barrier interval.

3.1 Focusing race detection to barrier intervals

Observe that at any point during execution of a GPU kernel, two threads in the same group must be executing instructions that lie in the same barrier interval: a sequence of instructions starting and ending with a barrier, but otherwise barrier-free.

The notion of a barrier interval is illustrated in Figure 4. In practice, a barrier interval may be more complex; for example, a barrier occurring inside a loop may form a barrier interval with itself. Arbitrary barrier intervals can be handled by:

- Logging and checking all accesses made by threads to the shared state, aborting if a data race is detected
- Resetting the access logs each time a barrier is reached

3.2 Restricting to a canonical thread schedule

Restricting analysis to barrier intervals has the potential to increase scalability, as it allows a kernel with multiple barriers to be analysed interval-by-interval. However, if there are $n$ threads and a barrier interval contains $k$ instructions then, assuming arbitrary thread interleavings can occur, there are $O(n^k)$ threads schedules. Exploring all such thread schedules to detect data races would not be feasible.

The next observation is that if checking for data races occurring during the barrier interval is performed for some fixed schedule then either a data race will be detected (in which case verification can abort with an error), or the schedule will be shown to exhibit no data races, in which case it can be concluded that no schedule between these barriers can exhibit a data race. This observation has been exploited in several works [LG10, CCK14, LLS+12, BCD+12] and a proof that the reasoning is sound is presented in [LLS+12]. Informally, the argument is that for a barrier interval there exist a set of earliest races: a race is an earliest race if there exists some thread schedule such that the race is the first race to be observed during the schedule. If a schedule exhibits a data race then it must exhibit an earliest race, and a little reasoning shows that, for a thread schedule $\sigma$ associated with barrier interval $I$:
σ exhibits an earliest race
⇔
every schedule for barrier interval I exhibits an earliest race

Considering an arbitrary canonical schedule thus suffices for race analysis.
Restricting race analysis to a single schedule has two related and significant advantages. For a barrier interval of length $k$ executed by $n$ threads:

- The number of schedules that need to be considered is reduced from $O(n^k)$ to 1.
- The chosen schedule can be used to rewrite the barrier interval as a sequential program consisting of $n \cdot k$ instructions, one for each thread, plus instructions to perform race logging and checking. The instructions appear in the order dictated by the schedule.

### 3.3 The two thread abstraction

Restriction to a canonical schedule has the benefit of bringing the analysis task into the realm of sequential program verification, on which steady progress has been made now for several decades. However, verifying a barrier interval via a sequential program of length $n \cdot k$, where $n$ is the number of threads and $k$ the length of the barrier interval, is problematic if $n$ is large. Because GPU kernels are often executed by thousands of threads, this is an issue in practice.

This sensitivity to the number of threads can be avoided by exploiting the fact that data races and barrier divergence are both defects that occur between pairs of threads: a data race occurs due to a conflict between precisely two threads; similarly barrier divergence occurs when two threads reach distinct barriers (or hit the same barrier having executed different numbers of iterations of loops enclosing the barrier).

Thus, when checking correctness of a barrier interval, it is sufficient to check the interval for every pair of threads separately. This can be achieved by checking that the barrier interval is free from races and divergence for an arbitrary pair of distinct threads.

If a data race exists then, as argued above, there must exist an earliest race between some pair of threads, $s$ and $t$ say. Attempting to check race-freedom for an arbitrary pair of threads $i$ and $j$ must include reasoning about the pair $s$ and $t$, which will lead to discovery of the earliest race between $s$ and $t$.

Care must be taken to extend this trick to the analysis of multiple barrier regions. After threads synchronise at a barrier, it is legitimate for a thread’s execution to depend on a value computed by a different thread before the barrier. If just a pair of threads, $i, j$ say, are modelled, then the effects across barriers of a thread $k$ different from $i$ and $j$ will not be accurately represented.

This can be accounted for by abstracting the shared state, i.e. local and global memory. The simplest approach, termed adversarial abstraction [BCD+12], is to treat the shared state as being completely abstract, so that each time a thread reads from the shared state an arbitrary value is retrieved; in this case writes to the shared state need not be modelled at all. A slightly richer abstraction, the equality abstraction, is discussed and compared with the adversarial abstraction in [BCD+12], and barrier invariants, a tunable shared state abstraction technique for establishing richer properties, are proposed in [CDK+13]. Adversarial abstraction is considered throughout this paper.
3.4 Transformation for straight-line kernels

The program transformation performed by GPUVerify is now explained for the case of single procedure straight-line kernels. Handling of conditionals and loops is described in Section 3.5.

Assume that a kernel has the following form:

```
kernel void foo(<parameters, including local arrays>) {
    <private variable declarations>
    S_1; S_2; ... S_k;
}
```

where each statement $S_i$ has one of the following forms, where $x$ denotes a private variable, $e$ an expression over private variables, and $A$ a local array:

- $x = e$ (private assignment)
- $x = A[e]$ (read from local array)
- $A[e] = x$ (write to local array)
- `barrier()` (barrier statement)

These assumptions ensure that a statement includes at most one read from local memory and at most one store to local memory. Pre-processing can be used to trivially transform statements that perform multiple loads and stores into this form. Furthermore, this is typical of the way a kernel is represented after compilation into a compiler intermediate representation such as LLVM bytecode.

Assume also that all local array parameters to the kernel refer to disjoint arrays.

The aim is to transform a kernel $K$ into a sequential program $P$ that:

- Captures execution of two arbitrary threads using some fixed schedule
- Detects data races
- Treats the shared state abstractly to model the effects of other threads

**Introduction of distinct thread ids** Two symbolic constants are introduced in $P$, tid$1$ and tid$2$, to represent the ids of two distinct but otherwise arbitrary threads. These conditions are encoded by the following preconditions on $P$, there $n$ denotes the number of threads executing the kernel:

```
0 <= tid$1$ && tid$1$ < n
0 <= tid$2$ && tid$2$ < n
tid$1$ != tid$2$
```

The first and second conditions require that the ids of the threads under consideration are within the valid range of thread ids. The third condition requires that the ids are distinct.

In what follows, “thread 1” or “the first thread” refers to the thread whose id is recorded by tid$1$, and “thread 2” or “the second thread” refers to the thread whose id is recorded by tid$2$. It is important to note that this does not refer specifically to the threads whose ids are 1 and 2.
Removal of array parameters  If $K$ has a local array parameter then this parameter does not appear in $P$. This is because the shared state of $K$ will be represented abstractly in $P$ by eliding writes, and replacing a reads into a private variable $x$ with a non-deterministic assignment to $x$. After this abstraction, local arrays have no role.

Dualisation of non-array parameters  If $K$ has a non-array parameter, $a$ of type $T$ say, then $P$ has two non-array parameters, $a\$1 and $a\$2, both with type $T$. Parameter $a\$1 represents the first thread’s copy of the original parameter $a$, and likewise $a\$2 represents the second thread’s copy. These parameters initially have identical values, enforced by the precondition $a\$1 == $a\$2.

Dualisation of private variables  If $K$ has a private, function-scope variable $x$ of type $T$ then $P$ has two private, function-scope variables, $x\$1 and $x\$2, each of type $T$. Because private variables initially have arbitrary values which may be distinct between threads there is no precondition relating $x\$1 and $x\$2.

Race logging and checking routines  For each local array $A$ in $K$, $P$ is equipped with two sets of integers, $R_A$ and $W_A$, which record reads from $A$ and writes to $A$ respectively. A precondition ensures that initially both $R_A$ and $W_A$ are empty.

The program $P$ is also equipped with four procedures:

- **LOG\_READ\_A**: takes an integer parameter and adds the parameter’s value to $R_A$
- **LOG\_WRITE\_A**: takes an integer parameter and adds the parameter’s value to $W_A$
- **CHECK\_READ\_A**: takes an integer parameter and aborts if the parameter’s value belongs to $W_A$
- **CHECK\_WRITE\_A**: takes an integer parameter and aborts if the parameter’s value belongs to either $R_A$ or $W_A$

In addition, $P$ is equipped with a **barrier** procedure, which has the effect of setting $R_A$ and $W_A$ to the empty set for every local array $A$.

In this paper a discussion of how the sets $R_A$ and $W_A$, the associated **LOG** and **CHECK** procedures, and the **barrier** procedure are efficiently implemented is omitted. Two alternative implementations, both of which avoid the use of quantifiers, have been presented in prior work [BCD+12, BBC+14].

Translation of statements  For an expression $e$ over private variables and $i \in \{1, 2\}$, we use $e\$i to denote the expression $e$ with every occurrence of a private variable $x$ replaced with $x\$i$. One can read $e\$i as ”$e$ in the context of thread $i$”. For instance, if $e$ is $a + \text{tid} - x$ then $e\$2 is $a\$2 + \text{tid}\$2 - x\$2$.

Table 1 shows how the forms of statement for a straight line kernel $K$ are translated into two-threaded form in the program $P$. Assignments and barriers are straightforward.

A read is translated as a call to the appropriate **LOG\_READ** procedure with thread 1’s offset, and a call to the appropriate **CHECK\_READ** procedure with thread 2’s offset. Thus thread 1 takes
Each thread executes the assignment

<table>
<thead>
<tr>
<th>Statement in K</th>
<th>Corresponding statement in P</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = c; \quad x$1 = $c$1; $x$2 = $c$2;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x = A[e]; \quad \text{LOG}_R A(e$1); \quad \text{CHECK}_R A(e$2); \quad \text{havoc}(x$1); \quad \text{havoc}(x$2);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A[e] = x; \quad \text{LOG}_W A(e$1); \quad \text{CHECK}_W A(e$2);</td>
<td></td>
<td></td>
</tr>
<tr>
<td>barrier();</td>
<td>barrier();</td>
<td>Barrier clears every $R_A$ and $W_A$</td>
</tr>
</tbody>
</table>

Table 1: Translation of kernel statements into sequential program statements, in the absence of conditionals and loops.

Figure 5: Illustration of the transformation process for a simple kernel. The parallel OpenCL kernel on the left is transformed into the sequential program on the right.

Due to shared state abstraction, there is no need to reflect the actual array write in $P$, because the array in question is not present.

A complete example of the translation process is shown in Figure 5.
3.5 Handling conditionals and loops using predicated execution

Loops and conditionals are handled using predicated execution. The essence of predicated execution is to flatten conditional code into straight line code. For example, the following fragment of C code:

```c
if (x < 100) {
    x = x + 1;
} else {
    y = y + 1;
}
```

can be flattened into straight line code through the introduction of two predicates, $P$ and $Q$, where $P$ records the truth of $x < 100$ and $Q$ the truth of $!(x < 100)$, as follows:

```c
P = (x < 100);
Q = !(x < 100);
x = P ? x + 1 : x;
y = Q ? y + 1 : y;
```

One can easily verify that, in this form, the code fragment computes the same result as the original. In effect, both sides of the conditional statement are executed, but the then side has no effect if $x < 100$ does not hold (in which case $P$ is false) and the else side has no effect if $x < 100$ does hold (in which case $Q$ is false).

Predication is used in the two-thread sequential encoding of a GPU kernel by flattening conditionals so that both threads execute both sides of every conditional, and modifying loops so that both threads execute the same number of iterations of each loop, but maintaining at all times a predicate for each thread determining whether the thread is enabled. If a thread is not enabled, it executes statements but these statements have no effect.

First, the the LOG and CHECK procedures introduced in Section 3.4 are adapted so that each is equipped with a Boolean predicate parameter.

- **LOG_READ_A**: takes a Boolean predicate and an integer parameter; adds the parameter’s value to $R_A$ if and only if the predicate holds
- **LOG_WRITE_A**: takes a Boolean predicate and an integer parameter; adds the parameter’s value to $W_A$ if and only if the predicate holds
- **CHECK_READ_A**: takes a Boolean predicate and an integer parameter; aborts if and only if the predicate holds and the parameter’s value belongs to $W_A$
- **CHECK_WRITE_A**: takes a Boolean predicate and an integer parameter; aborts if and only if the predicate holds and the parameter’s value belongs to either $R_A$ or $W_A$

The barrier procedure of $P$ is also adapted so that it additionally takes two predicate parameters, one for each thread. Execution of barrier aborts if and only if the predicate parameters differ. Otherwise the procedure sets $R_A$ and $W_A$ to the empty set if and only if both predicate parameters are true, leaving $R_A$ and $W_A$ untouched otherwise.

The rules of Table 2 show how a statement Stmt of $K$ is translated with respect to predicate $E$ to a statement translate(Stmt, $E$) in $P$. The predicate $E$ determines whether each of the threads
Table 2: Complete rules for translation of kernel statements into sequential program statements, using predication to handle loops and conditionals. The top-level program statement sequence is translated with respect to the predicate true.

<table>
<thead>
<tr>
<th>Statement in K</th>
<th>Corresponding statement in P, translate(Stmt,E)</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stmt = e;</td>
<td>x$1 = E$1 \iff e$1 : x$1; x$2 = E$2 \iff e$2 : x$2;</td>
<td>Each thread executes the assignment if its predicate holds, otherwise the thread performs a no-op.</td>
</tr>
<tr>
<td>Stmt = A[e];</td>
<td>LOG_READ_A(E$1, e$1); CHECK_READ_A(E$2, e$2); havoc(myp$1); havoc(myp$2); x$1 = E$1 \iff myp$1 : x$1; x$2 = E$2 \iff myp$2 : x$2;</td>
<td>Thread 1 logs the read if its predicate holds. Each thread chooses an arbitrary value, then sets its copy of x to the arbitrary value if its predicate holds and performs a no-op otherwise.</td>
</tr>
<tr>
<td>A[e] = x;</td>
<td>LOG_WRITE_A(E$1, e$1); CHECK_WRITE_A(E$2, e$2);</td>
<td>Thread 1 logs the write if its predicate holds. Thread 2 checks the write if its predicate holds. Because A has been removed, the write itself is not modelled in P.</td>
</tr>
<tr>
<td>barrier();</td>
<td>barrier(E$1, E$2);</td>
<td>If both predicates hold then barrier clears every R$A$ and W$A$.</td>
</tr>
<tr>
<td>if Stmt$_1$</td>
<td>translate(Stmt$_1$, E); translate(Stmt$_2$, E);</td>
<td>Statements in a sequence are translated one by one.</td>
</tr>
<tr>
<td>else Stmt$_2$</td>
<td>translate(Stmt$_1$, F); translate(Stmt$_2$, G);</td>
<td>The then branch, Stmt$_1$, is translated w.r.t. F, and the else branch, Stmt$_2$, w.r.t. G.</td>
</tr>
<tr>
<td>while Stmt$_1$</td>
<td>F$1 = E$1 \iff e$1; F$2 = E$2 \iff e$2;</td>
<td>Each thread sets fresh predicates $F$ and $G$ to the values of $e$ and $\neg e$ respectively if enclosing predicate $E$ holds, and to false otherwise.</td>
</tr>
<tr>
<td>do whileStmt$_2$</td>
<td>translate(Stmt$_1$, F); F$1 = E$1 \iff e$1; F$2 = E$2 \iff e$2;</td>
<td>Each thread sets fresh predicate $F$ to the value of loop condition $e$ if enclosing predicate $E$ holds, and to false otherwise.</td>
</tr>
<tr>
<td>while Stmt$_1$</td>
<td>translate(Stmt$_1$, F); F$1 = E$1 \iff e$1; F$2 = E$2 \iff e$2;</td>
<td>Threads loop until both are disabled, but a thread performs no-ops if its predicate $F$ is false. The predicate is re-evaluated on taking the loop back-edge.</td>
</tr>
</tbody>
</table>

is enabled during execution of the statement. The top-level statement sequence of the kernel is translated with respect to the predicate true because initially both threads are enabled. The rules for conditionals and loops introduce stronger predicates on enabledness.

The rules for assignment, read and write mirror the straight-line rules of Table 1. The difference is that all components of the translated statement are guarded by the predicate $E$, so that if $E$ does not hold for one of the threads then the thread performs no-ops. In the translation of reads, this difference requires the introduction of a special $tmp$ variable in each thread which is used to temporarily store the nondeterministic result of a read. The value stored in $tmp$ is then conditionally copied into receiving variable $x$ depending on whether predicate $E$ holds.

A barrier is transformed into a call to the version of barrier that accepts predicate arguments. If both predicates are true then barrier does its usual job of clearing all read and write sets; if both predicates are false then barrier has no effect. If the predicates $E$ and $ES2$ disagree then execution aborts because barrier divergence has been detected.

Sequences of statements are translated in the obvious way. Conditionals are handled by introducing fresh predicates to record enabledness for each side of the conditional: if enclosing predicate $E$ is false then both new predicates are also false, otherwise predicate $F$ is set to the
truth of the condition $e$, and $G$ to its negation. Each side of the conditional is translated according to the appropriate predicate, and the conditional itself is eliminated.

Arguably the most interesting translation rule is the rule for loops. Unlike in the case for conditionals, predication does not eliminate loops, but rather transforms a loop into a form where each thread is guaranteed to execute the same number of iterations, so that in the generated sequential program the threads being modelled can enter and leave the loop uniformly. This is achieved by using a fresh predicate $F$ to record whether each thread is still enabled during loop execution. This predicate is set to false if the enclosing predicate $E$ does not hold, and otherwise is set to the value of the loop condition. The loop iterates until $F$ becomes false for both threads. While $F$ remains true for one of the threads, both threads execute the loop body, but their execution is predicated with respect to $F$. This means that if one of the threads has become disabled, execution of the body by this thread has no effect. On iterating the loop, $F$ is recomputed: it is strengthened by the current value of the loop guard. Notice that once $F$ has been set to false it cannot become true, thus once a thread has become disabled the thread cannot become enabled until loop exit.

4 Implementation in GPUVerify tool

The transformation described in this tutorial is at the heart of the GPUVerify tool. The architecture of GPUVerify is illustrated in Figure 6. As input, GPUVerify accepts a kernel written in CUDA or OpenCL. A custom-built front-end based on the Clang/LLVM framework parses the kernel into an intermediate form which is fed to the kernel transformation engine. The transformation engine implements the transformation of Section 3 and applies race instrumentation as described in [BCD+12, BBC+14], generating a sequential program expressed in the Boogie intermediate verification language [Lei08]. The tool also generates candidate loop invariants using heuristics described in [BCD+12]. Verification of the sequential program is then delegated to the Boogie verification engine [BCD+05], an open source verifier developed primarily by Microsoft.

4 http://llvm.org/
Research.\textsuperscript{5} Boogie uses the Houdini algorithm [FL01] to perform invariant inference over the candidate loop invariants and discharges verification conditions to an SMT solver. GPUVerify supports the Z3 [MB08] and CVC4 solvers [BCD+11].

The architecture of Figure 6 shows that GPUVerify exhibits significant re-use: advantage is taken of Clang/LLVM, Boogie and Z3, which are widely used, robust components actively developed by expert research teams. This has significantly reduced the implementation effort required to make GPUVerify work, as well as increasing the reliability of the tool by virtue of other researchers and users continually improving the 3rd party components.

Two large experimental evaluations have demonstrated the scalability of GPUVerify, especially in terms of the number of threads that can be considered when verifying a kernel [BCD++12, BBC++14]. Notable aspects of the engineering effort associated with getting GPUVerify to work on realistic examples are also discussed in [BBC++14].

5 Related work

Extensions to and applications of GPUVerify The core GPUVerify method was presented in [BCD++12], with details on how to support unstructured control flow graphs, key to re-use of the Clang/LLVM infrastructure (see Figure 6), presented in [CDKQ13]. A richer shared state abstraction based on barrier invariants has been proposed and implemented in the tool [CDK++13]. GPUVerify has also been extended to support for warp-based execution and specific use cases for atomic operations [BD14]. Engineering details associated with the project have appeared [BBC++14], and the tool forms a key component in an automated verification method for parallel prefix sums [CDK14].

Other verification methods The closest work to the GPUVerify method is the PUG analyser for CUDA kernels [LG10]. Although GPUVerify and PUG have a similar goal, scalable verification of race-freedom for GPU kernels, the internal architecture of the two systems is very different. GPUVerify first translates a kernel into a sequential Boogie program that models the lock-step execution of two threads; the correctness of this program implies race- and divergence-freedom of the original kernel. Next, it infers and uses invariants to prove the correctness of this sequential program. Therefore it is only necessary to argue soundness for the translation into a sequential program; the soundness of the verification of the sequential program follows directly from the soundness of contract-based verification. On the other hand, PUG performs invariant inference simultaneously with translation of the GPU kernel into a logical formula. PUG provides a set of built-in loop summarisation rules which replace loops exhibiting certain shared array access patterns with corresponding invariants. Unlike GPUVerify, which must prove or discard all invariants that it generates, the loop invariants inserted by PUG are assumed to be correct. While this approach works for simple loop patterns, it has difficulty scaling to general nested loops in a sound way resulting in various restrictions on the input program required by PUG [LG10]. In contrast, GPUVerify inherits flexible and sound invariant inference from the Houdini invariant inference algorithm [FL01] regardless of the complexity of the control structure of the GPU kernel.

\footnote{http://boogie.codeplex.com/}
A verification approach for CUDA kernels [LGA+12] uses dynamic analysis to find data races at runtime. Then, if no data races are found, static analysis is used to determine whether control flow decisions at runtime were input-dependent. If not, the kernel is guaranteed to be data race-free. This method is in principle highly automatic for verifying race-freedom of input-independent kernels (though the implementation of the associated tool is not publicly available). However, it cannot be used to verify more complex examples where control flow can be input-dependent.

A permission-based separation logic has been developed for verifying GPU kernels, with a proof of race-freedom for a kernel being established through a consistent set of permission annotations for the kernel [HM13]. This approach provides a method for establishing richer functional properties than data race-freedom. However, the problem of automating the generation of permission annotations has not yet been studied in this line of work.

Symbolic execution and bounded-depth verification The GKLEE [LLS+12] and KLEE-CL [CCK14] tools perform dynamic symbolic execution of CUDA and OpenCL kernels, respectively, and are both built on top of the KLEE symbolic execution engine [CDE08]. A method for bounded verification of barrier-free GPU kernels via depth-limited unrolling to an SMT formula is presented in [TSL10]; lack of support for barriers, present in most non-trivial GPU kernels, limits the scope of this method. Symbolic execution and bounded unrolling techniques can be useful for bug-finding—both GKLEE and KLEE-CL have uncovered data race bugs in real-world examples—and these techniques have the advantage of generating concrete bug-inducing tests. A further advantage of GKLEE and KLEE-CL is that because they are based on KLEE, which works on LLVM bytecode, they can be applied to GPU kernels after optimisation and thus have the potential to detect bugs that result from incorrect compiler optimizations. The major drawback to these methods is that they cannot verify freedom of defects for non-trivial kernels.

Both GKLEE and KLEE-CL explicitly represent the number of threads executing a GPU kernel. This allows for precise defect checking, but limits scalability. A recent extension to GKLEE uses the notion of *parametric flows* to soundly restrict defect checking to consider only certain pairs of threads [LLG12]. This is similar to the two-thread abstraction employed by GPUVerify and PUG, and leads to scalability improvements over standard GKLEE, at the expense of a loss in precision for kernels that exhibit inter-thread communication.

Formal semantics for GPU kernels The relationship between the lock-step execution model of NVIDIA GPUs and the standard interleaved semantics for threaded programs presents a formal semantics for predicated execution has been studied [HK12]. This semantics shares similarities with a “synchronous delayed visibility” semantics used to present GPUVerify [BCD+12] but the focus of [HK12] is not on verification of GPU kernels. A recent paper studying Hoare logic for GPU kernels is in a similar vein [KI13].

Acknowledgements: I am grateful to the following for their contributions to the GPUVerify project: Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Jeroen Ketema, Egor Kyshtymov, Dan Liew, Shaz Qadeer, Paul Thomson, John Wickerson. Thanks to Andrei Lascu for providing corrections to an earlier draft of this material. The related
work section draws on material from [BCD+12] co-authored with Shaz Qadeer.
This work was supported by the EU FP7 project CARP and EPSRC grant EP/K011499/1.

Bibliography


The highs and lows of deploying Formal Methods in Industry

Guy H. Broadfoot

Silverdata Ltd.
United Kingdom

Abstract: I attended my first software conference in 1968; it was organised by NATO with the title “The Software Crisis.” Many of the papers presented then could have been written yesterday; the problems of the software industry in producing reliable, correct software in the face of increasing complexity and shrinking time to market pressures have not fundamentally changed that much.

In the intervening years as a community we have developed various tactics for trying to minimise software errors. Advances in theorem proving and model checking are good examples of systematic efforts to improve software correctness. Nevertheless, it remains the case that such approaches are rarely if ever encountered in the industrial workplace, with the possible exception of some safety critical domains, such as the software controlling nuclear power plants.

In spite advances in formal methods and supporting tools, the tools available to programmers for verifying assertions about program execution are complex and require knowledge and skills that most practicing programmers do not have. Formal proofs remain difficult to construct, especially for anything but the simplest of programs. Merely constructing assertions to characterise program correctness is a difficult challenge.

In 1998, I conceived the idea of combing model checking, code generation and the specification approach of Sequence-based Specification together to form an integrated software design platform for developing software components whose design (implementation) would be formally verified for correctness with respect to its specification. Other general correctness properties such as freedom from deadlocks, non-determinism, incomplete cases, etc. would also be verified. Verification would be performed by automatically translating Sequence-based specifications into semantically equivalent CSP process algebra and then applying the model-checking engine FDR2. After verification was completed, semantically equivalent source code would be generated in one of several supported high-level languages.

These ideas were developed further together with Philippa Hopcroft and in 2003 a company was founded to develop a commercial implementation of a development platform based on these ideas. In this talk, I will present an overview of the development platform and the technologies used. I will then discuss the experience gained during 10 years of trying to introduce this approach into industry and the lessons learned along the way.
Symbol Elimination for Automated Generation of Program Properties

Laura Kovács

Chalmers University of Technology

Abstract: Automatic understanding of the intended meaning of computer programs is a very hard problem, requiring intelligence and reasoning. In this talk we describe applications of our symbol elimination methods in automated program analysis. Symbol elimination uses first-order theorem proving techniques in conjunction with symbolic computation methods, and derives nontrivial program properties, such as loop invariants and loop bounds, in a fully automatic way. Moreover, symbol elimination can be used as an alternative to interpolation for software verification.

Keywords: program analysis, symbolic computation, automated reasoning, interpolation, invariant generation

Extended Abstract

Individuals, organizations, industries, and nations are increasingly depending on software and systems using software. This software is large and complex and integrated in a continuously changing complex environment. New languages, libraries and tools increase productivity of programmers, creating even more software, but the reliability, safety and security of the software that they produce is still low. We are getting used to the fact that computer systems are error-prone and insecure. Software errors cost world economies billions of euros. They may even result in loss of human lives, for example by causing airplane or car crashes, or malfunctioning medical equipment. To improve software and methods of software development one can use a variety of approaches, including automated software verification and static analysis of programs.

The results summarised in this abstract describe how the combination of automated reasoning and symbolic computation methods can be used for automatic program analysis. Program analysis aims to discover program properties preventing programmers from introducing errors while making software changes and can drastically cut the time needed for program development, making thus a crucial step to automated verification.

The common method of all results presented here is the so-called symbol elimination method. Although the symbol elimination terminology has been introduced only recently by us, we argue that symbol elimination can be viewed as a general framework for software verification. That is, various techniques used in software verification, such as Gröbner basis computation or quantifier elimination, can be seen as application of symbol elimination to safety verification of programs.

In a nutshell, symbol elimination is based on the following ideas. Suppose we have a program $P$ with a set of variables $V$. The set $V$ defines the language of $P$. We extend the language $P$ to

\* This work was partially supported by Swedish VR grant D0497701 and the Austrian research projects FWF S11410-N23 and WWTF ICT C-050.
a richer language $P_0$ by adding functions and predicates, such as loop counters. After that, we automatically generate a set $\Pi$ of first-order properties of the program in the extended language $P_0$, by using techniques from symbolic computation and theorem proving. These properties are valid properties of the program, however they use the extended language $P_0$. At a last step of symbol elimination we derive from $\Pi$ program properties in the original language $P$, thus “eliminating” the symbols in $P_0 \setminus P$.

The work summarized in this abstract is concerned with the algorithmic treatment of symbol elimination for generating computer program properties such as loop invariants, loop iteration bounds, interpolants, and postconditions.

We start by first presenting how symbol elimination is used in symbolic computation for analysing program loops and inferring loop invariants and postconditions. Our work uses algorithmic combinatorics and algebraic techniques, namely solving linear recurrences with constant coefficients, computing algebraic relations among exponential sequences, and eliminating variables from a system of polynomial equations using Gröbner basis computation and quantifier elimination techniques. We also describe applications of symbol elimination in the timing analysis of programs, or, more generally, for analysing the worse-case execution times of programs.

We further extend our work and present how symbol elimination is applied in first-order theorem proving for generating quantified loop invariants and interpolants. Unlike all previously known techniques, our method allows one to generate first-order invariants containing alternations of quantifiers. The method is based on automatic analysis of the so-called update predicates of loops. We observe that many properties of update predicates can be extracted automatically from the loop description and loop properties obtained by other methods such as a simple analysis of counters occurring in the loop, recurrence solving and quantifier elimination over loop variables. The key ingredient of symbol elimination for generating quantified program properties is then first-order saturation theorem proving. After observing that symbol-eliminating inferences extracted from first-order proofs of program properties can be used for automatic invariant generation and that interpolants obtained from proofs seem to be better for predicate abstraction and invariant generation than those obtained by quantifier elimination, we conclude that symbol elimination can be a key concept for applications of program analysis and verification.

Acknowledgements: The results described in this abstract are based on a joint work with Ioan Dragan (TU Vienna), Jens Knoop (TU Vienna), Andrei Voronkov (U. Manchester), and Jakob Zwirchmayr (IRIT).
Regular Papers
Conformance Testing of Cyber-Physical Systems: A Comparative Study

Morteza Mohaqeqi\textsuperscript{1}, Mohammad Reza Mousavi\textsuperscript{2} and Walid Taha\textsuperscript{3} \textsuperscript{*}

\textsuperscript{1} m.mohaqeqi@ut.ac.ir
School of Electrical and Computer Engineering, University of Tehran, Iran
Center for Research on Embedded Systems, Halmstad University, Sweden

\textsuperscript{2} m.r.mousavi@hh.se
Center for Research on Embedded Systems, Halmstad University, Sweden

\textsuperscript{3} walid.taha@hh.se
Center for Research on Embedded Systems, Halmstad University, Sweden
Rice University, USA

Abstract: For systematic and automatic testing of cyber-physical systems, in which a set of test cases is generated based on a formal specification, a number of notions of conformance testing have been proposed. In this paper, we review two existing theories of conformance testing for cyber-physical systems and compare them. We point out their fundamental differences, and prove under which assumptions they coincide.

Keywords: Hybrid systems, conformance testing, (hybrid) timed state sequences, and hybrid labeled transition systems.

1 Introduction

Cyber-physical systems are interdisciplinary by their nature and hence, experts from different disciplines have brought in different techniques for their specification, design, and analysis. This enables cross-fertilization of approaches among several disciplines but to achieve that, it requires reconciliation among the different techniques.

Model-based testing of cyber-physical systems is a relatively new concept that has been studied by few researchers from different backgrounds [BK06, vO06, vO09, ABW09, BWA10, Dan10, WLT13, AHF\textsuperscript{+}14, AFM\textsuperscript{+}14] and hence such a reconciliation has yet to take place.

Two notable examples of model-based testing theories for these systems in the literature are: the hioco relation by van Osch in [vO06, vO09] and the conformance relation by Abbas et al. in [AHF\textsuperscript{+}14], which we refer to as hconf. The hioco theory is based on hybrid transition systems formalism, which allows to specify both discrete input and output actions, and continuous behavior. On the other hand, under hconf, the conformance relation is considered as a measure of distance between the observable discretized (sampled) behavior of systems. These two approaches are substantially different since they stem, respectively, from the computer science-
and control theory point of view on model-based testing. Our general goal of research is to come up with a notion that reconciles these two views, and we start with a formal and in-depth comparison. Concerning the specific research question addressed in this paper, we were mainly intrigued by the following statement from [AHF+14]:

It can be verified that the hioco relation by van Osch is an exact version of \((T,J,(\tau,\epsilon))\)-closeness \((\tau = \epsilon = 0)\) with the role of inputs and outputs made explicit.

We aimed at formalizing and proving this claim. (Note that “the exact version of \((T,J,(\tau,\epsilon))\)-closeness” is what we call “hconf” in the remainder of this paper)\(^1\) However, even if we set aside the issue of explicit input and output, it turns out there are other fundamental differences between the two notions that went unnoticed in this statement. Hence, we will first point out the main differences between the two notions (which will also guide our future research in reconciling them) and subsequently, seek a restricted setting in which the above-mentioned claim does hold. We then formalize and prove the above-mentioned statement in this restricted setting.

As a common ground for our comparison, we use a restricted subset of Hybrid Automata, of which the restrictions are given and motivated in the remainder of this paper. We then translate this subset into the semantic domains of the two notions of conformance and show that the notions of conformance for the translated hybrid automata coincide in both domains.

1.1 Running Example

We consider a bouncing ball as a simple example of a system exhibiting both continuous and discrete behavior. The ball is dropped with an initial height and an initial velocity. Naturally, it is under the force of gravity. However, a vertical force can also be applied to the ball, which is considered as the input to this system. Collision with the floor leads to a bounce, which causes the ball to move in the reverse direction with the speed reduced by a factor \(\rho\). The movement of the ball between two consecutive collisions is accounted as a continuous behavior, while the collision with the floor is considered a discrete jump.

We can use a hybrid automaton to formally specify the behavior of the ball. We define three continuous variables \(h\), \(v\), and \(a\) for specifying the behavior of the ball, where \(h\) denotes the height, \(v\) denotes the vertical velocity, and \(a\) is the (vertical downward) acceleration caused by the force exerted on the ball. Consequently, the dynamics of this system can be captured by a hybrid automaton with one location, as shown in Fig. 1a. The dynamics associated to this location is specified as \(\dot{h} = v\) and \(\dot{v} = -g - a\), where \(g\) is the gravity constant.

Figure 1b shows the behavior of the system, starting with initial condition \(h = 5\) and \(v = 2\), while no vertical force is applied to the system.

1.2 Related Work

In the realm of model-based conformance testing, Tretmans proposed the theory of input/output (I/O) conformance (ioco) testing for discrete event systems [Tre96]. Afterwards, this theory was

\(^1\) Informally, two observed behavior satisfy \((T,J,(\tau,\epsilon))\)-closeness relation if in each interval of length \(\tau\), there exists one point in each of them with difference less than \(\epsilon\).
extended to real-time systems [BB05], where in addition to discrete states of the system, the evolution of a continuous variable, namely time, is modeled.

Conformance testing for cyber-physical systems (of which formal models are called hybrid systems in this paper), also, has been approached by a relatively small number of studies. In addition to the two aforementioned studies [vO06] and [AHF+14], which will be explained in detail in the next sections, conformance testing of hybrid systems using some notion of over-approximation has been proposed in [ABW09] and [BWA10]. These studies exploited a qualitative reasoning in order to obtain a discrete representation (abstraction) of a hybrid system. In a qualitative system, the continuous domains are abstracted into finite sets of quality levels and intervals. They proposed a method for obtaining a qualitative transition system from a hybrid system, which provides a discrete-event view to the system. Then, the ioco relation is applied to the obtained transition system. While this approach leads to finite representation of the hybrid system, it suffers from information loss occurred in the abstraction step.

Bringmann et al. [BK06] used stream-processing functions [MS97] for specifying test cases for hybrid systems. In this formalism, the continuous behavior of the variables is captured by the notion of streams. A stream-processing function or component relates a stream on input variables to a stream of output ones. Based on this notion, they propose a language for describing the test sets. They focus on how to specify test cases for hybrid systems. However, they provide no formal definition for conformance relation of hybrid systems.

In addition, Woehrle et al. [WLT13] investigated the problem of conformance testing of cyber-physical systems through measurement of a number of physical quantities like power consumption of the computing system. They employed timed automata to specify the system behavior. Then, the measured physical variable is compared against the automata to verify that if the system behavior conforms to its specification. Their method does not actually model the hybrid behavior of a system, but only considers time as a continuous variable. A conformance relation for hybrid systems has been also defined based on the notion of hybrid automata in [Dan10]. While the underlying formalism is different from the hybrid labeled transition system that is used by van Osch [vO06], the semantic model of the two conformance relations is similar [Dan10].
1.3 Paper Organization

The remainder of this paper is structured as follows. In Section 2, we review the hioco and hconf theories and point out the fundamental differences both between their semantic models and their conformance relations. In order to show the equivalence of the two conformance relations under specific assumptions, we first provide a method for translating a given hybrid automaton (in the restricted semantic domain) to each of the models in Section 3. Then, in Section 4, we show that, based on the proposed translation method, the two notions of conformance coincide. The paper concludes in Section 5, where we also describe our plan for future work.

2 Formal Definitions and Fundamental Differences

In this section, we first formally introduce system models that are used in the two aforementioned approaches, namely [vO06] and [AHF+ 14]. Afterwards, we point out the fundamental differences in their modeling approach and their assumptions. These will motivate the assumptions that we have made to pave the way for a meaningful comparison between the two conformance relations. Then, we review the conformance relations proposed by the two studies, and state the fundamental differences between them.

2.1 Formal Models

In the following, we first introduce some basic definitions. Then, we formally specify the system models employed in [vO06] and [AHF+14], namely, hybrid labeled transition system and hybrid-timed state sequence model, respectively.

2.1.1 Basic Definitions

A hybrid system is a system exhibiting both continuous and discrete behavior. The continuous behavior of the system is captured by the valuation of a set of continuous variables, denoted by $V$. We assume that $V$ is partitioned into a set of input variables, denoted by $V_I$, and a set of output variables, denoted by $V_O$. A valuation on $V$ is defined as a function which assigns a value to each variable $v \in V$; here, only variables of type real are considered. The set of all valuations of $V$ is denoted by $\text{Val}(V)$ and is defined as the set of all functions $V \rightarrow \mathbb{R}$. To describe the (non-interrupted) continuous evolution of the system, we use the following notion of trajectory [LSV03].

**Definition 1** (Trajectory [vO06]) Let $D \subset \mathbb{R}$ be an interval (here, singleton sets of numbers are also considered as intervals). A trajectory $\sigma$ is a function $\sigma : D \rightarrow \text{Val}(V)$ that maps each element in interval $D$ to a valuation. The set of all trajectories associated to $V$ is denoted by $\text{trajs}(V)$.

Beside continuous evolution, a hybrid system features discrete changes, called jumps or switches. A jump happens instantly, leading to a possibly noncontinuous change.

Further, we consider a restriction operator on the functions as follows.

**Definition 2** (Function Restriction) Consider an interval $D \subset \mathbb{R}$, a set $V$ of variables, and a
function $f : D \to \text{Val}(V)$; we define the restricted function $f \downarrow V'$ for some $V' \subseteq V$ as the function of type $D \to \text{Val}(V')$, such that for each $d \in D$, $f(d) \downarrow V'$ is obtained from $f(d)$ after removing variables not in $V'$.

2.1.2 Hybrid Labeled Transition System

In the approach adopted by van Osch [vO06], a hybrid system is modeled as a hybrid labeled transition system (HLTS). An HLTS, formally defined below, consists of a set of states with discrete (action) and continuous (trajectory) transitions between them. Actions are partitioned into three classes, namely, input-, output-, and internal actions. The latter class is not observable from outside.

**Definition 3** (Hybrid Labeled Transition System [vO06]) A hybrid labeled transition system (HLTS) $H$, over a given set $A$ of actions, is a 5-tuple $(S, s_0, V, L, \rightarrow)$, where

- $S$ is a (possibly infinite) set of states;
- $s_0 \in S$ is the initial state;
- $V = V_I \cup V_O$ is a set of (resp. input or output) continuous variables;
- $L = A \cup \text{trajs}(V)$ is a set of (resp. action or trajectory) labels;
- $\rightarrow \subseteq S \times (L \cup \{\xi\}) \times S$ specifies the transition relation, where $\xi$ denotes the internal action.

We may write $s \xrightarrow{l} s'$ instead of $(s, l, s') \in \rightarrow$. Moreover, we define a notion of generalized transition relation for an HLTS as follows. For the purpose of this paper, we assume the set of actions to be empty. We will also henceforth simplify the definitions for this restricted subset of HLTSs.

**Definition 4** (Generalized Transition Relation) Consider an HLTS $H = (S, s_0, V, L, \rightarrow)$. A generalized transition relation for $H$ is defined as the smallest relation $\Rightarrow \subseteq S \times L^* \times S$ where

- $s \xrightarrow{\xi} s$;
- if $s \xrightarrow{l} s'$, then $s \xrightarrow{l} s'$,
- $\forall l \in L$, if $s \xrightarrow{l} s$, then $s \xrightarrow{l} s$;
- $\forall \alpha, \beta \in L^*$, if $s \xrightarrow{\alpha} s''$ and $s'' \xrightarrow{\beta} s'$, then $s \xrightarrow{\alpha \beta} s'$.

We write $s \xrightarrow{\alpha} s'$ to denote that there exists a $s' \in S$ such that $s \xrightarrow{\alpha} s'$. The behavior of a system is specified by its set of traces, which are finite sequences of trajectories. (The internal actions are abstracted away in the traces.)

**Definition 5** (Trace) For HLTS $H$, a trace is a finite sequence $\alpha \in L^*$ such that $s_0 \xrightarrow{\alpha}$, where $s_0$ is the initial state of $H$. 
The length of a trace is defined as the number of elements of the sequence and is represented by $|\alpha|$. We denote the set of all traces of $\mathcal{H}$ by $\text{traces}(\mathcal{H})$.

**Example 1** Consider the bouncing ball example. Collision with the floor can be considered as an internal action. Also, the curves $f_1$, $f_2$, and $f_3$ in Fig. 1b denote some trajectories of the system (projected only to one variable $h$). Hence, the sequences $f_1f_2$ and $f_1f_2f_3$ are two traces of this system.

An HLTS can be input-enabled, as defined in the following.

**Definition 6** (Input-Enabled HLTS) An HLTS is input-enabled if

$$\forall s \in S, \forall \tau \in \text{trajs}(V_I): \exists \tau' \in \text{trajs}(V) \text{ such that } \tau' \downarrow V_I = \tau \land s \Rightarrow s'.$$

### 2.1.3 Timed State Sequence (TSS) Model

The approach proposed in [AHF+14] specifies a hybrid system as a mapping from pairs of initial conditions and input signals to output signals. The input and output signals of the system are described through a notion of timed-state sequence (TSS). (We slightly adapt the notation to make it consistent among the different semantic models.)

**Definition 7** (Hybrid-Timed State Sequence (TSS) [AHF+14]) Consider $N \in \mathbb{N}$, $T = \mathbb{R}_{\geq 0} \times \mathbb{N}$, and a set of variables $V$. A hybrid-timed state sequence (TSS) is defined as pair $(x,\sigma)$, where $x \in (\text{Val}(V))^N$ and $\sigma \in T^N$. The $i$th element of a TSS $(x,\sigma)$ is denoted by $(x_i,\sigma_i)$, where $\sigma_i = (t_i, j_i) \in T$. Also, we denote the set of all TSSs defined over the set of variables $V$, considering a specific $N \in \mathbb{N}$, by $\text{TSS}(V,N)$.

A TSS describes the valuation of a set of (input/output) variables in a finite number of time instants. In this notion, a time instant is denoted by a pair $(t, j)$, where $t$ is a real number denoting the real time, and $j$ is an integer which denotes the number of discrete jumps until that time instant. We refer to this semantic model as the TSS model.

**Example 2** The sequence $((1,1,1),((0,0),(4,5,0),(6,1)))$ is a TSS for the running example, defined for the variable $a$, where the tuple $(1,1,1)$ denotes the value of $a$ in three time instants $(0,0),(4,5,0)$, and $(6,1)$. For instance, the pair $(6,1)$ shows the instant that real time is 6 and the system has experienced one discrete jump. Similarly, the sequence $((5,6.55,7,1,6.65,5,2,2.75,0.19),((0,0),(1,0),(2,0),(3,0),(4,0),(5,0),(6,1)))$ denotes an output TSS for the considered system, denoting a valuation of the output variable $h$.

Assume a set of input variables $V_I$ with a corresponding compact set of possible values $\text{Val}(V_I)$ and a set of output variables $V_O$ with a set of possible valuations $\text{Val}(V_O)$. Accordingly, the set of input TSSs and output TSSs can be respectively defined by $\text{TSS}(V_I,N)$ and $\text{TSS}(V_O,N)$. Also, let $H$ be the set of initial conditions.

A hybrid system can be viewed as a mapping between the initial condition and the input TSSs $(u,\sigma_u) \in \text{TSS}(V_I,N)$ to the output TSSs $(y,\sigma_y) \in \text{TSS}(V_O,N)$. 

34 Conformance Testing of Cyber-Physical Systems: A Comparative Study
**Definition 8** (Hybrid System [AHF+14]) Hybrid system $\mathcal{H}$ is modeled as a mapping:

$$
\mathcal{H} : H \times \text{TSS}(V_I, N) \mapsto \text{TSS}(V_O, N)
$$

We write $y_{\mathcal{H}}(h_0, (u, \sigma_u))$ to denote the output TSS to which the pair $(u, \sigma_u)$ is mapped by $\mathcal{H}$, considering $h_0$ as the initial condition. In [AHF+14], it is assumed that the system is always input-enabled, with the following definition of input-enabledness.

**Definition 9** (Input-Enabled System) A hybrid system $\mathcal{H}$ is input-enabled if

$$
\forall h_0 \in H, \forall (u, \sigma_u) \in \text{TSS}(V_I, N) : \exists (y, \sigma_y) \in \text{TSS}(V_O, N) \text{ such that } (y, \sigma_y) = y_{\mathcal{H}}(h_0, (u, \sigma_u)).
$$

In other words, the system is input-enabled if it produces an output for every initial condition and input TSS.

### 2.2 Differences in Semantic Models

The following list provides an overview of the fundamental differences among the two underlying semantic models, reviewed in Subsection 2.1. After explaining each difference, we state the assumption we have made for the common semantic domains that makes a meaningful comparison possible.

- **Explicit discrete interactions vs. unlabeled jumps.** In HLTSs, an explicit notion of input/output actions is introduced, while TSSs provide no explicit means for modeling discrete input/output actions of the system. In order to have a unified semantic model for the two models, we do not consider explicit interactions in HLTSs and model all discrete jumps of the system as internal actions when using HLTSs. (A1)

- **Partial specifications vs. input-enabled models.** HLTSs used for hioco allow for partial specifications, in which the (output) response to certain (input) traces is left unspecified. However, the input-enabledness assumption made in the TSS model implies that for every sequence of input signal, the model should specify an output behavior of the system, expressed in terms of an output TSS. To allow for a meaningful comparison, we only consider input-enabled models. (A2)

- **Unique initial state vs. arbitrary initial condition.** HLTSs specify a unique initial state, while TSSs allow for arbitrary initial conditions. In order to make the two models compatible, we assume a singleton set of initial condition for the TSS model. (A3)

- **Nondeterministic specifications vs. deterministic models.** HLTSs allow for non-determinism both in the discrete and the continuous behavior, as an abstraction mechanism for unspecified / irrelevant details. However, hybrid systems in the hconf model are mappings from input TSSs to output TSSs and hence, are deterministic in nature. To make the comparison possible, we only consider deterministic models. (A4)

- **Continuous trajectories vs. discretized samples.** HLTSs allow for specification of continuous trajectories while hybrid systems in the hconf theory are necessarily discretized.
This alone may not be a major issue (i.e., hioco is also defined for discretized samples in [vO09, Chapter 6]); however, in combination with determinism and input-enabledness this assumption has far-reaching consequences, namely, if two continuous input trajectories have the same discretized behavior for one arbitrary sampling, they should lead to the same output behavior. To make the comparison feasible, we assume the latter property in the models studied in the remainder of this paper. (A5)

Considering these differences one comes away with the impression that the two models have distinct purposes and strengths. In particular, HLTSs as the semantic models of hioco are suitable for high-level partial specifications that leave some room for future design decisions and also only specify certain aspects of the system. However, TSSs used as the semantic models of hconf are suitable for low level specifications that provide a complete and deterministic (discretized) recipe for implementation. An ideal notion of conformance, in our view, should relax assumptions (A1)-(A5). In other words, an ideal theory of conformance should combine the liberal semantic domain of hioco with the practical conformance relation of hconf, as pointed out in the next section.

2.3 Conformance Relations

In this subsection, we review the two designated notions of conformance testing for hybrid systems, based on the formal models defined in Subsection 2.1.

2.3.1 HIOCO

In this section, we review the hybrid input-output conformance (hioco) theory [vO06], simplified according to the assumptions made in Section 2.2.

**Definition 10 (after Operator)** For an HLTS $H$ and a trace $\alpha \in \text{traces}(H)$, we define

$$H \text{ after } \alpha = \{ s \mid s_0 \xRightarrow{\sigma} s \}$$

**Definition 11 (Trajectories of a State)** For an HLTS $H$ and a state $s$, $\text{traj}(s)$ is defined as

$$\text{traj}(s) = \{ \sigma \in \text{trajs}(V) \mid s \xRightarrow{\sigma} \}$$

This definition can also be extended to a set of states $C \subseteq S$ as $\text{traj}(C) = \bigcup_{s \in C} \text{traj}(s)$

Using the above-given definitions, we are now ready to define the notion of hioco (simplified by neglecting discrete actions and restriction to input-enabled specifications).

**Definition 12 (Hybrid I/O Conformance [vO06])** Consider an HLTS $S$; an input-enabled HLTS $I$ is said to be hybrid input-output conforming to $S$, denoted by $I \text{hioco} S$, if and only if for all traces $\alpha \in \text{traces}(S)$:

$$\text{traj}(I \text{ after } \alpha) \subseteq \text{traj}(S \text{ after } \alpha)$$

The symmetric kernel of hioco, is denoted by $=_{\text{hioco}}$. Under the assumption of Section 2.2, hioco is a pre-order and $=_{\text{hioco}}$ is an equivalence relation.
2.3.2 HCONF

To define the notion of conformance, the notion of \((\tau, \varepsilon)\)-closeness is defined to measure how much a given TSS deviates from another TSS. Informally, two TSSs are said to be \((\tau, \varepsilon)\)-close if, for any time interval of length \(\tau\), and each sampled point in one, there exists a sample point in the other such that the difference between the valuations in the two points is less than \(\varepsilon\). Based on this notion, two systems are conforming if, for all combinations of initial conditions and input TSSs, the respective output TSS of the two systems are close to each other.

In order to be consistent with the hioco conformance relation, we only consider the conformance relation for TSS models for \((0, 0)\)-closeness [AHF+14]. Consider a maximum number of jumps \(N \in \mathbb{N}\) for which we are to perform the test.

**Definition 13** (hconf [AHF+14]) Two systems \(\mathcal{H}_M\) and \(\mathcal{H}_I\) are said to be conforming if for any input TSS \((u, \sigma_u) \in \text{TSS}(V_I, N)\)
\[ y_{\mathcal{H}_M}(h_0, (u, \sigma_u)) = y_{\mathcal{H}_I}(h_0, (u, \sigma_u)) \]

We write \(\mathcal{H}_M \stackrel{\text{hconf}}{\rightarrow} \mathcal{H}_I\) to denote that two systems \(\mathcal{H}_M\) and \(\mathcal{H}_I\) are conforming.

2.4 Differences in Conformance Relations

In addition to the fundamental differences in the semantic domain, which stated in Subsection 2.2, there are also fundamental differences in the way the two conformance relations are defined. Below, we provide a concise list of such fundamental differences:

- **Pre-order vs. equivalence relation.** The conformance relation hioco is a pre-order (for input-enabled specifications), while hconf is an equivalence relation. This stems from a fundamental difference in whether the implementation may choose from the alternative behaviors of the specification (pre-order view), or the implementation should implement the behavior prescribed by the specification (equivalence view). (The notion of closeness in hconf to some extent remedies this, and allows for deviations in the implementation.)

  In order to make our comparison possible, we consider the symmetric kernel of hioco for input-enabled specification, denoted by \(\cong_{\text{hioco}}\), and compare it with hconf.

- **Projecting on specification traces/trajectories vs. considering all traces.** Since the semantic model of hioco allows for partial specification, the conformance relation also exploits that by comparing the behavior of the implementation and specification only with respect to those traces specified in the specification. This can potentially create major differences between hioco and hconf. However, since we restricted our specifications to input-enabled and deterministic ones, this difference is not visible in our results to follow.

- **Exact vs. approximate comparison of trajectories.** Conformance relation hioco compares the continuous behaviors precisely, while hconf defines a notion of temporal and spatial closeness is used to represent and measure the deviation of the implementation from the specification. As indicated in the aforementioned claim of [AHF+14], we only consider the exact version of hconf and compare it with hioco.
• Sensitivity to quiescence. In order to reject implementations that do not produce any output when they should, the notion of quiescence is employed in hioco. Also, to denote existence or absence of continuous trajectories the notion of agile states is introduced. These notions are altogether absent in hconf, because hconf does not allow for the absence of outputs. In the case of deterministic input-enabled models (both for specifications and implementations, implied by our assumptions (A1)-(A5)), this difference is immaterial; however, once the models are relaxed to allow for non-deterministic and/or partial models, this difference may differentiate between the two notions of conformance.

3 Translating Hybrid Automata to the Other Models

In this section, we employ the notion of hybrid automata as a general formalism for hybrid systems and show how a hybrid automaton satisfying assumptions (A1)-(A5) can be translated to each of the two considered semantic domains. For the purpose of this paper, we deal with a minimal definition of hybrid automaton as considered in [GST09], in which there is no distinction between input and output actions.

Definition 14 (Hybrid Automata [GST09]) A hybrid automaton is defined as a tuple $(\text{Loc}, V, (l_0, v_0), \rightarrow, I, F)$, where

- $\text{Loc}$ is a finite set of locations;
- $V$ is the set of continuous variables;
- $l_0$ denotes the initial location and $v_0$ is an initial valuation of $V$;
- $\rightarrow \subseteq \text{Loc} \times \mathcal{B}(V) \times \text{Reset}(V) \times \text{Loc}$ is the set of jumps where:
  - $\mathcal{B}(V) \subseteq \text{Val}(V)$ indicates the guards under which the switch may be performed, and
  - $\text{Reset}(V) = \bigcup_{V' \subseteq V} \text{Val}(V')$ is the set of all value assignments to all or a subset of the variables $V$;
- $I : \text{Loc} \to \mathcal{B}(V)$ determines the allowed valuation of variables in each location (called the invariant of the location);
- $F : \text{Loc} \to \mathcal{B}(V \cup \dot{V})$ describes some constraints on variables and their derivatives and specifies the allowed continuous behavior in each location.

As before, we write $l \xrightarrow{E} l'$ for $(l, g, r, l') \in \rightarrow$.

The dynamic behavior of a hybrid automaton can be specified by a set of solutions. A solution to a hybrid automaton is a function on a hybrid time domain, which is defined as follows.

Definition 15 (Hybrid Time Domain [GT06, GST09]) A set $E \subseteq \mathbb{R}_{\geq 0} \times \mathbb{N}$ is a hybrid time domain if either:

- $E = \bigcup_{j=0}^{J-1} ([t_{j}, t_{j+1}], j)$, for a finite $J$, where $t_0 = 0$ and $t_0 \leq t_1 \leq t_2 \leq \ldots \leq t_J \leq \infty$; or
• $E = \bigcup_{j=0}^{\infty} ([t_j, t_{j+1}], j)$, with $t_0 = 0$ and $t_0 \leq t_1 \leq t_2 \leq \ldots$

For a hybrid time domain, we define time interval $I_j$ to be $[t_j, t_{j+1}]$. A time instant in the hybrid time domain $E$ is defined as a pair $(t, j) \in E$.

**Definition 16** (Solution of a Hybrid Automaton [Lem00]) A solution to the hybrid automaton $\mathcal{HA} = (\text{Loc}, V, (l_0, v_0), \rightarrow, I, F)$ is a function $x : E \rightarrow \text{Loc} \times \text{Val}(V)$, where $E$ is a hybrid time domain partitioned into a finite number $J$ of intervals, and

- $x(0, 0) = (l_0, v_0)$;
- for a fixed $j$, $x(t, j) : t \rightarrow I_j \times \text{Val}(V)$ is a function over real time that satisfies $I(l_j)$ and $F(l_j)$; and
- $\forall j \ (J - 1 > j \geq 0) : \exists g \in \mathcal{B}(V), r \in \text{Reset}(V)$ such that $(l_j \xrightarrow{E, r} l_{j+1}) \land (x(t_{j+1}, j) \downarrow V \text{ satisfies } g) \land (x(t_{j+1}, j + 1) \downarrow V = r)$.

The set of all solutions of a hybrid automaton $\mathcal{HA}$ is denoted by $\text{Solutions}(\mathcal{HA})$. For a given hybrid automaton $\mathcal{HA}$, the determinism assumption (A4) yields:

$$\forall s, s' \in \text{Solutions}(\mathcal{HA}) : (s \downarrow V_I = s' \downarrow V_I) \Rightarrow (s \downarrow V_O = s' \downarrow V_O)$$

The length of a solution $s$ is defined as the number of intervals in $\text{dom}\ s$.

### 3.1 Translating Hybrid Automata to Hybrid Labeled Transition Systems

According to **Definition 3** and **Definition 14**, a hybrid automaton denotes an HLTS, as follows.

**Definition 17** ($\|\cdot\|_{\text{HLTS}}$ [vO06]) A hybrid automaton $\mathcal{HA} = (\text{Loc}, V, (l_0, v_0), \rightarrow, I, F)$ can be converted to an equivalent HLTS $\|\mathcal{HA}\|_{\text{HLTS}} = (\text{Loc} \times \text{Val}(V), (l_0, v_0), \mathit{trajs}(V), \rightarrow)$, where

$$\rightarrow = \{(l, u) \xrightarrow{\sigma} (l', u') \mid (\exists r \in \mathcal{B}(V)) : (l, u) \rightarrow (l', u') \land (u \in g) \land (u \in I(l)) \land (u' \in I(l')) \land (u = \sigma.fval \land u' = \sigma.lval)\} \cup \{(l, u) \xrightarrow{\sigma} (l, u') \mid \exists t \in \mathit{trajs}(V) : (\sigma \in \mathcal{HA}\times\mathcal{V}(V)) \land (u = \sigma.fval) \land (u' = \sigma.lval) \land (\sigma \text{ satisfies } F(l)) \land (\forall t \in \text{dom}(\sigma) : \sigma(t) \in I(l))\}$$

where, $\sigma.fval$ and $\sigma.lval$ respectively denote the value of $\sigma$ at its starting (first) point and ending (last) point.

### 3.2 Translating Hybrid Automata to Timed State Sequences

In order to convert a hybrid automaton to a TSS, we use the concept of the solution of the hybrid automata.

**Definition 18** ($\|\cdot\|_{\text{TSS}}$) Consider a hybrid automaton $\mathcal{HA}$ with the set of input variables $V_I$ and the set of output variables $V_O$. Further, let $\text{Solutions}(\mathcal{HA})$ denote the set of all solutions of $\mathcal{HA}$. Thus, the TSS model which is defined by $\mathcal{HA}$, denoted by $\|\mathcal{HA}\|_{\text{TSS}}$, is the mapping:
\[ \mathcal{H} : (u, \sigma_u) \mapsto (y, \sigma_y) \]

that is constructed from the set of solutions in such a way that, for any solution \( x \in Solutions(\mathcal{H}CA) \), any input TSS \((u, \sigma_u)\) obtained by discretizing \( x \downarrow V_j \) is mapped to an output TSS \((y, \sigma_y)\), where

\begin{itemize}
  \item \( \sigma_y = \sigma_u; \)
  \item \( y = x(t_i, j_i) \downarrow V_O, \) where \((t_i, j_i)\) is the \( i \)th element of \( \sigma_u \).
\end{itemize}

### 4 Equivalence of hioco and hconf

**Theorem 1** Given a specification \( S \) in terms of a hybrid automaton and an implementation of which the behavior is expressed by a hybrid automaton \( \mathcal{I} \). If both \( S \) and \( \mathcal{I} \) satisfy conditions \( A1-A5 \), then it holds that \( \|S\|_{\mathcal{HTS}} \cong \|\mathcal{I}\|_{\mathcal{HTS}} \) if and only if \( \|S\|_{\mathcal{HTS}} =_{hioco} \|\mathcal{I}\|_{\mathcal{HTS}} \).

**Proof.** The idea of the proof is that we define a notion of hybrid conformance relation for hybrid automata and show that both hioco and hconf are equivalent to this notion of conformance. The proof comprises the following two steps:

1. \( \|S\|_{\mathcal{HTS}} =_{hioco} \|\mathcal{I}\|_{\mathcal{HTS}} \iff \forall s \in Solutions(S), \forall s' \in Solutions(\mathcal{I}) : (s \downarrow V_j = s' \downarrow V_j) \Rightarrow (s \downarrow V_O = s' \downarrow V_O) \)

2. \( \|S\|_{\mathcal{HTS}} \cong \|\mathcal{I}\|_{\mathcal{HTS}} \iff \forall s \in Solutions(S), \forall s' \in Solutions(\mathcal{I}) : (s \downarrow V_j = s' \downarrow V_j) \Rightarrow (s \downarrow V_O = s' \downarrow V_O) \)

To prove step 1, we provide a method for converting a solution in a hybrid automaton \( \mathcal{H} \) to an equivalent trace in the respective HLTS, and conversely, to convert a trace to a corresponding solution.

**Lemma 1** (Traces and Solutions Equivalence) Given a solution \( s \in Solutions(\mathcal{H}) \), one can uniquely obtain a trace \( \tau \in traces(\|\mathcal{H}\|_{\mathcal{HTS}}) \), with \( |\tau| = |s| \), and vice versa.

**Proof.** For a fixed \( N \geq 0 \), consider a trace \( \tau = \alpha_0 \alpha_1 ... \alpha_{N-1} \in traces(\|\mathcal{H}\|) \) with \( |\tau| = N \), and let \( D_j \) denote the continuous time interval over which \( \alpha_j \) is defined. We can associate a solution \( s \in Solutions(\mathcal{H}) \) with \( \tau \) as follows. We specify the domain of \( s \) as \( \text{dom} s = \bigcup_{j=0}^{N-1} I_j \), where \( I_j \) is defined as:

\begin{itemize}
  \item \( I_0 = [D_0, 0]; \)
  \item \( I_j = ([t_j, t_{j+1}], j), \) where \( t_{j+1} = t_j + |D_j| \) for \( j > 0; \)
\end{itemize}

in which \( |D_j| \) denotes the length of time interval \( D_j \). Furthermore, the value of the solution \( s \) at time instant \((t, j)\) is specified by the value of the respective trajectory at that instant, i.e., \( s(t, j) = \alpha_j(\tau - t_j) \). Conversely, it can be shown that given a solution \( s \in Solutions(\mathcal{H}) \), one can uniquely obtain a trace \( \tau \in traces(\|\mathcal{H}\|_{\mathcal{HTS}}) \), with \( |\tau| = |s| \). \( \Box \)
Next, we show that \(HLT_S = hconf HLT_I\) means that \(traces(HLT_S) = traces(HLT_I)\). Then, using Lemma 1, the first step of the proof can be readily concluded.

**Lemma 2**  
Consider two hybrid automata \(S\) and \(I\). It holds that \(HLT_S = hconf HLT_I\) if and only if \(traces(HLT_S) = traces(HLT_I)\).

**Proof.** For an HLTs \(\mathcal{H}\), let \(traces_N(\mathcal{H})\) denote the set of all traces of \(\mathcal{H}\) which are of length \(N\). In other words, \(traces_N(\mathcal{H}) = \{ \alpha \in traces(\mathcal{H}) : |\alpha| = N \}\). The goal is to show, given \(HLT_S = hconf HLT_I\), that \(traces_N(HLT_S) = traces_N(HLT_I)\) for any \(N \geq 0\). We proceed by induction on \(N\).

We consider \(N = 0\) as the base of induction, which includes empty trace \(\epsilon\). According to Definition 4, and also with regard to Definition 5, it holds that \(\epsilon \in traces(HLT_S)\) and \(\epsilon \in traces(HLT_I)\). Therefore, \(traces_0(HLT_S) = traces_0(HLT_I)\).

For the induction step, assuming that \(traces_N(S) = traces_N(I)\), we need to show \(traces_{N+1}(S) = traces_{N+1}(I)\). To this end, consider that \(traces_{N+1}(I)\) and \(traces_{N+1}(S)\), which can be defined as follows:

\[
traces_{N+1}(I) = \{ \tau \alpha | \tau \in traces_N(I) \land \alpha \in traj(I\text{ after } \tau) \}
\]

and

\[
traces_{N+1}(S) = \{ \tau \alpha | \tau \in traces_N(S) \land \alpha \in traj(S\text{ after } \tau) \}
\]

We have that \(traces_N(S) = traces_N(I)\), and also according to Definition 12, \(traj(S\text{ after } \tau) = traj(I\text{ after } \tau)\), we have \(traces_{N+1}(S) = traces_{N+1}(I)\), which completes the induction step. (Note that according to assumptions A2 and A4 in Subsection 2.2, Definition 12 establishes the former equality.)

For step 2 of the proof of Theorem 1, we first show that if \(Solutions(S) = Solutions(I)\) then \(\|S\|_{TSS} hconf \|I\|_{TSS}\). Note that the method for translating a hybrid automaton to a TSS described in Definition 18 is deterministic. Moreover, the translation method constructs the respective TSS solely based on the set of solutions of the hybrid automaton. Therefore, for two hybrid automata \(S\) and \(I\) with the same set of solutions, the resultant TSSs are trivially equal. According to Definition 13, this means that \(\|S\|_{TSS} hconf \|I\|_{TSS}\).

To prove step 2, we further need to show that if \(\|S\|_{TSS} hconf \|I\|_{TSS}\) then \(Solutions(S) = Solutions(I)\). We proceed with proof by contradiction. Assume that \(\|S\|_{TSS} hconf \|I\|_{TSS}\) and \(Solutions(S) \neq Solutions(I)\). Therefore, it holds that

\[
\exists s \in Solutions(I) \text{ such that } \forall s' \in Solutions(S) : s \neq s'.
\]

But, due to the assumption that the considered hybrid automata are input-enabled, there exists an \(s' \in Solutions(I)\) for which \(s \downarrow V_I = s' \downarrow V_I\). But, as \(s \neq s'\), we can conclude \(s \downarrow V_O \neq s' \downarrow V_O\), which means that there is a time instant \((t, j)\) for which \(s(t, j) \neq s'(t, j)\). Consider the input TSS obtained by discretization of \(s\), denoted as \(u\) that includes time instant \((t, j)\). According to the translation method from hybrid automaton to TSS described in Definition 18, the output TSS to which \(\|S\|_{TSS}\) maps the input TSS \(u\) differs from that of \(\|I\|_{TSS}\), which contradicts with the assumption of \(\|I\|_{TSS} hconf \|I\|_{TSS}\).
5 Conclusion

We studied two fundamental notions of conformance testing for cyber-physical systems, which are to our knowledge the most notable notions of their kind. We have pointed out fundamental differences, both in their semantic domains and in their definition of conformance. We identified a set of conditions under which the two notions are comparable. We proved that under such conditions, the two notions coincide.

While hioco is based on a richer and more expressive semantic domain, it suffers from practical concerns, which are related to its underlying infinite state-space. On the other hand, hconf provides a more practical approach to checking conformance, but it ignores some important aspects in modeling, including explicit discrete interactions, nondeterminism, and partial specifications.

The dichotomy reflected in the difference in the two notions suggests that a natural next step would be to define a more general notion of conformance testing for cyber-physical systems, which consolidates the theoretical expressiveness of hioco with the practical approach of hconf to conformance checking. Defining a test-case generation algorithm and proving soundness and adequacy of the generated test cases (given some testing hypothesis) are among the future milestones.

Acknowledgements: We would like to thank the anonymous reviewers of AVOCS’14, Houssam Abbas, Harsh Beohar, Pieter Cuijpers, Jim Kapinski, Mehdi Kargahi, and Mahsa Varshosaz for their helpful comments.

Bibliography


Analyzing Conflict Freedom For Multi-threaded Programs With Time Annotations

Jingshu Chen\textsuperscript{1}, Marie Duflot\textsuperscript{1,2}, Stephan Merz\textsuperscript{1,2}

\textsuperscript{1}Inria, Villers-lès-Nancy, F-54600, France
\textsuperscript{2}Université de Lorraine, LORIA, UMR 7503, Vandœuvre-lès-Nancy, F-54500, France

\textbf{Abstract:} Avoiding access conflicts is a major challenge in the design of multi-threaded programs. In the context of real-time systems, the absence of conflicts can be guaranteed by ensuring that no two potentially conflicting accesses are ever scheduled concurrently.

In this paper, we analyze programs that carry time annotations specifying the time for executing each statement. We propose a technique for verifying that a multi-threaded program with time annotations is free of access conflicts. In particular, we generate constraints that reflect the possible schedules for executing the program and the required properties. We then invoke an SMT solver in order to verify that no execution gives rise to concurrent conflicting accesses. Otherwise, we obtain a trace that exhibits the access conflict.

\textbf{Keywords:} multi-threaded program, access conflict, real-time system, time annotation, SMT solving

\section{Introduction}

Avoiding conflicting accesses to shared resources is a fundamental problem in concurrent programming, and it is particularly crucial in the development of controllers of real-time systems. Whereas the use of locks is the most common solution to this problem, it has well-known drawbacks, such as the run-time overhead associated with acquiring locks and being prone to errors and deadlocks. In real-time systems, the use of locks may be incompatible with the stringent requirements on the predictability of running times. Instead, programmers may rely on temporal conditions that ensure that statements with potentially conflicting accesses to resources are never scheduled concurrently.

In this paper, we assume that program code carries annotations that indicate the execution time allowed for each statement of the program [TW04]. Such annotations may for example be derived from a static analysis providing bounds on the execution time of the code on a specific execution platform (such as JOP [SPPH10] for safety-critical Java [JSR13], although the focus in this paper is on general principles rather than any specific language). Moreover, we assume that the platform provides mechanisms for ensuring that the actual execution of the program complies with these annotations. Our goal is to ensure that no conflicting accesses occur; such a specification can be expressed by precedence properties between statements of different threads.

\textsuperscript{*} This work has been supported by a grant from the Airbus Corporate Foundation and complementary funding by Région Lorraine, including funds for a post-doctoral contract for Jingshu Chen.
We present a technique for verifying whether all finite executions of an annotated multi-threaded program up to a fixed bound satisfy the required precedence properties. Similar to bounded model checking, the key idea is to reduce this verification problem to a constraint solving problem, by encoding the set of possible schedules of the given program that respect the timing annotations, as well as the required properties, as formulas in quantifier-free linear integer arithmetic. We then invoke off-the-shelf SMT solvers that efficiently decide the satisfiability of such formulas. In case the properties are violated, the solver generates a (counter-)model that corresponds to an execution violating the property, which can be analyzed by the program designer. Since the analysis is completely automatic, and the performance of the SMT solvers scales well, program designers can repeatedly analyze different variants of the program and understand the effect of changing timing parameters.

In this paper, we restrict attention to very simple programs where every thread consists of straight-line code, possibly contained in a single loop (which is unrolled for bounded verification). Such program structures are not uncommon in the real-time domain, for example when sensor inputs have to be sampled and processed at regular intervals. An extension to more complex control structures is straightforward by over-approximating the possible executions. For a more precise analysis, our technique could be combined with standard SMT-based program analysis [FP13, Lei13].

Outline. Section 2 presents a motivating example, and Section 3 describes the model of execution for the programs that we analyze. Section 4 represents the core of our paper, where we define how constraints are generated to represent the possible schedules of an annotated program, and its precedence properties. The results of some experiments, providing evidence for the scalability of the approach, are reported in Section 5. Section 6 presents related work, and Section 7 concludes the paper.

2 A Motivating Example

As a toy example, consider the following code snippet where $i$ and $j$ are two global variables.

```c
/* Thread t1 */
l_1,1 ://@1//@
    i = 2;
    l_1,2 ://@2//@
    i += 2;

/*Thread t2*/
l_2,1 :sleep(2);
    l_2,2 ://@2//@
    j = i;

/* post-condition: */
    j == i */
```

This program can be viewed as an implementation of the classic producer-consumer problem, which is representative for synchronization between threads. The two threads $t_1$ and $t_2$ update the values of the variables $i$ and $j$. It is intended that the values of $i$ and $j$ are equal at the end of the execution of the program, that is, the assignment $l_{2,2}$ in thread $t_2$ should be executed after the statements $l_{1,1}$ and $l_{1,2}$ of thread $t_1$.

The standard means for ensuring thread synchronization is the use of locks. However, the use of locks can be costly and error-prone. For programs written for real-time execution platforms...
where all threads share a common global time reference, such as Safety-Critical Java [JSR13], an alternative is to synchronize threads by scheduling constraints. In the above code, these constraints are indicated by the annotations at each statement, resp. by the argument of the sleep statement \$l_{2,1}\$. For example, the assignment statement \$l_{1,1}\$ is assumed to be scheduled for execution during exactly one time unit. We require these annotations to be present as an input for our analysis, and we assume that they are enforced by the execution platform. We assume that multi-threaded programs are scheduled on a single processor, subject to an eager scheduling policy where some thread executes whenever at least one thread is executable. Finally, we do not explicitly consider statements such as input and output that could execute in parallel to the CPU. The question whether the assumed scheduling constraints are feasible is out of the scope of this paper, but upper bounds for the execution of statements on specific processor architectures such as JOP [SPPH10] can be obtained by static analysis.

The annotated program in the above example indeed ensures its post-condition: initially, thread \$t_2\$ is sleeping, and thread \$t_1\$ is scheduled to execute \$l_{1,1}\$ for one time unit. After that, \$t_2\$ is still sleeping, so \$t_1\$ must again be scheduled for executing \$l_{1,2}\$, and only then \$l_{2,2}\$ can execute. However, if the timing annotation for statement \$l_{1,1}\$ were changed to 2, then the two threads would compete for execution after two time units, hence \$l_{2,2}\$ could be scheduled for execution in between statements \$l_{1,1}\$ and \$l_{1,2}\$, and the post-condition would be violated.

In the following, we describe an approach to mechanically analyze schedules of multi-threaded programs with timing annotations, with respect to properties that require temporal orders between program statements, typically ensuring the absence of race conditions for accessing shared variables. We generate constraints that describe the potential schedules, as well as required synchronization properties, and use off-the-shelf SMT solvers for verifying that all schedules respecting the constraints satisfy the properties. Otherwise, the solver generates a model that represents an execution of the program violating the properties.

### 3 Execution Model

The input to our analyzer is a multi-threaded program with timing annotations indicating the time allotted to the execution of (blocks of) statements. We distinguish between ordinary and sleep statements: the latter specify that scheduling of the adjacent ordinary statements must be separated by at least the indicated sleeping time. For simplicity, we assume that each thread consists of a sequence of (ordinary and sleep) statements, possibly enclosed in a loop. Without loss of generality, we assume that no thread contains two consecutive sleep statements: the sequence sleep\((m)\); sleep\((n)\) is equivalent to the single sleep statement sleep\((m + n)\).

We will generate constraints that describe all possible schedules of the program execution, up to a user-defined bound. A thread has four possible states: *executing* (a non-sleep statement), *waiting*, *sleeping*, and *terminated*. Threads are scheduled according to the following constraints:

- At any given instant, at most one thread is in state *executing*. It executes its current statement (or block of statements) without interruption by other threads, for the number of time units indicated by the corresponding timing annotation. After that lapse of time, the scheduler may choose to schedule a different thread for execution.
• Whenever there is at least one thread that is neither sleeping nor terminated, then some thread is executing.

• A statement sleep(n) following an ordinary statement causes the thread to enter the sleeping state as soon as the preceding statement has finished executing, and to remain in sleeping state for n time units. After that lapse of time, the thread moves to state waiting, unless it is immediately scheduled for execution or it has terminated. The scheduling of an initial statement sleep(n) is analogous, at the beginning of program execution. In particular, any number of threads may be sleeping simultaneously.

• Statements of every thread are scheduled in program order.

We leave relaxations of these constraints as interesting topics for future work. In particular, the execution semantics of modern programming languages on advanced architectures, including multi-core or multi-processor systems, does not adhere to all of the above assumptions.

4 Constraint Generation

We now describe constraints that encode the set of possible schedules for a given program, up to a fixed bound. We first list the variables that we use for representing schedules, then give a formula that represents the execution of an individual statement, and finally define the overall scheduling constraints as well as the formula representing the precedence properties to be verified.

4.1 Representing Program Schedules

Suppose that we are given a program with threads $Td = \{1, \ldots, T\}$, and that we want to represent schedules of length up to $N$ steps. We eliminate loops by unrolling every loop so that every thread $t$ consists of statements $s_{t,1}, \ldots, s_{t,n_t}$ executed sequentially. The number of ordinary (i.e., non-sleeping) statements in that sequence should be $N$, unless thread $t$ has less than $N$ such statements to execute even when loops are unrolled. We denote by $D_{t,i}$ be the duration of statement $s_{t,i}$, given as an integer constant that corresponds either to the timing annotation if $s_{t,i}$ is non-sleeping, or to the argument of the sleep statement $s_{t,i}$. Let $NS_t \subseteq \{1, \ldots, n_t\}$ denote the set of the corresponding indices for non-sleeping statements of thread $t$.

Our encoding is based on the following variables:

- $pc_t^{(k)}$, for $t \in Td$ and $k \in \{1, \ldots, N+1\}$, represents the “program counter” of thread $t$. Its value in $NS_t \cup \{n_t + 1\}$ denotes the next non-sleeping statement that thread $t$ will execute at round $k$ of the schedule; the value of $n_t + 1$ corresponds to a terminated thread.

- $Y^{(k)}$ and $X^{(k)}$, for $k \in \{1, \ldots, N+1\}$, indicate the global time at the beginning and the end, respectively, of round $k$ of the schedule. Except in situations where all threads are sleeping or have terminated, we will have $Y^{(k+1)} = X^{(k)}$.

1 The variables $Y^{(N+1)}$ and $X^{(N+1)}$ could be omitted, but their presence yields more uniform definitions.
\[ E_{t,i} , \text{ for } t \in T_d \text{ and } i \in \{1, \ldots, n_t \}, \text{ denotes the time at which the execution of statement } s_{t,i} \text{ ends (the starting time of execution is then obtained as } E_{t,i} - D_{t,i}). \text{ Observe that we have only one copy of these variables since each statement is executed at most once. Since the schedule ends after } N \text{ rounds, only the values of } E_{t,i} \text{ corresponding to statements that have actually been scheduled, are meaningful.} \]

The following formula \( \text{init} \) fixes some values for variables corresponding to the initial round of the schedule.\(^2\)
\[
\text{init} \triangleq \land \bigcap_{t \in T_d} \text{pc}^{(1)}_t = \begin{cases} 1 & \text{if } 1 \in NS_t \\ 2 & \text{otherwise} \end{cases} \land Y^{(1)} = \begin{cases} 0 & \text{if } 1 \in NS_t \text{ for some } t \in T_d \\ \min \{D_{t,i} : t \in T_d\} & \text{otherwise} \end{cases} \land \bigcap_{t \in T_d : 1 \notin NS_t} E_{t,1} = D_{t,1}
\]

The program counters of each thread are initialized to the first non-sleeping statements. The global time at which the first round starts is 0, except if the initial statements of all threads are sleep statements, in which case the first round starts at the end of the sleep statement(s) with the shortest duration. Finally, all initial sleep statements end after the sleeping time has elapsed.

### 4.2 Modeling Execution of a Non-Sleeping Statement

We now define a formula \( \text{exec}^{(k)}_{t,i} \) that models execution of the non-sleeping statement \( s_{t,i} \) (i.e., for \( i \in NS_t \)) at round \( k \). If statement \( s_{t,i} \) is not followed by a sleep statement, the formula is defined as
\[
\text{exec}^{(k)}_{t,i} \triangleq \land \text{pc}^{(k)}_t = i \land X^{(k)} = Y^{(k)} + D_{t,i} \land E_{t,1} = X^{(k)} \land \text{pc}^{(k+1)}_t = i + 1 \land \bigcap_{t' \in T_d \setminus \{t\}} \text{pc}^{(k+1)}_{t'} = \text{pc}^{(k)}_{t'}
\]

Formula \( \text{exec}^{(k)}_{t,i} \) requires that statement \( s_{t,i} \) be the next statement that thread \( t \) should execute at round \( k \). Then, round \( k \) ends at time \( Y^{(k)} + D_{t,i} \), which is also the time at which execution of \( s_{t,i} \) ends. The program counter for thread \( t \) at the next round moves to the subsequent statement, while the other program counters remain unchanged. The starting time of the subsequent round, i.e. the value of \( Y^{(k+1)} \), will be determined by the overall scheduling constraint defined in Section 4.3.

The formula \( \text{exec}^{(k)}_{t,i} \) is somewhat different if statement \( s_{t,i} \) is followed by a sleep statement \( s_{t,i+1} \) as described in Section 3, the sleeping time begins immediately after statement \( s_{t,i} \) has

---

\(^2\) We adopt the convention of writing multi-line conjunctions and disjunctions as lists bulleted with the operation sign, using indentation for indicating precedence [Lam94].
been executed, and the next statement to be executed is the statement following \( s_{t,i+1} \). We therefore define in this case
\[
\begin{align*}
\text{exec}^{(k)}_{t,i} & \triangleq \land pc^{(k)}_{t,i} = i \\
& \land X^{(k)} = Y^{(k)} + D_{t,i} \land E_{t,i} = X^{(k)} \land E_{t,i+1} = X^{(k)} + D_{t,i+1} \\
& \land pc^{(k+1)}_{t} = i + 2 \\
& \land \bigwedge_{t' \in T_d \setminus \{t\}} pc^{(k+1)}_{t'} = pc^{(k)}_{t}
\end{align*}
\]

4.3 Overall Scheduling Constraint

The overall constraint \( \text{sched} \) characterizing prefixes of schedules of length \( N \) asserts that at every round, some non-sleeping statement is executed, unless all threads have (and remain) terminated. This constraint also defines the starting time \( Y^{(k+1)} \) for the next round. The following definitions show the high-level structure and the case of termination.

\[
\begin{align*}
\text{sched} & \triangleq \bigwedge_{k=1}^{N} \text{round}^{(k)} \\
\text{round}^{(k)} & \triangleq \text{terminated}^{(k)} \lor \text{exec\_thread}^{(k)} \\
\text{terminated}^{(k)} & \triangleq \land \bigwedge_{t \in T_d} pc^{(k)}_{t} = n_t + 1 \land pc^{(k+1)}_{t} = pc^{(k)}_{t} \\
& \land Y^{(k+1)} = X^{(k)} \land X^{(k+1)} = X^{(k)}
\end{align*}
\]

Formula \( \text{sched} \) stipulates that the schedule should contain \( N \) rounds. The constraint characterizing round \( k \) distinguishes between two cases: either all threads are already terminated or some thread will execute at round \( k \). Termination means that the program counters of all threads point beyond the last statement; they then remain there, and the beginning and ending times of round \( k + 1 \) are set to \( X^{(k)} \), the ending time of round \( k \).

When program execution has not terminated, some thread \( t \) executes a non-sleeping statement, and we must determine the starting time of round \( k + 1 \). Using the formulas defined in Section 4.2, this suggests the definition
\[
\begin{align*}
\text{exec\_thread}^{(k)} & \triangleq \land \bigvee_{t \in T_d} \bigvee_{i \in NS_t} \left( \text{exec}^{(k)}_{t,i} \land (i = 1 \lor E_{t,i-1} \leq Y^{(k)}) \right) \\
& \land \text{fix\_starting\_time}^{(k)}
\end{align*}
\]

For the definition of the starting time \( Y^{(k+1)} \) of round \( k + 1 \), there are two cases to consider:

- If some non-sleeping statement can be executed at time \( X^{(k)} \), the ending time of round \( k \), then \( Y^{(k+1)} = X^{(k)} \). There is a statement to be executed at time \( X^{(k)} \) iff some non-terminated thread \( t \) is either at the beginning of its program, waiting for a non sleeping statement to occur, or execution of the statement preceding the current statement of thread \( t \) ended at time \( X^{(k)} \) or before.

- If no statement is executable at time \( X^{(k)} \), i.e. if all non-terminated threads are sleeping, then round \( k + 1 \) starts when the first thread(s) awake.
In the formal definition, we make use of a macro notation in order to refer to the ending time of the statement preceding the current one of a given thread. Specifically, we write $E_{\text{prev}}^{(k)} \sim e$ where $\sim \in \{=, \leq, \geq, >\}$ is a comparison operator, and $e$ is an arbitrary expression, as a shorthand for the formula

$$\bigvee_{i=1}^{n_t} \left( pc_i^{(k)} = i + 1 \land E_{i,i} \sim e \right)$$

and similarly for $e \sim E_{\text{prev}}^{(k)}$. With this notation, the intuition given above is concretized by the following formulas.

$$\text{fix}\_\text{starting}\_\text{time}^{(k)} \triangleq \bigvee \left( \text{some}\_\text{executable}^{(k)} \land Y^{(k+1)} = X^{(k)} \right) \land \neg \text{some}\_\text{executable}^{(k)} \land \text{set}\_\text{min}\_\text{end}\_\text{time}^{(k)}$$

$$\text{some}\_\text{executable}^{(k)} \triangleq \bigvee_{t \in T_d} \left( pc_t^{(k+1)} \leq n_t \land \left( 1 \lor E_{\text{prev}}^{(k+1)} \leq X^{(k)} \right) \right)$$

$$\text{set}\_\text{min}\_\text{end}\_\text{time}^{(k)} \triangleq \bigvee_{t \in T_d} \left( \bigwedge_{t' \in T_d\setminus\{t\}} pc_{t'}^{(k+1)} \leq n_{t'} \Rightarrow E_{\text{prev}}^{(k+1)} \leq E_{\text{prev}}^{(k+1)} \right) \land Y^{(k+1)} = E_{\text{prev}}^{(k+1)}$$

Observe that the formula $\text{sched}$ is expressed as a quantifier-free formula of the theory of linear integer arithmetic. Off-the-shelf SMT solvers such as Yices [DM06] or Z3 [MB08] provide very efficient decision procedures for such formulas, from which we can directly benefit.

### 4.4 Verifying Precedence Requirements

The properties that we are interested in assert precedence between the execution order of statements of different threads, such as that some statement should be executed before another one, or that it should not be executed in between two other statements. Formally, such properties can be expressed as Boolean combinations $\lambda$ of inequations between the ending times for statements. For the toy example of Section 2, we want to assert that the second statement $l_{1,2}$ of thread $t_1$ ends before the second statement $l_{2,2}$ of thread $t_2$ starts, which, as two statements are never executed at the same time, can be expressed as $E_{1,2} < E_{2,2}$. Since the ending times are meaningful only if the corresponding statements have actually been scheduled, we actually generate the formula

$$(pc_1^{(N+1)} > 2 \land pc_2^{(N+1)} > 2) \Rightarrow E_{1,2} < E_{2,2}.$$ 

Moreover, in case the statements of interest appear in loops, we actually want to verify that such constraints hold for all pairs of instances of these statements when the loops are unrolled.

In order to verify that the property holds over all possible schedules, we generate the formula $\text{sched} \land \neg \lambda$ and run an SMT solver that checks if that formula is satisfiable. If the answer is UNSAT, then the precedence property $\lambda$ holds over all prefixes of schedules of size (at most) $N$. Otherwise, the model computed by the SMT solver corresponds to a schedule that includes the relevant statements and that violates $\lambda$. 

AVoCS 2014

51
5 Experiments

We now illustrate the approach described in Section 4. We have developed a prototype that generates the constraints corresponding to given programs with timing annotations, as well as the desired precedence properties. In our experiments, we use the SMT solver Yices [DM06].

5.1 Generating the Constraints for a Toy Program

We will generate the constraints for the toy example considered in Section 2.

```plaintext
/* Thread t1 */
l1,1 : //@1@
    i = 2;
l1,2 : //@2@
    i += 2;
/* post-condition: j == i */
```

```plaintext
/* Thread t2 */
l2,1 : sleep(2);
l2,2 : //@2@
    j = i;
```

Since this program has three non-sleeping statements, we generate the constraints representing the possible schedules of length $N = 3$. The initial constraint is

\[
\text{init} \triangleq \land pc_1^{(1)} = 1 \land pc_2^{(1)} = 2 \land Y^{(1)} = 0 \land E_{2,1} = 2
\]

This constraint initializes the program counters for the two threads to their first non-sleeping statements. Since the initial statement of thread 1 is non-sleeping, the first round will start at time 0. The initial (sleep) statement of thread 2 ends after 2 time units.

Next, we define formulas that represent the execution of the three non-sleeping statements at round $k$, for $k = 1, 2, 3$, according to the schema given in Section 4.2.

```plaintext
exec_{1,1}^{(k)} \triangleq \land pc_1^{(k)} = 1 \land X^{(k)} = Y^{(k)} + 1 \land E_{1,1} = X^{(k)} \land pc_{1}^{(k+1)} = 2 \land pc_{2}^{(k+1)} = pc_{2}^{(k)}
```

```plaintext
exec_{1,2}^{(k)} \triangleq \land pc_1^{(k)} = 2 \land X^{(k)} = Y^{(k)} + 2 \land E_{1,2} = X^{(k)} \land pc_{1}^{(k+1)} = 3 \land pc_{2}^{(k+1)} = pc_{2}^{(k)}
```

```plaintext
exec_{2,2}^{(k)} \triangleq \land pc_2^{(k)} = 2 \land X^{(k)} = Y^{(k)} + 2 \land E_{2,2} = X^{(k)} \land pc_{2}^{(k+1)} = 3 \land pc_{1}^{(k+1)} = pc_{1}^{(k)}
```

Finally, the overall scheduling constraint is defined as

\[
\text{sched} \triangleq \text{round}^{(1)} \land \text{round}^{(2)} \land \text{round}^{(3)}
\]
where the formula \( round^{(k)} \) representing a single round is defined as

\[
\begin{align*}
\vee & \ \ pc_1^{(k)} = 3 \land pc_1^{(k+1)} = 3 \land pc_2^{(k+1)} = 3 \land Y^{(k+1)} = X^{(k)} \land X^{(k+1)} = X^{(k)} \\
\vee \land \vee & \ \ exec^{(k)}_{1,1} \\
\vee & \ \ exec^{(k)}_{1,2} \land E_{1,1} \leq Y^{(k)} \\
\vee & \ \ exec^{(k)}_{2,2} \land E_{2,1} \leq Y^{(k)} \\
\land \vee & \ \ some\_executable^{(k)} \land Y^{(k+1)} = X^{(k)} \\
\land \vee & \ \ \neg some\_executable^{(k)} \\
\land & \ \ pc_1^{(k+1)} \leq 2 \land (pc_2^{(k+1)} \leq 2 \Rightarrow E_{prev_1^{(k+1)}} \leq E_{prev_2^{(k+1)}}) \\
\land & \ \ Y^{(k+1)} = E_{prev_1^{(k+1)}} \\
\land & \ \ pc_2^{(k+1)} \leq 2 \land (pc_1^{(k+1)} \leq 2 \Rightarrow E_{prev_2^{(k+1)}} \leq E_{prev_1^{(k+1)}}) \\
\land & \ \ Y^{(k+1)} = E_{prev_2^{(k+1)}}
\end{align*}
\]

and \( some\_executable^{(k)} \) is

\[
\begin{align*}
\vee & \ \ pc_1^{(k+1)} \leq 2 \land (pc_1^{(k+1)} = 1 \lor E_{prev_1^{(k+1)}} \leq X^{(k)}) \\
\vee & \ \ pc_2^{(k+1)} \leq 2 \land (pc_2^{(k+1)} = 1 \lor E_{prev_2^{(k+1)}} \leq X^{(k)})
\end{align*}
\]

The property required of this program is that thread 2 executes its non-sleeping statement after all statements of thread 1, which is expressed as

\[
\lambda \triangleq (pc_1^{(4)} > 2 \land pc_2^{(4)} > 2) \Rightarrow E_{1,2} < E_{2,2}
\]

Yices reports that the formula \( sched \land \neg \lambda \) is unsatisfiable, confirming that the property holds for all executions of the program that respect the timing annotations, as discussed in Section 2. If the annotation of the first statement of thread 1 is changed to 2 time units, Yices reports satisfiability, corresponding to a schedule that first executes \( l_{1,1} \), then \( l_{2,2} \), and finally \( l_{1,2} \). In order to simplify experimentation with different values for the timing annotations and sleeping times, our implementation generates symbolic constants for them whose values can easily be changed in the header of the file.

### 5.2 Suggesting Sleeping Times

A programmer may propose a skeleton of a multi-threaded program with timing annotations for the ordinary (non-sleeping) statements. She may then be interested in finding appropriate sleeping times in order to ensure certain scheduling constraints, expressed by a formula \( \lambda \).

A simple approach with the help of SMT solvers can be employed for addressing this problem: instead of supplying fixed integer constants \( D_{t,i} \) for the durations of the sleep statements, we may use integer variables, possibly constrained to lie within certain bounds. We can then use the SMT solver for checking the satisfiability of the formula \( sched \land \lambda \). In case this formula is unsatisfiable, no choice of values for the sleep statements \( D_{t,i} \) within the given bounds suffices to guarantee the desired properties. Otherwise, we obtain a model that contains assignments of
integer values to the variables $D_{t,i}$ representing the duration of the sleeping statements. The solver thus guarantees the existence of at least one schedule that ensures property $\lambda$. While this does not mean that all schedules respect the property, we can assign the value obtained in this way to the sleeping duration $D_{t,i}$ and use the verification approach described earlier.

Consider again the example of the program analyzed in Section 2 and 5.1 but with a timing annotation 2 for the assignment statement $l_{1,1}$: we know that the statement `sleep(2)` in thread 2 does not ensure that statement $l_{2,2}$ will be executed after all statements in thread 1. We can use the SMT solver to check the satisfiability of the formula $schedule \land \lambda$, where $D_{2,1}$ is now a variable constrained by

$$2 \leq D_{2,1} \land D_{2,1} \leq 7.$$  

The SMT solver computes a model containing the assignment $D_{2,1} = 5$. This suggests replacing the statement in thread 2 by `sleep(5)`, and we can indeed verify that this modified program ensures that $l_{2,2}$ is executed as the last statement.

However, this simple approach exhibits an inherent inefficiency in performance. To figure out an efficient approach for suggesting sleeping times, we observe that our problem could be naturally reduced to the problem of integer parameter synthesis for timed automata. The latter problem has been well studied, e.g. in [JLR13]. Thus, with the idea of reduction, we can utilize existing efficient approaches to solve our problem. We plan to explore this idea in more details in our future work.

5.3 Evaluation of Scalability

We now present some more experimental results for evaluating how our approach scales. We performed 13 experiments on variations of the producer-consumer example introduced in Section 2 that we ran on a PC with a 1.7 GHz Intel Core i7 processor with 8GB memory, using Yices (version 1.0.39) as the core engine to perform constraint solving.

In particular, the results in Table 1 illustrate the scalability of our approach with respect to the number of threads. For these experiments, we consider a pipeline program, which consists of one producer thread $p$, and several copies of consumer threads $c_k$. As shown in the following code snippet, all threads maintain a local variable. The producer thread $p$ initiates and updates the value of its local variable $j_0$. Each consumer thread $c_k$ copies the value of its predecessor’s local variable $j_{k-1}$ into its own local variable $j_k$. Our experiments are performed for $k \in \{2,3,5,10,20,100\}$.

```plaintext
/* Thread p */
l1 : //@1@//
j0 = 0;
l2 : //@2@//
j0 += 2;

/* post-condition: \[ \bigwedge_{k \geq 1} j_k = j_{k-1} \] */
```

```plaintext
/* Thread c_k */
l3 : sleep(2 * k + 1);
l4 : //@2@//
```

The experiments in Table 2 demonstrate the scalability of our approach when programs have loops that are unrolled to different numbers of iterations. For experiments from 1 to 5, we
Table 1: Scalability in terms of threads

<table>
<thead>
<tr>
<th>Test</th>
<th>#threads</th>
<th>Encoding Time</th>
<th>Execution Time</th>
<th>Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.031s</td>
<td>0.003472s</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.031s</td>
<td>0.004341s</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0.033s</td>
<td>0.007635s</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>0.036s</td>
<td>0.037222s</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>0.071s</td>
<td>0.3413s</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>50</td>
<td>0.585s</td>
<td>6.87579s</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>3.970s</td>
<td>105.543s</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 2: Scalability in terms of loops

<table>
<thead>
<tr>
<th>Test</th>
<th>L</th>
<th>Encoding Time</th>
<th>Execution Time</th>
<th>Conflicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0.032s</td>
<td>0.00507s</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0.033s</td>
<td>0.007074s</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0.035s</td>
<td>0.036461s</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>0.036s</td>
<td>1.9266s</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>0.083s</td>
<td>314.761s</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>0.031s</td>
<td>0.017515s</td>
<td>Yes</td>
</tr>
</tbody>
</table>

consider a two-threaded producer and consumer program. Each thread has an infinite loop that is unwound $L$ times; $N$ is chosen as $2L+1$, which is big enough to let both threads perform $L$ loop iterations. The post-condition requires that the $i$-th instance of statement $l_5$ is executed between the $i$-th and $(i+1)$-st instances of statement $l_2$. We ran experiments for $L \in \{2, 3, 5, 10, 20\}$.

```c
/* t1: producer */
l_1: //@1@//
l_2: //l@2@//
i = 2;
loop {
l_3: sleep(2);
    i += 2;
} // k iterations

/* t2: consumer */
loop {
l_4: sleep(2);
l_5: //@2@//
j = i;
} // k iterations

/* post-condition: j == i in each round */
```

The experiment 6 in Table 2 is slightly different. The program skeleton is shown below. It is a modified producer-consumer program that consists of two threads. Each thread has one loop, unwound to 10 iterations. Thread $t_1$ updates $a$ using a value $t$, which is chosen randomly. Thread $t_2$ updates $b$ using the value of $a$. The correctness requirement is that the values of $a$ and $b$ are equal at the end of each iteration, that is, the assignment $l_6$ in thread $t_2$ should be executed after $l_2$ and $l_3$ of thread $t_1$ in every iteration. In this experiment, access conflicts are detected when the program enters into the second iteration of loop.
Analyzing Multi-Threaded Programs With Time Annotations

```c
/* t1: producer */
l1: /*@1@*/
i = 0;
loop {
    l2: /*@2@*/
    t = random();
    a = t + 2;
    l4: sleep(10);
    l5: /*@2@*/
    i++;  
} // 10 iterations
/* t2: consumer */
l6: /*@1@*/
j = 0;
loop {
    l7: sleep(9);
    l8: /*@4@*/
    b = a;
    l9: sleep(8);
    l10: /*@1@*/
    j++;  
} // 10 iterations
/* post-condition:  b == a in each round */
```

These experiments illustrate the scalability of our approach both in terms of number of threads and number of loops. To push the method to its limit, we intentionally focused on models without conflicts: in this case the solver has to check all possible schedules in order to conclude that the input formula is unsatisfiable. In general, this is more demanding than finding a counter model exhibiting the conflict, as can be seen from the last experiment in Table 2.

6 Related Work

SMT-based constraint solving [NOT06] has been an active area of research over the last decade, and has led to the existence of many highly efficient tools, such as Yices or Z3 [DM06, MB08]. Due to technological advances and industrial applications, these solutions have attracted much interest and been applied in different areas, including program analysis and property verification [HSIG10, BPS09, CG12]. Our work proposes an approach to reduce the problem of analyzing multi-threaded programs with time annotations to a constraint solving problem amenable to SMT techniques, which enables us to leverage the strength of existing powerful techniques to solve our problem.

As a fundamental challenge in designing reliable multi-threaded programs, data race detection has attracted significant research efforts. However, existing solutions target at analyzing programs that are based on the use of locks, critical sections and so on [CLL02, EA03, PFH06, SVEH11, RD13], whereas we aim at analyzing multithreaded programs that use time annotations to regulate the program execution and ensure the absence of access conflicts.

Some recent work studied data race detection using constraint solving. For example, ODR [AS09] utilizes constraint solving to determine a schedule that satisfies the recorded information. CLAP [HZD13] reproduces concurrency bugs by solving symbolic constraints and monitoring the local execution paths of threads. While that work is similar to ours concerning the use of constraint solving to analyze program execution for finding bugs related to data races or similar concurrency bugs, our problem is different in that it targets the coordination between threads based on time annotations. To the best of our knowledge, our work is the first that analyzes potential conflicts in multi-threaded programs with time annotations using off-the-shelf SMT
solvers.

7 Conclusion

The analysis of timing behavior is fundamental for ensuring the correctness of real-time programs. In particular, multi-threaded real-time programs can achieve synchronization by relying on global time and scheduling constraints. In this paper, we have proposed a representation of such constraints in the language of SMT solvers, and have shown how this encoding can be used to ensure that simple programs satisfy precedence properties. Although the size of the generated formulas is quadratic in the size of the programs, our experiments seem to indicate that modern SMT solvers are powerful enough for this analysis to scale reasonably well.

In this paper, we have only considered programs in which every thread consists of simple straight-line code, possibly within an infinite loop. In particular, we do not handle conditional statements. It is straightforward to extend the approach to non-deterministic choices between alternative branches, and this can be used to abstract from conditional choices by ignoring the values of program variables, and hence the predicates used in conditional statements. For values that can be represented in SMT solvers, such as (bounded or unbounded) integer variables, our approach can be combined with standard SMT-based program analysis techniques [FP13, Lei13] in order to obtain a more precise verdict, avoiding spurious counter-examples. It will also be interesting to consider scheduling constraints that specify lower and upper bounds, rather than precise running times.

The prospect of using an SMT solver for synthesizing sleep statements, rather than just verifying a fully annotated program, is an interesting avenue for future work. A full encoding of the synthesis problem requires quantification and is therefore a priori outside the scope of decision procedures for linear integer arithmetic. However, a divide-and-conquer approach that repeatedly applies our analyzer may be possible and practical.

Bibliography


Analyzing Multi-Threaded Programs With Time Annotations


Verification of Information Flow Properties
under Rational Observation

Béatrice Bérard\textsuperscript{1}\textsuperscript{*} and John Mullins\textsuperscript{2}\textsuperscript{†,‡}

\textsuperscript{1}Beatrice.Berard@lip6.fr
Université Pierre & Marie Curie
LIP6/MoVe, CNRS UMR 7606, Paris, France

\textsuperscript{2}john.mullins@polymtl.ca
École Polytechnique de Montréal
Campus of the Université de Montréal, Montreal (Quebec), Canada.

Abstract: Information flow properties express the capability for an agent to infer information about secret behaviours of a partially observable system. In a language-theoretic setting, where the system behaviour is described by a language, we define the class of rational information flow properties (RIFP), where observers are modeled by finite transducers, acting on languages in a given family $\mathcal{L}$. This leads to a general decidability criterion for the verification problem of RIFPs on $\mathcal{L}$, implying PSPACE-completeness for this problem on regular languages. We show that most trace-based information flow properties studied up to now are RIFPs, including those related to selective declassification and conditional anonymity. As a consequence, we retrieve several existing decidability results that were obtained by ad-hoc proofs.

Keywords: Information flow, Security predicates, Opacity, Declassification, Conditional anonymity, Rational transducers, Formal verification.

1 Introduction

Motivations. Generic models for information flow properties aim at expressing, in a uniform setting, the various capabilities of observers to infer information from partially observable systems. These models provide a description of the system behaviour, a parametric description of the observation by the environment and the secret parts of the system, and a security criterion. A security property is an instantiation of such a model, with the goal of avoiding a particular information flow. Generic models have been thoroughly investigated, for instance in [Man00, FG01, BKMR08]. They propose various classifications and comparisons of security properties, either for transition systems or directly for traces. In the case of transition systems [FG01, BKMR08], the branching structure permits to express security properties as

\textsuperscript{*} This author is supported by a grant from Coopération France-Québec, Service Coopération et Action Culturelle 2012/26/SCAC.
\textsuperscript{†} This author is supported by the NSERC Discovery Individual grant No. 13321 (Government of Canada), the FQRNT Team grant No. 167440 (Quebec’s Government) and the CFQCU France-Quebec Cooperative grant No. 167671 (Quebec’s Government).
\textsuperscript{‡} This work has been partially done while this author was visiting the LIP6, Université Pierre & Marie Curie.
equivalences like weak or strong (bi-)simulations. For trace-based models, properties are stated as relations between languages, also called security predicates in [Man00].

In addition to classification, an important question about security properties concern their verification: given a system $S$ and a security property $P$, does $S$ satisfy $P$? Since [FG01], much attention has been given to such questions for various classes of systems (or their sets of traces) and security properties [BKMR08, DHRS11, CDM12, DFK+12, BD12, MY14, CFK+14]. This is the problem we consider in this work, for a subclass of trace-based information flow properties.

**Contributions.** We first introduce the class of Rational Information Flow Properties (RIFP), in a language-theoretic setting. In this class, observations are modeled by rational transducers, called here rational observers. For a language $L$ in some family of languages $\mathcal{L}$, an RIFP is then defined as an inclusion relation $L_1 \subseteq L_2$, where $L_1$ and $L_2$ are obtained from $L$ by inductively applying rational observers, unions and intersections. This mechanism produces the set of properties $\text{RIF}(\mathcal{L})$, and a generic decidability result can be stated for the verification problem of these properties. In the particular case of the family $\text{Reg}$ of regular languages, generated by finite automata (also called labelled transition systems), we obtain a PSPACE-complete verification problem for the class $\text{RIF}(\text{Reg})$. We then proceed to show that this result subsumes most existing decidability results for security properties on regular languages, thus establishing the pertinency of our model. This involves expressing properties in our formalism by designing suitable rational observers. We first consider the particular case where observations are functions and we show that opacity properties with regular secrets are RIFPs. To illustrate the expressiveness of RIFPs, we introduce a subclass of functional rational observers that we call rational Orwellian observers and show that several properties including *intransitive non-interference* and *selective intransitive non-interference* for a language $L \in \mathcal{L}$ are in $\text{RIF}(\mathcal{L})$. We also reduce their verification to the verification of opacity w.r.t. Orwellian observers. These observers are more powerful than those considered so far in literature as they model not only observers constrained to a fixed *a priori* interpretation of unobservable events (static observers) or even to observers able to base this interpretation on observation of previous events (dynamic observers), but also able to re-interpret past unobservable events on the base of subsequent observation. We finally consider general observers and we show that all Mantel’s Basic Security Predicates (BSPs) are RIFPs. Finally, we illustrate the applicability of our framework by providing the first formal specification for *conditional anonymity* guaranteeing anonymity of agents unless revocation (for instance, the identity of an agent discovered to be dishonest can be revealed).

**Outline.** The rest of the paper is organized as follows. Rational Information flow properties are defined in Section 2, with the associated decidability results. RIFPs w.r.t. rational observation functions are investigated in Section 3: rational opacity properties as RIFP are presented in Subsection 3.1, Orwellian observers in Subsection 3.2 and their application to *intransitive non-interference* and *selective intransitive non-interference* in Subsection 3.3. RIFPs w.r.t. general rational observation relations are investigated in Section 4: BSPs as RIFPs are presented in Subsection 4.1 and an application of general rational observation relation to conditional anonymity is presented in Subsection 4.2. In Section 5, we discuss related work and we conclude in Section 6.
2 Rational Information flow properties

We briefly recall the notions of finite automata and finite transducers before defining rational information flow properties.

2.1 Automata and transducers

The set of natural numbers is denoted by \( \mathbb{N} \) and the set of words over a finite alphabet \( A \) is denoted by \( A^* \), with \( \varepsilon \) for the empty word and \( A^+ = A^* \setminus \{ \varepsilon \} \). The length of a word \( w \) is written \(|w|\) and for any \( a \in A \), \(|w|_a \) is the number of occurrences of \( a \) in \( w \). A language is a subset of \( A^* \).

Finite Labelled Transition Systems. A finite labelled transition system (LTS or automaton for short), over a finite set \( \text{Lab} \) of labels, is a tuple \( \mathcal{A} = (Q,I,\Delta,F) \), where \( Q \) is a finite set of states, \( I \subseteq Q \) is the set of initial states, \( \Delta \subseteq Q \times \text{Lab} \times Q \) is a finite transition relation and \( F \subseteq Q \) is a set of final states. Note that \( \text{Lab} \) can be an alphabet but also a (subset of a) monoid.

Given two states \( q,q' \in Q \), a path from \( q \) to \( q' \) with label \( u \), written as \( q \xrightarrow{u} q' \), is a sequence of transitions \( q \xrightarrow{a_1} q_1, \ xrightarrow{a_2} q_2, \ldots, q_{n-1} \xrightarrow{a_n} q' \), with \( a_i \in \text{Lab} \) and \( q_i \in Q \), for \( 1 \leq i \leq n - 1 \), such that \( u = a_1 \cdots a_n \). The path is accepting if \( q \in I \) and \( q' \in F \), and the language of \( \mathcal{A} \), denoted by \( L(\mathcal{A}) \), is the set of labels of accepting paths. A regular language over an alphabet \( A \) is a subset of \( A^* \) accepted by a finite LTS over the set of labels \( A \).

Finite Transducers. A finite transducer (or transducer for short) is a finite LTS \( \mathcal{T} \) with set of labels \( \text{Lab} \subseteq A^* \times B^* \) for two alphabets \( A \) and \( B \). A label \( (u,v) \in A^* \times B^* \) is also written as \( u|v \).

The subset \( L(\mathcal{T}) \) of \( A^* \times B^* \) is a rational relation [Sak09] from \( A^* \) to \( B^* \). The transducer \( \mathcal{T} \) is said to realize the relation \( L(\mathcal{T}) \) (see Fig. 1 for basic examples of transducers).

Given a rational relation \( R \), we write \( R(u) = \{ v \in B^* \mid (u,v) \in R \} \) for the image of \( u \in A^* \), \( R^{-1}(v) = \{ u \in A^* \mid (u,v) \in R \} \) for the inverse image of \( v \in B^* \), possibly extended to subsets of \( A^* \) or \( B^* \) respectively, and \( \text{dom}(R) = \{ u \in A^* \mid \exists v \in B^*, (u,v) \in R \} \) for the domain of \( R \). The relation \( R \) is complete if \( \text{dom}(R) = A^* \), it is a function if for each \( u \in \text{dom}(R) \), \( R(u) \) contains a single element \( v \in B^* \).

For a subset \( P \) of \( A^* \), the identity relation \( \{(u,u) \mid u \in P \} \) on \( A^* \times A^* \) is denoted by \( \text{Id}_P \). The composition of rational relations \( R_1 \) on \( A^* \times B^* \) and \( R_2 \) on \( B^* \times C^* \), denoted by \( R_1 \circ R_2 \) (from left to right) or by \( R_2 \circ R_1 \) (from right to left), is the rational relation on \( A^* \times C^* \) defined by \( \{(u,w) \mid \exists v \ (u,v) \in R_1 \land (v,w) \in R_2 \} \) ([EM65]). The family of regular languages is closed under rational relations [Ber79].

2.2 Rational observers

Information flow properties are related to what an agent can learn from a given system. In a language-based setting, the behavior of the system is described by a language \( L \) over some alphabet \( A \), and some function \( \mathcal{O} \) associates with each \( w \in L \) its observation \( \mathcal{O}(w) \) visible by the agent. We generalize the notion of observation by defining \( \mathcal{O} \) as a relation on \( A^* \times B^* \) for some alphabet \( B \), but we restrict \( \mathcal{O} \) to be a rational relation.
Definition 1 (Rational observer) A rational observer is a rational relation \( \mathcal{O} \) on \( A^* \times B^* \), for two alphabets \( A \) and \( B \). The observation of a word \( w \in A^* \) is the set \( \mathcal{O}(w) = \{ w' \in B^* \mid (w, w') \in \mathcal{O} \} \) and for any language \( L \subseteq \text{dom}(\mathcal{O}) \), the observation of \( L \) is \( \mathcal{O}(L) = \cup_{w \in L} \mathcal{O}(w) \).

As pointed out in [DHRS11], a large amount of information flow properties of a language \( L \) are expressed as relations of the form \( \mathcal{O}_1(L) \subseteq \mathcal{O}_2(L) \), for some language theoretic operations \( \mathcal{O}_1 \) and \( \mathcal{O}_2 \). Actually, we show below that \( \mathcal{O}_1 \) and \( \mathcal{O}_2 \) are often rational relations corresponding to some specific observations of \( L \). Also, we define the class of rational information flow properties as those using rational observers, and positive boolean operations:

Definition 2 (Rational information flow property) A rational information flow property (RIFP) for a language \( L \) is any relation of the form \( L_1 \subseteq L_2 \), where \( L_1 \) and \( L_2 \) are languages given by the grammar:

\[
L_1, L_2 ::= L \mid \mathcal{O}(L_1) \mid L_1 \cup L_2 \mid L_1 \cap L_2
\]

where \( \mathcal{O} \) is a rational observer.

Hence, from Def. 1, we recover information flow properties of \( L \) of the form \( \mathcal{O}_1(L) \subseteq \mathcal{O}_2(L) \) for two rational observers, as a particular case. However it has to be noted that Def. 1 does not reduce to these inclusions since rational relations are not closed under intersection [Ber79]. Given a family of languages \( \mathcal{L} \), we define \( \text{RIF}(\mathcal{L}) \) as the set of RIFPs for languages in \( \mathcal{L} \). We immediately have the following general result:

Proposition 1 Let \( \mathcal{L} \) be a family of languages closed under union, intersection, and rational transductions, such that the relation \( \subseteq \) is decidable in \( \mathcal{L} \). Then any property in \( \text{RIF}(\mathcal{L}) \) is decidable.

In particular, the class \( \mathcal{R}eg \) of regular languages satisfies the conditions above, with a PSPACE-complete inclusion problem. We then have:

Corollary 1 The problem of deciding a property in \( \text{RIF}(\mathcal{R}eg) \) is PSPACE-complete.

Proof. It follows from the remark above that the problem is in PSPACE. For PSPACE-hardness, recall that for a language \( K \), the relation \( \mathcal{O}_K \) defined by \( \mathcal{O}_K(w) = \{ w \} \cap K \) is a rational observer if (and only if) \( K \) is a regular language [Sak09]. Let \( L_1 \) and \( L_2 \) be two regular languages, and let \( \mathcal{O}_{L_1}, \mathcal{O}_{L_2} \) be the two corresponding relations, then for \( L = A^* \), we have \( L_1 \subseteq L_2 \) if and only if \( \mathcal{O}_{L_1}(L) \subseteq \mathcal{O}_{L_2}(L) \).

This corollary subsumes many existing decidability results for IF properties. The rest of the paper is devoted to establish reductions of some of these to the \( \text{RIF}(\mathcal{R}eg) \) verification problem.

3 RIF properties with rational functions

In this section, we consider the generic model of opacity introduced in [BKMR08] for transition systems. Opacity is parametrized with observation functions, that are classified in [BKMR08]
as static, dynamic or Orwellian to reflect the computational power of the observer. In a static observation, actions are always interpreted in the same way. It is defined as a morphism and hence, it is a rational function. A particular case of static observer is the projection \( \pi_B \) from \( A^* \) into \( B^* \) for a subalphabet \( B \) of \( A \), so that \( \pi_B(a) = a \) if \( a \in B \) and \( \pi_B(a) = \varepsilon \) otherwise. In a dynamic observation function, interpretation of the current action depends on the sequence of actions observed so far and hence, it is also a rational function.

**Example 1** In Fig. 1 (where all states are final states), the left hand side depicts a transducer realizing the projection from \( \{a,b\}^* \) onto \( \{b\}^* \) while the right hand side depicts a transducer realizing the following dynamic observation function (translated from [CDM12]): The first occurrence of the first action is observed, then nothing is observed until the first occurrence of the second action (\( b \) if the trace begins with \( a \) and \( a \) otherwise) and everything is observed in clear as soon as this second action occurs that is, \( O(aa^*bu) = abu \) and \( O(bb^*au) = bau \) for any \( u \in \{a,b\}^* \).

![Diagram](image)

**Figure 1:** Examples of transducers realizing basic observation functions

In Orwellian observation functions, the current observation depends not only on the prefix of actions observed so far but also on the complete trace. It reflects the capability of the observer to use subsequent knowledge to re-interpret past actions. In the rest of this section we will study opacity w.r.t. rational Orwellian observers.

### 3.1 Opacity w.r.t. rational functions

In its original setting, opacity is related to a language \( L \subseteq A^* \) modelling the behaviour of a system, a function \( O \) from \( A^* \) to \( B^* \) and in addition, a predicate \( \varphi \) given as a subset of \( L \), describing a secret. Two words \( w \) and \( w' \) of \( L \) are observationally equivalent for \( O \) if \( O(w) = O(w') \). The observation class of \( w \) in \( L \) is the set \( [w]_O = \{w' \in L \mid O(w) = O(w')\} = L \cap O^{-1}(O(w)) \).

The secret \( \varphi \) is opaque in \( L \) for \( O \) if for any word in \( \varphi \), there is another word in \( L \setminus \varphi \) such that \( w \) and \( w' \) are observationally equivalent. Hence, \( \varphi \) is opaque if and only if \( O(\varphi) \subseteq O(L \setminus \varphi) \), which we take as definition when \( O \) is a rational function:

**Definition 3** (Rational Opacity) Given a language \( L \subseteq A^* \), a language \( \varphi \subseteq L \) and a rational function \( O \), \( \varphi \) is rationally opaque in \( L \) for \( O \) if \( O(\varphi) \subseteq O(L \setminus \varphi) \).

The information flow deduced by an observer when the system is not opaque is captured by
the notion of secret disclosure: A word \( w \in L \) discloses the secret \( S \) w.r.t. \( O \) if \( [w]_O^L \subseteq \varphi \). We have:

**Proposition 2** Rational opacity properties on languages in some family \( \mathcal{L} \) for regular secrets belong to \( \text{RIF} (\mathcal{L}) \).

**Proof.** As already seen in the proof of Corollary 1, intersection with a regular set \( K \) is a rational observation \( O_K \). Since the secret \( \varphi \) is regular, opacity of \( \varphi \) in \( L \) for \( O \) is equivalent to \( O(\varphi(L)) \subseteq O(\neg \varphi(L)) \).

Non-interference and weak and strong anonymity have been shown to reduce to opacity w.r.t. suitable observers (see [BKMR08]). In [CDM12], PSPACE-hardness is established for opacity of regular secrets for regular languages w.r.t. static and dynamic observers.

### 3.2 Rational Orwellian observers

In the sequel, we denote the disjoint union by \( \cup \). In our context, Orwellian observation functions from [BKMR08] are realized by rational Orwellian observers:

**Definition 4** (Rational Orwellian Observer) A rational Orwellian observer is a rational function, given as a disjoint union of functions: \( O = \bigcup_{1 \leq i \leq n} O_i \), where the domains \( \{\text{dom}(O_i), 1 \leq i \leq n\} \) form a partition of \( A^* \). The partial functions \( O_i \) are called views.

Note that \( O \) is a function because the domains of the views are disjoint. We simply call these functions Orwellian observers for short, since there is no ambiguity in our context. The terminology Orwellian comes from the ability of the observer to somehow see in the future, as illustrated in the following example.

**Example 2** (A simple example) The function \( O = O_a \cup O_b \cup O_\epsilon \) is an Orwellian observer on \( \{a, b\} \) realized by the transducer depicted in Fig. 2. The function is defined by \( O(\epsilon) = \epsilon \) and:

\[
O(w) = \begin{cases} 
\pi_{\{b\}}(w) & \text{if the last letter of } w \text{ is } a \\
\pi_{\{a\}}(w) & \text{if the last letter of } w \text{ is } b.
\end{cases}
\]

Hence, the observer interpretation of the current event depends on the last event of the trace. If it is \( a \) then \( O \) interprets the trace as its projection over \( \{b\} \) and the other way around, if it is \( b \) then it interprets the trace as its projection over \( \{a\} \).

![Figure 2: The Orwellian observer \( O = O_a \cup O_b \cup O_\epsilon \).](image-url)
Despite its observational power, this observer is not able to deduce whether the first event in the trace in $L = (a+b)(a^*+b^*)(a+b)$ is an $a$. Indeed, let $\varphi = a(a^*+b^*)(a+b)$ be the secret, corresponding to the set of traces in $L$ with $a$ as the first event. Then $\varphi$ is opaque w.r.t. $O$ in $L$. To see this, if a secret trace $w$ is observed, examine what $O$ can deduce from this observation.

- If $w$ ends with an $a$ then $O(w) = b^n$ for some $n \geq 0$ but $b^na \notin \varphi$ is also observed by $b^n$.
- If $w$ ends with a $b$ then $O(w) = a^n$ for some $n \geq 0$ but $ba^nb \notin \varphi$ is also observed by $a^n$.

**Example 3 (Static and dynamic observers)**  Static and dynamic observations are of course special cases of Orwellian observers, where $O$ consists of a single complete view. Note that static and dynamic observations preserve prefixes while it is not necessarily the case for Orwellian observations (see examples 2 and 4).

**Example 4 (Intransitive non-interference)**  Let $A = V \cup C \cup D$ be a partition of the alphabet into visible actions in $V$, confidential actions in $C$ and declassification actions in $D$. When a declassification action occurs in a word, the prefix is observed in clear. The corresponding observation function is called in [MY14] the projection on $V$ unless $D$, and defined as a mapping $\pi_{V,D} : A^* \to A^*$ such that $\pi_{V,D}(\epsilon) = \epsilon$ and

$$\pi_{V,D}(ua) = \begin{cases} ua & \text{if } a \in D, \\ \pi_{V,D}(u)a & \text{if } a \in V, \\ \pi_{V,D}(u) & \text{otherwise.} \end{cases}$$

A language $L$ satisfies intransitive non-interference (INI) if $\pi_{V,D}(L) \subseteq L$. Again:

**Proposition 3**  The function $\pi_{V,D}$ is an Orwellian observer, hence INI for languages in $\mathcal{L}$ belongs to RIF($\mathcal{L}$).

**Proof.** The function $\pi_{V,D}$ is a sum of two views: $\pi_{V,D} = O_\epsilon \cup O_D$, realized by the transducers depicted in Fig. 3.

![Figure 3: The Orwellian observer $\pi_{V,D} = O_\epsilon \cup O_D$.](image)

It has been shown in [MY14] that a language $L$ satisfies intransitive non-interference (INI) if and only if $\varphi_{INI} = \{w \in L \mid \pi_{V,D}(w) \neq w \}$ is opaque in $L$ w.r.t. the observer $\pi_{V,D}$.

This can be generalized as follows, showing that many non-interference like properties reduce to opacity w.r.t. Orwellian observers.
Proposition 4 Let $\mathcal{O}$ be a rational idempotent function (i.e. $\mathcal{O}^2 = \mathcal{O}$). Then $\mathcal{O}(L) \subseteq L$ if and only if $\mathcal{O}_\mathcal{O} = \{ w \in L \mid \mathcal{O}(w) \neq w \}$ is opaque in $L$ for $\mathcal{O}$.

Proof. First assume that $\mathcal{O}(L) \subseteq L$ and let $w \in \mathcal{O}_\mathcal{O}$. Then $\mathcal{O}(w) \neq w$. For $w' = \mathcal{O}(w)$, we have: $w' \in L$ and $\mathcal{O}(w') = \mathcal{O}_\mathcal{O}(w) = w'$, hence $w' \notin \mathcal{O}_\mathcal{O}$. Opacity of $\mathcal{O}_\mathcal{O}$ follows.

Conversely, assume that $\mathcal{O}_\mathcal{O}$ is opaque and let $w$ be an element of $L$. If $w \in \mathcal{O}_\mathcal{O}$, then there exists $w' \in L \setminus \mathcal{O}_\mathcal{O}$ such that $\mathcal{O}(w) = \mathcal{O}(w')$. Since $w' \notin \mathcal{O}_\mathcal{O}$, $\mathcal{O}(w') = w'$, hence $w' = \mathcal{O}(w) \in L$. Otherwise, $w \notin \mathcal{O}_\mathcal{O}$ implies $\mathcal{O}(w) = w \in L$. In all cases, $\mathcal{O}(w) \in L$ and $\mathcal{O}(L) \subseteq L$.

Finally, we can state the following:

Proposition 5 Given an Orwellian observer $\mathcal{O}$, deciding opacity of regular secrets w.r.t. $\mathcal{O}$ for regular languages is PSPACE-complete.

Proof. Corollary 1 implies that the problem is in PSPACE. For the PSPACE-hardness, it suffices to observe that dynamic or static observers are particular Orwellian observers for which the problem is already PSPACE-hard.

In the next paragraph, we show that the observation function defined for selective declassification is an Orwellian observer.

3.3 Selective declassification

Intransitive non-interference with selective declassification (INISD) generalizes INI by allowing to each downgrading action to declassify only a subset of confidential actions. It has recently been proposed in [BD12] for a class of Petri net languages (that does not include rational languages). To formalize INISD, the alphabet is partitioned into $A = V \cup C \cup D$ as in example 4.

In addition, with each declassification action $d \in D$ is associated a specific set $C(d) \subseteq C$ of confidential events, with the following meaning: An occurrence of $d$ in a word $w$ declassifies all previous occurrences of actions from $C(d)$, hence these actions are observable while other events in $C$ are not.

Let $\Sigma(D) = \{ \sigma \in D^* \mid |w|_d \leq 1 \text{ for all } d \in D \}$ be the set of repetition-free sequences of downgrading actions in $D$. With any $\sigma = d_1d_2 \ldots d_\sigma \in \Sigma(D)$, we associate the sets:

$A_\sigma = V \cup C \cup \{d_1, \ldots, d_\sigma\}$

$W_\sigma = A_\sigma^* \cdot d_1 \cdot (A_\sigma \setminus \{d_1\})^* \cdot d_2 \cdot \ldots \cdot d_\sigma \cdot (A_\sigma \setminus \{d_1, \ldots, d_\sigma\})^*$

$V_{\sigma,i} = V \cup \{d_j, i + 1 \leq j \leq n\} \cup \bigcup_{j=i+1}^n C(d_j)$, for every $i \in \{0, \ldots, n\}$

with the convention $V_{\sigma,0} = V$, and the projections $\pi_{\sigma,i} : A^* \to V_{\sigma,i}^*$ for every $i \in \{0, \ldots, n\}$.

For a given $\sigma = d_1 \ldots d_n \in \Sigma(D)$, the set $W_\sigma$ contains the words $w$ in $A^*$ where the set of all downgrading actions is precisely $\{d_1, \ldots, d_n\}$ and such that the last occurrence of $d_i$ precedes the last occurrence of $d_{i+1}$ for any $1 \leq i < n - 1$. Note that the family of all these sets $\{W_\sigma, \sigma \in \Sigma\}$ form a partition of $A^*$. Besides, the projection $\pi_{\sigma,i}$ observes in clear any confidential event in $\bigcup_{j=i+1}^n C(d_j)$, in addition to the visible events in $V$ and the declassifying events from $\sigma$. 

68 Verification of Information Flow Properties under Rational Observation
Now the property called INISD in [BD12] can be stated in our general context for a language $L$ as follows: For any $\sigma \in \Sigma(D)$ and for any word $w = w_0d_1w_1 \ldots d_nw_n$ in $L \cap W_\sigma$, there exists a word $w' = w_0d_1w'_1 \ldots d_nw'_n$ in $L \cap W_\sigma$ such that for every $i \in \{0, \ldots, n\}$, $w'_i \in V_{\sigma,i}^*$ and $\pi_{\sigma,i}(w_i) = \pi_{\sigma,i}(w'_i)$. We have:

**Proposition 6** The INISD property for languages in $\mathcal{L}$ belongs to RIF($\mathcal{L}$).

**Proof.** We build an (idempotent) Orwellian observer $\mathcal{O}_{SD}$ such that a language $L$ satisfies INISD if and only if $\mathcal{O}_{SD}(L) \subseteq L$. Let $\mathcal{O}_{SD} = \bigcup_{\sigma \in \Sigma(D)} \mathcal{O}_\sigma$, where the view $\mathcal{O}_1$ and a generic view $\mathcal{O}_\sigma$ for some non empty $\sigma = d_1 \ldots d_n \in \Sigma(D)$ are depicted in Fig. 4.

$$\begin{align*}
\mathcal{O}_1 : & \quad \cdots \quad q_0 \quad d_1 \quad q_1 \quad d_2 \quad \cdots \quad d_n \quad q_n \\
\mathcal{O}_\sigma : & \quad \cdots \quad p_0 \quad c \in C \quad \cdots \quad c \in \Sigma(D) \quad \cdots \quad c \in A_\sigma \setminus V_{\sigma,0} \quad \cdots \quad c \in A_\sigma \setminus V_{\sigma,1} \quad \cdots \quad c \in A_\sigma \setminus V_{\sigma,\sigma} \\
\end{align*}$$

Figure 4: Views of the observation $\mathcal{O}_{SD}$

Let $w = w_0d_1w_1 \ldots d_nw_n$ be a word in $L \cap W_\sigma$, the observation of $w$ is

$$\mathcal{O}_\sigma(w) = \pi_{\sigma,0}(w_0)\pi_{\sigma,1}(w_1) \ldots \pi_{\sigma,1}(w_n).$$

Then $L$ satisfies INISD if and only if $\mathcal{O}_\sigma(L \cap W_\sigma) \subseteq L \cap W_\sigma$ for any $\sigma \in \Sigma(D)$. Since the family $\{W_\sigma, \sigma \in \Sigma\}$ is partition of $A^*$, the family $\{L \cap W_\sigma, \sigma \in \Sigma\}$ is a partition of $L$ and the result follows. Each view $\mathcal{O}_\sigma$ is idempotent and the partitionning also ensures that $\mathcal{O}_{SD}$ itself is idempotent. As a consequence, proposition 4 applies here.

**Remark 1** Also note that a secret $\phi$ is opaque for a language $L$ w.r.t. $\mathcal{O}_{SD}$ if and only if for all $\sigma \in \Sigma(D)$, $\phi \cap W_\sigma$ is opaque for $L \cap W_\sigma$ w.r.t. $\mathcal{O}_\sigma$. Indeed, the result again holds because the family $\{L \cap W_\sigma, \sigma \in \Sigma\}$ is partition of $L$: for all $\sigma \in \Sigma(D)$, $\mathcal{O}_\sigma(W_\sigma) \subseteq W_\sigma$, we have that $\phi$ is opaque for $L$ w.r.t. $\mathcal{O}_{SD}$ if and only if for all $\sigma \in \Sigma(D)$,

$$\mathcal{O}_\sigma(\phi \cap W_\sigma) \subseteq \mathcal{O}_\sigma((L \setminus \phi) \cap W_\sigma) = \mathcal{O}_\sigma((L \cap W_\sigma) \setminus (\phi \cap W_\sigma)).$$

Like before, for regular languages, decidability of INISD as well as opacity under $\mathcal{O}_{SD}$, are consequences of corollary 1 and proposition 6 above. This property is studied in [BD12] for the prefix languages of (unbounded) labelled Petri nets. This family is closed under intersection, inverse morphisms and alphabetical morphisms, hence it is also closed under rational transductions (by Nivat’s theorem [Ber79]), but it has an undecidable inclusion problem. A very nice proof is given in [BD12] for the decidability of the INISD property: it relies on the decidability of the inclusion problem for the particular case of free nets (where all transitions have distinct labels, different from $\varepsilon$).

The following example (inspired from [BD12]) tries to explain the essence of selective declassification.
Example 5 (The Dining Raptors) A circuit followed by a herd of goats is divided in three sections. Each section is guarded by a gate. When gate \(i\) is open, goats can move clockwise from section \(i\) to section \(i+1 \mod 3\). The center of the circuit is occupied by a den of raptors. When gate \(i+1 \mod 3\) is open, a raptor can leave the den and hide around gate \(i\) after opening it and closing gate \(i+1 \mod 3\) to increase chance of success. When a raptor is embushed near a section and there is a goat in this section, the raptor can catch prey and come back to the den.

This scenario is modelled with the transition system

\[
DR(n,m) = \prod_{i=1}^{n} \text{Goat}(i) \times \prod_{j=1}^{m} \text{Raptor}(j) \times \prod_{k=1}^{3} \text{Gate}(k)
\]

obtained by synchronizing the components depicted in Figure 5 on the complementary actions. Goats’ move from gate \(i\) to gate \(i+1 \mod 3\) is modelled with visible action \(l_1\), raptors’ embush action at section \(i\), with the confidential action \(h_i\) and the raptors’ catch action in section \(i\), by the declassification action \(d_i\). Opacity of \(\phi_{DR}\) w.r.t. \(\Theta_{SD}\) in \(L(DR(m,n))\) where

\[
\phi_{DR} = \{u \in L(DR(m,n)) \mid \Theta_{SD}(u) \neq u\}
\]

comes down to absence of information the goats can get from environment about the moment they will be caught until this happens. Hence there is no strategy that they can oppose to the raptors. In the case where initially goats are in section 2 and gates 1 and 3 are opened, as shown in Figure 5, \(L(DR(n,m))\) is not opaque w.r.t. \(\phi_{DR}\) since \(l_3h_1l_2\) reveals the secret \((h_2l_3l_1l_2, l_3h_2l_1l_2 \text{ and } l_3h_2l_2l_1)\) and this, for any number of raptors and goats. This example may be of course modified in various ways as follows. If all three gates are open, goat 1 never realizes she dies since \(l_3h_1d_1\) does not reveals the secret but following this, as gate 2 is now close, goat 2 after \(l_3l_1l_2\) will know that a raptor is embushed at gate 2 since \(l_3l_1h_1d_1l_3l_1h_2l_2\) reveals the secret. If only gate 3 is open, \(l_3h_2d_2h_1l_1\) reveals to the herd,
that one of them is now trapped in section 2. Finally, if we dismantle all three gates, the only synchronizing actions are now the declassification ones and $\varphi_{DR}$ becomes opaque w.r.t. $\mathcal{O}_{SD}$.

4 RIF properties with full rational relations

In this section, we first revisit Basic Security Predicates (BSP) presented in [Man00, Man01] and used as building blocks of the Mantel’s generic security model. In the second part, we investigate anonymity properties.

4.1 Basic Security Predicates

For BSPs, the alphabet $A$ is partitioned into $A = V \cup C \cup N$, where $V$ is the set of visible events, $C$ is the set of confidential events and $N$ is a set of internal events. Informally, a BSP for a given language $L$ over $A$, is an implication stating that for any word $w$ in $L$ satisfying some restriction condition, there exists a word $w'$ also in $L$ which is observationally equivalent to $w$ and which fulfills some closure condition describing the way $w'$ is obtained from $w$ by adding or removing some confidential events. The conditions are sometimes parametrized by an additional set $X \subseteq A$ of so-called admissible events. We prove:

Proposition 7 Any BSP over languages in some family $\mathcal{L}$ belongs to RIF($\mathcal{L}$).

Due to lack of space, the proof is omitted (but will be found in a long version). It mainly consists in exhibiting rational observers together with an inclusion relation such that a language $L$ satisfies a given BSP if and only if this relation holds.

In [DHRS11], the decidability results for all 14 BSPs on regular languages are obtained by ad-hoc proofs establishing that regularity is preserved by the various $op_1$, $op_2$ operations. These include auxiliary functions on languages (like $mark$, $unmark$, etc.) that are unnecessary in our setting. Actually, we show how decidability of BSPs is an immediate consequence of corollary 1 and proposition 7 above. The more difficult case of pushdown systems (generating prefix-closed context-free languages) is also investigated in [DHRS11]: Although context-free languages are closed under rational transductions, they are not closed under intersection and the inclusion problem is undecidable for context-free languages [Ber79]. Finally, several undecidability results are presented in [DHRS11]. In particular, they exhibit an information flow property called Weak Non Inference (WNI) shown to be undecidable even for regular languages. Hence, WNI cannot be expressed neither as a conjunction of BSPs, and as matter of fact, neither as an RIFP. Also, in order to get decidable cases, authors had to restrict the languages and/or the class of properties like reducing the size of the alphabet ($\text{card}(V) \leq 1$ and $\text{card}(C) \leq 1$).

4.2 Conditional anonymity

Conditional or escrowed anonymity is concerned with the revocation of the guarantee, under well-defined conditions, to which an agent agrees, that his identification w.r.t. a particular (non-secret) action will remain secret and in such case, conditional anonymity guarantees the unlinkability of revoked users in order to guarantee anonymity to “legitimate” agents [DS08].
suggested in [BKMR08], Orwellian observation can be used to model conditional anonymity but [BKMR08] contains neither a definition of such a property, nor any investigation of its decidability. We close the gap in this paper.

The alphabet is partitioned into \( A = V \cup P \cup R \) where \( P \) is the set of actions performed by anonymous participants, \( V \) is the set of visible actions and \( R \) is the set of anonymity revocation actions, such that for each participant corresponds a dedicated revocation action \( r \) allowing to reveal the subset \( P(r) \) of all its anonymous actions. Hence the sets \( P(r) \) are mutually disjoint.

In [SS96], definitions of weak and strong anonymity are given in the setting of the process algebra CSP. A language is strongly anonymous (SA) if it is stable under any “perturbation” of anonymous actions where an anonymous action in \( P \) can be replaced by any other element of \( P \). It is weakly anonymous (WA) if it is stable under any permutation on the set of anonymous actions. For a finite set \( Z \), we denote by \( S_Z \) the set of all permutations on \( Z \). We first have:

**Proposition 8** Weak and strong anonymity on languages in \( \mathcal{L} \) belong to \( \text{RIF}(\mathcal{L}) \).

**Proof.** For these two properties, the subalphabet \( R \) of revocation actions is empty. We express the properties in our language-based setting, similarly as in [BKMR08].

A language \( L \) is strongly anonymous w.r.t. \( P \) if \( \theta_{SA}^P(L) \subseteq L \) where \( \theta_{SA}^P \) is the mapping defined on \( A = V \cup P \) by: \( \theta_{SA}^P(a) = P \) if \( a \in P \) and \( \theta_{SA}^P(a) = \{a\} \) otherwise. Such mappings (called rational substitutions in [Ber79]) are well known to be rational relations, hence the result follows.

A language \( L \) is weakly anonymous w.r.t. \( P \) if \( \theta_{WA}^P(L) \subseteq L \) where \( \theta_{WA}^P = \bigcup_{a \in R} \theta_{a} \) and \( \theta_{a} \) is the morphism which applies the permutation \( \alpha \) on letters of \( P \):

\[
\theta_{a}(a) = \alpha(a) \text{ if } a \in P \text{ and } \theta_{a}(a) = a \text{ otherwise.}
\]

With any \( \sigma \subseteq R \), we associate:

- \( W_\sigma = \{ w \in A^* \mid \pi_\sigma(w) \in \sigma^+ \} \), the set of words \( w \) in \( A^* \) where the set of revocation actions appearing in \( w \) is \( \sigma \),
- \( P_\sigma = P \setminus \bigcup_{r \in \sigma} P(r) \), the set of actions of legitimate agents.

We denote by \( 2^R \) the powerset of \( R \) and remark that here also, the sets \( W_\sigma \) for \( \sigma \in 2^R \) form a partition of \( A^* \). In order to provide at any moment strong (weak) anonymization to legitimate agents, we define conditional anonymity as follows:

**Definition 5** With the notations above, a language \( L \) on \( V \cup P \cup R \) is:

- **conditionally weakly anonymous** (CWA) if for any \( \sigma \subseteq R \), \( L \cap W_\sigma \) is WA w.r.t. \( P_\sigma \),
- **conditionally strongly anonymous** (CSA) if for any \( \sigma \subseteq R \), \( L \cap W_\sigma \) is SA w.r.t. \( P_\sigma \).

Now we have:

**Proposition 9** Weak and strong conditional anonymity on languages in \( \mathcal{L} \) belong to \( \text{RIF}(\mathcal{L}) \).

**Proof.** We build rational observers, with a view-like component for each possible subset \( \sigma \) of revoked users, corresponding to \( \theta_{SA} \) (resp. \( \theta_{WA} \)) localized to \( W_\sigma \), i.e. revocation actions are those in \( \sigma \), anonymous actions are restricted to \( P_\sigma \) and visible actions are extended to \( V \cup \bigcup_{r \in \sigma} P(r) \):
Then $L$ is conditionally strongly anonymous (CSA) if and only if $\mathcal{O}_{CSA}(L) \subseteq L$ and $L$ is conditionally weakly anonymous (CWA) if and only if $\mathcal{O}_{CWA}(L) \subseteq L$, which yields the result.

\section{Related works.}

Along the lines, important connections between RIFPs and information flow properties have been established, hence in this section, we will focus on extending the picture.

Algorithms for verifying opacity in Discrete Event Systems w.r.t. projections are presented together with applications in [BBB$^+$07, TK09, SH11, Lin11]. In [BBB$^+$07], the authors consider a concurrent version of opacity and show that it is decidable for regular systems and secrets. In [TK09], the authors define what they called secrecy and provide algorithms for verifying this property. A system property satisfies secrecy if the property and its negation are state-based opaque. In [Lin11] the author provides an algorithm for verifying state-based opacity (called strong opacity) and shows how opacity can be instantiated to important security properties in computer systems and communication protocols, namely anonymity and secrecy. In [SH11], the authors define the notion of K-step opacity where the system remains state-based opaque in any step up to depth-k observations that is, any observation disclosing the secret has a length greater than k. Two methods are proposed for verifying K-step opacity. All these verification problems can be uniformly reduced to the RIFP verification problem.

In [FG01], the authors provide decision procedures for a large class of trace-based security properties that can all be reduced to the RIFP verification problem for regular languages. In [MZ07], decision procedures are given for trace-based properties like non-deducibility, generalized non-interference and forward correctability. The PSPACE-completeness results for these procedures can be reduced to our results.

Concerning intransitive information flow (IIF), non-interference (NI) and intransitive non-interference (INI) for deterministic Mealy machines have been defined in [Rus92]. In [Pin95], an algorithm is provided for INI. A formulation of INI in the context of non-deterministic LTSs is given in [Mul00], in the form of a property called admissible interference (AI), which is verified by reduction to a stronger version of NI. This property, called strong non-deterministic non-interference (SNNI) in [FG01], is applied to $N$ finite transition systems where $N$ is the number of downgrading transitions of the original system. This problem was also reduced to the opacity verification problem w.r.t. Orwellian projections in [MY14]. In [BPR04], various notions of trace-based INI declassification properties are considered and compared. In contrast, our generic model is instantiable to a much larger class of IIF properties.

In [vdM07], the author has argued that Rushby’s definition of security for intransitive policies suffers from some flaw, and proposed some stronger variations. The considered flaw relies to the fact that, if $u \in W_{d_1}$ and $v \in W_{d_2}$, that is $u$ (resp. $v$) declassifies only $h_1 \in H(d_1)$ (resp. $h_2 \in H(d_2)$), then the shuffle of $u$ and $v$ resulting of their concurrent interaction will reveal the order in which $h_1$ and $h_2$ have been executed. The proof techniques used in this paper for deciding the RIFP
verification problem relies on their end-to-end execution semantics and hence does not address this problem.

6 Conclusion

In this paper we have introduced a language-theoretic model for trace-based information flow properties, the RIFPs where observers are modelled by rational transducers. Given a family \( \mathcal{L} \) of languages, our model provides a generic decidability result to the \( \text{RIF}(\mathcal{L}) \) verification problem: Given \( L \in \mathcal{L} \) and a security property \( P \) in \( \text{RIF}(\mathcal{L}) \), does \( L \) satisfy \( P \)? When \( \mathcal{L} \) is the class \( \text{Reg} \) of regular languages, the problem is shown PSPACE-complete. This result subsumes most decidability results for finite systems. In order to prove that, we have shown that opacity properties and Mantel’s BSPs, two major generic models for trace-based IF properties, are RIFPs. We have illustrated the expressiveness of our model by showing that the verification problem of INI and INISD can be reduced to the verification problem of opacity w.r.t. a subclass of rational observers called rational Orwellian observers. Finally we have illustrated the applicability of our framework by providing the first formal specification of conditional anonymity.

References


Using SMT for dealing with nondeterminism in ASM-based runtime verification

Paolo Arcaini¹, Angelo Gargantini¹, Elvinia Riccobene²

¹Dipartimento di Ingegneria, Università degli Studi di Bergamo, Italy
²Dipartimento di Informatica, Università degli Studi di Milano, Italy

Abstract: In runtime verification, operational models describing the expected system behavior offer some advantages with respect to declarative specifications of properties, especially when designers are more accustomed to them. However, nondeterminism in the specification usually affects performances of those operational methods that explicitly represent all the possible conformant states. In this paper, we tackle the problem of dealing with nondeterminism in an operational runtime verification approach based on the use of Abstract State Machines (ASMs). We propose an SMT-based technique in which ASM computations are symbolically represented and conformance verification is performed by means of satisfiability checking. Experiments show that, in most of the cases, the symbolic approach performs better than a technique for ASM-based runtime verification explicitly representing the conformant states.

Keywords: runtime verification, nondeterminism, Abstract State Machines, SMT

1 Introduction

Runtime verification in the form of conformance monitoring consists in checking during runtime that the monitored system behaves like specified. Monitors [FHR13] are used to assess the correctness of a system behavior by checking whether the observed state of the implemented system is conformant to the expected state provided by an abstract specification of the system.

In most approaches dealing with runtime verification of software, the required behavior of the system is specified by means of correctness properties [DGR04]. Temporal logic-based formalisms are very popular in runtime verification [CDR04], especially variants of linear temporal logic [BLS11]. However, operational notations can be used in runtime verification as well [AGR12, BS03a, LDSW09]. Operational specifications offer some advantages with respect to declarative specifications of properties, especially when designers are more accustomed to them. Since operational models may be executable and easier to write and understand, they can be used starting from the first stages of the software development both for documentation purposes and validation activities, as simulation and model-based testing [BS03a]. Moreover, operational approaches usually permit to trace, by means of a step-wise model refinement [BS03b], the relation between the specification and the implementation.

The Abstract State Machine (ASM) operational method [BS03b] has been used in [AGR12], where we developed a framework (called CoMA) for runtime verification of Java programs. The technique proposed for conformance monitoring makes use of Java annotations to link the concrete implementation to its ASM formal model.
In case of not fully predictable systems, models specifying system behaviors are nondeterministic. Nondeterminism in the specification can also be due to underspecification, when some implementation choices are left abstract, or model abstraction used to reduce complexity. ASMs permit to model nondeterministic behaviors in a concise and natural way, thanks to the built-in `choose` operator that allows to nondeterministically select a computation path among all possible ones. However, nondeterminism poses further challenges to the ASM-based runtime monitoring approach. In [AGR13], we extended CoMA in order to deal with nondeterminism by explicitly building and keeping track of all the possible next states: that approach becomes very inefficient when the number of next states is high. Although this limitation is unavoidable if one needs to keep track of all the nondeterministic choices performed, techniques can be developed to mitigate this shortcoming. For example, symbolic representation can help to specify in a concise way the relation between a state and the set of reachable next states.

Following our ongoing research on ASM-based runtime verification of Java programs, to overcome the limits of CoMA, we here present an approach exploiting a symbolic representation of a machine computation [VBGS09], similarly to what is done in bounded model checking where a program trace is represented by means of a propositional formula [Bie09].

Starting from the theoretical framework we presented in [AGR12, AGR13], we here provide a novel definition of conformance in the presence of nondeterminism, and propose CoMA-SMT, an SMT-based technique in which ASM computations are symbolically represented and the conformance verification is performed by means of satisfiability checking.

The devised approach requires the designer to write an ASM specification of the system and to link it to the implementation to be monitored. The specification is automatically translated into an SMT logical context. During runtime, the SMT context is step by step extended with the current transitions, and the values observed in the monitored system are asserted. The implementation behaves correctly as long as the SMT context stays satisfiable: if the context becomes unsatisfiable, a runtime fault is observed.

We have compared CoMA-SMT with CoMA, and our preliminary experiments show that, in most of the cases, the symbolic approach performs better than techniques based on explicit state representation. Moreover, other experiments show that CoMA-SMT can handle a more general notion of conformance, not supported by CoMA, that is much more difficult to check.

Section 2 presents a background about ASMs, nondeterminism in runtime monitoring, and CoMA. In Section 3, to deal with conformance in case of nondeterminism, we improve the theoretical framework CoMA is based on. How to symbolically represent an ASM is described in Section 4. Section 5 presents the proposed runtime monitoring approach. Preliminary experiments are reported in Section 6. Some related work is presented in Section 7, and Section 8 concludes the paper.

2 Background

2.1 ASMs

Abstract State Machines (ASMs), whose complete presentation can be found in [BS03b], are an extension of FSMs, where unstructured control states are replaced by states with arbitrary complex data. The states of an ASM are multi-sorted first-order structures, i.e., domains of
objects with functions and predicates defined on them. ASM states are modified by transition relations specified by “rules” describing the modification of the function interpretations from one state to the next one. There is a limited but powerful set of rule constructors that allow to express guarded actions (if-then), simultaneous parallel actions (par) or sequential actions (seq). Appropriate rule constructors also allow modeling nondeterminism (existential quantification choose) and unrestricted synchronous parallelism (universal quantification forall).

An ASM state \( s \) is represented by a set of couples \((location, value)\). ASM locations, namely pairs \((function-name, list-of-parameter-values)\), represent the abstract ASM concept of basic object containers (memory units). Location updates represent the basic units of state change and are given as assignments, each of the form \( loc := v \), where \( loc \) is a location and \( v \) its new value.

Functions are classified as derived, i.e., those coming with a specification or computation mechanism given in terms of other functions, and basic which can be static (never change during any run of the machine) or dynamic (may change as a consequence of agent actions or updates). Dynamic functions are distinguished between monitored (only read by the machine and modified by the environment), and controlled (read and written by the machine).

A computation of an ASM is a finite or infinite sequence \( s_0, s_1, \ldots, s_n, \ldots \) of states of the machine, where \( s_0 \) is an initial state and each \( s_{n+1} \) is obtained from \( s_n \) by simultaneously firing all the transition rules which are enabled in \( s_n \). The (unique) main rule is a transition rule and represents the starting point of the computation. An ASM can have more than one initial state. It is possible to specify state invariants which are checked during simulation.

For our purposes, we use a definition of ASM adapted from [BS03b]: \( ASM = < signature, funcDefs, funcInit, r\_main > \), where signature contains the function declarations, funcDefs the derived functions definitions, funcInit the definitions of initials values for the controlled functions, and \( r\_main \) is the main rule.

### 2.2 Runtime monitoring and nondeterminism

Runtime verification of nondeterministic behaviors using state-based specifications, like ASMs, is particularly complex since the specification takes into account all the possible correct system evolutions. The nondeterminism due to monitored quantities (e.g., the system inputs or external actions), called external, is still easy to monitor: once these quantities are fixed by the environment, the system behaves deterministically. However, in most cases, the specification is internally nondeterministic, sometimes even when the system is deterministic. The following scenarios can be identified:

- The system has a nondeterministic behavior (for instance a Java program containing a call to a method in the class \texttt{java.util.Random}), as well as the abstract specification.
- The system has a deterministic behavior, while the specification is nondeterministic. This situation arises when the model is more abstract (with less implementation details) than the corresponding system, and is frequent in the object oriented context. Bekkaert and Steegmans [BS01] have shown that nondeterminism in the behavioral specifications of object oriented conceptual models can simplify the representation of complex functionalities and achieve a better separation of concerns.
A simple example of nondeterministic system  As running case study, we consider a tank that can be either filled or emptied (see Fig. 1). At every instant, the level cannot be increased/decreased more than fifty units of product. The tank is full when it contains one thousand units of product. Such tank can be modeled by a simple ASM, as shown in Code 1.

2.3 CoMA: Conformance Monitoring through ASMs

CoMA [AGR12] is a technique for runtime monitoring of Java programs through ASMs; Fig. 2 shows the structure of the framework. The runtime monitor observes the behavior of a Java code and determines its correctness w.r.t. the ASM specification working as an oracle of the expected behavior: while the software system is executed, the monitor checks conformance between the values of the observed elements and the expected state. A link between a Java class and an ASM is established using a set of Java annotations (1). The Observer detects when the Java object observed state is changed (2), and leads the corresponding ASM to perform a machine step (3). The Analyzer evaluates the conformance between the Java execution and the ASM behavior (4). A complete description of CoMA can be found in [AGR12]. Here only some basic definitions are reported.

Let $C$ be a Java class, $O_C$ an object of $C$, and $ASM_C$ an ASM model of the expected behavior of any object of $C$.

$OS(C)$ is the set of observed elements, i.e., all public fields, and pure\(^1\) public methods of the class $C$ the user wants to observe. Observable elements are linked to ASM functions by the function $\text{link} : OS(C) \rightarrow \text{Funcs}$.

\(^1\)A method is pure when its execution does not affect the program state.
Changing methods, CM(C), are the non-pure methods whose execution can change the element values of OS(C) and that the user wants to monitor. A changing step is defined by the triple \((s, m, s')\), being \(m\) a method in CM(C), and \(s\) and \(s'\) the states of an object \(OC\) before and after the method execution.

**Definition 1** (State and step conformance) A state \(s\) of \(OC\) conforms to a state \(S\) of ASM\(_C\) if all observed elements of \(C\) have values in \(OC\) conforming to the values of the functions in ASM\(_C\) linked to them, i.e.,

\[
\text{conf}(s, S) \equiv \forall e \in OS(C) : \text{val}_\text{Java}(e, s) = \text{val}_{\text{ASM(link(e),S)}}
\]

A change step \((s, m, s')\) of an instance \(OC\), with \(m\) a method of CM(C), conforms with a step \((S, S')\) of ASM\(_C\) if \(\text{conf}(s, S) \land \text{conf}(s', S')\).

**Definition 2** Univocal runtime conformance A class \(C\) is univocally runtime conforming to its specification ASM\(_C\) if the following conditions hold:
1) the initial state \(s_0\) of the computation of \(OC\) conforms to *one and only one* initial state \(S_0\) of the computation of ASM\(_C\), i.e., \(\exists! S_0\) initial state of ASM\(_C\) such that \(\text{conf}(s_0, S_0)\);
2) for every change step \((s, m, s')\) with \(s\) the current state of \(OC\), \(\exists! (S, S')\) step of ASM\(_C\) with \(S\) the current state of ASM\(_C\), such that \((s, m, s')\) is step conforming with \((S, S')\).

Univocal runtime conformance requires that the next step of \(OC\) is state-conforming with *one and only one* of the next states of the specification. Therefore, in case of nondeterminism, during the runtime monitoring CoMA chooses, among the next states of the ASM, the unique state that conforms to the Java state. Fig. 3a depicts this situation.

Since CoMA represents the states in an explicit way, as in explicit state model checkers (e.g., SPIN), we will refer to it as an explicit state monitoring approach.
Example Let’s consider the Tank case study. If the conformance between the ASM specification and the Java program is based on the value of the level, then the conformance is univocal. At each step, the ASM has between fifty and a hundred one possible next states; such states, however, are uniquely identified by the value of level. So, if the implementation is correct, only one of the possible next ASM states is conformant.

3 Dealing with multiple conformance

Definition 2 assumes that, at every time, there is only one possible current ASM state which is conformant to the current Java state. However, the implementation should be considered conformant also when there exists at least one conformant state.

Definition 3 Conformant set Given a Java object $O_C$, let $s_n$ be the state obtained after $n$ executions of changing methods. We call $\text{confSet}(s_n)$ the set of ASM states reachable in $n$ steps and conformant with $s_n$.

Definition 4 Multiple runtime conformance A class $C$ is multiply runtime conforming to its specification $ASM_C$ if the following conditions hold:
1) the initial state $s_0$ of the computation of $O_C$ conforms to at least one initial state $S_0$ of the computation of $ASM_C$, i.e., $\exists S_0$ initial state of $ASM_C$ such that $\text{conf}(s_0, S_0)$;
2) for every change step $(s, m, s')$ with $s$ the current state of $O_C$, $\exists (S, S')$ step of $ASM_C$, $S \in \text{confSet}(s)$, such that $(s, m, s')$ is step conforming with $(S, S')$.

Multiple runtime conformance can be depicted as in Fig. 3b: The current Java state $s$ is conformant with the ASM states $\text{confSet}(s) = \{S_1, \ldots, S_k\}$; the Java state $s'$ is produced by the execution of the method $m$ at the state $s$; ASM states $S'_1, \ldots, S'_n$ are reachable in one step from $\text{confSet}(s)$; the Java state $s'$ is conformant with ASM states $\{S'_i, \ldots, S'_j\} \subseteq \{S'_1, \ldots, S'_n\}$.

Example Let’s consider the Tank case study. If the conformance between the ASM specification and the Java program is only based on the value of function full, then the conformance is multiple. As seen before, at each step the ASM has between fifty and a hundred one possible next states; at most one next state can have value true for full. Therefore, if the implementation is correct and the value of full is false, more than one of the possible next ASM states can be conformant with the implementation.

Supporting multiple runtime conformance in monitoring Dealing with multiple conformance in an explicit state runtime monitoring approach as CoMA, would require to keep track of all the possible states to which the monitored system can be conformant. At the $i$th step of monitoring, the framework should record in the set $\text{confSet}(s_i)$ (see Def. 3) the ASM states reachable in $i$ steps of simulation that are conformant with the current Java state (as proposed in [FHR13]). If $\text{confSet}(s_i)$ becomes empty, then an error is found. In our approach, instead, we would like to represent such reachable and conformant states in a symbolic way.

In order to do this, we describe how to symbolically represent the set of states $RS_i$ reachable in $i$ steps of the ASM execution, and the transition relation induced by the ASM transition rules.
between states in $RS_i$ and their successor states in $RS_{i+1}$. These formulae establish a logical context.

At runtime, in order to perform the conformance checking, we extend the context by asserting a set of formulae stating the values of the observed elements in the implementation current state. A Satisfiability Modulo Theories (SMT)\(^2\) solver can be used to check the satisfiability of the obtained context. If the context becomes unsatisfiable, then the implementation is not conformant. As SMT solver we use Yices [DM06].

Note that we could use BDDs to symbolically represent ASM states; however, it has been shown that for bounded model checking a SAT/SMT approach scales better [Bie09, Kur08]. So, since the approach we are proposing has several commonalities with BMC, we adopt the SMT approach.

**Example** Let’s consider the Tank case study. The ASM states reachable in one step can be symbolically described as $\text{level} = 0 \land (\text{level} - 50 \leq \text{level'} \leq \text{level} + 50)$, where $\text{level'}$ represents the updated version of $\text{level}$.

### 4 ASM symbolic representation

This section describes how to build the logical context, symbolically representing any computation of length $n$ of a given ASM. Section 4.1 describes how to represent the set of states $RS_i$ at level $i$ of the ASM computation – i.e., those reachable by the machine in $i$ steps – and the transition relation induced by the ASM transition rules between states in $RS_i$ and their successor states in $RS_{i+1}$. Some mapping functions permit to obtain, from an ASM model, a sequence of Yices definitions and assertions (commands) parameterized with the index $i$. In order to symbolically represent ASM computations of depth $n$, the Yices commands must be instantiated $n$ times with concrete values for $i$ (i.e., $i = 0, \ldots, n - 1$), as described in Section 4.2.

#### 4.1 Mapping from ASM to Yices

For the lack of space, we do not report the mapping of ASM domains to Yices types, that, however, is not relevant for the understanding of the work. The signature symbols are defined by applying the mapping reported in Table 1 ($T_d$), being $i$ the current level; note that, for each ASM function, a fresh Yices constant is created at every step. Function definitions are asserted by applying the correspondence given in Table 2 ($T_d$), at level $i$. Transition rules are symbolically represented by a formula describing the transition relation between states in $RS_i$ and those in $RS_{i+1}$. This formula is asserted by recursively applying the mapping reported in Table 3 ($T_r$), starting from the main rule.

Both function definitions and transition rules contain terms: the symbolic representation of terms in states of $RS_i$ is given in Table 4 ($T_e$).

Note that in an update rule (first row of Table 3) the location term on the left-hand side of the rule refers to states in $RS_{i+1}$, while the term on the right-hand side of the rule refers to states in

---

\(^2\) An SMT problem is a decision problem for logical formulae with respect to combinations of background theories expressed in classical first-order logic with equality. An SMT instance is a generalization of a boolean SAT instance in which various sets of variables are replaced by predicates from a variety of underlying theories.
Using SMT for dealing with nondeterminism in ASM-based runtime verification

**Table 1**: $T_i$: Mapping schema of the ASM function declarations to Yices at step $i$

<table>
<thead>
<tr>
<th>ASM function declaration</th>
<th>Yices</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>funcType</code> ∈ {...}</td>
<td><code>(define f' :: ...)</code></td>
</tr>
<tr>
<td><code>funcType</code> ∈ {controlled, monitored, derived, static}</td>
<td><code>(assert (= f' $T_i(fd, i)$))</code></td>
</tr>
<tr>
<td><code>funcType</code> ∈ {D1, ..., Dn}</td>
<td><code>(assert (and (= $T_i(fd_1, ..., d_{m_1}$), i) $T_i(fd_1)$))</code></td>
</tr>
<tr>
<td><code>funcType</code> ∈ {D, D_1, ..., D_n}</td>
<td><code>(assert (and (= $T_i(fd_1, ..., d_{n_1}$), i)) $T_i(fd_1)$))</code></td>
</tr>
</tbody>
</table>

**Table 2**: $T_i$: Mapping schema of ASM function definitions to Yices at step $i$

<table>
<thead>
<tr>
<th>ASM transition rule</th>
<th>Yices</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>updLoc := updTer</code></td>
<td>$T_i(updLoc, i + 1) = T_i(updTer, i)$</td>
</tr>
<tr>
<td><code>par R_1 ... R_n endpar</code></td>
<td>`(and $T_i(R_1, i ... T_i(R_n, i))$)</td>
</tr>
<tr>
<td><code>if guard then Rthen else Reelse endif</code></td>
<td>`(if $T_i(guard, i) T_i(Rthen, i) T_i(Reelse, i))$)</td>
</tr>
<tr>
<td><code>if guard then Rthen endif</code></td>
<td><code>($=&gt; T_i(guard, i) T_i(Rthen, i)$) $T_i(Rthen, i)$) </code></td>
</tr>
<tr>
<td><code>for all x_1 in D_1, ..., x_n in D_n</code></td>
<td>`(and $r_1 ... r_m$ with $m = \prod_{j=1}^n</td>
</tr>
<tr>
<td><code>choose x_1 in D_1, ..., x_n in D_n</code></td>
<td><code>(define cv_i :: D_i)</code></td>
</tr>
<tr>
<td><code>for each x_j: (define cv'_j :: D_j)</code></td>
<td><code>(=&gt; (exists (d_1 :: D_1 ... d_n :: D_n) $T_i(guard[x_1 \mapsto d_1, ..., x_n \mapsto d_n], i)$) $(and T_i(guard[x_1 \mapsto cv'_1, ..., x_n \mapsto cv'_n], i)$ $T_i(R[x_1 \mapsto cv'_1, ..., x_n \mapsto cv'_n], i)$) $T_i(R[x_1 \mapsto cv'_1, ..., x_n \mapsto cv'_n], i)$) </code></td>
</tr>
<tr>
<td><code>main rule r_main = mainBody</code></td>
<td>`(assert $T_i(mainBody, i)$)</td>
</tr>
</tbody>
</table>

**Table 3**: $T_i$: Mapping schema of ASM transition rules to Yices at step $i$

For this reason, the term on left-hand side is mapped with parameter $i + 1$, whereas the term on the right-hand side is mapped with parameter $i$.

In order to guarantee the semantics of ASM steps, we must impose that locations that are not updated by the computation step keep their value unchanged. The formula obtained by applying the mapping function $T_i$ to the transition rules (Table 3) does not guarantee such condition. Therefore, for each controlled location $f^i$, we need to add to the context the following formula

\[ (= \not (or \ guard_1 \ ... \ guard_n)) = f^{i+1} f^i \)
being guard₁, …, guardₙ the conditions upon which ℓᵢ is updated, and ℓᵢ₊₁ the location in the next state. The conditions are statically derived from the transition rules leading to the updates of the location. Let unchLocsᵢ be the assertion of the set of formulae obtained by instantiating the previous formula for all the controlled locations of the ASM model at level i.

### 4.2 Building the logical context

The logical context is initialized and then extended at every level i along the ASM computation.  

*Context initialization* consists of a set of formulae symbolically representing the ASM initial state(s), i.e., signature, and function definitions and initializations. Context initialization `contInit` is built as follows

\[
contInit = T_ℓ(signature, 0), T_d(funcDefs, 0), T_d(funcInit, 0)
\]

where the comma is to be intended as a sequential concatenation operator.

*Context extension* consists of a set of formulae representing the transition relation between states in `RSᵢ` and their successor states in `RSᵢ₊₁`. This requires to define the signature at level `i + 1`. Context extension `contExtᵢ` at level `i` is built as follows

\[
contExtᵢ = T_ℓ(signature, i + 1), T_d(funcDefs, i + 1), T_ℓ(r_main, i), unchLocsᵢ
\]

At every step, `T_ℓ(signature, i + 1)` represents a fresh copy of the signature and it must be added to the context before the assertion of the rules `T_ℓ(r_main, i)`, because `T_ℓ(r_main, i)` describes the relation between states in `RSᵢ` and states in `RSᵢ₊₁`.

Code 2 reports the context initialization and context extension at level 0 of the Tank case study.
5 SMT-based Runtime Verification

We here describe an approach to perform runtime monitoring of Java programs, in which the expected behavior is given in terms of ASMs; the approach, called CoMA-SMT, exploits the symbolic representation of ASMs introduced in Section 4. We keep the overall architecture of the CoMA framework (Fig. 2). The technique, described in Section 2.3, for linking an ASM specification to a Java class is the same. However, the way to simulate the ASM (the simulator in Fig. 2) and to perform the conformance checking (the analyzer in Fig. 2) differ. The approach is also able to check multiple runtime conformance (see Def. 4), not supported by CoMA.

Alg. 1 depicts the monitoring procedure of the proposed approach. In order to ease the description, Alg. 1 reports both the execution of the monitored Java program (reported in a frame in the algorithm) and the execution of the monitoring framework. When a Java object of the monitored class is created (line 1), the framework creates a logical context (line 2) and add the context initialization to it (line 3). The monitoring consists in a never ending loop in which, when a Java changing method \( m \) is executed (line 6), the following actions are executed:

- the context is extended for describing the transition relation between ASM states at the current level \( i \) and the possible next states at level \( i + 1 \) (line 7);
- from the Java state \( s_{i,\text{Java}} \), obtained after the changing method execution (line 8), and from the linking between the specification and the code, the framework builds the formula \( \text{javaValuesConstr} \) in which the linked ASM locations are forced to assume the actual values of the corresponding Java elements (line 9). Let \( f_1^{i+1}, \ldots, f_g^{i+1} \) be the locations linked to Java fields or methods (i.e., the observed elements) and \( v_1, \ldots, v_g \) the values of the linked fields and methods at state \( i + 1 \). Formula \( \text{javaValuesConstr} \) is built as follows:

\[
\text{assert } (\text{and } (f_1^{i+1} = v_1) \ldots (f_g^{i+1} = v_g))
\]

- formula \( \text{javaValuesConstr} \) is asserted in the logical context (line 10);
- the logical context is checked for satisfiability (line 11):
Algorithm 1 CoMA-SMT: monitoring procedure

1: \( oC \leftarrow \text{new} \ C() \) \hspace{1cm} \triangleright \text{Monitored program: Java object instantiation}
2: \( ctx \leftarrow \text{mk\_context}() \) \hspace{1cm} \triangleright \text{Logical context creation}
3: \( \text{add\_to\_context}(\text{contInit}, ctx) \) \hspace{1cm} \triangleright \text{Context initialization}
4: \( i \leftarrow 0 \)
5: \textbf{while} true \textbf{do}
6: \( \triangleright \text{Monitored program: execution of a changing method } m \)
7: \( \text{add\_to\_context}(\text{contExt}_i, ctx) \) \hspace{1cm} \triangleright \text{Context extension at level } i
8: \( s_{Java} \leftarrow \|oC\| \) \hspace{1cm} \triangleright \text{Observed Java state after step } i
9: \( \text{javaValuesConstr} \leftarrow \text{getValues}(s_{Java}) \) \hspace{1cm} \triangleright \text{Observed elements values}
10: \( \text{add\_to\_context}(\text{javaValuesConstr}, ctx) \) \hspace{1cm} \triangleright \text{Assertion of the observed values}
11: \( \text{if check}(ctx) = \text{UNSAT} \) then
12: \( \triangleright \text{Is SAT?} \)
13: \textbf{return} \texttt{NotConformantException} \hspace{1cm} \triangleright \text{The Java state is not conformant}
14: \( i \leftarrow i + 1 \)
15: \textbf{end while}

- if the context is unsatisfiable, it means that the implementation is not conformant with the specification. In this case, the monitoring is interrupted by throwing an error message (line 12);
- otherwise, if the context is still satisfiable, it means that the implementation is conformant and the monitoring can continue.

Example Let’s consider a Java implementation of the Tank case study, having a pure method \( \text{getLevel()} \) returning the level of the tank, and a boolean pure method \( \text{isFull()} \) reporting whether the tank is full; the two methods are respectively linked (by means of the Java annotation \@MethodToFunction) with ASM functions level and full. The implementation has also a changing method \( \text{add}(\text{int quantity}) \) (annotated with \@RunStep) that permits to increase/decrease the tank level of a given quantity. After having detected the object instantiation and a call of method add, the monitoring framework has built the logical context as shown in Code 2. Let’s suppose that the values returned by the observed elements getLevel and isFull are respectively 23 and \texttt{false}. The formula built by the framework for checking the conformance is \( (\text{and} (= \text{level1} 23)(= \text{full1} \texttt{false})) \).

6 Experiments

We run all the experiments on a Linux PC, Intel(R) Core(TM) i7, and 8 GB of RAM. The result of each experiment is the average of 20 runs.

6.1 Comparison with CoMA

As first experiment, we have successfully executed CoMA-SMT on all the case studies provided by the CoMA benchmarks. We have used univocal conformance, since it is the only kind of
Using SMT for dealing with nondeterminism in ASM-based runtime verification

(a) Tank case study

(b) Tic-tac-toe

Figure 4: Univocal conformance – CoMA-SMT and CoMA

conformance supported by CoMA.

Then, we have compared the two frameworks in terms of execution times. Fig. 4a shows the time taken for monitoring a Java implementation of the Tank case study in the presence of univocal conformance (i.e., linking both level and full functions) using the two frameworks. We have executed the Java code by calling the method `add` an increasing number of times. We can see that, in both frameworks, the execution time grows linearly with the number of steps. However, CoMA-SMT always performs better than CoMA, and the gain increases with the number of steps. We have observed that, for the case studies in which the number of possible next conformant states is always high, CoMA-SMT always performs better than CoMA. This means that, when the number of reachable states is high, a symbolic representation is more performant than an explicit representation.

Nonetheless, we found that, when the number of possible reachable states is low, CoMA may perform better than CoMA-SMT. We took the Tic-tac-toe case study introduced in [AGR13] as an example of nondeterministic system; it models the Tic-tac-toe game in which, at each step, the user and the computer alternatively make a move. The user makes a move by calling a given method (the actual parameters of the method represent the move coordinates); the computer makes a move by nondeterministically choosing an empty cell of the board. The linking between the implementation and the specification is based on the configuration of the board. Fig. 4b shows the comparison of the monitoring times in the two frameworks: CoMA-SMT performs better for less than around 15 moves, while, for more than 15 moves, CoMA is advantageous. Notice that, when a player has won or there is a tie, there is always only one possible next state because the game has terminated and any method invocation cannot change the board configuration; therefore, we can argue that after around 15 moves the game is very likely terminated. So, the results in Fig. 4b suggest that the explicit state representation is advantageous when the number of reachable states is low; using the symbolic representation, instead, always requires to check for satisfiability at each step, and so the monitoring time grows linearly in any case. As future work, we could devise some optimization techniques able to detect states in which the monitoring could be stopped (since no violation can be found in the future) or, at least, in which the logical context could be simplified.
6.2 Univocal and multiple conformance

We have then experimented the influence of multiple conformance in monitoring; since CoMA does not support multiple conformance, we have performed the experiments only using CoMA-SMT. We have monitored the Java implementation of the Tank case study in the presence of univocal conformance (i.e., linking both level and full functions) and in the presence of multiple conformance (i.e., linking only function full). We have executed the Java code by calling the method `add` an increasing number of times. Fig. 5a shows the results in terms of execution times.

We have also executed the Tic-tac-toe implementation in the presence of univocal conformance (i.e., the linking is based on the board configuration) and in the presence of multiple conformance (i.e., the linking is only based on a boolean flag specifying whether the game is terminated).

As expected, the runtime monitoring is more computationally onerous in the presence of multiple conformance rather than in the presence of univocal conformance. Indeed, in univocal conformance only one state is selected when asserting the values of the Java values (line 10 in Alg. 1): therefore the solver, when checking for satisfiability, must handle a logical context in which most of the variables are fixed. In multiple conformance, instead, the solver must handle a logical context that is much more complex.

7 Related work

Extended literature exists about runtime verification [FHR13, DGR04]. Declarative specifications are more used in runtime monitoring [BLS11, CDR04, KLHN09] and they also deal with nondeterminism. Some attempts to use operational specifications exist and we relate to them; however, as far as we know, using SMT for doing runtime verification through operational specifications has never been proposed.

In [LDSW09], a formal specification-based software monitoring system is presented. In that approach the behavior of a concrete implementation (a Java code) is checked for compliance with a Z specification. The execution of a program is monitored by a debugger and the formal specification is executed in parallel with a specification animator. Although the approach is similar to ours in the use of an operational specification, it does not support nondeterminism.

Operational approaches based on Abstract State Machines can be found in [AGR12, BS03a].
CoMA [AGR12] has been already described in Section 2.3 and we have extensively compared to it. The approach in [BS03a] handles runtime verification of .NET programs. It uses AsmL, a dialect of ASMs, to describe the expected behavior. As CoMA, that approach only supports univocal conformance (the authors say that the choices must be angelical), whereas we also support multiple conformance. Moreover, the technique adopts an explicit state approach to do the conformance checking, while we use a symbolic one.

Several works handle the problem of runtime verification of concurrent programs, in which nondeterminism derives from threads’ interleaving [BENS11]; instead, we focus on single thread programs that can be internally nondeterministic.

Symbolic representation of ASMs has already been presented in [VBGS09], for doing bounded model checking of AsmL models (a concrete syntax for ASMs). The main difference with our approach is that the transformation in [VBGS09] requires an intermediate representation as model program (i.e., a collection of guarded update rules), while we directly support ASM models.

8 Conclusions and Future work

We have proposed CoMA-SMT, an approach for runtime monitoring of nondeterministic Java programs using the ASM formal method. The approach improves an existing framework (CoMA), based on the explicit representation of ASM states. The new approach proposes a symbolic representation of the ASM computation, and exploits an SMT solver for checking the conformance between the monitored Java program and its ASM specification. While CoMA requires that, at each step, only one of the next ASM states is conformant with the observed Java state, CoMA-SMT is also able to monitor Java programs that, at a given step, are conformant with more than one ASM state. Preliminary experiments show that the symbolic approach can sometimes perform better than CoMA, but we plan to apply CoMA-SMT to more complex case studies.

As future work, we plan to devise some optimization techniques for dealing with the complexity of formulae in multiple conformance. A context simplification could be achieved by asserting, during monitoring, further constraints on the set of conformant states; in case of unsatisfiability, backtracking techniques should be necessary, although backtracking has never been used in runtime monitoring so far [FHR13]. Moreover, we could adapt some techniques of BMC [Bie09] for defining stopping monitoring policies able to detect situations in which any possible further system evolution cannot violate the conformance relation.

Bibliography


QBF with Soft Variables

Sven Reimer, Matthias Sauer, Paolo Marin, Bernd Becker

Institute of Computer Science, Albert-Ludwigs-Universität Freiburg
Georges-Köhler-Allee 051, D-79110 Freiburg, Germany
{reimer|sauerm|marin|becker}@informatik.uni-freiburg.de

Abstract: QBF formulae are usually considered in prenex form, i.e. the quantifier block is completely separated from the propositional part of the QBF. Among others, the semantics of the QBF is defined by the sequence of the variables within the prefix, where existentially quantified variables depend on all universally quantified variables stated to the left.

In this paper we extend that classical definition and consider a new quantification type which we call soft variable. The idea is to allow a flexible position and quantifier type for these variables. Hence the type of quantifier of the soft variable can also be altered.

Based on this concept, we present an optimization problem seeking an optimal prefix as defined by user-given preferences. We state an algorithm based on MaxQBF, and present several applications – mainly from verification area – which can be naturally translated into the optimization problem for QBF with soft variables. We further implemented a prototype solver for this formalism, and compare our approach to previous work, that differently from ours does not guarantee optimality and completeness.

Keywords: QBF, MaxQBF, prefix, dependency, optimization problem

1 Introduction

For design automation tasks in safety-critical or other domains where precise answers are necessary, applications employing quantified Boolean formulae (QBF) logic have been demonstrated to be an effective solution: In contrast to the traditional SAT-based 01X-encoding [JBM+00], QBF delivers accurate answers by accurately considering unknown and unspecified signals [SB01], which is also named Zr-encoding [HB07]. In particular universally quantified variables in QBF are used to accurately model the behavior of unknown circuit lines. The encoding of such problems is often not trivial and the effort required to solve them strongly depends on the prefix order, i.e. the user-given dependencies between the existential and universal variables. Additionally, there is an increasing interest for optimization problems with QBF [BLV08, IJM13] in the game theory domain [CP04]. However, in the classical definition of a QBF, the prefix is known and fixed.

Research interests considering the possibility of changing the given prefix structure is focused on simplifying the solving to a given QBF without changing its meaning, e.g. by optimizing decision strategies in search-based solving [GNT07]. More generalized approaches apply so-called dependency schemes [Sam08] focusing on tractable algorithms, i.e. heuristics with polynomial complexity, optimizing the prefix order wrt. the dependencies between the variables.

As we will show in this paper, there is a relation between the prefix order and the optimization problem for maximizing unknown values within a circuit [RS04, NSB07, PR00]. E.g., one seeks for a maximum number of don’t care signals at the input of a circuit in order to generalize a found solution [RS04]. Since in QBF these don’t care or unknown values are modeled by universally quantified variables, one needs the possibility to change the quantifier for a particular variable from existential to universal and vice versa in order to optimize the number of unknowns. In pure QBF, quantifiers have to be given a-priori and cannot be changed.

This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center “Automatic Verification and Analysis of Complex Systems” (SFB/TR 14 AVACS).
In this paper we present a new formalism based on QBF supporting a dynamic prefix using so-called soft variables. In contrast to previous perceptions, our approach allows to change the prefix position of such soft variables dynamically. In particular, the quantifier from existential to universal can be altered and vice versa. Technically the mechanism provides a set of possible QBF prefixes. The challenge is to find a satisfiability preserving prefix such that the soft variables are quantified following an ordering as close as possible to user-given preferences.

We define an optimization problem for QBF with soft variables and an algorithm returning the optimal prefix. The algorithm is based on the optimization problem called MaxQBF \cite{CFLS93}. We highlight the importance of our formalism by showing applications from different verification areas, which are naturally covered by our formalism. Previous work on these applications has two main disadvantages:

- Most approaches use a SAT-based encoding, which is less precise than a QBF representation (with soft variables).
- Applications using QBF formulations usually need more encoding effort than approximate SAT encodings. As we will show, the optimization problem for solving QBF with soft variables is PSPACE-complete (as QBF), but allows a more compact and in many cases "easier" representation of the problem statement than pure QBF.

We developed a prototype solver for solving both: 1) MaxQBF based on \cite{IJM13} and 2) QBF with soft variables based on MaxQBF. First experimental results demonstrate the applicability of our implementation and the advantage of the solution compared to heuristic approaches.

The paper is structured as follows: In §2 we introduce basic information, terminology, and notation used throughout the paper. In §3 the concept of QBF with soft variables is introduced as well as an algorithm to solve the problem. We introduce applications for the formalism and first experimental results for some of those in §4. Lastly, §5 concludes the paper and discusses future work on this topic.

2 Preliminaries

In this section we introduce the notation and some background on solving techniques, and some further details necessary for a good comprehension of this paper. We assume that propositional logic and the SAT problem is familiar to the reader. The interested reader is referred to \cite{BHMW09} for further insight in optimization problems in propositional logic and QBF.

2.1 QBF

The logic of quantified Boolean formulae (QBF) is an extension of SAT by bounding the variables to quantifiers $Q \in \{\exists, \forall\}$. In the following, we will consider prenex conjunctive normal form formulae (PCNF) $\psi = Q_1 X_1 \ldots Q_n X_n \varphi$, where $\varphi$ is a quantifier free matrix in CNF. We denote with $\mathcal{V}$ the set of variables occurring in $\varphi$. We call $P = Q_1 X_1 \ldots Q_n X_n$ the prefix of $\psi$. W.l.o.g. $Q_i \neq Q_{i+1}$ for all $i \in \{1, \ldots, n-1\}$, i.e. the existential and universal quantifiers have an alternating order, and $X_1, \ldots, X_n$ are disjoint sets of variables with $X_1 \cup \cdots \cup X_n := \mathcal{x}$.\footnote{Note that $X_i = \emptyset$ or $\mathcal{x} \neq \mathcal{V}$ may hold}

We define a quantifier level function $\delta : X \to \mathbb{N}$ for a variable $x$ as $\delta(x) = i$ for $x \in X_i$. In the following, w.l.o.g. we set $Q_1 = \exists$, in particular, variables on odd quantification levels are always existentially quantified and on even levels universally quantified. We say that an existential variable $x \in X_i$ depends on all universal variables $y \in X_j$ with $j < i$ and $Q_j = \forall$, i.e. the assignment of $x$ depends on the assignment of $y$.

A variable $x \in W = \mathcal{V} \setminus \mathcal{x}$ is not bound by any quantifier and is called free variable. If $\psi$ contains free variables, then $\psi$ is an open QBF, otherwise it is a closed formula. Given a set of variables $\mathcal{V}$
and an open QBF ψ, we say that some variable valuation f ∈ (W → B) is a model of ψ iff ψ(f) = ⊤, and ψ(W) is satisfiability equivalent to the closed QBF where the free variables W are existentially quantified at level 1.

The co-factor of a propositional formula is defined as φ|x = φ[x = ⊤] and φ|⊥ = φ[x = ⊥] respectively, where φ[x = c] denotes a propositional formula with every occurrence of x replaced by the value c ∈ {⊤, ⊥}. Similarly, ψ|x is defined as the QBF where the matrix φ is replaced by φ|x and all superfluous quantifications are excluded from the prefix (and analogously for ψ|⊥). We define the semantics of QBF recursively as follows:

$$Q_x \psi = \begin{cases} 
\psi|x \land \psi|\neg x & \text{if } Q = \forall \\
\psi|x \lor \psi|\neg x & \text{if } Q = \exists 
\end{cases}$$

If φ does contain an empty clause, ψ is unsatisfied, otherwise satisfied in case no variable is left unassigned.

2.2 MaxQBF

MaxQBF (also referred to as MaxQSAT in the literature) is an extension of QBF for optimization problems [CFLS93]. The semantics is similar to the analogous optimization problem for SAT, called MaxSAT. A MaxSAT procedure tries to satisfy as many clauses of a propositional formula as possible. We denote these clauses as soft clauses. The optimization problem for QBF ensures that these soft clauses have to be satisfied for every branch of the universal variables. There are some extensions of MaxSAT which can be adapted quite naturally to MaxQBF:

- Weighted MaxSAT/QBF: Each soft clause c is associated with a non-negative integer weight ω(c). If a clause is satisfied, the clause gains ω(c) as score, otherwise the score of the clause is 0. The objective is to maximize the sum of the scores.

- Partial MaxSAT/QBF: There are two types of clauses: hard clauses and soft clauses. Hard clauses must be satisfied, while soft clauses may be satisfied. The objective is to maximize the number of satisfied soft clauses.

- Weighted Partial MaxSAT/QBF: Combination of the two concepts stated above.

To the best of our knowledge there exist only two references proposing exact algorithms to solve optimization problems over quantified formulæ: [BLV08] considers quantified constraint optimization problems (QCOP) as extension of constraint optimization problems (COP). In [IJM13] the authors propose two algorithms for solving the related QMaxSAT problem: Given a QBF instead of optimizing the number of satisfied soft clauses, the goal is to maximize a linear pseudo-Boolean cost function. The MaxQBF problem can be easily expressed by this formalism and vice versa.

3 QBF with Soft Variables

In this section we describe the concept of soft variables in QBF, as well as an algorithmic approach for solving this formalism using MaxQBF, and an algorithm for solving MaxQBF problems. Finally, we state some details of our implementations.

3.1 Soft Variables

The syntax of soft variables in the context of a QBF is defined as follows.

**Definition 1** Consider a QBF ψ = P.φ, a Boolean variable s ∈ V and a set of natural numbers L. We call s a soft variable iff the following properties hold:
1. \( s \) does not occur in \( P \), and

2. \( L \) is a set of quantification levels on which \( s \) is designated to be quantified, also called the quantification set of \( s \).

We denote with \( \mathbb{F}_L s \) a soft variable \( s \), where \( L \) contains all possible prefix positions. Furthermore we write \( \psi(\mathbb{F}_{L_1}s_1,\ldots,\mathbb{F}_{L_n}s_n) \), indicating that \( s_1,\ldots,s_n \) are soft variables of \( \psi \) with possible prefix positions \( L_j \), for each \( j \in \{1,\ldots,n\} \). We write \( S_{\psi} \) indicating the set of all soft variables in \( \psi \).

Moreover, we allow soft variable groups \( S \), written \( \mathbb{F}_L S \). We write \( \psi(\mathbb{F}_{L_1}S_1,\ldots,\mathbb{F}_{L_n}S_n) \) for a QBF \( \psi \) with different groups of soft variables \( S_j \). We also allow combinations of soft variables and groups of soft variables within a QBF. In the following we distinguish the two cases by using capital letters for groups and small letters for variables.

**Definition 2** Let \( \psi(\mathbb{F}_{L_1}s_1,\ldots,\mathbb{F}_{L_n}s_n) \) be a QBF with soft variables. We call \( \Lambda : S_{\psi} \rightarrow \mathbb{N} \) the level function of \( \psi \) which maps each variable \( s_j \) to a quantification level \( l \in L_j \). Given such a level function \( \Lambda \) we denote \( \psi(\mathbb{F}_{L_1}s_1,\ldots,\mathbb{F}_{L_n}s_n)_{\Lambda} \) as the QBF where every soft variable is mapped to one possible quantification level within \( P \) of \( \psi \) according to \( \Lambda \). For each soft variable of one soft variable group \( s_j \in S \) the level function has to be same value, i.e. \( s_j = l \) for a fixed level \( l \) and for all \( s_j \in S \).

Intuitively \( \psi(\mathbb{F}_{L_1}s_1,\ldots,\mathbb{F}_{L_n}s_n) \) is a set of QBF with different prefixes for the soft variables, to be more precise every QBF resulting from every possible level function \( \Lambda \) of \( \psi \).

**Example 1** Consider the QBF with soft variables \( \psi_1 \) with a matrix \( \varphi_1 = (s_1 \vee y) \land (s_1 \vee z) \land (\overline{y} \vee z) \land (\overline{y} \land \overline{z}) \), where \( s_1 \) is a soft variable in the scope of all existential levels:

\[
\psi_1(\mathbb{F}_{(1,3)}s_1) = \forall y \exists z. (s_1 \vee y) \land (s_1 \vee z) \land (\overline{y} \vee z) \land (\overline{y} \land \overline{z})
\]

By definition we are allowed to set \( s_1 \) either to the first or third (i.e. an existential) level, resulting in the two possible level functions with \( \Lambda_1(s_1) = 1 \) and \( \Lambda_2(s_1) = 3 \), and therefore in two possible prefixes \( P_1 = s_1 \land y \exists z \) and \( P_2 = y \exists z \exists s_1 \). The QBF \( \psi_1(\mathbb{F}_{(1,3)}s_1)_{\Lambda_1} = P_1 \cdot \varphi_1 \) is unsatisfiable, whereas \( \psi_1(\mathbb{F}_{(1,3)}s_1)_{\Lambda_2} = P_2 \cdot \varphi_1 \) results in a satisfied matrix for all branches of the universal variable \( y \), i.e. is satisfied.

Now consider the formula \( \psi_2 \) with a matrix \( \varphi_2 = (y \vee z) \land (s_2 \vee z) \land (\overline{y} \lor \overline{z}) \), where \( s_2 \) is a soft variable with level 2 and 3 as possible prefix positions:

\[
\psi_2(\mathbb{F}_{(2,3)}s_2) = \forall y \exists z. (y \vee z) \land (s_2 \vee z) \land (\overline{y} \lor \overline{z})
\]

By applying the soft variable concept we obtain two possible level functions with \( \Lambda_3(s_2) = 2 \) and \( \Lambda_4(s_2) = 3 \) and the respective prefixes \( P_3 = y \forall z \exists s_2 \) and \( P_4 = y \exists z \exists s_2 \). Both \( \psi_2(\mathbb{F}_{(2,3)}s_2)_{\Lambda_3} = P_3 \cdot \varphi_2 \) and \( \psi_2(\mathbb{F}_{(2,3)}s_2)_{\Lambda_4} = P_4 \cdot \varphi_2 \) are satisfiable.

Based on this syntax we briefly define the semantics of a QBF with soft variables.

**Definition 3** A QBF with soft variables \( \psi(\mathbb{F}_{L_1}s_1,\ldots,\mathbb{F}_{L_n}s_n) \) is satisfied if there exists a level function \( \Lambda \) such that \( \psi(\mathbb{F}_{L_1}s_1,\ldots,\mathbb{F}_{L_n}s_n)_{\Lambda} \) is satisfied. If for all possible level functions the resulting QBF is unsatisfied, we say \( \psi(\mathbb{F}_{L_1}s_1,\ldots,\mathbb{F}_{L_n}s_n) \) is unsatisfiable.

We want to consider an optimization problem for QBF with soft variables. To do so, we first define a score function \( \sigma \) as follows:

**Definition 4** Let \( \psi(\mathbb{F}_{L_1}s_1,\ldots,\mathbb{F}_{L_n}s_n) \) be a QBF with soft variables \( s_1,\ldots,s_n \). The score function \( \sigma : (S_{\psi} \times \mathbb{N}) \rightarrow \mathbb{N} \) is defined for each variable \( s_j \) as \( \sigma(s_j,l) = \chi_{s_j,l} \), where \( \chi_{s_j,l} \) is a user-given score.
and \( l_j \) the corresponding level with \( \Lambda(s_j) = l_j \). The overall score \( \chi_\Lambda \) is the sum of all scores for one level function \( \Lambda \): \( \chi_\Lambda = \sum_{j=1}^{n} \chi_{s_j,l_j} \).

The score function allows to define the optimization problem \( \Omega(\psi(\mathcal{L}_1s_1, \ldots, \mathcal{L}_ns_n)) \) as follows:

**Definition 5** Given a QBF with soft variables \( \psi(\mathcal{L}_1s_1, \ldots, \mathcal{L}_ns_n) \) and a score function \( \sigma \) of \( \psi \), the optimization problem \( \Omega(\psi(\mathcal{L}_1s_1, \ldots, \mathcal{L}_ns_n)) \) is to find a level function \( \Lambda \) maximizing the score \( \chi_\Lambda \) such that \( \psi(\mathcal{L}_1s_1, \ldots, \mathcal{L}_ns_n)_{\Lambda} \) is satisfied, i.e. \( \Omega(\psi(\mathcal{L}_1s_1, \ldots, \mathcal{L}_ns_n)) = \max_\Lambda \chi_\Lambda \). If \( \psi(\mathcal{L}_1s_1, \ldots, \mathcal{L}_ns_n)_{\Lambda} \) is unsatisfied, the score is \( \chi_\Lambda = 0 \).

**Example 2** Consider the QBF \( \psi_1 \) of Example 1. We chose the following scores: \( \sigma(s_1,1) = 2 \) and \( \sigma(s_1,3) = 1 \). The QBF \( \psi_1(\mathcal{L}_1s_1)_{\Lambda_1} \) is unsatisfied, and therefore the score evaluates to \( \chi_{\Lambda_1} = \chi_{s_1,1} = 0 \). Since \( \psi_1(\mathcal{L}_1s_1)_{\Lambda_2} \) is satisfied, we yield the score \( \chi_{\Lambda_2} = \chi_{s_1,3} = 1 \), which is also the maximum score over all possible level functions \( \Lambda \) of \( \psi_1 \) and thus the result of the maximization problem \( \Omega(\psi(\mathcal{L}_1s_1)) = 1 \) with \( \Lambda_2 \).

Considering \( \psi_2 \) from Example 1, we chose the scores \( \sigma(s,2) = 2 \) and \( \sigma(s,3) = 1 \). \( \psi_2(\mathcal{L}_2s_2)_{\Lambda_3} \) as well as \( \psi_2(\mathcal{L}_2s_2)_{\Lambda_4} \) are satisfied and gain the scores \( \chi_{\Lambda_3} = \chi_{s_2,2} = 2 \) and \( \chi_{\Lambda_4} = \chi_{s_2,3} = 1 \). Hence, we obtain \( \Omega(\psi_2(\mathcal{L}_2s_2)) = 2 \) with level function \( \Lambda_3 \) as solution to the optimization problem.

Please note, in this example we chose the weights for both formulae in a meaningful way: The QBF where the soft variable is quantified on a level which is more likely to be unsatisfiable gets the higher score.

**Proposition 1** Solving the optimization problem \( \Omega(\psi(\mathcal{L}_1s_1, \ldots, \mathcal{L}_ns_n)) \) is PSPACE-complete.

**Proof.** (Sketch) To decide the optimization problem we need to solve a QBF problem for each possible level function. QBF is in PSPACE [SM73] and since we only need to store the currently optimal level function and its score, which needs polynomial space, the optimization problem for QBF with soft variables is also in PSPACE.

For PSPACE-hardness we reduce from QBF which is PSPACE-hard [SM73]. Let \( \psi \) be a QBF \( \psi = Q_1X_1 \ldots Q_nX_n.\varphi \). Let \( m \) be a new existential level with \( m > n \). We define a QBF with soft variables: \( \psi'(\mathcal{L}_1mX_1, \ldots, \mathcal{L}_nmX_n) = \varphi \), i.e. \( \psi' \) is obtained from \( \psi \) by eliminating \( P \) and declaring each variable in \( X \) as part of a soft variable group \( X_i \). The possible prefix positions are chosen such that the whole group \( X_i \) is either quantified at level \( i \) as in \( \psi \), or at the new level \( m \). We define the score function \( \sigma \) such that for each soft variable group \( X_i \) both \( \sigma(X_i,m) = 0 \) and \( \sigma(X_i,i) = 1 \) holds, i.e. the preference is to set the group to the level it belonged to in \( \psi \). The maximization problem for QBF with soft variables tries to set all groups of \( \psi' \) to the corresponding position in \( \psi \). From the result of the QBF with soft variables problem \( \Omega(\psi') = \max_\Lambda \chi_\Lambda \) we directly obtain the satisfiability value of \( \psi \): If at least one soft variable group is not set at level \( i < m \) in \( \lambda \) of \( \psi' \) or max \( \chi_\Lambda = 0 \) holds, the QBF \( \psi \) is unsatisfiable. Otherwise, if all groups \( X_i \) are set to level \( i \), the QBF \( \psi \) is satisfiable.

\( \square \)

### 3.2 Algorithm

In this section we describe an algorithm to solve the optimization problem for QBF with soft variables which is based on Weighted Partial MaxQBF. To do so, we present an approach for solving MaxQBF (and its extensions) based on iterative MaxSAT algorithms.

#### 3.2.1 Solving QBF with Soft Variables

Let \( \psi(\mathcal{L}_1s_1, \ldots, \mathcal{L}_ns_n) = P.\varphi \) be a QBF with soft variables \( s_1, \ldots, s_n \), \( P \) a prefix over the variables \( X \), and \( \varphi \) a quantifier free matrix over the variables \( V \). In the following we show how to transform
this problem into a Weighted Partial MaxQBF problem. Therefore, we need to transform our formalism into a QBF without any soft variables.

We denote $e_{\text{max}}$ of $\psi$ as the innermost existential level with no further (existential or universal) quantification level right of $e_{\text{max}}$. Hence, $e_{\text{max}}$ is the right-most existential level wrt. the current prefix $P$ and all levels $l \in L^j$ for all $j \in \{1, \ldots, n\}$.

The following extensions/modifications have to be applied for each soft variable $s_j$, $j \in \{1, \ldots, n\}$: We quantify $s_j$ on level $e_{\text{max}}$ existentially such that the appearances of $s_j$ in the matrix is well-defined. For each level $l \in L^j$ we introduce a new quantification of a helper variable $s'_j$ on level $l$ and a new free variable $f'_j$ of $\psi$. These helper variables allows us to alter the quantification levels by setting $f'_j$ appropriately. Therefore, we connect these variables to $\varphi$ by the constraint $f'_j \Rightarrow (s'_j \equiv s_j)$ for each $l \in L^j$, i.e. we set $s_j$ to $s'_j$ (and hence $s_j$ quantified on level $e_{\text{max}}$ semantically “behaves” like the variable $s'_j$ quantified on level $l$) if the free variable $f'_j$ is set to $\top$. We have to ensure that exactly one $f'_j$ for all $l \in L^j$ is set to $\top$ and all other $f'_j$, $k \neq l$ are set to $\bot$, which is also known as an exactly-one constraint. We add the encoding of such a constraint for each new free variable of every soft variable to $\varphi$. We declare every clause of $\varphi$ as well as every additional constraint we introduced so far as hard clause of the Weighted Partial MaxQBF instance. Finally, we add an additional unit clause for each free variable $f_j$, declared as soft clause with weight $\sigma(s_j, l)$.

A Weighted Partial MaxQBF solver maximizes these weights, i.e. the score function $\sigma$ is directly mapped to a the maximum number of satisfied unit soft clauses triggering a soft variable to be set on the corresponding position in the prefix. For the free variables $f_j$ we obtain a model from which we can extract the level function $\Lambda$: if $f'_j$ is set to $\top$, $\Lambda(s_j) = l$ holds. By construction, $\chi_\Lambda$ is the maximum score of $\psi(\mathcal{W}_{L^1, s_1}, \ldots, \mathcal{W}_{L^n, s_n})$.

**Remark 1** Handling groups $S$ of soft variables is analogous to the method described above. Instead of introducing a free variable $f'_j$ for each variable and level, we only have to introduce one $f''_S$ for each level and group, i.e. every variable of a group $S$ is either quantified simultaneously on $l$ or not. The additionally introduced constraint is $f''_S \Rightarrow (s'_j \equiv s_j)$ for each $s_j \in S$.

**Example 3** Consider $\psi_1$ from our running example. First, we introduce the soft variable $\mathcal{W}_{(1,3), s_1}$ into $P$ by introducing an existential quantification $\exists s_1$ on level $e_{\text{max}} = 3$. By definition, $\mathcal{W}_{(1,3), s_1}$ can be quantified on both existential levels 1 and 3, therefore we introduce two existential helper variables $s'_1$ and $s''_1$ as well as two new free variables $f'_1$ and $f''_1$. Moreover, we introduce a CNF representation for the constraints $f'_1 \Rightarrow (s'_1 \equiv s_1)$ and $f''_1 \Rightarrow (s''_1 \equiv s_1)$, the constraints for the exactly-one-constraint, and the unit clauses for the free variables into $\varphi_1$, resulting in:

\[
\exists s'_1 \forall y \exists s''_1 \exists z \exists s_1. \quad \begin{align*}
& (s_1 \lor y) \land (s_1 \lor z) \land (y \lor z) \land (s_1 \lor y) \land \\
& (f'_1 \lor s'_1 \lor s_1) \land (f''_1 \lor s''_1 \lor s_1) \land (f'_1 \lor s'_1 \lor s_1) \land (f''_1 \lor s''_1 \lor s_1) \land \\
& (f'_1 \land s'_1) \land (f''_1 \land s''_1) \land (f'_1 \land (f''_1) \land (f''_1) \land (f''_1))
\end{align*}
\]

This instance is passed to a Weighted Partial MaxQBF solver, where $(f'_1)$ and $(f''_1)$ are declared as soft clauses with weight 2 and 1 respectively. All other clauses are declared as hard clauses.

Analogously we obtain for $\psi_2$ the following Weighted Partial MaxQBF instance:

\[
\forall s'_2 \forall y \exists s''_2 \exists z \exists s_2. \quad \begin{align*}
& (y \lor z) \land (s_2 \lor z) \land (s_2 \lor y) \land \\
& (f'_2 \lor s'_2 \lor s_2) \land (f''_2 \lor s''_2 \lor s_2) \land (f'_2 \lor s'_2 \lor s_2) \land (f''_2 \lor s''_2 \lor s_2) \land \\
& (f'_2 \land s'_2) \land (f''_2 \land s''_2) \land (f'_2) \land (f''_2)
\end{align*}
\]
with the soft clauses \((f_2^3)\) and \((f_3^3)\) and weights 2 and 1, respectively.

3.2.2 Solving MaxQBF

In this section we present an approach for solving MaxQBF and its extensions based on iterative methods from MaxSAT [ZSM03]. The method is also mentioned in [LJM13], but we present some further details. First, we describe the iterative approach in MaxSAT and then we introduce how to adapt this technique for MaxQBF.

Iterative MaxSAT solvers as introduced in [ZSM03] add a new relaxation literal \(r\) to each soft clause. If \(r\) is set to \(\bot\), the corresponding soft clause has to be satisfied, otherwise \((r = \top)\) the clause is relaxed, i.e., it is satisfied by the relaxation literal and hence, the (original) soft clause does not have to be satisfied by the set of variables belonging to the original CNF. Let \(\varphi\) be the original MaxSAT instance with \(m\) soft clauses including a unique relaxation literal \((r_1,\ldots,r_m)\) for each soft clause. In an iterative approach the relaxation literals are connected to the inputs of a cardinality network [Sin05] and the instance \(\varphi \land \beta(r_1,\ldots,r_m) \land (\bar{\pi})\) is handed over to a SAT solver, where \(\beta(r_1,\ldots,r_m)\) is the CNF encoding for such a cardinality network, with \(r_1,\ldots,r_m\) as inputs and \(o_1,\ldots,o_m\) as outputs. The additional unit clause \((\bar{\pi})\) demands at least \(i\) arbitrary inputs of the network to be set to \(\bot\), therefore \(i\) soft clauses have to be satisfied. If this SAT instance is satisfiable, there are at least \(i\) simultaneously satisfied soft clauses, otherwise there exists no solution with \(i\) satisfied soft clauses. The solution is narrowed by the incremental usage of the underlying SAT solver until a value \(k\) is identified with \(\varphi \land \beta(r_1,\ldots,r_m) \land (\bar{\pi})\) being satisfiable and \(\varphi \land \beta(r_1,\ldots,r_m) \land (\bar{\pi}_{k+1})\) being unsatisfiable. This value \(k\) is the maximum number of simultaneously satisfied soft clauses.

This procedure can be easily extended to the Partial MaxSAT and Weighted MaxSAT concepts: In Partial MaxSAT, hard clauses are not connected to the network, and in Weighted MaxSAT a soft clause \(c\) is connected \(\omega(c)\) times into the network, where \(\omega(c)\) is the weight of the clause \(c\). The algorithm for Weighted Partial MaxSAT is obtained by a straight-forward combination of both methods.

We adapt this concept for MaxQBF by using an incremental QBF solver. The cardinality network and the relaxation variables are encoded as in MaxSAT. The variables for the encoding are added as free variables of the QBF, since a soft clause has to be satisfied for every branch of the universal variables and we are able to obtain a model for these variables as a result from solving the open QBF. Likewise to MaxSAT we call the QBF solver incrementally in order to find a value \(k\) such that the QBF with \(k\) satisfied soft clauses is satisfied, but unsatisfied with \(k + 1\) clauses. The extension for Partial MaxQBF, Weighted MaxQBF and Weighted Partial MaxQBF is done in the same manner as for the MaxSAT extension.

3.2.3 Implementation Details

We implemented a Weighted Partial MaxQBF solver as well as the algorithm for solving QBF with soft variables.

For the MaxQBF solver we used the QBF solver quantum [RPSB12] and implemented incremental functionality based on [MMR12]. For the model of the free variables (representing the level function \(\Lambda\)) we adapt techniques of [BELM12]. For scalability reasons we also use a preprocessor for QBF. The preprocessor is implemented for incremental usage (see also [MMLB12]) as well as model preservation techniques known from SAT [EB05] adapted to QBF. All variables which are introduced in context of a soft variable are set as “Don’t touch” [KLSB11] in the preprocessor, i.e. these variable are excluded from several preprocessing techniques, among others variable elimination and pure literal detection.

For an exactly-one-constraint with more than 5 possible levels (i.e. \(|L^j| > 5\)), we use the encoding for \(LT^n_{SEQ}^1\) as presented in [Sin05] using \(O(|L^j|)\) clauses and \(O(|L^j|)\) additional auxiliary variables.
only. Otherwise we introduce a standard one-hot encoding using $O(|L|^2)$ additional clauses.\(^2\)

For some instances (cf. §4.1) the encoding can be simplified to a pure Partial MaxQBF problem if the following requirements hold: 1) the soft variables are only defined over two different levels and 2) for all soft variables the weight function assigns for one level 0 and for the other level a constant value $c > 0$. If this is the case we can add just one free variable per soft variable with the constraints: $f^l_j \Rightarrow (s^l_j \equiv s_j)$ and $f^k_j \Rightarrow (s^k_j \equiv s_j)$, where $l$ is the preferred level with weight $c$ and $k$ is the level with weight 0. Moreover, just one soft clause ($f^l_j$) for each soft variable is added.

4 Applications

In this section we present several applications, mainly from verification and testing of circuits, covered by QBF with soft variables. In §4.1 we present optimization problems considering unknowns in a circuit, and in §4.2 we present further applications, for example optimal solutions for dependency schemes.

The measurements for all case studies were performed on a machine using one core of a 3.3 GHz Intel Xeon, and limiting the memory to 4 GB.

4.1 Maximizing Unknowns in a Circuit

There are plenty of applications which ask for maximizing or optimizing unknown values within a circuit. There are two general problem statements, given a circuit together with a property which has to hold, we want to optimize: 1) The solution by introducing unknown values for the internal circuit lines or the inputs, and 2) secondary objectives in presence of unspecified values such that the property still holds. In the first case the unknown values generalize (or uniform) the solution, whereas in the second case some parts of the circuit are abstracted by introducing universal quantifications in an appropriate QBF and we want to optimize further objectives in presence of these unknowns.

In the following part of this section we briefly review classical algorithms discussing differences to the QBF with soft variable formulation. Finally, we introduce specific problem statements as well as first case studies for some application with our prototype solver. We define a metric representing the quality loss of heuristic methods compared to our approach as:

$$\text{loss(}\text{Method}\text{)} = 1 - \frac{X_{\text{Method}}}{X_{\text{QBF}}}$$

where $X_{\text{Method}}$ is the number of maximized unknowns computed by an approximate method, and $X_{\text{QBF}}$ is the value computed by QBF using soft variables.

4.1.1 Comparison to Previous Work

To model unknown values in a circuit, commonly 01X-encoding [JBM+00] applying SAT-based methods is used. Another popular SAT-based heuristic is lifting [RS04] which was introduced for minimal counterexamples (cf. §4.1.6). This method does not need any 01X-encoding, instead the optimization is done directly on the extracted model of a satisfied SAT instance.

However, 01X-logic is a pessimistic abstraction. In many applications, more precise solutions which then need a QBF formulation for the unknown values [SB01] are preferred. And although in an optimal case the lifting technique yields solutions as exact as a QBF formulation, it does not guarantee optimal ones due to its heuristic methodology. In QBF each unknown value is associated to a universally quantified variable. In contrast to SAT-based methods, the maximization with classical QBF-based methods is a harder problem, since the semantics of QBF binds each variable to its quantifier, i.e. the quantifier is statically declared and cannot be changed to the purpose of optimizing over unknown lines.

\(^2\) As shown in [Sin05] the encoding for $\text{LT}_{1\text{SEQ}}^n$ is superior to the naïve encoding for $n > 5$. 
QBF with soft variables overcomes this issue. For any line which is part of the unknown maximization a soft variable is introduced, and this allows to switch between existential (known) and universal (unknown) quantification (values). Shortly, the QBF with soft variables concept provides a compact representation for modelling unknowns, which is more precise than classical 01X-based approaches and guarantees optimal solution in general. More details of the encoding are presented in the following subsections.

4.1.2 Uniform Counterexamples

Exploiting partial designs is useful e.g. in the following cases: 1) Abstraction of complex parts of the system, 2) early step of the design process where not all parts are done, and 3) for diagnosis – in case a bug is present in the design and it is still present while parts are removed, there must be errors outside the black boxes. There exist different encoding methods for these hidden parts (usually called black boxes) [SB01], and the approach for using them in a BMC context [NSB07] is the so-called Black Box Bounded Model Checking (BBBMC).

One question in BBBMC is whether a property of the system is satisfied for each possible realization of the black boxes. A BBBMC can be written as the following QBF problem: \( ψ = \exists X_1 \forall Z_1 \exists X_2 \forall Z_2 \ldots \exists X_k \forall Z_k \land T_{0,1} \ldots \land T_{k-1,k} \land \neg P_k \), where \( X_i \) are the inputs of the transition relation and \( Z_i \) are the outputs of the black box at depth \( i \). If the property cannot be fulfilled, it is interesting to obtain a counterexample for the inputs \( X_i \). Using this encoding, only the assignments for the inputs \( X_1 \) are the same for each possible black box implementation – all other inputs may differ depending on the black box implementation. To overcome this issue one can define a uniform QBF prefix as follows: \( ψ_{uni} = \exists X_1 \ldots \exists X_k \forall Z_1 \ldots \forall Z_k \land T_{0,1} \ldots \land T_{k-1,k} \land \neg P_k \). Note that these two representations are not equivalent: \( ψ \) ensures that the primary inputs of each unrolling depth can “react” to the black box output, whereas in \( ψ_{uni} \) the inputs are independent from the black box implementation. Hence, \( ψ \) is more accurate than \( ψ_{uni} \), nonetheless \( ψ_{uni} \) returns a more general counterexample. In [NSB07] the authors specify partially uniform QBF, i.e., parts of a black box implementation. Hence, \( ψ \) is violated regardless of the excluded parts, the failure has to occur in the non-excluded part.

4.1.3 Diagnosis

As described in §4.1.2, black boxes can be used for diagnosis tasks: If a property of the system is violated regardless of the excluded parts, the failure has to occur in the non-excluded part. A challenge is to identify the black boxed part automatically, such that the remaining sub-circuit is as small as possible. In this case already the non-abstracted circuit is sufficient for the violation of
the property which would simplify the diagnosis task. To the best of our knowledge this problem statement is not tackled so far, even with pessimistic 01X-encoding.

Using QBF with Soft Variables Using the standard Tseitin-encoding [Tse68] to produce a propositional formula, every line \( l_i \) is associated to a Boolean variable. Hence we can define a soft variable for each line obtaining the following QBF with soft variables: 
\[
\psi(\exists (2,3) l_1, \ldots, \exists (2,3) l_k) = \exists \emptyset \forall \exists x_1 \ldots \exists x_n. \varphi \land \neg P,
\]
where \( x_i \) are inputs of the circuit, \( \varphi \) its encoding, and \( \neg P \) the violated property. By preferring universal quantification in the score function, the result of the optimization problem indicates a maximal number of lines, which can be excluded such that the bug is still present. Note, that this result is not accurate anymore in this scenario as discussed in §5. In order to reduce the complexity one may not consider every line but only specific ones, or consider different groups of lines separately and examine them incrementally.

4.1.4 Circuit Initialization

The problem of circuit initialization is a well known problem in the area of testing [PR00] and is closely related to state reachability problems known from BMC [RSSB14]. The problem asks whether a sequential circuit is initializable, i.e. all flip-flops can be set to a known value assuming the initial state is (completely) unknown. If this is not the case (which usually applies), one can seek for two related optimization questions: 1) starting from a unknown state what is the maximum number of initialized flip-flops, or 2) what is the smallest number of initially controlled flip-flops needed to initialize the complete circuit. For both objectives a minimal trace length is preferred.

Classical approaches either solve this problem heuristically (e.g. [PR00]) or are complete, but only with the pessimistic 01X-logic (cf. [RSSB14]). So far there is no approach that applies a complete method using the accurate QBF modelling for the unknown values.

Using QBF with Soft Variables Based on the method in [RSSB14], we can define a BMC problem based on QBF with soft variables. Let \( g_i^j \in G_i \) be a variable representing the value of a flip-flop at depth \( i \) and \( |G_i| \) the number of variables in \( G_i \). For the objective 1) we obtain: 
\[
\psi(\exists (1,2) g_1^1, \ldots, \exists (1,2) g_{|G_i|}^k) = \exists \emptyset \forall G_0 \exists G_1 \ldots \exists G_{k-1}. I_0 \land T_{0,1} \ldots \land T_{k-1,k} \land P_k.
\]
If the score functions recommend to quantify a soft variable existentially on the first level, the maximum number of initialized flip-flops in time step \( k \) is given by the number of these existentially quantified variables. The second objective 2) can be expressed as: 
\[
\psi(\exists (2,3) g_0^0, \ldots, \exists (2,3) g_{|G_i|}^0) = \exists G_k \forall \emptyset \exists G_1 \ldots \exists G_{k-1}. I_0 \land T_{0,1} \ldots \land T_{k-1,k} \land P_k.
\]
Here, \( G_k \) is quantified on the first level since we have to ensure that the value for the flip-flops in the \( k \)’th time step are fixed to a specific known value independently from the potentially unknown flip-flop values of the initial time step 0. By applying a score function favoring the quantification of soft variables at the universal level 2, the result for this optimization problem is equivalent to the minimum number of controlled flip-flops in the first time step.

Note, in contrast to the method in [RSSB14], this approach does not contain a proof whether more flip-flops can be initialized in further time steps, but it provides more accurate results as shown in the following case study.

Case Study We applied the encoding to the benchmark b06 from the ITC99 benchmark series which is commonly used in the EDA community and previously proven to be not completely initializable using 01X-encoding [RSSB14]. However, using QBF with soft variables, we have identified an initialization sequence of 5 time steps driving each flip-flop to a specified value.

\[3\] It may be meaningful to define groups of lines (e.g. buses or outputs of a specific module such as multiplier), which can be defined analogously.
4.1.5 Test Pattern Relaxation

In the area of testing [JG03] and in particular Automatic Test Pattern Generation (ATPG) the number of specified input bits in a test pattern is a major quality metric. Partially specified test patterns, so-called test cubes, serve as a foundation for further post-processing steps controlling secondary objectives, e.g. compaction, or test power reduction. As the quality of these methods depends on the number of unspecified values, test patterns with a greater amount of unspecified inputs are preferred.

Using QBF with Soft Variables Once a test pattern is found, we introduce soft variables for all inputs $x_1, \ldots, x_n$ in order to preferably treat them as unknowns. A resulting QBF may look like: $\psi(\bigwedge_{i=1}^{2,3} x_1, \ldots, \bigwedge_{i=1}^{2,3} x_n) = \exists \emptyset \forall \emptyset \exists Y. \varphi$, where $\varphi$ is the encoding of the circuit using variables $V$ and the property justifying the test pattern. If the score function is built favoring the universal quantification level, one obtains the maximum number of lines that can be excluded from the test pattern such that the fault is still visible.

Case Study In [SRP+13] an approach using an accurate QBF modeling was already presented, however without using the soft variable concept. For the common ISCAS89 and ITC99 benchmark circuits we generated minimal test cubes for 100 test pattern detecting small delay faults [JG03]. We used QBF with soft variables to document the quality loss of common heuristics (01X, lifting, and simulation) in the number of unspecified inputs compared to the optimal solution. Fig. 1 shows the quality loss for different benchmarks. As it can be seen, depending on the internal structure of the individual benchmark the heuristics can identify up to 23% less unspecified inputs for benchmark circuit b09.

4.1.6 Minimal Counterexamples

SAT-based Bounded Model Checking (BMC) [BCC+03] is a formal verification technique for designs modeled as finite state machines (FSM). The transition relation of the FSM is unrolled step by step and a property is verified at the current depth. The depth $k$ is incremented until either the property is violated for the given depth, or a user-defined bound for the depth is reached. A SAT formula for the BMC problem can be written as: $\varphi = I_0 \wedge T_{0,1} \wedge \ldots \wedge T_{k-1,k} \wedge \neg P_k$, where $I_0$ is the initial state of the system, $T_{i,i+1}$ is the transition relation for step $i$ and $P_k$ is the property at step $k$. Note that $\varphi$ is true iff the property is violated.

In case of a violated property the model of $\varphi$ represents a full trace of assignments to the primary inputs of the transition relation (i.e. a counterexample) of the current unrolling depth. This trace can be used for diagnosis or abstraction refinements. In order to improve the diagnosis routines
the number of primary inputs, which are needed to justify the violation of the property, can be delimited.

Note that the test pattern relaxation problem as described in §4.1.5 can be seen as a special case of the minimal counterexample problem. Instead of asking whether a property is violated, we ask whether a property (detection of a fault) holds minimizing the number of specified inputs (test cube).

Using QBF with Soft Variables Consider a BMC problem $\varphi$ as stated above with a counterexample at depth $k$. We denote $X = x_1, \ldots , x_n$ as the variables representing all inputs of the transition relations and $Y = y_1, \ldots , y_m$ all other variables in $\varphi$. In order to minimize the number of specified inputs (or maximize the unknown inputs), we state the following QBF with soft variables instance: $\psi(\mathcal{F}(x_1, \ldots , x_n)) = \exists \forall \exists Y. \varphi$, i.e. a QBF containing only existentially quantified variables $Y$ on level 3 and the soft variables $X$. If we define the score function $\sigma$ such that the universally quantified position gets a larger score than the existential position, the result of $\Omega(\psi)$ returns the maximum number of inputs which can be excluded from the counterexample such the property is still unsatisfied. Hence, we obtain a counterexample with a minimal number of specified inputs.

Case Study We considered 94 unsatisfiable BMC benchmarks from 11 classes of the Hardware Model Checking Competitions (2010, 2011 and 2012). We selected the Bounded Model Checker cip [KLSB11] which is able to extract counterexamples from buggy benchmark designs. For some groups with smaller number of inputs we computed the minimal counterexamples for these benchmarks, and compared them with the results obtained by applying a lifting algorithm.

The results are given in Table 1. In the first three columns the family name, the number of overall / non-solved QBF instances within a timeout of 30 CPU minutes, and the average number of inputs of the original counterexample are given. In order to compare results, the following columns contains only results for the instances where the QBF approach was able to provide a solution. The following two columns show the average run time in seconds and the average number of inputs we can exclude from the counterexample using QBF with soft variables. The next two columns show the same for the lifting approach, and the last column indicates the average quality loss of the lifting approach.

On average the loss of accuracy with lifting is about 6% compared to the exact QBF with soft variables. However, for some benchmarks the loss is over 70% (instance brpptimeonenegnv). The run times for these smaller examples are very reasonable with just 4 non-solved benchmarks, but our approach still has scaling issues, especially if the number of lines to maximize is large.

Table 1: Results for HWMCC benchmarks

<table>
<thead>
<tr>
<th>Family</th>
<th>Inst.</th>
<th>Inputs</th>
<th>QBF</th>
<th>Lifting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Time</td>
<td>X</td>
</tr>
<tr>
<td>beemi*</td>
<td>14 / 0</td>
<td>489.71</td>
<td>0.60</td>
<td>17.50</td>
</tr>
<tr>
<td>beemsch*</td>
<td>16 / 0</td>
<td>320.38</td>
<td>1.03</td>
<td>13.66</td>
</tr>
<tr>
<td>bj*</td>
<td>12 / 1</td>
<td>16.27</td>
<td>0.07</td>
<td>14.36</td>
</tr>
<tr>
<td>brpp*</td>
<td>20 / 0</td>
<td>455.35</td>
<td>6.95</td>
<td>301.20</td>
</tr>
<tr>
<td>counter*</td>
<td>2 / 0</td>
<td>82.00</td>
<td>0.01</td>
<td>21.00</td>
</tr>
<tr>
<td>dme*</td>
<td>20 / 0</td>
<td>477.25</td>
<td>91.64</td>
<td>282.50</td>
</tr>
<tr>
<td>ken*</td>
<td>2 / 0</td>
<td>268.68</td>
<td>3.48</td>
<td>161.10</td>
</tr>
<tr>
<td>pci*</td>
<td>5 / 0</td>
<td>477.25</td>
<td>91.64</td>
<td>282.50</td>
</tr>
<tr>
<td>srg*</td>
<td>3 / 0</td>
<td>81.00</td>
<td>0.29</td>
<td>25.67</td>
</tr>
<tr>
<td>texas*</td>
<td>9 / 1</td>
<td>117.63</td>
<td>6.29</td>
<td>102.88</td>
</tr>
<tr>
<td>vis*</td>
<td>6 / 2</td>
<td>78.00</td>
<td>49.52</td>
<td>45.00</td>
</tr>
<tr>
<td>total</td>
<td>94 / 4</td>
<td>288.32</td>
<td>25.74</td>
<td>102.88</td>
</tr>
</tbody>
</table>
4.2 Further applications

For the task of finding optimal dependency schemes [Sam08] one can use the optimization problem for QBF with soft variables. Therefore, we declare a single existential variable of the original problem as soft variable allowing to be quantified on all existential prefix position. In the score function we prefer the position with the least dependencies on universally quantified variables as possible.

Another application area where QBF with soft variables may be a useful mechanism is planning and decision making, which are common problems evolving from the game theory domain. In a multi-agent environment, where more than one player is present, such problems ask to determine the best action. Hence, the goal is to find the action model with the highest score. In [YWJ07] a weighted MaxSAT solver is used to determine these models. But this approach is limited if we consider uncertainty of other players’ action, the environment or possibly unknown actions performed earlier. Using MaxQBF one’s own action score could be maximized with accurately modeled behaviour of the environment. Moreover, by using QBF with soft variables it can be determined whether the order of the actions can be altered or even if an action can be left out in order to decrease the overall costs.

A related problem are two-players games [AMN05], which can be naturally formulated with QBF logic by giving the existential quantifier (resp. the universal quantifier) the role of the system player (resp. the environment player). As in the planning problem one could try to make own moves independent from the opponent ones using QBF with soft variables for the existential variables. This would lead to more generalized winning strategies. This problem is related to the uniform counterexamples (cf. §4.1.2) and can be tackled in a similar manner.

5 Conclusions

We presented first results on the novel concept of soft variables for quantified Boolean formulae. The related optimization problem allows to optimize the prefix according to a user-given preference. Furthermore, we introduced a MaxQBF solver and a sound and complete algorithm to solve QBF with soft variables using MaxQBF.

Recent improvements in classical algorithms for SAT and QBF lead to new research interests in answering beyond yes/no questions. Such optimization problems using quantified formulas are often hard to encode due to the static prefix or rather predefined dependencies of the variables. The concept of soft variables overcomes this issues and opens a wide field of such applications requiring an accurate model. We demonstrated the applicability of our formalism by introducing several applications in the area of formal verification, debugging, testing and artificial intelligence.

However, also QBF is limited if the abstracted part modeled by universally quantified variables either show sequential behavior or there are multiple black box parts with overlapping cones of the black box inputs and outputs. In these cases QBF (hence also QBF with soft variables) is not accurate anymore [GRS+13]. To overcome the latter issue one can define explicit dependencies of the variables rather than defining a linear prefix. This extension is known as Dependency QBF (DQBF) using so-called Henkin quantifiers, where variables are quantified with a explicitly given variable set of dependencies [Hen61]. The definition of a DQBF with soft variables is straightforward: instead of possible prefix positions a soft variable is allowed to be quantified with different dependency sets4. Research interest starts on focusing DQBF algorithms and applications [GRS+13], but the scalability of these algorithms is still not reasonable for tackling even more complex optimization problems (deciding pure DQBF is already NEXPTIME-complete [PRA01]). Therefore we do not consider soft variables for DQBF in this paper, but it may become relevant if appropriate algorithms for solving DQBF are present.

As future work we want to investigate the new application areas covered by our soft variable.

4 Actually, the prefix of a QBF is just a convenient notation for a linear dependency relation between the variables.
mechanism as well as dedicated solving mechanisms beyond the techniques used for MaxSAT/QBF-based algorithms to increase the scalability.


Exact and Approximate Abstraction for Classes of Stochastic Hybrid Systems

Jeremy Sproston∗

Dipartimento di Informatica, University of Turin, Italy

Abstract: A stochastic hybrid system contains a collection of interacting discrete and continuous components, subject to random behaviour. The formal verification of a stochastic hybrid system often comprises a method for the generation of a finite-state probabilistic system which either represents exactly the behaviour of the stochastic hybrid system, or which approximates conservatively its behaviour. We extend such abstraction-based formal verification of stochastic hybrid systems in two ways. Firstly, we generalise previous results by showing how bisimulation-based abstractions of non-probabilistic hybrid automata can be lifted to the setting of probabilistic hybrid automata, a subclass of stochastic hybrid systems in which probabilistic choices can be made with respect to finite, discrete alternatives only. Secondly, we consider the problem of obtaining approximate abstractions for discrete-time stochastic systems in which there are continuous probabilistic choices with regard to the slopes of certain system variables. We restrict our attention to the subclass of such systems in which the approximate abstraction of such a system, obtained using the previously developed techniques of Fränzle et al., results in a probabilistic rectangular hybrid automaton, from which in turn a finite-state probabilistic system can be obtained. We illustrate this technique with an example, using the probabilistic model checking tool PRISM.

Keywords: Probabilistic verification, stochastic hybrid systems, bisimulation

1 Introduction

Our increasing reliance on complex embedded and cyber-physical systems calls for the development of methods for the verification of hybrid systems, which are systems in which behaviour is described as an interplay between discrete and continuous components. In this paper, we consider a well-known formalism for the description of hybrid systems, namely hybrid automata [ACH+95], which comprise a finite-state graph, to represent the discrete part of the system, and a finite set of real-valued variables, to represent the continuous part of the system. Interaction between the discrete and continuous parts of the system is represented by labelling the graph with conditions on the variables and their first derivatives. Hence we can express that, as time passes and the system resides in a particular node of the graph, the rate of change of the variables is described in a certain way; we can also describe how the system moves from node to node when the

∗ Supported by the project AMALFI (University of Turin/Compagnia di San Paolo), the MIUR-PRIN project CINA, and the EU ARTEMIS Joint Undertaking under grant agreement no. 332933 (HoliDes).
value of the variables satisfies certain conditions. Hybrid automata can be subject to automatic verification procedures, which have been implemented in a number of tools [HHW97, Fre08].

In many contexts, a representation of a hybrid system without information regarding the likelihood of its various behaviours is insufficient. For example, a verification method for a particular hybrid system may show the existence of a behaviour corresponding to an error or to a decrease in performance, but without representing the fact that the behaviour is unlikely to occur. This observation has led to a number of formalisms for stochastic hybrid systems, in which the likelihood of behaviours is represented, for example [HLS00, Spr00, Buj04, APLS08, FHH+11, Hah13]. In particular, probabilistic hybrid automata [Spr00] extend hybrid automata by associating probabilities with edges of the graph. Stochastic hybrid automata [FHH+11, Hah13] extend probabilistic hybrid automata by allowing the values of the continuous variables to be reset according to continuous probability distributions.

An example of a probabilistic hybrid automaton modelling a faulty thermostat is shown in Figure 1 (the model is a modification of an example of [Spr11]). We use a number of the usual conventions for illustrating hybrid automata, and refer to the nodes of the graph as locations. The ambient temperature is represented by the variable \( x \), and variable \( y \) is a timer. When the heater is on (location \( \text{l_{on}} \) or location \( \text{l_{malf}} \)), the temperature increases at a rate between 1 and 6; when the heater is off (location \( \text{l_{off}} \)), the temperature changes at a rate between -4 and -1. The locations \( \text{l_{on}} \) and \( \text{l_{off}} \) correspond to non-faulty behaviour, whereas the location \( \text{l_{malf}} \) corresponds to the heater being on in the presence of a fault in the temperature sensor that means that the measurement of the temperature is temporarily unavailable. The system passes from \( \text{l_{on}} \) to \( \text{l_{off}} \), with probability \( \frac{9}{10} \), or to \( \text{l_{malf}} \), with probability \( \frac{1}{10} \), when the temperature is between 10 and 15. The sensor fault means that the temperature can increase to a higher level in \( \text{l_{malf}} \) than in \( \text{l_{on}} \). After a malfunction, either the system is deactivated if the temperature reaches an excessive level (location \( \text{l_{deact}} \)), or the system times-out exactly 20 time units after the location \( \text{l_{malf}} \) was entered, in which case the heater is switched off. All edges of the probabilistic hybrid automaton correspond to reaching a certain location with probability 1, apart from the probabilistically branching edge from \( \text{l_{off}} \).

Two main approaches for the verification of probabilistic and stochastic hybrid automata have been presented. On the one hand, exact methods generally consider the construction of a finite-state probabilistic system (more precisely, a probabilistic automaton or a Markov decision pro-

![Figure 1: A probabilistic hybrid automaton modelling a faulty thermostat](image-url)
cess [Put94]) that represents faithfully the behaviour of the original system. This approach has been adopted for restricted subclasses of probabilistic hybrid automata in [Spr00, Spr11] using probabilistic bisimulation [LS91, SL95] to obtain a finite number of equivalence classes from which a finite-state system can be constructed. On the other hand approximate methods consider generally the construction of a finite-state probabilistic system that overapproximates the behaviour of the original system. This approach has been adopted for probabilistic hybrid automata in [Spr01, ZSR+12], and for stochastic hybrid automata in [FHH+11, Hah13].

The contributions of this paper are twofold. First, in Section 2, we generalise the results of [Spr00, Spr11] concerning exact verification methods. Any probabilistic hybrid automaton can be translated into a non-probabilistic hybrid automaton in which information concerning probability distributions is encoded in labels on edges of the graph. Consider a probabilistic hybrid automaton $H$: we show that if the non-probabilistic hybrid automaton counterpart of $H$ has a finite bisimulation equivalence quotient, then $H$ has a finite probabilistic bisimulation equivalence quotient. This result unifies and generalises previous results, and has the consequence that we can identify classes of probabilistic hybrid automata with a finite bisimulation equivalence quotient on the basis of whether members of the corresponding class of hybrid automata have finite bisimulation quotients. For example, we can conclude that the class of probabilistic STORMED hybrid automata, which are probabilistic hybrid automata to which the restrictions of STORMED hybrid automata [VPVD08] apply to the non-probabilistic characteristics of the system, have a finite probabilistic bisimulation equivalence quotient, because STORMED hybrid automata have finite bisimulation equivalence quotients. Any future results on the identification of classes of hybrid automata with finite bisimulation quotients will also imply that the corresponding class of probabilistic hybrid automata has finite probabilistic bisimulation quotients.

The second contribution, in Section 3, concerns approximate verification methods. In stochastic hybrid automata, the rate of change of continuous variables as time passes is not (directly) chosen probabilistically. However, the rate of change of a continuous variable $x$ may be equal to the value of another continuous variable $y$ (that is, $\dot{x} = y$), and $y$ could be subject to a reset by a continuous probability distribution: hence, the rate of change of $x$ is indirectly dependent on a continuous probabilistic choice. We consider this approach in the context of stochastic hybrid automata and the approximation method of [FHH+11, Hah13], which uses probabilistic hybrid automata as approximate models of stochastic hybrid automata. We show that there exists a class of probabilistic hybrid automata that features dependence of variable’s rate of change on the value of other variables (where these variables are constant as time passes) that is equivalent to probabilistic rectangular automata, a subclass of probabilistic hybrid automata which has a finite probabilistic bisimulation equivalence quotient when considering a discrete-time semantics [Spr11]. We apply this approach to the case of the faulty thermostat of Figure 1: for example, when the heater is on, rather than increase nondeterministically with a rate in $[1, 6]$, we consider that the rate of increase of the temperature is chosen from the normal distribution with mean $3.5$ and standard deviation $1$, truncated to the interval $[1, 6]$. This continuous distribution is approximated according to the approach of [FHH+11, Hah13], which then results in an intermediate probabilistic hybrid automaton that we can show is equivalent to a probabilistic rectangular automaton, which (assuming a discrete-time semantics) can then be transformed into a finite-state probabilistic system, which is then analysed using the probabilistic model checking tool PRISM [KNP11]. We show that there is a significant difference in the results obtained.
from the probabilistic rectangular automaton in Figure 1 and from the probabilistic rectangular automaton obtained as an approximation of an automaton involving continuous distributions, as described above.

2 Exact Abstraction of Probabilistic Hybrid Automata

2.1 Preliminaries

We use \( \mathbb{R} \) to denote the set of real numbers, \( \mathbb{R}_{\geq 0} \) to denote the set of non-negative real numbers, \( \mathbb{Z} \) to denote the set of integers, \( \mathbb{N} \) to denote the set of natural numbers and \( \mathbb{Q} \) to denote the set of rational numbers. Given a set \( Q \) and a function \( \mu : Q \to \mathbb{R}_{\geq 0} \), we define \( \text{support}(\mu) = \{ q \in Q \mid \mu(q) > 0 \} \). A (discrete) probability distribution over a countable set \( Q \) is a function \( \mu : Q \to [0, 1] \cap \mathbb{Q} \) such that \( \sum_{q \in Q} \mu(q) = 1 \). Let \( \text{Dist}(Q) \) be the set of distributions over \( Q \). If \( Q \) is an uncountable set, we define \( \text{Dist}(\mathbb{Q}) \) to be the set of functions \( \mu : Q \to [0, 1] \), such that \( \mu(q_1) = \lambda_1, \ldots, \mu(q_n) = \lambda_n \). Given a set \( Q' \subseteq Q \), we let \( \mu(Q') = \sum_{q \in Q'} \mu(q) \).

A probabilistic automaton (PA) \( \mathcal{P} = (S, \text{Act}, \xrightarrow{\cdot}) \) comprises the following components: a set of states \( S \); a set of actions \( \text{Act} \); and a probabilistic, nondeterministic transition relation \( \xrightarrow{\cdot} \subseteq S \times \text{Act} \times \text{Dist}(S) \). Each of \( S, \text{Act} \) and \( \xrightarrow{\cdot} \) may be uncountable.

An infinite path of a PA is an infinite sequence \( r = s_0a_0s_1a_1\cdots \) such that \( (s_i, a_i, \mu_i) \in \xrightarrow{a} \) and \( \mu_i(s_{i+1}) > 0 \) for each \( i \in \mathbb{N} \). Similarly, a finite path of a PA is a finite sequence \( r = s_0a_0s_1a_1\cdots a_{n-1}s_n \) such that \( (s_i, a_i, \mu_i) \in \xrightarrow{a} \) and \( \mu_i(s_{i+1}) > 0 \) for each \( i < n \). We use \( \text{Path}_{\text{fin}}(s) \) to denote the set of infinite paths of \( \mathcal{P} \), and \( \text{Path}_{\text{fin}}(s) \) to denote the set of finite paths of \( \mathcal{P} \).

When clear from the context we omit the superscript \( \mathcal{P} \). If \( r \) is a finite path, we denote by \( \text{last}(r) \) the last state of \( r \). Let \( \text{Path}_{\text{fin}}(s) \) and \( \text{Path}_{\text{fin}}(s) \) refer to the sets of infinite and finite paths of \( \mathcal{P} \), respectively, commencing in state \( s \in S \).

Let \( \mathcal{P} = (S, \text{Act}, \xrightarrow{\cdot}) \) be a PA. A strategy of \( \mathcal{P} \) is a function \( \sigma \) mapping every finite path \( r \in \text{Path}_{\text{fin}}(r) \) to a transition \( (\text{last}(r), a, \mu) \in \xrightarrow{a} \). We write as \( \Sigma_{\mathcal{P}} \) the set of strategies of \( \mathcal{P} \). For any strategy \( \sigma \), let \( \text{Path}_{\text{fin}}(s) \) and \( \text{Path}_{\text{fin}}(s) \) denote the sets of infinite and finite paths, respectively, resulting from the choices of \( \sigma \). For a state \( s \in S \), let \( \text{Path}_{\text{fin}}(s) = \text{Path}_{\text{fin}}(s) \cap \text{Path}_{\text{fin}}(s) \) and \( \text{Path}_{\text{fin}}(s) = \text{Path}_{\text{fin}}(s) \cap \text{Path}_{\text{fin}}(s) \). Given a strategy \( \sigma \in \Sigma_{\mathcal{P}} \) and a state \( s \in S \), we define the probability measure \( \text{Prob}^\sigma \) over \( \text{Path}_{\text{fin}}(s) \) in the standard way [KSK76]. Note that we generally consider pure strategies (that is, strategies that do not make randomized choices), the choices of which may depend on the history of the system. The cases in which randomized strategies (which map from finite paths \( r \to \text{Dist}(\xrightarrow{a}) \)) are considered will be signalled in the text.

Given state set \( S' \subseteq S \), we write \( \text{Reach}(S') \) for the set of paths that reach \( S' \), that is \( \text{Reach}(S') = \{ r \mid r \in \text{Path}_{\text{fin}} \land \exists i \in \mathbb{N} . r(i) \in S' \} \) where \( r(i) \) is the \( (i+1) \)-th state along \( r \). Then, given state \( s \in S \), we write

\[
\text{MaxReach}^\sigma_s(S') = \sup_{\sigma \in \Sigma_{\mathcal{P}}} \text{Prob}^\sigma_s(\text{Reach}(S')) , \quad \text{MinReach}^\sigma_s(S') = \inf_{\sigma \in \Sigma_{\mathcal{P}}} \text{Prob}^\sigma_s(\text{Reach}(S'))
\]

to be the maximum and minimum probability of reaching a state in \( S' \) from \( s \), respectively.
Let \( P = (S, \text{Act}, \Rightarrow) \) be a PA. For distributions \( \mu, \nu \in \text{Dist}(S) \) and for an equivalence relation \( \equiv \subseteq S \times S \), we denote by \( \mu \equiv \nu \) the condition that, for each equivalence class \( C \) of \( \equiv \), the equality \( \mu[C] = \nu[C] \) holds. Let \( \equiv \subseteq S \times S \) be an equivalence relation on \( S \). A probabilistic bisimulation respecting \( \equiv \) on \( P \) [LS91, SL95] is an equivalence relation \( \simeq \subseteq S \times S \) such that \( s \simeq t \) implies that (1) \( s \equiv t \), and (2) if \( (s, a, \mu) \in \Rightarrow \), then there exists \( (t, a, \nu) \in \Rightarrow \) such that \( \mu \simeq \nu \). States \( s \) and \( t \) are called probabilistically bisimilar with respect to \( \equiv \) in \( P \) if there exists a probabilistic bisimulation \( \simeq \) respecting \( \equiv \) such that \( s \simeq t \). Probabilistic bisimilar states exhibit the same maximum and minimum probabilities of reachability (or, more generally, \( \omega \)-regular properties [BK08]), and satisfy the same properties of the probabilistic temporal logic \( \mathsf{PCTL}^* \) [SL95].

A PA \( P = (S, \text{Act}, \Rightarrow) \) for which all transitions \( (s, a, \mu) \in \Rightarrow \) are such that \( \mu \) is of the form \( \{s' \mapsto 1\} \) for some \( s' \in S \) is called a nondeterministic automaton. In the case of nondeterministic automata, we often write \( (s, a, s') \) to denote the transition \( (s, a, \{s' \mapsto 1\}) \). If \( P \) is a nondeterministic automaton, then we can simplify the definition of probabilistic bisimulation, which, in this context, is called simply bisimulation, in the following way: a bisimulation respecting \( \equiv \) on a nondeterministic automaton is an equivalence relation \( \approx \subseteq S \times S \) such that \( s \approx t \) implies that (1') \( s \equiv t \), and (2') if \( (s, a, s') \in \Rightarrow \), then there exists \( (t, a, t') \in \Rightarrow \) such that \( s' \approx t' \). States \( s \) and \( t \) are called bisimilar with respect to \( \equiv \) in the nondeterministic automaton \( P \) if there exists a bisimulation \( \approx \) respecting \( \equiv \) such that \( s \approx t \).

### 2.2 Probabilistic Hybrid Automata

Let \( \mathcal{X} \) be a finite set of real-valued variables. A valuation \( v : \mathcal{X} \to \mathbb{R} \) is a function that assigns a value to each variable of \( \mathcal{X} \). Let \( \mathcal{V}(\mathcal{X}) \) be the set of valuations of \( \mathcal{X} \). When the set \( \mathcal{X} \) is clear from the context, we generally write \( \mathcal{V} \).

A probabilistic hybrid automaton (PHA) \( \mathcal{H} = (L, \mathcal{X}, \text{Events}, \text{post}, \text{prob}) \) consists of the following components: a finite set \( L \) of locations; a finite set \( \mathcal{X} \) of variables; a finite set \( \text{Events} \) of events; a post operator \( \text{post} : L \times \mathcal{V} \times \mathbb{R}_{\geq 0} \to 2^\mathcal{V} \); a finite set \( \text{prob} \subseteq L \times 2^\mathcal{V} \times \text{Events} \times \text{Dist}(\text{Upd}(\mathcal{X}) \times L) \) of probabilistic edges, where \( \text{Upd}(\mathcal{X}) \) is the set of functions \( \vartheta : \mathcal{V} \to 2^\mathcal{V} \). A probabilistic edge \( (l, g, e, p) \in \text{prob} \) comprises (1) a source location \( l \), (2) a set \( g \) of valuations, called a guard, (3) an event \( e \), and (4) a probability distribution \( p \) that assigns probability to pairs of the form \( (\vartheta, l') \), where \( \vartheta \in \text{Upd}(\mathcal{X}) \) is a function describing the manner in which variables are updated and \( l' \in L \) is a target location.

The behaviour of a PHA takes a similar form to that of a classical, non-probabilistic hybrid automaton [ACH+95]. If the PHA is currently in location \( l \), as time passes, the value of the variables in \( \mathcal{X} \) change according to the post operator \( \text{post} \): more precisely, if the current valuation is \( v \) and \( \delta \in \mathbb{R}_{\geq 0} \) time units elapse, the subsequent valuation belongs to the set \( \text{post}(l, v, \delta) \). If the current valuation of the variables belongs to the guard \( g \) of a probabilistic edge \( (l, g, e, p) \), then the probabilistic edge can be taken. This involves a probabilistic choice according to the distribution \( p \): if the pair \( (\vartheta, l') \) is chosen, then the PHA goes to location \( l' \), updating the variables according to the function \( \vartheta \). More precisely, if the current valuation of the variables is \( v \) and the pair \( (\vartheta, l') \) is chosen, then the state after taking the probabilistic edge will be \( (l', v') \) for some \( v' \) that is chosen nondeterministically from the set \( \vartheta(v) \). To summarise, the following choices made by the PHA are nondeterministic: the amount of time to let advance in the current location \( l \); the valuation used to describe the values of the variables after time has elapsed, according to
post; the probabilistic edge taken (provided that the guard of the probabilistic edge is satisfied by the current variable valuation); and, finally, the values to which the variables are updated when a probabilistic edge is taken. Instead, the only probabilistic choice featured in the model concerns the choice of pair $(\vartheta, l')$ once a probabilistic edge has been chosen.

We make a number of standard assumptions on the components of a PHA [Spr01, Hah13].

- (Assumptions on post.) For all locations $l \in L$ and valuations $v \in \mathcal{V}$, we require the following: (1) $\text{post}(l, v, 0) = \{v\}$; (2) for all $\delta, \delta' \in \mathbb{R}_{\geq 0}$ such that $\delta \geq \delta'$, we have $\text{post}(l, v, \delta) = \bigcup_{e \in \text{post}(l, v, \delta)} \text{post}(l, v, \delta)$; (3) there exists $\delta \in \mathbb{R}_{\geq 0}$ such that $\delta > 0$ and $\text{post}(l, v, \delta) = \emptyset$.

- (Probabilistic edges can be taken when no more time can elapse.) If $l \in L$ and $v \in \mathcal{V}$ are such that, for all $\delta \in \mathbb{R}_{\geq 0}$ such that $\delta > 0$, we have $\text{post}(l, v, \delta) = \emptyset$, then there must exist some probabilistic edge $(l, g, e, p) \in \text{prob}$ such that $v \in g$.

- (Non-empty updates.) For all probabilistic edges $(l, g, e, p) \in \text{prob}$ and each $(\vartheta, l') \in \text{support}(p)$, we have $\vartheta(v) \neq \emptyset$ for all $v \in g$.

We now introduce formally the semantics of PHA in terms of PA. The (dense-time) semantics of the PHA $\mathcal{H} = (L, \mathcal{X}, \text{Events}, \text{post}, \text{prob})$ is the PA $[[\mathcal{H}]] = (S, \text{Act}, \Rightarrow)$ defined in the following way. The set of states of $[[\mathcal{H}]]$ is defined as $S = L \times \mathcal{V}$. The set of actions of $[[\mathcal{H}]]$ is $\text{Act} = \mathbb{R}_{\geq 0} \cup \text{Events}$. To define the transition relation $\Rightarrow$, we first define a transition relation for each time duration and event.

- (Time elapse.) Let $\delta \in \mathbb{R}_{\geq 0}$. Then $\delta \subseteq S \times \mathbb{R}_{\geq 0} \times \text{Dist}(S)$ is the largest set such that $\{(l, v), (l', v') \mapsto 1\} \in \delta$ implies that (1) $l = l'$ and (2) $v' \in \text{post}(l, v, \delta)$.

- (Jumps.) Let $e \in \text{Events}$. Consider a distribution $p \in \text{Dist}(\text{Upd}(\mathcal{X}) \times L)$, where $\text{support}(p) = \{(\vartheta_1, l_1), \ldots, (\vartheta_n, l_n)\}$. Then, for valuation $v$, we write $\text{Bundle}(v, p) \subseteq \mathcal{V}^n$ to denote the largest set of vectors of valuations such that $[v_1, \ldots, v_n] \in \text{Bundle}(v, p)$ implies $v_i \in \vartheta_i(v)$ for each $1 \leq i \leq n$. Then $\delta \subseteq S \times \text{Dist}(S)$ is the largest set of transitions such that $((l, v), e, \mu) \in \delta$ implies that there exists a probabilistic edge $(l, g, e, p) \in \text{prob}$ such that (a) $v \in g$ and (b) there exists $[v_1, \ldots, v_n] \in \text{Bundle}(v, p)$ such that, for each $(l', v') \in S$:

$$
\mu(l', v') = \sum_{1 \leq i \leq n \text{ s.t. } v' = v_i} p(\vartheta_i, l') .
$$

Then we define $\Rightarrow$ as the transition relation $(\bigcup_{\delta \in \mathbb{R}_{\geq 0}} \delta \cup (\bigcup_{e \in \text{Events}} \varepsilon_e))$. We note that the summation in the definition of jump transitions is necessary for the case in which the same state can be obtained by more than one element $(\vartheta, l)$ in the support set of the distribution of a probabilistic edge.

In the next section, we generally consider a time-abstract semantics of $\mathcal{H}$, in which actions corresponding to durations of time-elapse transitions are replaced by a single action $\tau$ (where we assume that $\tau \notin \text{Events}$). Formally, the time-abstract semantics of $\mathcal{H}$ is the PA $[[\mathcal{H}]]^{\tau} = (S, \text{Act}, \Rightarrow)$, where the set $S$ of states is the same as for the dense-time semantics of $\mathcal{H}$, the set of actions is defined as $\text{Act} = \{\tau\} \cup \text{Events}$, and the transition relation $\Rightarrow$ is defined as $\Rightarrow \cup (\bigcup_{e \in \text{Events}} \varepsilon_e)$, where $\varepsilon_e = \{(s, \tau, \mu) | \exists \delta \in \mathbb{R}_{\geq 0} (s, \delta, \mu) \in \delta\}$. 

114 Exact and Approximate Abstraction for Classes of Stochastic Hybrid Systems

Proc. AVoCS 2014 6 / 15
2.3 From Non-Probabilistic to Probabilistic Bisimulation

A hybrid automaton (HA) is a PHA \((L, \mathcal{X}, \text{Events, post, prob})\) for which all probabilistic edges \((l, g, e, p) \in \text{prob}\) correspond to a trivial probabilistic choice over a single element; more precisely, each \((l, g, e, p) \in \text{prob}\) is such that \(p\) is of the form \(\{(\varnothing, l') \mapsto 1\}\) for some \(\varnothing \in \text{Upd}(\mathcal{X})\) and \(l' \in L\). We refer to probabilistic edges of the above form as edges.

Consider an arbitrary PHA \(\mathcal{H} = (L, \mathcal{X}, \text{Events, post, prob})\). We let \(\text{ind}(\text{prob})\) be the smallest set of edges such that, if \((l, g, e, p) \in \text{prob}\) then, for each \((\varnothing, l') \in \text{support}(p)\), there exists the probabilistic edge \((l, g, (e, p, \varnothing), \{(\varnothing, l') \mapsto 1\}) \in \text{ind}(\text{prob})\). Let \(\text{ind}(\mathcal{H}) = (L, \mathcal{X}, \text{Events} \times \text{Dist}(\text{Upd}(\mathcal{X}) \times L) \times \text{Upd}(\mathcal{X}), \text{post, ind}(\text{prob}))\) be the HA induced by the PHA \(\mathcal{H}\). Note that the location, variable and post sets are the same in \(\mathcal{H}\) and \(\text{ind}(\mathcal{H})\). The events of \(\text{ind}(\mathcal{H})\) feature tuples comprising an event of \(\mathcal{H}\), the distribution over updates and locations, and an update. The set of probabilistic edges of \(\text{ind}(\mathcal{H})\) is \(\text{ind}(\text{prob})\), i.e., a set of edges in which the events encode information derived from probabilistic edges in \(\text{prob}\).

We present our first result, namely that bisimilar states of the time-abstract semantics of \(\text{ind}(\mathcal{H})\) are probabilistically bisimilar in the time-abstract semantics of \(\mathcal{H}\). In this paper, we regard locations as being observable from the point of view of properties (such as PCTL\(^*\) formulae or \(\omega\)-regular objectives), and hence we consider (probabilistic) bisimulation with respect to an equivalence relation that considers as equivalent states with the same location. We define the equivalence relation \(\equiv_{\text{loc}} \subseteq S \times S\) in the following way: \((l, v) \equiv_{\text{loc}} (m, w)\) if and only if \(l = m\), for all states \((l, v), (m, w) \in S\).

**Proposition 1** Let \(\approx \) be a bisimulation with respect to \(\equiv_{\text{loc}}\) on \(\llbracket \text{ind}(\mathcal{H}) \rrbracket^\text{ta}\). Then \(\approx\) is a probabilistic bisimulation with respect to \(\equiv_{\text{loc}}\) on \(\llbracket \mathcal{H} \rrbracket^\text{ta}\).

**Proof.** We use \(\llbracket \mathcal{H} \rrbracket^\text{ta} = (S, \text{Act}(\mathcal{H})^u, \Rightarrow, \llbracket \text{ind}(\mathcal{H}) \rrbracket^u)\) and \(\llbracket \text{ind}(\mathcal{H}) \rrbracket^\text{ta} = (S, \text{Act}[\text{ind}(\mathcal{H})]^u, \Rightarrow, \llbracket \text{ind}(\mathcal{H}) \rrbracket^u)\) be the time-abstract semantics of \(\mathcal{H}\) and \(\text{ind}(\mathcal{H})\), respectively (note that \(\llbracket \mathcal{H} \rrbracket^\text{ta}\) and \(\llbracket \text{ind}(\mathcal{H}) \rrbracket^\text{ta}\) have the same set of states, \(S = L \times \mathcal{V}\)). Let \(\approx\) be a bisimulation respecting \(\equiv_{\text{loc}}\) on \(\llbracket \text{ind}(\mathcal{H}) \rrbracket^u\).

Consider states \((l, v), (m, w) \in S\), and assume that \((l, v) \approx (m, w)\). This implies that the two conditions in the definition of bisimulation are satisfied: more precisely, we have \((1') l = m\), and \((2')\) if \(((l, v), a, (l', v')) \in \Rightarrow_{\llbracket \text{ind}(\mathcal{H}) \rrbracket^u}\), then there exists \(((m, w), a, (m', w')) \in \Rightarrow_{\llbracket \text{ind}(\mathcal{H}) \rrbracket^u}\) such that \((l', v') \approx (m', w')\).

Given that \(l = m\), condition \((1')\) in the definition of probabilistic bisimulation is satisfied. Therefore it remains to show condition \((2')\) in the definition of probabilistic bisimulation. Recall the definition of the action sets \(\text{Act}[\mathcal{H}]^u = \{\tau\} \cup \text{Events}\) and \(\text{Act}[\text{ind}(\mathcal{H})]^u = \{\tau\} \cup (\text{Events} \times \text{Dist}(\text{Upd}(\mathcal{X}) \times L) \times \text{Upd}(\mathcal{X}))\). We first consider transitions of \(\llbracket \mathcal{H} \rrbracket^\text{ta}\) and \(\llbracket \text{ind}(\mathcal{H}) \rrbracket^\text{ta}\) that correspond to the action \(\tau\), i.e., the transitions corresponding to time elapsing. The definition of time-elapse transitions depends on \(\text{post}\), which is identical in both \(\mathcal{H}\) and \(\text{ind}(\mathcal{H})\). Hence, for \((l, v)\), the existence of a transition \(((l, v), \tau, (l', v')) \in \Rightarrow_{\llbracket \text{ind}(\mathcal{H}) \rrbracket^u}\) implies the existence of a transition \(((l, v), \tau, (l', v') \mapsto 1) \in \Rightarrow_{\llbracket \mathcal{H} \rrbracket^u}\). Similarly, for \((m, w)\), the existence of a transition \(((m, w), \tau, (m', w') \mapsto 1) \in \Rightarrow_{\llbracket \text{ind}(\mathcal{H}) \rrbracket^u}\) implies the existence of a transition \(((m, w), \tau, (m', w') \mapsto 1) \in \Rightarrow_{\llbracket \mathcal{H} \rrbracket^u}\). Recalling that \((l', v') \approx (m', w')\), we conclude that, in the case of action \(\tau\), we have that condition \((2')\) in the definition of bisimulation implies condition \((2')\) in the definition of probabilistic bisimulation.
We now consider jump transitions. Consider an edge \((l, g, (e, p, \vartheta), \{(\vartheta, l') \mapsto 1\}) \in \text{ind}\{\text{prob}\}\) such that \(v \in g\). We write \(\text{support}(p) = \{(\vartheta_1, l_1), \ldots, (\vartheta_n, l_n)\}\). From the definition of the edge set \(\text{ind}\{\text{prob}\}\) we have that there exist edges \((l, g, (e, p, \vartheta), \{(\vartheta, l) \mapsto 1\}) \in \text{ind}\{\text{prob}\}\) for all \(1 \leq i \leq n\), where \(\vartheta = \vartheta_i\) and \(l' = l_i\) for some \(1 \leq i \leq n\). Consider some vector \(a = [v_1, \ldots, v_n] \in \mathcal{V}^n\) such that \(v_i \in \vartheta_i(v)\) for each \(1 \leq i \leq n\). We then consider the set of transitions corresponding to \(a\), namely \(T_a = \{((l, v), (e, p, \vartheta), (l_i, v_i)) \mid v_i \in \vartheta_i(v) \land 1 \leq i \leq n\}\). We have that \(T_a \subseteq \Rightarrow_{\text{ind}(\mathcal{H})}^a\) for the following reasons: first, we have assumed above that \(v \in g\); second, for each \(1 \leq i \leq n\), noting that Bundle\((v, \{(\vartheta_i, l) \mapsto 1\})\) contain vectors of length 1, namely those valuations \(v'\) such that \(v' \in \vartheta_i(v)\), we obtain that \(v_i \in \text{Bundle}(v, \{(\vartheta_i, l) \mapsto 1\})\), which then implies that \(\{(l, v), (e, p, \vartheta), (l_i, v_i) \} \in \Rightarrow_\text{ind}(\mathcal{H})^a\) (furthermore, recall that we simplify the notation of such transitions to \(\{(l, v), (e, p, \vartheta), (l_i, v_i)\}\)). Informally, \((l_i, v_i)\) is the unique state which corresponds to the traversal of edge \((l, g, (e, p, \vartheta), \{(\vartheta, l) \mapsto 1\}) \) from \((l, v)\).

Now, by condition \((2')\) of the definition of bisimulation, the existence of each transition in \(T_a\) implies the existence of an equally-labelled transition from \((m, w)\) leading to a bisimilar state. Formally, we can obtain a set \(U = \{(m, w), (e, p, \vartheta), (m_i, w_i) \} \mid \{(l, v), (e, p, \vartheta), (l_i, v_i)\} \in T_a \land (l_i, v_i) \approx (m_i, w_i)\} \) and \(U \subseteq \Rightarrow_{\text{ind}(\mathcal{H})}^a\).

Next, we show that the transition sets \(T_a\) and \(U\) imply the existence of probabilistic transitions from \((l, v)\) and \((m, w)\) in \([\text{ind}(\mathcal{H})]^a\). First note that \(a \in \text{Bundle}(v, p)\), because \(v_i \in \vartheta_i(v)\) for each \(1 \leq i \leq n\). Then \(a\) induces the transition \(\{(l, v), e, \mu_a\} \in \Rightarrow_{\text{ind}(\mathcal{H})}^a\) where the distribution \(\mu_a\) is defined as \(\mu_a(l', v') = \sum_{1 \leq i \leq n, v' = a[i]} p(\vartheta_i, l_i)\) for each \((l', v') \in S\).

Let \(b = [w_1, \ldots, w_n]\). Note that, for each \(1 \leq i \leq n\), we have \((l_i, a[i]) \approx (m_i, b[i])\) (because \((l_i, v_i) \approx (m_i, w_i)\)). We also have that \(b \in \text{Bundle}(v, p)\) because, for each \(1 \leq i \leq n\), the existence of the transition \(\{(m, w), (e, p, \vartheta), (m_i, w_i)\}\) implies that \(w_i \in \vartheta_i(w)\). In a similar manner to the case of \(a\), we have that \(b\) induces the transition \(\{(m, w), e, \nu_b\} \in \Rightarrow_{\text{ind}(\mathcal{H})}^a\) where \(\nu_b(m', w') = \sum_{1 \leq i \leq n, w' = b[i]} p(\vartheta_i, m_i)\) for each \((m', w') \in S\).

We now show that \(\mu_a[C] = \nu_b[C]\) for all equivalence classes \(C\) of \(\approx\). Recall that:

\[
\mu_a[C] = \sum_{(l', v') \in C} \mu_a(l', v') = \sum_{(l', v') \in C} \sum_{1 \leq i \leq n, v' = a[i]} p(\vartheta_i, l_i)
\]

\[
\nu_b[C] = \sum_{(m', w') \in C} \nu_b(m', w') = \sum_{(m', w') \in C} \sum_{1 \leq i \leq n, w' = b[i]} p(\vartheta_i, m_i).
\]

Given that \(\approx\) respects \(\equiv_{\text{loc}}\), then, for all \((l', v'), (l'', v'') \in C\), we have \(l' = l''\). We use \(l_C\) to denote the location component of the states in \(C\). Now consider the sets \(\mathcal{F}_a^C = \{i \in \mathbb{N} \mid 1 \leq i \leq n \land (l_C, a[i]) \in C\}\) and \(\mathcal{F}_b^C = \{i \in \mathbb{N} \mid 1 \leq i \leq n \land (l_C, b[i]) \in C\}\). Note that we can write:

\[
\sum_{(l_C, v') \in C} \sum_{1 \leq i \leq n, v' = a[i]} p(\vartheta_i, l_C) = \sum_{i \in \mathcal{F}_a^C} p(\vartheta_i, l_C).
\]

\[
\sum_{(l_C, w') \in C} \sum_{1 \leq i \leq n, w' = b[i]} p(\vartheta_i, l_C) = \sum_{i \in \mathcal{F}_b^C} p(\vartheta_i, l_C).
\]

Hence, to show that \(\mu_a[C] = \nu_b[C]\), it suffices to show that \(\sum_{i \in \mathcal{F}_a^C} p(\vartheta_i, l') = \sum_{i \in \mathcal{F}_b^C} p(\vartheta_i, m')\).

Given that we established above that, for each \(1 \leq i \leq n\), we have \((l_C, a[i]) \approx (l_C, b[i])\), we also conclude that \((l_C, a[i]) \in C\) if and only if \((l_C, b[i]) \in C\). This implies that \(\mathcal{F}_a^C = \mathcal{F}_b^C\). We then have that \(\sum_{i \in \mathcal{F}_a^C} p(\vartheta_i, l') = \sum_{i \in \mathcal{F}_b^C} p(\vartheta_i, m')\). Hence \(\mu_a[C] = \nu_b[C]\). We thus conclude that \(\mu_a \approx \nu_b\). Condition (2) of the definition of probabilistic bisimulation has been satisfied.
3 Approximate Abstraction of a Class of Stochastic Hybrid Automata

In this section, we consider the analysis of a restricted class of stochastic hybrid automata (SHA) by a reduction to probabilistic rectangular automata (PRA), which can then be transformed into a finite-state PA. We describe briefly SHA in Section 3.1 and PRA in Section 3.2. As an intermediate step between SHA and PRA, we introduce a class of PHA called probabilistic slope-update hybrid automata (PSUHA) in Section 3.3. Given certain assumptions, we show that PSUHA can be reduced to PRA. We illustrate the approach with respect to the example of Figure 1.

3.1 Stochastic Hybrid Automata: A Brief Overview

We now describe briefly the SHA formalism; for a more technical introduction, see [FHH+11, Hah13]. A SHA features the same components as a PHA, but also features a set of continuous probabilistic edges. A continuous probabilistic edge \((l, g, e, \tilde{p})\) comprises (1) a source location \(l\), (2) a guard \(g\), (3) an event \(e\), and (4) a measurable function \(\tilde{p}\) mapping each state to a probability measure over locations and valuations. We assume that \(\text{post}\), guards and update sets of standard probabilistic edges, and guards of continuous probabilistic edges are measurable in the sense of [FHH+11, Hah13].

We informally describe the semantics of SHA. As in the case of PHA, from state \((l, v)\) time can pass with the values of the continuous variables changing as described by the operator \(\text{post}\). The conditions in which standard probabilistic edges can be taken, and their effects on the location and continuous variables, are also as in the case of \(\text{prob}\) of PHA. Similarly to standard probabilistic edges, a continuous probabilistic edge \((l, g, e, \tilde{p})\) can be taken if the current state is \((l, v)\) for \(v \in g\), and the location and valuation is updated according to a probabilistic choice. However, for a continuous probabilistic edge, a probability measure (which depends on the current state) described by \(\tilde{p}\) is used; in particular, this allows us to update variables according to infinite-support probability distributions, such as the continuous normal or uniform distributions. In [FHH+11, Hah13], continuous probabilistic edges have been used in the modelling of hybrid systems that are subject to measurement errors.

Note that, as with PHA, SHA do not feature probabilistic choice over elements of the post operator: therefore, choices regarding the amount of time to let elapse, and regarding the change to the continuous variables corresponding to time elapsing, are nondeterministic. However, the change to continuous variables corresponding to time elapsing can be made to be dependent on continuous probabilistic choice indirectly. Consider an example of a stochastic hybrid system in which the rate of change of a continuous variable \(x\) as time passes in location \(l\) is determined by a continuous probabilistic choice, say a uniform distribution over the interval \([1, 3]\), made when location \(l\) is entered. Then we can model this system as a SHA with a continuous variable \(\tau\) that is updated by continuous probabilistic edges according to a uniform distribution over \([1, 3]\) on entry to location \(l\). The post operator then specifies that the rate of change of \(x\) as time passes in \(l\) is equal to the value of \(\tau\), and that the value of \(\tau\) does not change as time passes in \(l\).

We now explain briefly the methodology of [FHH+11, Hah13] for the construction of approximate abstractions of SHA. This approach concerns the construction of a PHA that is identical to the SHA except for the fact that each continuous probabilistic edge is replaced by a standard probabilistic edge with the same source location, guard and event, but for which the distribution
ranges over update sets and target locations rather than over variable values and target locations. For example, consider the case of a continuous probabilistic edge \((l, g, e, \tilde{p})\), where \(\tilde{p}\) corresponds to the choice of a single target location \(l'\), and updates the value of a variable \(x\) according to a uniform distribution over \([1, 3]\), while leaving the values of the other variables unchanged. The continuous probabilistic edge can be replaced by a discrete probabilistic edge \((l, g, e, p)\), where \(p\) is defined in the following manner. The interval \([1, 3]\) is represented by a finite number of intervals, the union of which contains \([1, 3]\), and each of which corresponds to a probability derived from the original distribution. For example, we can consider the intervals \([1, 2]\) and \([2, 3]\), which each correspond to probability \(\frac{1}{2}\), in accordance with the original uniform distribution. Then we let \(p = \{(\vartheta_1, l') \mapsto \frac{1}{2}, (\vartheta_2, l') \mapsto \frac{1}{2}\}\), where \(\vartheta_1\) (\(\vartheta_2\), respectively) corresponds to a nondeterministic choice of which value to update \(x\) to within the interval \([1, 2]\) (\([2, 3]\), respectively), while leaving the values of the other variables unchanged. It can be seen that this construction results in a conservative approximation of a SHA, in the sense that any strategy of the SHA can be emulated by a randomised strategy of the constructed PHA (see [Hah13], Theorem 4.22). Hence the minimum (maximum, respectively) probability of reaching a certain location in the PHA will be no greater than (no less than, respectively) the minimum (maximum, respectively) probability of reaching the location in the SHA. This approximate abstraction methodology then allows tools for the analysis for PHA to be applied to the analysis of SHA.

3.2 Probabilistic Rectangular Automata

Let \(\mathcal{I}\) be the set of intervals \(\{(a, b) \mid a, b \in \mathbb{Z}\} \cup \{(−\infty, a) \mid a \in \mathbb{Z}\} \cup \{(a, \infty) \mid a \in \mathbb{Z}\} \cup \{−\infty, \infty\}\). In the following, we describe a set \(V\) of valuations \(\nu'(X)\) over \(X \subseteq \mathcal{X}\) to be rectangular over \(X\) if, for each variable \(x \in X\), we have that there exists an interval \(I_x \in \mathcal{I}\) such that \(v \in V\) if and only if \(v(x) \in I_x\) for all \(x \in X\). In this case, we often write \(%v\) to denote \(V\). If a set of valuations is rectangular over \(\mathcal{X}\), we simply describe the set as being rectangular.

In a PRA, all guards and updates are described in terms of rectangular sets over subsets of the set \(\mathcal{X}\) of continuous variables. The post operator of a PRA is characterised in the following way: for each location, (1) the rate of change of each variable belongs to an interval, and (2) the variable values that can be obtained as time passes are constrained to be within a rectangular set. Formally a PRA \(\mathcal{R} = (L, \mathcal{X}, \text{Events}, \text{post}, \text{prob})\) is defined as a PHA with the following restrictions. For each probabilistic edge \((l, g, e, p) \in \text{prob}\), we have that \(g\) is rectangular. Furthermore, writing \(\text{support}(p) = \{((\vartheta_1, l_1), \ldots, (\vartheta_n, l_n))\}\), for each \(1 \leq i \leq n\) there exists variable set \(\text{Reset}(\vartheta_i) \subseteq \mathcal{X}\) and a rectangular set \(%\vartheta_i\) such that, for each \(v \in \nu', \nu' \in \vartheta_i\), we have \(\nu'(x) \in \vartheta_i\) for each \(x \in \text{Reset}(\vartheta_i)\) and \(\nu'(x) = \nu(x)\) for each \(x \in \mathcal{X} \setminus \text{Reset}(\vartheta_i)\). The post operator \(\text{post}\) takes the following form: for each location \(l \in L\), there exists a rectangular set \(\text{inv}(l) = \prod_{x \in \mathcal{X}} I_x^l\), and there exists an interval \(\prod_{x \in \mathcal{X}} \tilde{I}_x^l\) with \(\tilde{I}_x^l, I_x^l \in \mathbb{Z}\) for each \(x \in \mathcal{X}\) such that, for each valuation \(v \in \nu'\) and delay \(\delta \in \mathbb{R}_{\geq 0}\), we have \(\text{post}(l, v, \delta) = \nu'(x) \in \nu(x) + \delta \times \tilde{I}_x^l\). As a PRA is a subclass of PHA, we can define the dense-time semantics \(\langle \mathcal{R} \rangle^k\) of a PRA as in Section 2. The discrete-time semantics of the PRA \(\mathcal{R} = (L, \mathcal{X}, \text{Events}, \text{post}, \text{prob})\) with respect to \(k \in \mathbb{N}\) \(\setminus\{0\}\) is the PA \((\langle \mathcal{R} \rangle)^k = (S, \text{Act}, \Rightarrow)\) defined as for the dense-time semantics except for \(\Rightarrow\), which is defined as \(\Rightarrow = \bigcup_{e \in \text{Events}} \Rightarrow_{\mathcal{E}}\). We say that the variable \(x \in \mathcal{X}\) is nondecreasing if
both $I^l_x \subseteq \mathbb{R}_{\geq 0}$ and $[\underline{f}^l_x, \bar{f}^l_x] \subseteq \mathbb{R}_{\geq 0}$ for all locations $l \in L$. The variable $x \in \mathcal{X}$ is bounded if $I^l_x$ is a bounded set, for all locations $l \in L$. The variable $x \in \mathcal{X}$ has nondecreasing or bounded variables if all variables in $\mathcal{X}$ are either nondecreasing or bounded. Given a PRA $\mathcal{R}$ with nondecreasing or bounded variables, the discrete-time semantics $(\langle \mathcal{R} \rangle)^k$ of $\mathcal{R}$ with respect to any $k \in \mathbb{N}$ has a finite number of classes for probabilistic bisimulation with respect to $\equiv_{loc}$ [Spr11]. This result permits the construction of a finite-state PA that is equivalent to the PRA, and which can then be analysed using tools for finite-state PA, such as PRISM [KNP11].

### 3.3 Probabilistic Slope-Update Hybrid Automata

We now consider PSUHA, a class of PHA that generalises PRA, and in which the rate of change of some variables can be described as the value of other (constant) variables. A PSUHA $\mathcal{U} = (L, \mathcal{X}, \text{Events}, \text{post}, \text{prob})$ is a PHA where the components $L, \mathcal{X}, \text{Events}$ and $\text{prob}$ are defined as in the case of PRA, and where the post operator $\text{post}$ is defined in the following way. First, we identify a subset $C \subseteq \mathcal{X}$ of variables that remain constant as time passes, in each location (variables in $C$ may be reset when probabilistic edges are taken). For each location $l \in L$, there exists a subset $\text{Dep}(l) \subseteq \mathcal{X}$ of variables that have a rate of change that is equal to the value of a variable in $C$ (note that $\text{Dep}(l) \cap C = \emptyset$). For variable $x \in \text{Dep}(l)$, we write $\text{DepOn}(l, x) \in C$ to denote the variable on which the rate of change of $x$ depends in $l$. Let $\text{Rectangular}(l) = \mathcal{X} \setminus (C \cup \text{Dep}(l))$. We can now proceed to define the post operator $\text{post}$: for each location $l \in L$, there exists a rectangular set $\text{inv}(l) = \prod_{x \in \mathcal{X}} I_x$, and an interval $[\underline{f}^l_x, \bar{f}^l_x]$ with $\underline{f}^l_x, \bar{f}^l_x \in \mathbb{Z}$ for each $x \in \text{Rectangular}(l)$ such that, for each valuation $v \in \mathcal{Y}$ and delay $\delta \in \mathbb{R}_{\geq 0}$, we have:

$$\text{post}(l, v, \delta) = \{ v' \mid (\forall x \in \text{Rectangular}(l). v'(x) \in [v(x) + \delta \ast \underline{f}^l_x, v(x) + \delta \ast \bar{f}^l_x])$$

$$\land (\forall x \in \text{Dep}(l). v'(x) = v(x) + \delta \ast v(\text{DepOn}(l, x)))$$

$$\land (\forall x \in C. v'(x) = v(x))$$

$$\land v' \in \text{inv}(l) \} .$$

The set $\text{prob}$ of probabilistic edges of a PSUHA is subject to the following assumptions:

1. (Variables in $C$ are reset on entry to each location.) Each $(l, g, e, p) \in \text{prob}$ is such that, for each $(\vartheta, l') \in \text{support}(p)$ and for each $x \in C$, we have that $x \in \text{Reset}(\vartheta)$.

2. (On entry to a given location by multiple probabilistic edges, the same interval is used to define the value of a variable in $C$.) For each $l \in L$ and each $x \in C$, there exist $\underline{u}^l_x, \bar{u}^l_x \in \mathbb{Z}$ such that, for each $(\vartheta, l) \in \bigcup_{(l', g, e, p) \in \text{prob}} \text{support}(p)$, we have $\underline{u}^0_x = \underline{u}^l_x$ and $\bar{u}^0_x = \bar{u}^l_x$.

3. (No probabilistic edge features a probabilistic choice between updates associated with the same location.) For each $(l, g, e, p) \in \text{prob}$, for each pair $(\vartheta', l'), (\vartheta'', l'') \in \text{support}(p)$, we have $l' \neq l''$.

4. (The guards of probabilistic edges do not constrain values of variables in $C$.) For each $(l, g, e, p) \in \text{prob}$ where we write $g = \prod_{x \in \mathcal{X}} I_x$, and each $x \in C$, we have that $I_x = (-\infty, \infty)$.

We now show that, for any PSUHA, we can construct a PRA with the same state set such that the maximum reachability probabilities are identical in the PSUHA and PRA, and similarly for
the minimum reachability probabilities. We present the result in the context of the discrete-time semantics, although we note that an analogous result holds for the dense-time semantics. Let \( k \in \mathbb{N} \setminus \{0\} \), and let \( \mathcal{U} \) and \( \mathcal{R} \) be PSUHA and PRA, respectively, such that \( \langle \langle \mathcal{U} \rangle \rangle^k \) and \( \langle \langle \mathcal{R} \rangle \rangle^k \) have the same state set. Then we introduce the following equivalence relation on states: for state \((l,v)\) of \( \langle \langle \mathcal{U} \rangle \rangle^k \) and state \((m,w)\) of \( \langle \langle \mathcal{R} \rangle \rangle^k \), we write \((l,v) \equiv_\mathcal{C} (m,w)\) if and only if (1) \( l = m \) and (2) \( v(x) = w(x) \) for all \( x \in \mathcal{X} \setminus C \).

**Proposition 2** Let \( \mathcal{U} \) be a PSUHA, \( k \in \mathbb{N} \setminus \{0\} \) and \( F \subseteq L \) be a set of locations. Then there exists a PRA \( \mathcal{R} \) such that \( \langle \langle \mathcal{U} \rangle \rangle^k \) and \( \langle \langle \mathcal{R} \rangle \rangle^k \) have the same state set and, for each pair \((l,v),(m,w)\) of states such that \((l,v) \equiv_\mathcal{C} (m,w)\), we have:

\[
\text{MaxReach}_{\langle\langle \mathcal{U} \rangle \rangle^k}^{\langle\langle \mathcal{U} \rangle \rangle^k}(S_F) = \text{MaxReach}_{\langle\langle \mathcal{R} \rangle \rangle^k}^{\langle\langle \mathcal{R} \rangle \rangle^k}(S_F), \quad \text{MinReach}_{\langle\langle \mathcal{U} \rangle \rangle^k}^{\langle\langle \mathcal{U} \rangle \rangle^k}(l,v) = \text{MinReach}_{\langle\langle \mathcal{R} \rangle \rangle^k}^{\langle\langle \mathcal{R} \rangle \rangle^k}(l,v).
\]

where \( S_F = \{ (l,v) \mid (l,v) \in S \land I \subseteq F \} \).

**Proof sketch.** For details of the proof, see [Spr14]. Let \( \mathcal{U} = (L, \mathcal{X}, \text{Events}, \text{post}, \text{prob}) \) be a PSUHA. We construct the PRA \( \mathcal{R} = (L, \mathcal{X}, \text{Events}, \text{post'}, \text{prob}) \) to be identical to \( \mathcal{U} \) except for the post operator. For each state \((l,v) \in S\) and each duration \( \delta \in \mathbb{R}_{\geq 0}\), we define:

\[
\text{post'}(l,v,\delta) = \{v' \mid (\forall x \in \text{Rectangular}(l), v'(x) \in [v(x) + \delta \ast f^*_l, v(x) + \delta \ast \bar{f}^*_l],
\land (\forall x \in \text{Dep}(l), v'(x) \in [v(x) + \delta \ast f^*_{\text{DepOn}(l,x)}], v(x) + \delta \ast \bar{f}^*_{\text{DepOn}(l,x)}),
\land (\forall x \in C, v'(x) = v(x)),
\land v' \in \text{inv}(l) \}.
\]

Note that, for each variable \( x \) in \( \text{Dep}(l) \), the slope of \( x \) in \( l \) is chosen the interval \([f^*_{\text{DepOn}(l,x)}, \bar{f}^*_{\text{DepOn}(l,x)}]\), i.e., the interval used to determine the value of the variable \( \text{DepOn}(l,x) \) on which \( x \) depends on entry to \( l \). It can be seen that \( \text{post'} \) conforms to the definition of possible post operators of PRA, hence \( \mathcal{R} \) is a PRA.

The proof then involves showing the following condition: a jump transition of \( \langle \langle \mathcal{U} \rangle \rangle^k \) followed by a sequence of time transitions can be replicated by a jump transition of \( \langle \langle \mathcal{R} \rangle \rangle^k \) followed by a sequence of time transitions, and vice versa, where the states before and after the transitions are related by \( \equiv_\mathcal{C} \). Furthermore, the jump transitions are made using the same probabilistic edges, which means that corresponding transitions can be shown to have the same probability. Note that we consider such a multi-step equivalence relation (that is, we compare \( \langle \langle \mathcal{U} \rangle \rangle^k \) and \( \langle \langle \mathcal{R} \rangle \rangle^k \) by considering a jump transition and sequences of time-elapse transitions), rather than a 1-step relation such as probabilistic bisimulation, for the following reason: when considering the case in which \( \langle \langle \mathcal{U} \rangle \rangle^k \) must emulate a jump transition of \( \langle \langle \mathcal{R} \rangle \rangle^k \), the rates of change chosen by \( \langle \langle \mathcal{R} \rangle \rangle^k \) in subsequent transitions must be known by \( \langle \langle \mathcal{U} \rangle \rangle^k \), so that \( \langle \langle \mathcal{U} \rangle \rangle^k \) knows what value to set the constants on which the rates of change of variables will depend.

Using this multi-step equivalence property, given that time-elapse transitions correspond to probability 1, it follows that, for state \((l,v)\) of \( \langle \langle \mathcal{U} \rangle \rangle^k \) and state \((m,w)\) of \( \langle \langle \mathcal{R} \rangle \rangle^k \) such that \((l,v) \equiv_\mathcal{C} (m,w)\), we have \( \text{MaxReach}_{\langle\langle \mathcal{U} \rangle \rangle^k}^{\langle\langle \mathcal{U} \rangle \rangle^k}(l,v) = \text{MaxReach}_{\langle\langle \mathcal{R} \rangle \rangle^k}^{\langle\langle \mathcal{R} \rangle \rangle^k}(m,w) \) and \( \text{MinReach}_{\langle\langle \mathcal{U} \rangle \rangle^k}^{\langle\langle \mathcal{U} \rangle \rangle^k}(l,v) = \text{MinReach}_{\langle\langle \mathcal{R} \rangle \rangle^k}^{\langle\langle \mathcal{R} \rangle \rangle^k}(m,w) \). \( \square \)
Proposition 2 suggests the following approximate analysis method for the class of discrete-time SHA that, when abstracted using the method of [FHH+11, Hah13], result in a discrete-time PSUHA with nondecreasing or bounded variables: from the PSUHA, then obtain an equivalent PRA according to Proposition 2; subsequently, the obtained PRA is transformed into a finite-state PA using the results of [Spr11].

We also mention that we can encode in PSUHA (and hence in the associated PRA) the periodic resampling of variables in C, which can be done every \( \frac{1}{k} \) time units, for some \( k \in \mathbb{N} \setminus \{0\} \). The intuition underlying the encoding is that at most \( \frac{1}{k} \) time units can elapse in each location before taking a probabilistic edge. This can be enforced by adding a clock variable \( x \) and by defining the post operator so that the value of \( x \) cannot exceed \( \frac{1}{k} \). Each location has a “self-loop” probabilistic edge that is enabled when \( x \) is equal to \( \frac{1}{k} \), resets \( x \) to 0, resets variables in \( C \) and does not change any other variable.\(^1\)

### 3.4 Example: Faulty Thermostat

We now illustrate the approximate analysis method proposed in the previous section with an application to the PRA model of a faulty thermostat presented in Figure 1. As described in Section 1, when the heater is on, we consider that the rate of increase of the temperature is chosen from the normal distribution with mean 3.5 and standard deviation 1, truncated to the interval \([-4, 1]\). Similarly, when the heater is off, instead of a nondeterministically chosen decrease of within \([-4, -1]\), we consider the normal distribution with mean \(-2.5\) and standard deviation 0.5, truncated to the interval \([-4, -1]\). This system can be modelled as a SHA in the sense of Section 3.1. Now we consider how the approximate abstraction approach of [FHH+11, Hah13] can be used to obtain a PSUHA. For the case in which the heater is on, we consider three subintervals, \([1, 3]\), \([3, 4]\) and \([4, 6]\), which correspond approximately to the probabilities 0.312, 0.376 and 0.312, respectively, of the aforementioned normal distribution. For the case in which the

\(^1\) Note that this construction involves probabilistic edges that choose between alternatives involving the same location: to satisfy the assumption (3) in the definition of the set of probabilistic edges of a PSUHA, we can consider multiple copies of the location, each corresponding to a different alternative associated with each probabilistic edge, w.l.o.g.
heater is off, we consider the three subintervals $[−4,−3]$, $[−3,−2]$ and $[−2,−1]$, corresponding to the probabilities 0.159, 0.682 and 0.159, respectively. We then constructed the PSUHA according to the discrete-time resampling construction, then transformed the resulting model to a PRA and, in turn, to a finite-state PA, which was analysed with PRISM.

In Figure 2, we present the maximum and minimum probabilities of visiting location $l_{\text{deact}}$ within time bound $T$, both in the “original” PRA model shown in Figure 1, and in the “new” PRA model obtained by the method described above. In both cases, we use the time granularity $k = 10$, resulting in a PA with 136112 states. It is clear that the results obtained for the original PRA model bound those obtained by the new PRA model, often by a substantial amount.

4 Conclusions

We have presented general results on obtaining finite bisimulation quotients for the exact verification of PHA, and a method for the approximate verification of a restricted subclass of SHA, based on a combination of the approximation technique of [FHH+11, Hah13] and the discrete-time verification method of [Spr11]. We mention some limitations of the latter approach. First, the approach of choosing probabilistically a slope of a variable on entry to a location, which then remains constant in that location, may be unrealistic for some applications, in which the slope of a variable may be subject to stochastic fluctuation as time passes. This criticism applies principally to the case of the dense-time semantics: in a discrete-time context, only the target state after $\frac{1}{k}$ time units is important, rather than the trajectory used to reach it, because nondeterministic choice is disabled as time passes during the $\frac{1}{k}$ time units. Second, our restriction of bounded slopes in the context of PSUHA leads to the necessity of truncation of some continuous distributions. We envisage that this restriction can be lifted. In future work we plan to apply the results of Section 3 to more realistic case studies.

Acknowledgements: Thanks to Ahmed Bouajjani for comments regarding the results of Section 2.

Bibliography


Reachability and Reward Checking for Stochastic Timed Automata†

E. Moritz Hahn1, Arnd Hartmanns2 and Holger Hermanns2

1 State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences, China

2 Saarland University – Computer Science
Saarbrücken, Germany

Abstract: Stochastic timed automata are an expressive formal model for hard and soft real-time systems. They support choices and delays that can be deterministic, nondeterministic or stochastic. Stochastic choices and delays can be based on arbitrary discrete and continuous distributions. In this paper, we present an analysis approach for stochastic timed automata based on abstraction and probabilistic model checking. It delivers upper/lower bounds on maximum/minimum reachability probabilities and expected cumulative reward values. Based on theory originally developed for stochastic hybrid systems, it is the first fully automated model checking technique for stochastic timed automata. Using an implementation as part of the MODEST TOOLSET and four varied examples, we show that the approach works in practice and present a detailed evaluation of its applicability, its efficiency, and current limitations.

Keywords: stochastic timed automata, probabilistic reachability, expected rewards

1 Introduction

Proper consideration of quantitative aspects is crucial to formally model and analyse many complex safety-critical or economically vital systems. Key quantities are time, to represent e.g. timeouts and delays, and probabilities, to model the quantified uncertainty that appears, for example, in randomised algorithms, as disturbances like random failures, and as randomised delays. Additionally, nondeterminism is a key feature for verification that enables abstraction, concurrency, and the specification of unquantified uncertainty. We need to analyse properties like the probability of (un)desired behaviour, the expected time to success, or the probability of an error within a given amount of time.

A suitable model for these kinds of systems are stochastic timed automata (STA). They allow nondeterministic decisions, real time aspects, continuous and discrete probabilistic choices, and any combination thereof. STA had been introduced as the original formal semantics of the high-level compositional modelling language MODEST [BDHK06]. They are at the heart of a large spectrum of compositional models, summarised in Figure 1, rooted in labelled transition

† This work is supported by the EU 7th Framework Programme under grant agreements 295261 (MEALS) and 318490 (SENSATION), the DFG Transregional Collaborative Research Centre SFB/TR 14 AVACS, the CAS/SAFEA International Partnership Program for Creative Research Teams, the Chinese Academy of Sciences Fellowship (grant no. 2013Y1GB0006), and the research fund for International Young Scientists (grant no. 61350110518).
systems and Markov chains. MODEST has since been extended with support for continuous dynamics [HHHK13] based on the model of stochastic hybrid automata [FHH+11]. The compositionality properties of STA in turn rest on results established by Strulo, Bravetti and especially D’Argenio [BD04, BG02, DK05, HS00]. STA can also be viewed as generalised semi-Markov processes (GSMP) extended with discrete and continuous nondeterminism.

The MODEST TOOLSET, which is available at www.modestchecker.net, provides analysis tools for a variety of these models [HH14]. However, so far it did not support the genuine analysis of full STA models with nondeterministic decisions, and that is what this paper is about: We present an algorithm to compute upper/lower bounds on maximum/minimum reachability probabilities and expected cumulative reward values in a given STA. It uses abstraction to convert the STA into a PTA, which can then be analysed using existing PTA model checking techniques [NPS13]. We show the correctness of the abstraction for the considered properties. The underlying theory was originally developed for stochastic hybrid systems [FHH+11, Hah13]; we explain how we take advantage of the specialisation to timed systems to improve scalability, usability and applicability. We implemented the new approach in the MODEST TOOLSET, which allows us to investigate its effectiveness and efficiency using four different example models.

**Related work.** Kwiatkowska et al. [KNSS00] have pioneered the foundational basis of STA model checking with their work on timed automata with generally distributed clocks, verified against properties in probabilistic timed CTL. They use a semantics based on the region graph where regions are further partitioned to cater for the stochastic behaviour. The main differences to what we present in this paper are that our approach can handle distributions with unbounded support (e.g. the exponential and normal distributions), supports expected rewards, and that we avoid the region construction. We also show a working implementation, which instead currently uses a digital clocks semantics, but this can be interchanged with other approaches. In case an STA only uses bounded-support distributions (e.g. the continuous uniform one), our approach provides the same error bounds. However, we do not provide error bounds for the general case.

Other related approaches that we find are based on statistical model checking [DLL+11], numerical discretisation [LHK01], plain discrete event simulation [HS00], or state classes [BBH+13] (on a different model also called STA). However, all of these either implicitly or explicitly ex-
clude the presence of nondeterminism, and thus work in the GSMP realm instead. As an example, consider the “STA” model of [BBJM12] (which is closely related to the one of [BBH+13]): There, a single distribution is sampled on every edge, the result being the exact sojourn time in the following location. In comparison, our model of STA also supports continuous and discrete nondeterminism as well as multiple samplings per edge and multiple sampled variables that can memorise their values over several edges/locations. In particular, the method we present in this paper is geared towards correctly handling the general combination of stochastics and nondeterminism. Dedicated approaches for deterministic models provide better precision or performance for that special case. We return to this tradeoff in our evaluation in Section 6, where we look at two deterministic models for comparison, and two nondeterministic case studies that can only be handled correctly with our new approach.

2 Preliminaries

We use $\mathbb{R}_0^+$ to denote the set of nonnegative real numbers and $\mathbb{N}^+$ for the positive natural numbers. For a set $S$, $\mathcal{P}(S)$ denotes its powerset. We assume familiarity with general notions and constructions from probability theory. Due to space constraints, we do not consider possible measurability issues (see e.g. [Hah13, Chapter 5] for discussions concerning a more general model). For all probability distributions, we assume an according (Borel) space to be given. By $\text{Prob}(\Omega)$ we denote the set of all probability measures on the sample space $\Omega$. The Dirac distribution $\Dirac(x) \in \text{Prob}(\Omega)$ is s.t. we have $\Dirac(x)(A) = 1$ if $x \in A$ and $\Dirac(x)(A) = 0$ otherwise. By $[\forall i: x_i \mapsto p_i]$ or $[x_1 \mapsto p_1, \ldots, x_n \mapsto p_n]$ we denote the distribution $\sum p_i \Dirac(x_i)$.

Given a set $\text{Var}$ of variables where each variable $x$ has an associated domain (or type) $\text{Dom}(x)$, we let $\text{Val}$ denote the set of variable valuations, i.e. of functions $\text{Var} \to \bigcup_{x \in \text{Var}} \text{Dom}(x)$ where $v \in \text{Val} \Rightarrow \forall x \in \text{Var}: v(x) \in \text{Dom}(x)$. $\mathbf{0} \in \text{Val}$ assigns zero to every variable. By $\text{Exp}$ we denote the set of expressions over the variables in $\text{Var}$. We write $e(v)$ for the evaluation of expression $e$ in valuation $v$. We consider three restricted classes of expressions: Boolean expressions $\text{Bxp}$, arithmetic expressions $\text{Axp}$ and sampling expressions $\text{Sxp} \supseteq \text{Axp}$ that may include probability distributions. The set of assignments is $\text{Asgn} = \text{Var} \times \text{Sxp}$ such that $(x, e) \in \text{Asgn} \Rightarrow \forall v \in \text{Val}: e(v) \in \text{Dom}(x)$. The modification of $v \in \text{Val}$ according to $u \in \text{Asgn}$ is written as as $[u](v)$. A set of assignments is called an update, and notation for assignments can be lifted to updates. A symbolic probability distribution for a set $S$ is a function $f \in S \to \text{Axp}$ that maps elements of $S$ to weights s.t. the support $\{s \in S \mid f(s) \neq 0\}$ is countable. Given a valuation for the variables appearing in these weights, a symbolic distribution can be turned into the concrete probability distribution given by the ratios of individual weights over the sum of all weights in the support. We only consider proper symbolic distributions: those where all weights evaluate to positive numbers and the sum of all weights is finite (i.e. convergent) and nonzero, for all relevant valuations.

3 Stochastic Timed Automata

As a generalisation of timed automata, stochastic timed automata deal with time through clock variables (or clocks). Clocks take values in $\mathbb{R}_0^+$ and advance synchronously over time with rate 1. If $v \in \text{Val}$ and $t \in \mathbb{R}_0^+$, then $v + t$ denotes the valuation where all clocks have been incremented.
 Reachability and Reward Checking for Stochastic Timed Automata

by \( t \). Clock constraints are expressions in \( Bxp \) of the form

\[
\mathcal{CC} ::= b \mid \mathcal{CC} \wedge \mathcal{CC} \mid \mathcal{CC} \vee \mathcal{CC} \mid c \sim e \mid c_1 - c_2 \sim e
\]

where \( \sim \in \{ >, \geq, <, \leq, =, \neq \} \), \( c, c_1, c_2 \) are clocks and \( b \in Bxp, e \in Axp \) are clock-free expressions. If all \( e \) are of integer type, we have an integer clock constraint. A clock constraint that does not contain the last case (where two clocks are compared) is diagonal-free. If all comparison operators \( \sim \) used in a clock constraint are in \( \{ \geq, \leq \} \), it is closed.

**Definition 1** A stochastic timed automaton (STA) is a 6-tuple \( \langle \text{Loc}, \text{Var}, A, E, \text{init}, \text{Inv} \rangle \) where \( \text{Loc} \) is a countable set of locations, \( \text{Var} \supseteq \mathcal{C} \) is a finite set of variables with a subset of clocks \( \mathcal{C} \), \( A \) is the automaton’s finite alphabet, \( E \in \text{Loc} \to \mathcal{P}(\mathcal{C} \times A \times Wxp) \) is the edge function, \( \text{init} \in \text{Loc} \) is the initial location, and \( \text{Inv} \in \text{Loc} \to \mathcal{C} \) is the invariant function. An edge consists of a guard that determines when the edge is enabled, an action label, and a symbolic probability distribution over updates and target locations in \( Wxp = \mathcal{P}(\text{Asgn}) \times \text{Loc} \to Axp \). We also write \( I \xrightarrow{e \cdot \text{Asgn}} \mathcal{W} \) for \( \langle g, a, \mathcal{W} \rangle \in E(I) \). The invariant function maps each location to an expression that allows time to pass as long as it evaluates to true.

We can equip STA with rewards, which can be seen as real-valued variables available to external observers only (i.e., they can be used during verification, but not be read in guards etc.). They advance at a certain rate in locations and can be increased when taking an edge:

**Definition 2** A reward \( r = \langle \text{Rew}_{\text{Loc}}, \text{Rew}_{E} \rangle \in (\text{Loc} \to Axp) \times (E \to Axp) \) for an STA as above assigns rate rewards to its locations and edge rewards to edges.

**Example 1** The graphical representation of an example STA with reward \( r \) is shown in Figure 2. Locations contain their name, invariant and rate reward (when not zero). Edges are shown either as simple arrows labeled with guard, action and update if they lead to a single update/location pair with probability 1, or as split arrows with an intermediate node otherwise. Edge rewards are included in updates. The example automaton contains a probabilistic choice on the edge labeled \( a \). Out of \( l_2 \), the edge to \( l_4 \) can only be taken after a deterministic delay of 16 time units, while the one back to \( l_0 \) can be taken after any delay nondeterministically chosen out of \([8, 16] \). After 16 time units, the choice of edge in \( l_2 \) thus becomes nondeterministic. The delay incurred in \( l_1 \), on the other hand, is stochastic: \( x := \text{Exp}(\lambda) \) assigns to \( x \) a value sampled from the exponential distribution with rate \( \lambda \), thus the delay is exponentially distributed with rate \( \lambda \). The reward \( r \) keeps track of the time spent in \( l_1 \), and is increased by 16 upon entering \( l_4 \).
The semantics of STA is given in terms of timed probabilistic transition systems [BDHK06]:

**Definition 3**  A timed probabilistic transition system (TPTS for short) is a 4-tuple \( \langle S, A, T, s_{init} \rangle \) where \( S \) is an (uncountable) set of states, \( A = \mathbb{R}_0^+ \cup A' \) is the system’s (uncountable) alphabet that can be partitioned into delays in \( \mathbb{R}_0^+ \) and discrete actions in \( A' \), \( T \in S \to \mathcal{P}(A \times \text{Prob}(S)) \) is the transition function, and \( s_{init} \in S \) is the initial state. We also write \( s \xrightarrow{a, \mu} s' \) for \( (a, \mu) \in T(s) \). For every delay-labelled transition \( \langle x, \mu \rangle \in T(s), x \in \mathbb{R}_0^+ \), we require that \( \exists s' \in S: \mu = \mathcal{D}(s') \), \( \langle x, \mu' \rangle \in T(s) \Rightarrow \mu = \mu' \) (time determinism), and \( \langle x + x', \mathcal{D}(s') \rangle \in T(s) \Leftrightarrow \exists s'' \in S: \langle x, \mathcal{D}(s'') \rangle \in T(s) \land \langle x', \mathcal{D}(s') \rangle \in T(s') \) for \( x' \in \mathbb{R}_0^+ \) (time additivity).

**Definition 4**  A reward structure for a TPTS is a function \( \text{rew} \in T \to \mathbb{R}_0^+ \) assigning a nonnegative reward to each of its transitions.

**Definition 5**  The semantics of an STA \( M = \langle \text{Loc}, \text{Var}, A, E, \text{Init}, \text{Inv} \rangle \) is defined as the TPTS \( \llbracket M \rrbracket = \langle \text{Loc} \times \text{Val}, \mathbb{R}_0^+ \cup A, T_M, \langle \text{Init}, 0 \rangle \rangle \) where \( T_M \) is the smallest function that satisfies

\[
\begin{align*}
\frac{E}{\mu_w^v(l, v) \xrightarrow{a, T_M} \mu_w^{v'}} \quad \text{(jump)} & \quad t \in \mathbb{R}_0^+ \quad \forall t' \leq t: (\text{Inv}(l))(v + t') \\
\frac{E}{\mu_r^v(l, v) \xrightarrow{T_M} \mu_r^{v'}} \quad \text{(delay)} & \quad \forall l' \in \text{Loc} \text{ and measurable } V' \subseteq \text{Val} \text{ we have }
\end{align*}
\]

where for \( l' \in \text{Loc} \) and measurable \( V' \subseteq \text{Val} \) we have

\[
\mu_w^{v'}(l', V') \overset{\text{def}}{=} \sum_{l \in \text{Loc}, U \in \mathcal{P}(\text{Assign})} \pi_w^v(U, l)(l', V')
\]

where \( \pi_w^v(U, l) \) is the discrete probability distribution for the symbolic distribution \( \mathcal{W} \) in valuation \( v \) and \( \mu_r^v(l, v) \) returns the probability of \( V' \) corresponding to the sampling expressions in update \( U \).

The jump inference rule creates action-labelled transitions for the discrete jumps corresponding to taking an edge in the STA. These transitions therefore go from a state into a continuous distribution over target states according to the sampling expressions in the assignments. Inference rule delay creates real-labelled transitions that represent the passage of time whenever this is allowed by the invariants. They always lead into Dirac distributions, i.e. a single target state.

**Definition 6**  The semantics of a reward \( r \) for an STA \( M \) is a reward structure \( \llbracket r \rrbracket : T_M \to \mathbb{R}_0^+ \) for the TPTS semantics \( \llbracket M \rrbracket \). For transitions labelled with time actions \( t \in \mathbb{R}_0^+ \), it assigns a reward of \( t \) times the location reward rate according to \( \text{Rew}_{\text{Loc}} \). For \( A \)-labelled transitions, the reward value is as defined by \( \text{Rew}_{\mathcal{A}} \) for the STA edge inducing the TPTS transition.

### 3.1 Reachability Probabilities and Expected Rewards

For a given STA, we want to answer questions of the form “what is the probability of reaching a certain set of states from the initial state” and “what is the expected accumulated reward when a certain set of states is reached for the first time”. These properties ask for the computation of reachability probabilities and expected rewards. Since STA may be nondeterministic, we quantify over the resolutions of nondeterminism by asking for maximum or minimum values. For a given TPTS \( M = \langle S, A, T, s_{init} \rangle \), we now define paths and schedulers:

**Definition 7**  The set of finite paths is \( \text{Paths}_{M}^{\text{fin}} \overset{\text{def}}{=} S \times (A \times \text{Prob}(S) \times S)^* \). The last state of the finite path \( \beta = s_0(a_0, s_0) a_1(s_1, a_1) \ldots s_n \) is last(\( \beta \)) \overset{\text{def}}{=} s_n$. A scheduler is a function \( \sigma \in \text{Paths}_{M}^{\text{fin}} \to \)
Reachability and Reward Checking for Stochastic Timed Automata

Prob(A × Prob(S)) so that for each \( \beta \in \text{Paths}_{\text{fin}}^{\text{fin}} \) we have \( \sigma(\beta)(A × \text{Prob}(S)) \setminus T(\text{last}(\beta))) = 0. \)

A scheduler \( \sigma \) induces the stochastic processes \( X^\sigma_M(\cdot) \) of the current state of \( M \) and \( Y^\sigma_M(\cdot) \) of the transition chosen by \( \sigma \) in the current state. It is time-divergent if \( \text{Prob}(\sum_{i=0}^\infty f(Y^\sigma_M(i)) = \infty) = 1 \) for \( f(s \xrightarrow{a} \mu) = a \) if \( a \in \mathbb{R}^*_0 \) and \( f(s \xrightarrow{a} \mu) = 0 \) otherwise. We denote the set of all time-divergent schedulers of \( M \) by \( \mathcal{S}_M \).

A scheduler assigns probabilities to sets of enabled action-distribution pairs depending on the history seen so far. It resolves the nondeterminism in a TPTS so as to obtain probability measures, allowing to derive according stochastic processes. The semantics of the two kinds of properties we consider for STA can then be defined on the TPTS semantics in the usual way using measurable sets of paths and the cylinder construction. Given a set of states \( B \), we are interested in minimal/maximal values, that is infima/suprema over all \( \sigma \in \mathcal{S}_M \). The reachability probability induced by \( \sigma \) is defined as \( \text{Prob}(\exists i \geq 0: X^\sigma_M(i) \in B) \), i.e. the measure of paths with a state in \( B \).

The expected accumulated reward is \( \mathbf{E}(\sum_{i=0}^\infty r(Y^\sigma_M(i))) \) if \( \text{Prob}(\exists i \geq 0: X^\sigma_M(i) \in B) = 1 \) and \( \infty \) otherwise. It is thus the expected reward accumulated along paths provided \( B \) is reached eventually; otherwise the expected value is infinity. As the values of clocks are explicit in TPTS, timed properties can be specified by referring to these values directly in the characterisation of \( B \), e.g. referring to an extra clock that is never reset to specify time bounds.

**Example 2** We are interested in the probability of reaching \( l_3 \) or \( l_4 \) within at most \( t \) time units in the STA of the previous example. The minimum probability is 0 because the invariant of \( l_0 \) allows us to stay there forever. If \( t < 8 \), we can only reach \( l_3 \) and thus compute the maximum probability using the cdf of the exponential distribution: it is \( p = \frac{1}{2} \cdot (1 - e^{-\lambda t}) \). If \( t \geq 16 \), we can also reach \( l_4 \) and the result is \( p = \frac{1}{2} + p \). For \( t \in [8, 16) \), we get \( p' = \frac{1}{2} \cdot (1 - e^{-\lambda t}) + \frac{1}{2} \cdot (1 - e^{-\lambda (t - 8)}) \) by going back to \( l_0 \) from \( l_2 \) as soon as possible. Observe that \( p = p' \) for \( t = 8 \), but for \( t = 16 \), \( p' \neq \frac{1}{2} + p' \) here, the nondeterministic choice available in \( l_2 \) makes an important difference.

Now, let us look at the (time-unbounded) minimum and maximum expected reward \( r \) when we reach \( l_3 \) or \( l_4 \). By definition, since there is a scheduler that reaches those locations with probability less than 1 (by staying in \( l_0 \) forever), the maximum value is \( \infty \). If \( \lambda \geq \frac{1}{10} \), the minimum value that we can achieve is \( \frac{1}{\lambda} \) by always returning to \( l_0 \) from \( l_2 \); otherwise, it is \( \frac{1}{2} \cdot (16 + \frac{1}{\lambda}) \).

### 3.2 Model Context

STA are related to many other automata models (cf. Figure 1). Of particular interest for this paper are stochastic hybrid automata (SHA) and probabilistic timed automata (PTA): The analysis technique we present is based on an existing one for SHA, and it involves the transformation of STA into PTA that are subsequently model checked using the digital clocks approach for PTA.

SHA [FHH+11] add continuous variables to STA. These can change over time according to differential (in)equations specified by the invariants. In contrast to clocks, they can also appear on the right-hand side of assignments, in particular in sampling expressions. SHA thus combine hybrid system behaviour (as in hybrid automata) with stochastic sampling and delays (as in STA).

PTA are the special case of STA where all clock constraints are integer and no continuous probability distributions are used. All delays and choices are thus based on discrete (usually finite-support) distributions, or nondeterministic. A number of techniques to model check PTA
exist [NPS13]. In this paper, we use the digital clocks approach because it supports both reachability probabilities and expected rewards: Clocks are replaced by (bounded) integer variables, and self-loop edges are added to increment them synchronously as long as the location invariant is satisfied. This turns the PTA into a (finite) Markov decision process (MDP) where reachability probabilities and expected rewards can be computed using standard techniques. The results are correct for the original PTA whenever all clock constraints are closed and diagonal-free.

4 Checking Reachability and Rewards

We use a combination of abstraction and probabilistic model checking to compute bounds on reachability probabilities and expected reward values for STA. This works as follows: First, the continuous distributions that occur in the STA are abstracted by a combination of discrete probabilistic choices and continuous nondeterminism. The result is a PTA. The digital clocks approach is used to convert that into a finite MDP. Standard techniques like value iteration can now be used to derive maximum/minimum reachability probabilities and expected rewards. The results are upper/lower bounds on the corresponding values in the original STA. This approach is a special case of a technique developed for SHA safety verification [FHH +11] and reward-based analysis [Hah13], which was (partly) implemented in the prohver tool [HHHK13]. By specialising for STA, we gain scalability, improve usability by requiring less user input and improving automation, and are able to compute useful lower bounds on minimum probabilities.

4.1 Abstracting Continuous Distributions

In the first step, the support of a continuous distribution is divided into a number of intervals and the probability of each interval is computed. The continuous sampling is then replaced by a probabilistic choice over the intervals with the computed probabilities, followed by a nondeterministic choice of which concrete value to pick from the chosen interval. When using prohver, the probabilities for the intervals had to be concrete real values due to the PHAVER backend used. In our new approach, we can map to PTA with probabilities that depend on state variables (but not on clocks or variables that were previously sampled). Since PTA allow only integer clock constraints, the choice of intervals is limited to those with integer bounds. We always overapproximate continuous distributions with intervals of unit width 1 aligned on integer bounds in the current implementation; all integer time points are anyway enumerated in the resulting MDP’s state space. For distributions with unbounded support, such as the exponential or normal distribution, we generate as many unit width intervals as needed to cover a probability mass of $1 - \rho$ and then add half-open intervals for the residual of the support. Instead of a set of intervals as with prohver, the only parameter of our approach therefore is this residual probability $\rho$. We use a default of $\rho = 0.05$ unless stated otherwise.

Example 3 For the STA of Example 1, we show the PTA overapproximation for the case that a single unit-width interval is sufficient to cover $1 - \rho$ probability in Figure 3. With $\rho = 0.05$, this is ensured provided $\lambda \geq 3$. We use $\geq 3$ and $\leq 3$ to denote interval comparisons. They are satisfied whenever there exists some value in the interval such that the concrete comparison is satisfied. This amounts to a comparison with the upper bound for $\leq 3$ and with the lower bound for $\geq 3$.
when the interval operand is on the right-hand side.

4.2 Correctness

We now show that, in the PTA that is constructed as described above, the maximum/minimum reachability probabilities and expected reward values are indeed upper/lower bounds for the corresponding values in the original STA. We first define the effect of abstraction more formally:

Definition 8 Consider an STA $M = \langle \text{Loc}, \text{Var}, A, E, l_{\text{init}}, \text{Inv} \rangle$ and a (potentially infinite) family of sets $\mathcal{A} = \{B_i\}_{i \in I}$. Each abstract state $B_i \subseteq \text{Loc} \times \text{Val}$ subsumes certain concrete states of $[M]$, we have $\bigcup_i B_i = \text{Loc} \times \text{Val}$ so that all states are covered. We require that an abstract state only subsume concrete states of the same location. Assume $B_{\text{init}} \ni \langle l_{\text{init}}, v \rangle$, and $B_i, B_j$ with $i \neq j$ disjoint. The abstraction TPTS is defined as $\text{abs}(M, \mathcal{A}) \equiv \langle \mathcal{A}, A \cup \mathbb{R}_0^+, \mu^\text{abs}_{M}, B_{\text{init}} \rangle$ where $\mu^\text{abs}_{M}$ is defined similar to Definition 5 with the jump rule being

$$I, v \xrightarrow{a, J} B_i$$

$$B_i \xrightarrow{a, \mu^\text{abs}_{M}} \forall j : B_j \rightarrow \mu^\text{abs}_{M}(B_j)$$

where $\mu^\text{abs}_{M}$ is as in Definition 5. We require $\mathcal{A}$ to be defined s.t. all induced $[\forall j : A_i \rightarrow \mu^\text{abs}_{M}(A_j)]$ have finite support. Timed transitions are defined accordingly. We assign rewards to abstract states according to the rate for its location and the rewards of the edges originating from there.

In the context of this paper, $\mathcal{A}$ is obtained by splitting the possible values sampling variables can take into unit width or half-open intervals. This construction ensures the finite-support requirement. For instance, for a single sampling variable $x$, all concrete states where $x$ is sampled to take values between 1 and 2 are subsumed by a single abstract state. For multiple sampling variables, abstract states are built from the cross product of intervals.

Lemma 1 For an STA $M$ with abstraction set $\mathcal{A}$ and some set of states $B$ the maximal (minimal) probability/reward value to reach $B$ in $\text{abs}(M, \mathcal{A})$ is not lower (not higher) than the maximal (minimal) probability/reward value in $[M]$.

Proof. We only consider disjoint abstract states. Non-disjoint ones (from overlapping intervals) would however not affect correctness, yet induce imprecision due to additional transitions in the abstraction. Let $M = \langle \text{Loc}, \text{Var}, A, E, l_{\text{init}}, \text{Inv} \rangle$ and $\mathcal{A} = \{B_i\}_{i \in I}$. We define the intermediate
abstraction $M’ \equiv \langle \text{Loc} \times \text{Val}, A \sqcup \mathbb{R}^+_0, T_{M’}, \langle \text{init}, 0 \rangle \rangle$ by replacing $\text{jump}$ of Definition 5 by

$$I \xrightarrow{g \circ \mathcal{W}} \langle l’ \rangle \quad \forall j \in J \quad (\forall j : s_j’ \rightarrow \mu_{\mathcal{W}}^j(B_j))$$

Let $f$ map paths from the intermediate abstraction to the semantics $[M]$, so for a path $\beta = s_0a_0[\forall j : s_j’ \rightarrow \mu_{\mathcal{W}}^j(B_j)]s_1a_1 \ldots$ we have $f(\beta) \equiv s_0a_0\mu_{\mathcal{W}}^0s_1a_1 \ldots$

For $\sigma \in \mathcal{S}[M]$ we construct $\sigma’ \in \mathcal{S}_{M’}$. Consider path $\beta$ with last($\beta$) = $(l, v)$. W.l.o.g. consider a subset $A = \{a\} \times A_{\text{dist}} \subseteq A \times \text{Prob}(S)$ of the possible successors when choosing edge $e = I \xrightarrow{s \circ \mathcal{W}} \mathcal{W} \in E$ with $(l, v) \xrightarrow{s \circ \mathcal{W}} \mu_{\mathcal{W}}$. Let $(S_i) \subseteq \mathcal{A}$ be the finite set of abstract states for which $\mu_{\mathcal{W}}(S_i) > 0$. Define $\mu_i \in \text{Prob}(S_i)$ as $\mu_i(A_i) \equiv \mu_{\mathcal{W}}(A_i) / \mu_{\mathcal{W}}(S_i)$ for measurable $A_i \subseteq S_i$ and denote their product measure by $\mu_{\text{prod}} \in \text{Prob}(\mathcal{X}, S)$. Define $U \equiv \{ [\forall i : s_i’ \rightarrow \mu_{\mathcal{W}}^i(S_i)] \mid \forall i : s_i’ \in S_i \}$, function $g([s_1’ \rightarrow p_1, \ldots, s_n’ \rightarrow p_n]) \equiv (s_1’, \ldots, s_n’)$, and $\mu(B) \equiv \mu_{\text{prod}}(g(B))$. Then we set $\sigma’(\beta)(A) \equiv \mu(A_{\text{dist}} \cap U)\sigma(f(\beta))(\{\text{edge e chosen}\})$. This way $\sigma’$ for $M’$ simulates the continuous distributions in $[M]$ s.t. measures on paths with $\sigma$ and $\sigma’$ agree [Hah13, Theorem 4.22]. This implies that reachability probabilities and reward values when using equivalent reward structures agree.

Because distributions in $M’$ and abs$(M, \mathcal{A})$ have finite support, one can define a finite automata simulation relation $[SL95]$ such that $(l, v) \preceq B_i$ if $(l, v) \in B_i$ from which one concludes that abs$(M, \mathcal{A})$ also bounds reachability probabilities of $M’$. Using extensions of simulation relations similar to e.g. [Hah13, Definition 7.26] one can also bound reward values in this way.

4.3 Digital Clocks and Scaling Time

We model-check the resulting PTA using the existing digital clocks approach [NPS13]. Let us illustrate this approach on our running example:

**Example 4** The digital clocks MDP for the PTA from the previous example is shown in Figure 4. The clock-incrementing self-loops are labelled $\varepsilon \in \mathcal{C}k$. We have excluded the non-stochastic part (locations $l_2$ and $l_4$) and merged the interval-valued variable $x$ into the locations to show the concrete comparisons on the edges of $l^t_1$ and $l^f_1$. We have also included the concrete probabilities for $\lambda \approx 3$. The maximum probability of reaching $l_4$ or $l_2$ in this MDP in at most $t \in \mathbb{N}$ time units is 0.475 for $t = 0$ and 0.5 for $1 \leq t \leq 7$. We know from Example 2 that the actual probability in the STA is $\frac{1}{2} \cdot (1 - e^{-\frac{\lambda}{2}}) < 0.5$. In our case of $\lambda \approx 3$, this is $0$ for $t = 0$, approx. 0.475 for $t = 1$ and very close to 0.5 for $t = 7$. The error is thus between 0.475 and almost 0 depending on $t$.

For reward $r$, the maximum value is $\infty$ even if we remove the $\varepsilon \in \mathcal{C}k$-edge from $l_0$: We can stay in $l^f_1$ forever due to the right-open interval created for the unbounded exponential distribution. The minimum value computed in this MDP is $0.475 \cdot 0 + 0.025 \cdot 1 = 0.025$, whereas the actual value for $\lambda \approx 3$ is $\approx \frac{1}{3}$.

The example shows that the error introduced by the abstraction of the continuous distributions depends on the variance of the distributions in relation to the interval width of at least 1 required to use PTA. In models where the dependence between time and property values is similarly direct as in this example, we can get more accurate results at the cost of larger MDP state spaces by scaling time: Both the results of the sampling and the non-interval values that clocks are compared to (including those in properties) are multiplied by some factor $d \in \mathbb{N}^+$. (For the
example, the former can be achieved by dividing the rate by $d$.)

**Example 5** By scaling time by a factor of $d = 2$ in our running example STA, two unit width intervals are used for $r = 0.05$ and $\lambda \approx 3$, with probabilities 0.388 and 0.087. The upper bound for the reachability probability drops to 0.388 for $t = 0$ and 0.475 for $t = 1$; the lower bound for the minimum expected reward rises to 0.137.

Although scaling time can lead to tighter bounds, there is another, independent cause of overapproximation error, which is due to the digital clocks requirement of closed clock constraints: All adjacent intervals have a singleton overlap, and we can only refer to exactly these overlapping values in clock constraints and properties. They have probability 0 in the STA, but not in the PTA, which leads to e.g. the upper bounds for time-bounded reachability probabilities being “one step ahead”. In Example 5, the upper bound computed for $t = 0$ is the actual probability for $t = 1$, the bound for $t = 1$ is the probability for $t = 2$, and so on.

## 5 Implementation

We have implemented our STA analysis approach in the new mcsta tool within the MODEST TOOLSET [HH14]. It relies neither on mcpta [HH09] nor on PRISM for PTA model checking.

It currently supports the continuous uniform, exponential and normal distributions as follows, where $x$ is a variable of type $\texttt{real}$ and sampling expressions may reference other state variables:

- $x := \texttt{Uni}(\text{lower, upper})$ for the uniform distribution, where lower resp. upper are expressions of type $\texttt{real}$ for which a concrete lower bound $lb$ resp. a concrete upper bound $ub \in \mathbb{R}$ can be determined with $lb \leq ub$. The intervals are then $\lfloor lb \rfloor, \lfloor lb \rfloor + 1, \ldots, \lfloor ub \rfloor - 1, \lfloor ub \rfloor$ with probability expressions constructed according to $\text{cdf}_{\text{Uni}}(x) = (x - \text{lower})/(\text{upper} - \text{lower})$.

- $x := \texttt{offset + EXP(rate)}$ for the exponential distribution, where offset is an expression of type $\texttt{int}$ and rate is an expression of type $\texttt{real}$ for which a concrete lower bound $\lambda \in \mathbb{R}^+$ can be determined. The intervals are then $\lfloor \text{offset}, \text{offset} + 1 \rfloor, \ldots, \lfloor \text{offset} + n - 1, \text{offset} + n \rfloor$ and $\lfloor \text{offset} + n, \infty \rfloor$ where $n = \lceil -\ln p \rceil / \lambda$ (using the quantile function of the exponential distribution). The probability expressions of the intervals are constructed according to $\text{cdf}_{\text{EXP}}(x) = 1 - e^{-\text{rate} \cdot x}$.

- $x := \texttt{NORM}(m, \sigma)$ for the normal distribution, where the mean $m$ is an expression of type $\texttt{int}$ and the standard deviation $\sigma$ is a concrete value in $\mathbb{R}^+$. The intervals are $(-\infty, m - n], \ldots, [m - 1, m], [m, m + 1], \ldots, [m + n, \infty)$. Since neither the quantile function nor the cdf of the normal

\[ P(x < c | \sigma) = \Phi \left( \frac{c - m}{\sigma} \right) \]

\[ P(x \geq c | \sigma) = 1 - \Phi \left( \frac{c - m}{\sigma} \right) \]

Figure 4: Digital clocks MDP of the PTA abstraction (explicit intervals)
distribution have a closed-form solution, we require $\sigma$ to be a concrete value to precompute $n$ and the actual interval probabilities based on $\sigma$ and $\rho$ close to double precision. These examples show a general recipe to support other continuous distributions using their quantile function and cdf. In case a distribution is parameterised by an expression that contains state variables, we may generate more intervals than necessary for some valuations, which then have zero probability. For example, we generate two intervals for $x = U\{0,2i\}$ when $i$ has domain $\{0,1\}$ since the upper bound of expression $2i$ is 2. However, since the probabilities are preserved as expressions, the probability of $[1,2]$ will evaluate to 0 for all states where $i \neq 2$.

6 Evaluation

We have applied mcsta to four different examples. We are interested in how close the computed bounds are to the actual values (effectiveness), and how large the state spaces of the underlying MDP become\(^1\) (efficiency). All measurements were performed on the same 1.7 GHz Intel Core i5-3317U system with 4 GB of RAM running 64-bit Windows 8.1. The first two models we present are deterministic. As mentioned, our method is not targeted for this special case, so we expect correct and useful, but not very tight, computed bounds. Specialised methods will perform better or be more precise in these cases. The last two models, however, contain continuous and discrete nondeterminism, so our technique is currently the only one available for verification.

6.1 M/G/1 Queueing System with Normal Distribution

Our first example models an M/G/1/6 queueing system as STA where the service time is normally distributed with mean 10 and standard deviation 2. Since clocks cannot be negative, it is implicitly truncated to values $\geq 0$ when we compare the result to a clock. The time between customer arrivals is exponentially distributed with rate $\frac{1}{6}$. The queue has length 5, not counting the customer being served, and is initially empty. We are interested in the following values:
– the probability $p$ that the queue is full and $\leq t_p$ time units have elapsed,
– the expected time $t$ until the queue is full for the first time, and
– the expected number $c$ of customers served before the queue becomes full.
Since nondeterminism is absent by construction, we can use statistical model checking with the modes simulator from the MODEST TOOLSET to obtain good approximations of $p$, $t$ and $c$.

\(^1\) Memory was the limiting factor in all examples; runtime was always below 3 minutes.
Reachability and Reward Checking for Stochastic Timed Automata

The results of computing upper and lower bounds on $p$ using mcsta are shown in Figure 5. On the left, we show the computed bounds for different values of $t_p$ as black triangles. We see that there is a noticeable approximation error, but the general evolution of the probability over time is preserved. After $t_p \approx 80$, the lower bound shows no significant improvements. For $t_p \geq 90$, we ran out of memory, so we increased the residual probability parameter $\rho$ to 0.1. The number of concrete states in the MDP of the digital clocks semantics is shown on the right of Figure 5. We see that it increases linearly with $t_p$ and can be reduced significantly by increasing $\rho$, i.e. by lowering the number of intervals for the abstraction of the exponential and normal distributions.

Asking for minimum expected rewards, we compute bounds $t \geq 43.4$ and $c \geq 3.52$ for the other two values. As we do not need a global clock to check a time bound like $t_p$ here, the underlying MDP has just 136767 states. State-space exploration and computation of both bounds takes only 2.3 s in total. If we ask for maximum expected rewards, we get bounds $\infty$ due to the right-open intervals created by the abstraction of the unbounded distributions (cf. Example 4). Simulation with modes tells us that $t \approx 61$ and $c \approx 6.2$ for this deterministic model.

6.2 Tandem Queueing Network

We next look at a model from the PRISM benchmark suite [KNP12]: the tandem queueing network of an M/Cox$_2$/1/4 followed by an M/M/1/4 queue [HMS99]. It is a CTMC and we can thus model it as an STA without nondeterminism. We experiment with scaling time as described in Section 4.3. We compute the maximum probability $p_{ff}$ of the first queue being full in time $t$, trying to use a value of $\rho \geq 0.05$ as low as possible and a time scaling factor as high as possible without running out of memory. The result is shown on the left of Figure 6.

The second property we look at is the maximum probability $p_{af}$ of both queues becoming full within time $t$. This happens at a vastly different time scale: $p_{af}$ only starts to approach 0.5 when $t$ is on the order of 50. We thus focus on the effect of scaling time on the approximation error for fixed time bound $t = 2$. The results are shown on the right of Figure 6. We see that the error can be significantly reduced by scaling up time.

Finally, we compute bounds on the expected times $t_{ff}$ until the first queue becomes full and $t_{af}$ until both are full. As we increase the time scaling, we go from lower bounds $t_{ff} \geq 0.000012$ and $t_{af} \geq 0.56$ for time scale $d = 1$ with 9557 MDP states, computed in 0.1 s, to $t_{ff} \geq 0.108710$ and $t_{af} \geq 5.87$ for $d = 10$ with 3662958 states, computed in 108 s. Again, upper bounds (i.e. maximum expected rewards) are all $\infty$. From simulation, we get $t_{ff} \approx 0.29$ and $t_{af} \approx 17.9$.
Table 1: Results and comparison for the WLAN example

<table>
<thead>
<tr>
<th>model</th>
<th>type</th>
<th>( P_{\text{max}} )</th>
<th>([E_{\text{min}}^{\land}, E_{\text{max}}^{\land}] )</th>
<th>([E_{\text{min}}^{\lor}, E_{\text{max}}^{\lor}] )</th>
<th>([E_{\text{min}}^{1}, E_{\text{max}}^{1}] )</th>
<th>states</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>wlan</td>
<td>PTA</td>
<td>0.18359</td>
<td>[1325, 6280] ( \mu s )</td>
<td>[450, 4206] ( \mu s )</td>
<td>[450, 5586] ( \mu s )</td>
<td>104804</td>
<td>8 s</td>
</tr>
<tr>
<td>wlan-uni</td>
<td>STA</td>
<td>0.13659</td>
<td>[2325, 4607] ( \mu s )</td>
<td>[950, 3018] ( \mu s )</td>
<td>[950, 3880] ( \mu s )</td>
<td>264240</td>
<td>15 s</td>
</tr>
</tbody>
</table>

### 6.3 Wireless LAN with Uniform Transmission Time

Departing from queueing systems, we now look at the model of a communication protocol: the carrier-sense multiple-access with collision avoidance (CSMA/CA) part of IEEE 802.11 WLAN. We take the MODEST PTA model [HH09] and replace the nondeterministic choice of transmission delay out of \([200, 1250] \mu s\) (with a unit of time representing 50 \( \mu s\)) by a uniformly distributed choice over the same interval. The result is still nondeterministic, and an STA instead of a PTA.

Model-checking results for the original PTA (“wlan”) and the new STA (“wlan-uni”) are shown in Table 1. We see that the state space of the underlying MDP is larger when the uniform distribution is used. This is because the states not only contain explicit values for all clocks as in the original PTA, but additionally 21 different concrete intervals that overapproximate the result of sampling from \(\text{Unif}(4, 25)\). The blowup thus stays far below the worst-case factor of 21.

We analyse six time-unbounded properties: \( P_{\text{max}} \), the maximum probability that either of the two modelled senders’ backoff counters reaches the upper bound of 2, as well as \( E_{\text{min}}^{\land}/E_{\text{max}}^{\land}, E_{\text{min}}^{\lor}/E_{\text{max}}^{\lor} \), and \( E_{\text{min}}^{1}/E_{\text{max}}^{1} \), the minimum/maximum expected times until both senders, either of them, or the one with id 1 correctly deliver their packets. Due to the nondeterminism, we cannot use simulation or any other technology to compute the actual values. However, the computed bounds are plausible if we assume that in the PTA, the longest/shortest transmission delay maximises/minimises the values. The STA is thus indeed expected to show less extremal behaviour.

### 6.4 File Server

As a final example, we analyse another model that combines all essential features of STA and cannot be model checked with any other approach we know of (except prohver). It represents a single-threaded file server with slow archival storage:

- Requests arrive to a single queue of length \( C = 5 \) with interarrival times following \( \text{Exp}(1/8) \).
- File sizes are uniformly distributed over some range such that sending the file back to a client takes time uniformly distributed over \([1, 3] \).
- 2% of all files are in slow archival storage. Retrieving a file is instantaneous for normal storage, but takes between 30 and 40 time units nondeterministically for archival storage.

We thus have continuous stochastic delays, a probabilistic choice and nondeterministic delays. Additionally, we model the initial queue length as uniformly distributed in \(\{0, \ldots, \lfloor C/2 \rfloor\} \). The model is part of the MODEST TOOLSET download.

We are interested in the probability \( p \) that the request queue becomes full within time \( t_p \), and the minimum (i.e. worst-case) expected time \( t \) until this happens. For \( t \), we obtain a lower bound of 462 time units from an MDP with 107742 states in 6s. For \( p \), the results are shown in Figure 7. On the right, we see that the number of MDP states again grows linearly with the time bound.
On the left, we have plotted the computed upper/lower bounds using small triangles. Due to the nondeterministic delay, we cannot use simulation. However, we can instruct modes to resolve that delay by scheduling events either as soon or as late as possible (ASAP/ALAP). Simulating these deterministic variants of the model gives us $t \approx 1012$ for ASAP and $t \approx 721$ for ALAP. For $p$, the simulation results are included on the right of Figure 7. The results that we get via our new approach are clearly useful: They are safe bounds whereas we do not know anything about the relationship between simulation results and the actual values.

7 Conclusion

We presented the first fully-automated model checking approach for STA with general, unbounded distributions and support for nondeterminism. It provides upper bounds for maximum and lower bounds for minimum reachability probabilities and expected rewards. We investigated causes of approximation error and showed that scaling time can effectively reduce the error. In experiments performed with our implementation, mcsta, we saw that the approach works well in practice, but state-space explosion is a significant problem for time-bounded properties.

References


On the Random Structure of Behavioural Transition Systems

Jan Friso Groote\textsuperscript{1}, Remco van der Hofstad\textsuperscript{2}, and Matthias Raffelsieper\textsuperscript{3}

\textsuperscript{1}J.F.Groote@tue.nl, http://www.win.tue.nl/~jfg
\textsuperscript{2}R.W.v.d.Hofstad@tue.nl, http://www.win.tue.nl/~rhofstad

Department of Mathematics and Computer Science, Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, The Netherlands

\textsuperscript{3}raffelsm@ethz.ch
D-MTEC, ETH Zürich, Weinbergstrasse 56/58, 8092 Zürich, Switzerland

Abstract: Random graphs have the property that they are very predictable. Even by exploring a small part reliable observations are possible regarding their structure and size. An unfortunate observation is that standard models for random graphs, such as the Erdős-Rényi model, do not reflect the structure of the graphs that describe distributed systems and protocols. In this paper we propose to use the parallel composition of such random graphs to model ‘real’ state spaces. We show how we can use this structure to predict the size of state spaces, and we can use it to explain that software bugs are in practice far easier to find than predicted by the standard random graph models. By some practical experiments we show that our new random model is an improvement over the standard model in predicting properties of transition systems representing realistic systems.

Keywords: Random graph, \(P\)-parallel random transition system, state space size

1 Introduction

Modelling the behaviour of systems is gaining popularity. An unpleasant side effect is that the transition systems of models of realistic systems easily become very large. We ran into such an example while modelling an UART (universal asynchronous receiver/transmitter) for a company called NXP. Using highway search \cite{4}, a parallel simulation technique far more efficient than random simulation in finding problematic situations, we did \textit{not} find a suspected error. The question that we needed to answer was how large the probability was that the error really did not occur. A typical derived question that immediately jumps to mind is to estimate the size of the state space.

In order to answer such questions, one can resort to random graphs \cite{2}. The Erdős-Rényi model is a commonly used model. It has a set \(S\) of \(N\) states (nodes, vertices) and a set of transitions \(\rightarrow\). There are two highly similar variants, one where each conceivable edge is present with some probability \(p\), and one where \(M\) transitions are chosen out of the \(N^2\) possibilities.

Erdős-Rényi random graphs are a little counterintuitive if it comes to modelling transition systems which represent behaviour. Transition systems have an initial state and this initial state has outgoing transitions to states that in general also have outgoing transitions. In the Erdős-Rényi random graph the initial state may not have outgoing transitions (actually with a fairly
high probability $e^{-\lambda}$ where $\lambda$ is the fan-out, i.e., the expected number of transitions leaving a state). Therefore, we choose a slightly different model, where each state has a fixed number $\lambda$ of outgoing transitions each of which goes to randomly selected states of the transition system. All choices are made independently of each other.

Given this model of a random transition system we estimate the size of a transition system by a random walk through the graph. By random simulation we have evidence that these estimates are very good. However, by applying this technique to realistic models (e.g., Firewire P1394 protocol [9]) it becomes obvious that the structure of these random graphs is not really a reflection of a ‘real state space’.

As an alternative model for the structure of realistic systems, we propose to use the Cartesian product of $P$ parallel random transition systems, reflecting that a realistic system often consists of $P$ more or less independent components. One could not only think of the components as independent parallel processes, but one can also consider the behaviour of subtasks or even variables as potential parallel components.

We develop techniques to estimate the sizes and fanout of the different components. Again, using random simulation we verified that these estimation techniques are correct and precise. More importantly, we estimate the sizes of ‘realistic state spaces’ and find that these are far better than those we obtain using the ‘single threaded’ random model. There are also some disadvantages, in particular, the predictions are less stable and the calculational effort for the estimates is higher.

Our experiments provide evidence that $P$-parallel random transition systems could be a good representation of ‘realistic’ state spaces. Of course, the state spaces of real applications do not have a random structure. But having a random model which reflects ‘real’ state spaces reasonably well and which is sufficiently simple to allow mathematical analysis is really a great asset, because it enables the use of the power of random analysis to substantially increase our generic insight in the behaviour of real systems.

As an illustration of the potential power of the $P$-parallel model we apply it to the question how effective testing is. In our experience it is remarkably easy to detect a known error by running a random test. According to the single threaded random model this is not possible. The probability of hitting an erroneous state by a random walk is far too small. However, if the error occurs in one of the states of one of the $P$-parallel components, it is far easier to find. Even stronger, if we know the sizes of the different components, we can come up with small numbers of required test runs to guarantee with a given confidence that realistic systems are error free.

Acknowledgments. Thanks go to Jesse Goodman, Wil Kortsmidt and the referees for remarks, support and discussion. This research was in part supported by NWO and Agentschap NL.

Related Work.

As far as we know, there is not much work on the random structure of transition systems representing behaviour. The following is what we are aware of. Estimating the size of a Petri Net’s state space has been investigated in [11]. That work makes explicit use of the structure of a Petri Net and is only applicable when the Petri Net is constructed from a set of supported building blocks.
A more general approach was presented in [14] where, as in our work, a state space is seen as a directed graph. However, the authors do not compute an estimate, but instead only classify state spaces into one of three classes: small models, large models, and models that are too large and hence out of reach. In this work classification trees, neural networks and techniques similar to the Lincoln Index [13] are employed.

Inspired by [14], the authors of [3] present a method to compute the estimated state space size. There, the observed measure is the size of the breadth-first frontier that is still to be explored in relation to the number of states that have already been explored. By visual inspection, the authors determine that this curve should be approximated with a quadratic function and use least-squares fitting to compute the parameters and thereby an estimate for the state-space size.

2 Random State Spaces

In this section we define the basic notions that we employ. We use directed graphs or transition systems without labels, as we do not need the labels in our exposition.

A state space is seen as a graph $G = (S, \to)$, with $S$ being an arbitrary set of states (nodes, vertices) and $\to \subseteq S \times S$ being a multi-set of transitions (edges). If $(s, s') \in \to$ holds, we generally denote this by $s \to s'$. For the edges in the set $\to$ we assume that every state has a fixed degree of outgoing edges, i.e., there exists a fixed $\lambda \in \mathbb{N}$ such that $|\{s' \mid s \to s'\}| = \lambda$ for all $s \in S$ (where $\{s' \mid s \to s'\}$ is a multi-set) and $|E|$ denotes the size of a multi-set $E$. One can consider transition systems with a variable fan-out, but this will make the random model more complex, and therefore harder to use, and less predictive. If $s \to s'$, then $s$ is called the source and $s'$ the target state of that edge. In a random state space it is assumed that for every such edge, given its source state $s$, every other state $s'$ is equally likely to be the target state. Furthermore, we define $N = |S|$ and $M = |\to|$ to denote the number of states and transitions, respectively.

A tuple $T = (G, s_0)$ is called a random transition system, where $G = (S, \to)$ is a random state space as described above and $s_0 \in S$ is an arbitrary, randomly chosen initial state. For such a random transition system, only the part reachable from the initial state is of interest, i.e., those states $s' \in S$ for which $s_0 \xrightarrow{}^* s'$ holds (where $\xrightarrow{}^*$ denotes the reflexive transitive closure of $\to$). Note that the number of reachable states is at most $N$.

This paper considers state spaces being the graph product of two or more random transition systems. Since taking the graph product is associative, we only consider the case of two random transition systems, which can then be repeated for more components. Thus, a product transition system $T_{1 \times 2} = (G, s_0)$ with graph $G = (S, \to)$ and initial state $s_0 \in S$ is assumed to be composed from two random transition systems $T_1 = (G_1, s_1, 0)$ and $T_2 = (G_2, s_2, 0)$, with $G_1 = (S_1, \to_1)$, $G_2 = (S_2, \to_2)$, such that $S = S_1 \times S_2$, $s_0 = (s_1, 0, s_2, 0)$, and $(s_1, s_2) \to (s'_1, s'_2)$ iff either $s_1 \to_1 s'_1$ and $s_2 = s'_2$, or $s_1 = s'_1$ and $s_2 \to_2 s'_2$.

Note that it is assumed that the states in the product transition system $T_{1 \times 2}$ are opaque, i.e., from a state $s = (s_1, s_2) \in S$ the individual components $s_1$ and $s_2$ of the state cannot be recovered. Note also that we do not consider ‘synchronisation’, i.e., a transition in the product transition system which consists of the simultaneous occurrence of transitions in the constituent transition systems. Our random approximations of realistic state spaces do not contain such synchronised transitions.
3 Estimation Based On Duplicates

In this section, a technique is described to estimate the size of a single random transition system \( T = (G, s_0) \) with graph \( G = (S, \rightarrow) \). For this purpose, the process of exploring the state space is analysed. Starting from the initial state \( s_0 \), the edges that have their source state in the already explored part of the state space are iteratively explored. Thus, a target state of an edge that is being explored can either be a state that has already been explored previously, or it is a state that is new, i.e., has not been seen previously. Note that the method in this section does not rely on a particular state space exploration strategy, contrary to approach in the next section which requires breadth-first search.

3.1 A stochastic model

We develop a stochastic model where we introduce random variables \( Y_k \), which represent the number of unique states seen after exploring \( k \) transitions. Then, the probability distribution of the state space size \( N \) after exploring \( m \) transitions is

\[
P[N = n \mid \bigwedge_{k=0}^{m} Y_k = i_k],
\]

where \( i_k \) is the observed number of unique states seen after exploring \( k \) transitions. Applying Bayes’ law, this probability can be rewritten as follows:

\[
P[N = n \mid \bigwedge_{k=0}^{m} Y_k = i_k] = \frac{P[N = n \land \bigwedge_{k=0}^{m} Y_k = i_k]}{P[\bigwedge_{k=0}^{m} Y_k = i_k]} = \frac{P[\bigwedge_{k=0}^{m} Y_k = i_k \mid N = n]P[N = n]}{\sum_{r=n}^{\infty} P[\bigwedge_{k=0}^{m} Y_k = i_k \mid N = r]P[N = r]} \tag{1}
\]

In the above equation (1), the probability \( P[N = n] \) is the so-called \textit{a-priori} probability for the size of the state space. To characterise the probabilities \( P[\bigwedge_{k=0}^{m} Y_k = i_k \mid N = n] \), we use the following recursive identity:

\[
P[m \mid \bigwedge_{k=0}^{m} Y_k = i_k \mid N = n] = P[Y_m = i_m \mid \bigwedge_{k=0}^{m-1} Y_k = i_k \land N = n]P[m \mid \bigwedge_{k=0}^{m-1} Y_k = i_k \mid N = n] \tag{2}
\]

Thus, we need to analyse the first factor at the right-hand side of equation (2) further. In case \( m = 0 \), then we find that

\[
P[0 \mid Y_k = i_k \mid N = n] = \begin{cases} 1 & \text{if } i_0 = 1 \\ 0 & \text{otherwise} \end{cases}
\]

since only the initial state is seen in the beginning.

For \( m > 0 \), we make a case distinction based on whether the target state of the newly explored edge has been seen before or not. In case \( i_m = i_{m-1} \), i.e., the target state has already been seen
before, one of the \(i_{m-1}\) already seen states has been chosen:

\[
P[Y_m = i_m \mid \bigwedge_{k=0}^{m-1} Y_k = i_k \land N = n] = \frac{i_{m-1}}{n} = \frac{i_m}{n} \tag{3}
\]

Otherwise, if the target state of the newly explored edge has not been seen before, then \(i_m = i_{m-1} + 1\) holds and we have chosen one of the \(n - i_{m-1}\) states we have not yet seen, giving the following equality:

\[
P[Y_m = i_m \mid \bigwedge_{k=0}^{m-1} Y_k = i_k \land N = n] = \frac{n - i_{m-1}}{n} = \frac{n - i_m + 1}{n} \tag{4}
\]

Next, we solve the recurrence in equation (2) using equations (3) and (4). To do so, we introduce variables \(q_j^{(m)}\) for \(1 \leq j \leq i_m\) to represent the multiplicity of number \(j\) in the sequence \(i_0, i_1, \ldots, i_m\) of length \(m + 1\). For example, in the sequence 1, 1, 2, 3, 4 it holds that \(q_1^{(5)} = 2\), \(q_2^{(5)} = 1\), \(q_3^{(5)} = 2\), and \(q_4^{(5)} = 1\). It should be noted that \(\sum_{j=1}^{i_m} q_j^{(m)} = m + 1\) always holds. Using these variables, the right hand side of equation (2) becomes

\[
\prod_{j=1}^{i_m} (n - j + 1) \prod_{j=1}^{i_m} q_j^{(m)} - 1 \quad \frac{1}{n^{m+1}}. \tag{5}
\]

Substituting (5) for (2) in (1) gives the following result, where remarkably enough the variables \(q_j^{(m)}\) disappear. Apparently, only the information about the number of unique states (the \(i_k\)) and the total number of states explored (which is \(m + 1\), i.e., the number of target states of explored edges plus the initial state) is required:

\[
P[N = n \mid \bigwedge_{k=0}^{m} Y_k = i_k] = \frac{\left(\prod_{j=1}^{i_m} (n - j + 1)\right) P[N = n]/n^{m+1}}{\sum_{r = i_m}^{\infty} \left(\prod_{j=1}^{i_m} (r - j + 1)\right) P[N = r]/r^{m+1}} \tag{6}
\]

We used that the maximally observed number of unique states \(i_m\) is a lower bound for the total size \(N\). Often, an upper bound \(\pi\) on the total number of states can be obtained. We assume that \(N\) is a priori uniformly distributed in the interval \([i_m, \pi]\). In this case, equation (6) reduces to

\[
P[N = n \mid \bigwedge_{k=0}^{m} Y_k = i_k] = \frac{\left(\prod_{j=1}^{i_m} (n - j + 1)\right) /n^{m+1}}{\sum_{r = i_m}^{\pi} \left(\prod_{j=1}^{i_m} (r - j + 1)\right) /r^{m+1}}, \tag{7}
\]

which gives the following expected size of the state space:

\[
\mathbb{E}[N \mid \bigwedge_{k=0}^{m} Y_k = i_k] = \frac{\sum_{r = i_m}^{\pi} n \left(\prod_{j=1}^{i_m} (n - j + 1)\right) /n^{m+1}}{\sum_{r = i_m}^{\pi} \left(\prod_{j=1}^{i_m} (r - j + 1)\right) /r^{m+1}} \tag{8}
\]
3.2 Simulation experiments

We want to establish the effectiveness of the estimation procedure for the monolithic model by predicting the sizes of a randomly generated state spaces and comparing the results with the actual sizes used to generate them.

For the experiments with the estimation procedure presented in the previous section, we explored the example graphs in a breadth-first fashion, as this is the commonly used strategy of many model checking tools, an example being the tool LPS2LTS contained in the mCRL2 toolset [5, 8]. When used in verbose mode, the tool will output the currently explored number of states and transitions, which therefore allows to apply equation (8) to estimate the total size of the state space.

We present here the results on a random transition system with 10,000 states, each with 2 outgoing transitions. For other results see [7]. After randomly designating an initial state, 8,011 states were reachable. To compute the estimates, we used \( \bar{\pi} = 1,000,000 \) as upper bound on the size of the state space.

The estimation results are shown in figure 1. The x-axis shows the number of unique states seen at a certain point, i.e., the maximal \( i_m \) used in that computation. On the y-axis, the result of evaluating equation (8) is depicted. The values of the maximal \( i_m \) were increased in steps of 10 and two adjacent points were connected by a straight line segment. It can be observed that the estimation yields results close to the actual value of 10,000 after having observed a few hundred
Table 1: Estimates for the number of states of the firewire protocol. The actual number of reachable states is 188569. The estimates for $N$ under the assumption that the state space has a random structure are not very accurate.

unique states. After that, it slightly overshoots, but always stays below 11,500 estimated states. From this we conclude that our estimates are quite accurate.

3.3 Application to the firewire protocol

We use a description of a real time bus access in the firewire or P1394 protocol provided in [9] to observe what happens when we predict the number of states for a realistic protocol assuming that it is randomly generated. The protocol consists of two protocol entities that alternatingly obtain access to a data bus resolving contention conflicts on the way.

The typical values for $i_m$ and $m$ are given in table 1. The estimates for the expected number of states $N$ and number of reachable states $N_{\text{reachable}}$ are also provided. The actual number of reachable states is 188569 and there are 340608 transitions. There is an average fan-out of 1.8.

In table 1 we observe that the estimates for $N_{\text{reachable}}$ while traversing the state space structurally underestimate the actual number of reachable states and they only approach the actual number of states when all states have been traversed. This is quite different from what we observe in figure 1. But this pattern is very similar to what we see in other state spaces representing real systems. From this we conclude that our random model does not represent real systems sufficiently well.

4 Estimates of the Sizes of Product Transition Systems

In order to have a random model that approximates real state spaces better, we study transition systems that are the product of two or more random transition systems in this section. Since we assume that in a product transition system the identity of a component state cannot be recovered, there is no way to obtain the original graphs. If we were able to recover the constituent graphs, we could have applied the technique of section 3 to the components and easily derive a prediction for the size of the whole transition system.

Our estimation technique is completely different from that of section 3. We require that the exploration of a state space is performed in a breadth-first fashion, i.e., first all states at a certain distance from the initial state are considered, before dealing with those at higher distances. This allows to consider layers of a transition system $T = (G, s_0)$ with $G = (S, \rightarrow)$. We define the layer $\partial B_T(j)$ at some distance $j \in \mathbb{N}$ by

$$\partial B_T(j) = \{s \in S \mid s_0 \xrightarrow{j} s \land \forall k < j : s \notin \partial B_T(k)\}$$
where \( s_0 \rightarrow^j s \) means that state \( s \) is reachable from state \( s_0 \) in exactly \( j \) steps. We also call \( \partial B_T(j) \) the boundary ball on intrinsic distance \( j \). Note that by the above definition, every state \( s \) reachable from the initial state \( s_0 \) is contained in exactly one layer, namely that with the minimal distance \( j \).

The set of those states that have been seen up to some distance \( j \in \mathbb{N} \) is defined as \( B_T(j) = \bigcup_{i=0}^j \partial B_T(i) \) and is called the ball of radius \( j \). To obtain the layer \( \partial B_T(j) \), only edges from states from the layer \( \partial B_T(j-1) \) have to be considered, since otherwise, if there was an edge from a state \( s \in \partial B_T(j-k) \) with \( k > 1 \) to a state \( s' \in \partial B_T(j) \), then this would imply \( s' \in \partial B_T(j-k+1) \), which would contradict \( s' \in \partial B_T(j) \).

### 4.1 Estimating the size of a single component

We first concern ourselves with the estimation of the size of a single random transition system based on balls and layers. In section 4.2 we use this result to estimate the sizes of the transition systems in parallel products of transition systems. Breadth-first generation of a state space explores edges having their source state in the current layer and adds the target states to the surrounding ball. Thus, to obtain a layer at distance \( j+1 \), a total of \( \lambda |\partial B_T(j)| \) edges are explored where \( \lambda \) is the fan out of each state.

We are interested in the expected size of the reachable state space, which is the same as the size of the ball with maximal radius. Thus, the expected size of the balls should be investigated.

**Definition 1** We define \( G(j) = \mathbb{E}(|B_T(j)|) \) to be the expected size of the breadth-first graph with states at a distance \( \leq j \) from the initial state and \( R(j) = \mathbb{E}(|\partial B_T(j)|) = G(j) - G(j-1) \), where \( G(-1) = 0 \).

We use the convention that the subscript of \( B_T \) carries over to \( G \) and \( R \). For example, we write \( G_{1 \times 2}(j) \) for \( \mathbb{E}(|B_{1 \times 2}(j)|) \).

To analyse the function \( G(j) \), we introduce random variables \( X_i \) that denote the size of the partial state space after exploring \( i \) edges. Let \( n_i \) for \( i = 1, 2, \ldots \) denote the number of unique states observed after exploring \( i \) edges. Thus, \( n_0 = 1 \) and for \( n_{i+1} \) we have either \( n_{i+1} = n_i \) in case the target state of the additionally explored edge was already in the explored part of the state space, or \( n_{i+1} = n_i + 1 \) if the target state of the edge is new. Since the target state of an edge is picked uniformly at random, the probability of the first case \( (n_{i+1} = n_i) \) to occur is

\[
\mathbb{P}(X_{i+1} = n_{i+1} \mid \bigwedge_{j=0}^i X_j = n_j) = \frac{n_i}{N} = \frac{n_{i+1}}{N},
\]

which amounts to the probability to pick one of the \( n_i = n_{i+1} \) states that were already explored from the total \( N \) states. In the second case, where \( n_{i+1} = n_i + 1 \), the probability is

\[
\mathbb{P}(X_{i+1} = n_{i+1} \mid \bigwedge_{j=0}^i X_j = n_j) = 1 - \frac{n_i}{N} = \frac{N - n_{i+1} + 1}{N},
\]

where a state is picked that is not among the \( n_i = n_{i+1} - 1 \) already explored states.
The expected value of these random variables can be computed as:

\[
E[X_{i+1}] = \sum_{k=0}^{\infty} k P(X_{i+1} = k)
\]

\[
= \sum_{k=0}^{\infty} k \frac{k}{N} P(X_i = k) + \sum_{k=0}^{\infty} \frac{k - 1}{N} P(X_i = k - 1)
\]

\[
= \sum_{k=0}^{\infty} k \frac{k}{N} P(X_i = k) + \sum_{k=0}^{\infty} (k + 1) \frac{N - k}{N} P(X_i = k)
\]

\[
= \sum_{k=0}^{\infty} k \left(1 - \frac{1}{N}\right) P(X_i = k) + \sum_{k=0}^{\infty} P(X_i = k)
\]

\[
= \sum_{k=0}^{\infty} k \left(1 - \frac{1}{N}\right) P(X_i = k) + 1
\]

\[
= \frac{N - 1}{N} E[X_i] + 1.
\]

In equation (9) we used the above observations that either an already explored state is reached or a new state was explored. Solving the recurrence equation (10) using the boundary condition that \(E[X_0] = 1\), gives a closed formula:

\[
E[X_i] = N \left(1 - \left(\frac{N - 1}{N}\right)^{1+i}\right).
\]

This equation can also be used to predict \(N\) in table 1 with almost equal estimates for \(N\).

Another use of equation (11) is to derive the number of reachable states \(s\). All reachable states have been explored if there are no other states that have been visited. Then there are \(\lambda s\) visits, as for each explored state each outgoing transition is investigated. So, equation (11) becomes

\[
s = N \left(1 - \left(\frac{N - 1}{N}\right)^{1+\lambda s}\right).
\]

From this \(s\) can easily be solved, although care is required as there is also a small negative solution for \(s\).

From the observation that \(\lambda |B_T(j)|\) edges are explored to obtain \(B_T(j + 1)\), together with equation (11) the following recursive formula for the function \(G(j)\) can be derived, namely the expected size of the ball with radius \(j\) for a random transition system, where \(N\) and \(\lambda\) denote the total size and the fan-out of each state, respectively:

\[
G(j + 1) = E[X_{i,B_T(j)}] = N \left(1 - \left(\frac{N - 1}{N}\right)^{1+\lambda G(j)}\right).
\]

Equation (13) gives a means to compute the size \(N\) given two consecutive balls, assuming that we have a way to estimate \(\lambda\) as the average fan-out in each state. With more balls available, \(N\), and even \(\lambda\), can be estimated using the least square method.
4.2 Product Graph Size Estimation

As the estimates of the size of the state spaces are way off if we assume that they are homogenous random state spaces, we are interested in viewing a state space as the product of two random state spaces. We assume that the two components cannot be distinguished in the product graph and therefore we need to formulate a relation between the observable ball and layer sizes of the product graph and the component sizes. For this purpose, we note that due to the definition of the graph product, which only performs a transition in one of the components, a state in the layer of depth \( j \) can be reached by either only performing \( j \) steps in the first component, or \( j - 1 \) steps in the first component and one step in the second component, etc. Furthermore, steps from different components are independent of each other, i.e., if \((s_1, s_2) \rightarrow (s_1', s_2') \rightarrow (s_1'', s_2'')\), then also \((s_1, s_2) \rightarrow (s_1', s_2') \rightarrow (s_1'', s_2'')\), where \( \rightarrow_i \) is a step done by component \( i \). Hence, we can re-order the steps such that first all steps in the first component are performed, and then all steps in the second component. Thus, in the product graph, the following equation holds:

\[
|\partial B_{T_1 \times T_2}(j)| = \sum_{k=0}^{j} |\partial B_{T_1}(k)| \cdot |\partial B_{T_2}(j-k)|. \tag{14}
\]

From equation (14) for the expected value \( R_{1 \times 2}(j) \) of the product graph we derive:

\[
R_{1 \times 2}(j) = \sum_{k=0}^{j} R_1(k) \cdot R_2(j-k). \tag{15}
\]

In the following, the expected sizes of the component layers, \( R_1 \) and \( R_2 \), are considered in more detail. Let \( i \in \{1, 2\} \) and recall that \( R_i(j+1) = G_i(j+1) - G_i(j) \). Using equation (13), we can also obtain a recursive formula for the component layer sizes:

\[
R_i(j + 1) = G_i(j + 1) - G_i(j) = N_i \left( \left( \frac{N_i - 1}{N_i} \right)^{1+\lambda_i G_i(j-1)} - \left( \frac{N_i - 1}{N_i} \right)^{1+\lambda_i G_i(j)} \right) \\
= N_i \left( \left( \frac{N_i - 1}{N_i} \right)^{1+\lambda_i G_i(j-1)} \left[ 1 - \left( \frac{N_i - 1}{N_i} \right)^{\lambda_i G_i(j)} \right] \right) \\
= (N_i - G_i(j)) \left[ 1 - \left( \frac{N_i - 1}{N_i} \right)^{\lambda_i R_i(j)} \right] \\
= \left( N_i - \sum_{k=0}^{j} R_i(k) \right) \left[ 1 - \left( \frac{N_i - 1}{N_i} \right)^{\lambda_i R_i(j)} \right]. \tag{16}
\]

Substituting equation (16) into equation (15) for \( R_1 \) and \( R_2 \) yields an equation for the observable layer sizes of the product graph, in which the values of \( N_1, N_2, \lambda_1, \) and \( \lambda_2 \) are unknown. Note however that the combined fan-out \( \lambda_1 + \lambda_2 \) can be observed in the product graph, since this is the fan-out of each state in that graph. Therefore, one of the fan-out values can be eliminated, for example by replacing \( \lambda_2 \) with \( \lambda = \lambda_1 + \lambda_2 \) is the observed constant fan-out in the product graph. By observing the sizes of three layers, three equations can be obtained
from which the values of $N_1$, $N_2$ and $\lambda$ can be solved, which is enough to provide the desired estimates. As before if more layers are available, the parameters can be estimated using the least square method. Using random simulation we established that this formula is effective and correct.

It should be noted that the above can easily be extended to more than two components. For example, in case of a product graph consisting of three random transition systems, equation (14) becomes:

$$|\partial B_{T_1 \times T_2 \times T_3}(j)| = \sum_{k=0}^{j} |\partial B_{T_1}(k)| \cdot |\partial B_{T_2}(l)| \cdot |\partial B_{T_3}(j-k-l)|.$$  \hfill (17)

4.3 Experiments with state spaces of realistic systems

![Figure 2: Estimates of the reachable state space of the firewire protocol](image)

We are interested in whether the estimates using product state spaces give a better prediction for the sizes of state spaces that we encounter in practice. In figures 2 and 3 the results are provided for the firewire protocol [9] and the concurrent alternating bit protocol [10]. They are representative for other similar experiments that we have done. The sizes of the layers used in the experiments are provided in [7].

On the $x$-axis the index of each layer is indicated. On the $y$-axis the layer sizes, the estimated number of reachable states and the estimated fan-out are depicted. For the estimated number of states we use a logarithmic scale. The red squares denote the estimates assuming that the state space is a single random state space. The blue circles indicate the estimated number of states assuming that the state space consists of two random parallel systems. The estimated fan-outs...
for two parallel systems are drawn using purple ellipses and they use a linear scale. In figure 2 $1e5$ corresponds to a fan-out of thousand and in figure 3 100 corresponds to a fan-out of 10. The layer sizes are depicted by a brown dot, also on a linear scale. In figure 2 100 corresponds to a layer size of 5000 states. In figure 3 10 corresponds to a layer size of 25. The numbers are calculated using the least square method using Matlab [1] (see [7] for the actual MATLAB code used). To prevent the individual sizes of the components to become very different in the estimations, we add a small penalty $(N_1 - N_2)^2 \times 10^{-6}$ to the squared difference of input data and estimation. The knowledge of the fan-out $\lambda$ is not used. So, for the blue circles variables $\lambda_1$, $\lambda_2$ and $N_1$ and $N_2$ are estimated. Using these the size of the reachable state space is calculated and put in the figure.

It is obvious that the red boxes structurally underestimate the size of the reachable state space, where the blue circles do quite well. The figures in table 1 and figure 2 for the estimated reachable state space assuming that the state space is not parallel are different (compare $N_{\text{reachable}}$ with the red squares). This difference is due to the different estimation techniques. In section 3 it is explicitly assumed that the number of states cannot be less than the number of observed states (concretely, the lowerbound $i_m$ in equation 6). In the estimation in this section, using the least square method, this information is not used. But the common denominator, namely structural underestimation of the reachable state space, is visible in both cases.

The fact that the obtained blue estimates are doing much better supports the claim that ‘real systems’ behave as parallel non communicating state spaces. That the results contain variations is to be expected. If we look at 3D visualisations of the state space of the firewire protocol, then the two peaks in the blue circles at layers 30 and 63 match very neatly with the disks in this 3D visualisation [6]. It is unexpected that these local peaks are very high, especially because one expects a dampening effect of taking all layers into account. In this light it is also strange that the red boxes are hardly influenced by these two peaks in layer sizes.

5 Estimating the presence of residual bugs

In this section we show that it matters very much if the system has a product graph structure if it comes to the effectiveness of testing for software bugs. It shows that if graphs have a single-
threaded random structure, testing for the presence of bugs is virtually impossible, but if graphs have a product structure this is quite feasible.

Assuming that realistic systems have a parallel structure explains why in practice many bugs are found quite easily by testing. Only rarely bugs are hard to find and such bugs typically occur when a number of components are in very specific states simultaneously. It suggests that bugs in programs must be classified depending on the number of components that must be in a specific state for such a bug to occur.

We calculate how long a successful random walk through the state space must be to conclude with error $\alpha$ that there are no buggy states in the system. For this purpose we introduce two stochastic variables, namely $M$ which represents the number of states in a walk without encountering a bug, and $K$ which is the number of states with a bug. We assume that $K$ is uniformly distributed where there are between 0 and $np$ bugs. This upper bound on the number of bugs has little influence on the length of the random walk satisfying small $\alpha$. So, we could as well assume that we know that there is a small upper bound on the number of states with bugs, without altering the results.

We compare a single random state space $G$ of size $np$ with the product of $p$ state spaces $G_1, \ldots, G_p$, each of size $n$. We first calculate the probability that although there are states with bugs in the system, a random walk of size $m$ does not encounter an error state in $G$. We want this probability to be smaller than $\alpha$ such that if we conclude that the system is free of bugs on the basis of a random walk of a certain size, the probability that this is incorrect is smaller than $\alpha$. Our estimation assumes that each time we visit a state its outgoing transitions go to random other states, possibly different than in a previous visit. Strictly spoken this is not correct as the outgoing transitions of a state are static, and when we revisit a state in a random traversal, the probability that the outgoing state is also already visited is higher than in our estimation. Taking the already visited states into account is a known difficult problem, and our slightly simplified model is already very useful to heuristically show the effect of the graph structure on testing. We compute
The estimates are depicted for a system with 1000 states versus three systems with 1000 states.

For the Cartesian product graph of \( G_1 \) to \( G_p \) we come to the following estimation where we assume that graph \( G_i \) has \( K_i \) error states, uniformly distributed from 0 to \( n \). We use the notation \( \text{if}(c,x,y) \) to represent \( x \) if \( c \) holds, otherwise it is \( y \).

\[
\mathbb{P}[\sum_{i=1}^{p} K_i > 0 | M = m] = \frac{\sum_{k=0}^{p} \mathbb{P}[M = m | K = k] \mathbb{P}[K = k]}{\sum_{k=0}^{p} \mathbb{P}[M = m | K = k] \mathbb{P}[K = k]}
\]

At (18) we used that we assume that errors are uniformly distributed over the interval \([0,n^p]\). So, \( \mathbb{P}[K = 0] \) equals \( \mathbb{P}[K = k] \).

To obtain the third expression in the derivation above, we observe that a step that does not reach a state with a bug, does not hit such a state in any of the components. A component \( i \) has probability \((n - k_i)/n\) to avoid a state with a bug. Assuming that with equal probability each component can do a step, the probability to avoid a state with a bug is \( \sum_{i=1}^{p} (n - k_i)/pn \). So, the probability to avoid \( m \) times a buggy state is \( 1 - \sum_{i=1}^{p} (k_i/pn)^m \).

In figure 4 the estimates are depicted for a system with 1000 states versus three systems with 10 states. The difference is quite obvious. In the monolithic system a test run far longer than the number of states must be traversed to declare the system error free with \( \alpha = 0.05 \). With the parallel system a test run of far less than 200 steps is more than sufficient.

Notably, the number of required test runs only increases with the number of states per component. So, for four or more components with each 10 states test runs of approximately 200 are also enough to obtain a certainty of 95% on the freedom of errors, whereas with the monolithic system test runs must have a length of millions of steps to obtain the same effect.
6 Conclusions and future work

Despite the in our view promising experiments showing that $P$-parallel random transition systems are a good candidate to act as a model for real behaviour, more experimental data is required to determine whether this is really as general as we think. Besides this, there are quite a number of open technical questions. Solving $N_i$ and $\lambda_i$ in equation (14) is too time consuming, taking hours for all 107 layers of the firewire protocol. It is unclear how to determine the index of parallelism $P$ from experimental data. We fixed $P$ a priori to 2 and 3 [7]. It is unclear how to calculate the probability of a random run in a directed graph to hit a bug (even when $P = 1$).

Understanding the effectiveness of ‘highway’-search as mentioned in the introduction is still far beyond reach. This paper only contains a possible single step towards answering it.

References

A Formal Co-Simulation Approach for Wireless Sensor Network Development

Adisak Intana, Michael R. Poppleton, Geoff V. Merrett

ai1n10,mrp.gvm@ecs.soton.ac.uk
Electronics and Computer Science
University of Southampton, Southampton, SO17 1BJ, UK

Abstract: This paper proposes a Formal Co-simulation (FoCoSim-WSN) framework to provide a good software engineering practice for wireless sensor networks (WSNs) including high-level abstraction, separation of concerns, strong verification and validation (V&V) techniques. This provides an iterative interworking framework which combines the benefits of existing simulation and proof-based formal verification approaches. The complexity of software development for the sensor node controller is reduced by separating the controller model from the simulation environment. Controller Algorithms from application through network and MAC layers can be formally developed and verified in a layered manner using the refinement method of the Event-B language and its RODIN toolkit. The absence of certain classes of faults in controller models which cannot be guaranteed by simulation testing techniques, can be proved by formal methods. On the other hand, the MiXiM simulation of physical environment provides full confidence about reliability and performance analysis through long running simulation via wireless channels. Our prototype development confirms the flexibility of the framework for interworking between formal, simulation and co-simulation modelling.

Keywords: formal modelling and analysis, Event-B, proof, simulation, wireless sensor network, co-simulation

1 Introduction

A Wireless Sensor Network (WSN) is a distributed system of cooperating devices that performs distributed monitoring applications in a physical environment over a self-organised wireless network topology. In traditional WSN development, WSN requirements are tackled with a “code-and-fix” process [Pie10] in which the code is implemented on the real hardware. The WSN application is developed under the constraints of display-less, low-level specific platform and low-power design. In safety-critical application domains where WSNs are increasingly being adopted - from healthcare to military - the demand on verification is high [Pie10, BRWR10, ACB10]. Functional requirements including safety/liveness properties have to be considered together with performance and reliability requirements of the network.

Simulation is usually used at an early stage of designing and testing communication protocols because it provides the higher level of abstraction [KM07, ISH10]. It abstracts away from specific operating system platforms whereas other testing techniques such as emulation and laboratory testbeds do not. In current simulation practice, protocols and algorithms are layered to
create a communication networking protocol suite by a standard protocol stack scheme. These are integrated with a stochastic environment framework of wireless channel, radio and analogue models to generate the long running testing scenarios. The simulation and performance analysis such as network latency and energy consumption are conducted independently from any specific platform. However, code for simulation is developed monolithically; current practice is a long way from model-based software engineering process. The specification of the behaviours of the software controller algorithm and the behaviour of the environment are implemented in simulation at the same time. This gives significant complexity to manage during development and makes it hard to understand the code. Thus, a clear separation of concerns is required in this aspect. Furthermore, critical design errors are not guaranteed to be discovered during simulation. This technique cannot guarantee the absence of certain classes of faults as discovered in [IPM13].

Formal Methods have been considered to design and verify the WSN application and protocol. For example, formal analysis is proposed in [MRDD10] to detect critical network elements with OMNeT++. The framework proposed by [WBL09] also indicates the translated formal specification in PVS (Prototype Verification System) from the logic-based Network Datalog language (NDlog) to guarantee the protocol behaviour. In [IPM13], the proof-based formal method gives a strong guarantee for the absence of faults. Certain functional requirements and safety properties are encoded as invariants in Event-B and it is proved that this invariant is always satisfied by the system behaviours before actual implementation. However, the complexity and scalability of applications need long running simulation behaviours to give full confidence about reliability and performance requirements in formal models.

To increase the quality of current SE practice for WSN development, this paper is implementing the vision proposed in [PM12]. We construct the infrastructure for co-simulation between formal Event-B WSN models and MiXiM environment simulation engines. This provides an integrated set of methodologies for WSNs: (S)imulation, (F)ormal and (C)o-simulation; see Figure 1. The FoCoSim-WSN framework is proposed which is a formal co-simulation method for Event-B and MiXiM for WSNs.

(S) S-style development is the traditional WSN development style that layers the protocol algorithms and evaluates the network performance as mentioned earlier. Target code based on the simulation model is generated together with standard platform-specific libraries. Node level simulation or emulation takes place to test the correctness and performance of the real code before the real world deployment.

(F) F-style modelling represents the requirement specification, modelling and verification in a formal modelling language [IPM13, Abr07, ABHV06]. Each protocol algorithm is layered and verified through refinement steps at network level development. The verified network model is produced before different refinement paths are encoded with requirements for the specific dependent platform at node level. The final, verified node code is generated with standard libraries for a specific hardware platform from this verified node model.

(C) our C-style prototype framework enables the complexity of development to be reduced by separating the software controller from the environment. Formal methods provide the controller, a formal model of code in the real nodes, containing the protocol algorithms
separately. An environment simulator provides stochastic sensed data and radio environment, allowing simulation scenarios to be defined as required. The verified controller model for each layer of protocol stack - ultimately, down to verified generated code - is co-simulated with the environment model to perform the performance analysis. A master co-simulation language and algorithm is required to integrate and manage the component simulators. The formal Event-B controller model simulated by ProB can co-simulate with a sensor environment provided by MiXiM via this master.

In this work, our FoCoSim-WSN framework co-simulates between node controller models on the Event-B simulation and sensor environment models on MiXiM simulation. Each network algorithm at each protocol layer is separated from the wireless and physical environment and modelled in Event-B language. A master algorithm is developed by Groovy language to coordinate between network algorithm and environment models. As MiXiM can provide a socket interface in order to integrate its network models together with other simulation environment such as the Vehicles in Network Simulation (Veins) [SGD11] framework, we mock-up our own interfaces for MiXiM containing a socket interface for co-simulation. This work focuses on the network level of the development. We exercise our FoCoSim-WSN framework with the two abstractions layers at the network level development, application (app) and network (net). The two lower layers, MAC and physical (phy), are our work in progress. The node level development remains work for the future.

The remainder of this paper is organised as follows. Sections 2 and 3 discuss related work and introduce the running case study, an environment monitor system. In Section 4, we demonstrate the strengths and weaknesses of S-style development. Section 5 discusses the F-style approach showing the benefit of specifying and verifying the network algorithm model. We apply a shared-event decomposition to separate the controller from the environment. The main section is Section 6 which introduces our prototype C-style framework (FoCoSim-WSN) for co-simulation and demonstrate with the case study. Finally Section 7 give some conclusion and future work.

1 see Veins -http://veins.car2x.org/

Figure 1: Vision of co-simulation approach for WSN development
2 Related Work

Recently, co-simulation frameworks have been proposed in order to co-ordinate between the software controller model and physical simulation environment. DESTECS ² [BKL+10] is an integrative co-simulation framework for co-modelling and co-simulating between discrete-event (DE) and continuous-time (CT) of physical models via XML-RPC Interface. The formal simulation of ADVANCE³ [ADV13] provides a framework for integrating multi cyber-physical systems using different simulation engines via Functional Mock-up Interfaces (FMIs)⁴. A master-slave algorithm to execute the co-simulation is implemented in the Groovy language of the ProB tool.

A hybrid design framework is needed for WSN development when the WSN application closely interacting physical environments has become more complex. The model-based system design (MBSD) framework for WSNs proposed by [WB12] co-simulates event-triggered components illustrating network algorithms together with continuous dynamic behaviour exhibited by physical environment. HybridSim[WB13] adopts FMI standard to co-simulate between sensor application models provided by TinyOS and simulation environment generated by Modelica. SysML (based on UML) is applied to the work described above [WB12, WB13] to express the application abstraction. Their work is similar to our work in which the model of the node is co-simulated with the environment. However, their node models do not contain formal elements that leads to lack of formal precise semantics. The closest to our co-simulation framework is NMlab [HSB10] which provides a co-simulation framework for Matlab and ns-2 simulator. The system controller implemented in Matlab co-simulates with the network models provided by ns-2 by using socket interfaces. Similar to this, HarvWSNet [DBMS13], a framework for energy harvesting WSNs, combines the strengths of two development toolkits via a standard socket interface. The power management model is implemented in Matlab to communicate to the wireless sensor network communication model provided by WSNet. However, the communication algorithms at each protocol layer are still implemented in WSNet simulator.

3 Case Study

To demonstrate the effectiveness of our approach, we have extended an environment monitoring system from our preliminary work [IPM13]. This case study is derived from deployed projects described in [BIS+08]. Each sensor node in a network senses data such as temperature and wind speed periodically. This environmental data is regularly sensed and routed wirelessly via multi-hop from the source node to a sink node. As a small number of nodes was initially deployed in SensorScope project to evaluate the first use of multi-hop [BIS+08], we implemented our preliminary models consisting of 7 nodes (6 sensor nodes with node 0 representing a sink in Figure 2a) to evaluate the first demonstration of a multi-hop network.

Nodes in the simulation represent the wireless devices with their protocol stacks as shown in Figure 2b. The data that has to be sent to a data sink is collected by Application Layer before sending down to lower layers. The task of the Network Layer is to manage the route tree used to

---

² see DESTECS -http://www.destecs.org/
³ see ADVANCE -http://www.advance-ict.eu/
⁴ see FMIs -https://www.fmi-standard.org/
Figure 2: An example of (a) a multi-hop network topology and (b) node’s structure

decide the next hop for transmission towards the sink. MAC Layer manages power consumption by switching radio on/off to sending/receiving packets and provides an acknowledgement (ACK) mechanism for reliable transmission. The lowest layer, the Physical layer, performs the radio propagation for packet sending and receiving.

In this work, we focus on the experiment at the two upper layers, application and network layers. SensorApp, a periodic sensing protocol in which the sensed packet is transmitted periodically down to the lower layer, is chosen to implement at the application layer. For the network layer, MintRoute, a link quality protocol is selected to be an efficient routing protocol.

MintRoute [WTC03] is a routing algorithm to build the route tree from every node towards a sink. The route tree is dynamically changed based on the link quality between nodes. This link quality is adjusted by the successful rate of transmitted packet delivery. MintRoute performs four major steps: (1) neighbourhood discovery - one neighbour discovers another neighbour based on broadcast beacon messages, (2) link quality estimation - the estimation of reception link quality ratio is calculated periodically by observing the successful rate of receiving packets, (3) route broadcast - the transmission link quality ratio of each neighbour is estimated periodically based on the reception link quality ratio attached in “route update message”, and (4) parent selection - this is performed periodically to specify one of the neighbours for routing. The path cost towards a sink is calculated based on both two ratios (reception and transmission). Link which has these two ratios less than the quality threshold is not considered. A neighbour with the smallest path cost is chosen as a parent.

4 S-style modelling

To explore the benefits and drawbacks of S-style development, this section describes the simulation experiment on the case study described in the previous section.

MiXiM5[KSW+08] provides a development framework for the simulation and performance analysis of wireless networks including WSNs. It provides generic and flexible component architecture for models based on a standard network simulation engine, OMNeT++6. This layers the development environment into the standard IP protocol stack as shown in Figure 2b. Each

\footnote{5} see MiXiM -http://mixim.sourceforge.net/
\footnote{6} see OMNeT++ -http://www.omnetpp.org/
layer can communicate with the adjacent layer via communication interfaces named gates. In our simulation model, we extended the base modules (the general structure) for application and network layers provided by MiXiM to implement SensorApp and MintRoute respectively. The configuration parameters were replicated from the real configuration used in SensorScope [BIS+08], as discussed in our previous work [IPM13].

This exercise expresses the strength and weakness of S-style development. The network algorithms can be analysed with the performance evaluation such as the load distribution of the network and the network latency as described in the introduction. However, based on our previous work [IPM13], simulation enables us to discover the loop problem occurring in the route tree but this problem cannot be revealed at all running experiments we performed. This leads us to fix and prove this problem in formal models. Thus, simulation cannot guarantee that the fault in the algorithm will be discovered.

Furthermore, this modelling style causes complexity in development. The controller representing the specific protocol algorithm for each layer in protocol stack has to be completed together with environment elements provided by the standard interface in simulation toolkit (wireless channel communication and connectivity, the library functions - implementing sending/receiving packet and packet encapsulation/decapsulation) to form a single simulation model. This leads us to encounter difficulties of managing such a complex simulation model (especially in MintRoute algorithm implemented at the network layer). The next section will demonstrate how to reduce this complexity by using Event-B modelling techniques. Each single controller for each protocol stack in a simulation model is separated and implemented into multiple layers by using the refinement technique.

5 F-style modelling

5.1 Overview of Event-B Modelling

Event-B [Abr07] is a proof-based formal method for specification and verification based on set theory and first order predicate logic. An Event-B model consists of two parts: context and machine. The context describing the static part contains carrier sets, constants and axioms. The machine represents the behavioural part which consists of three elements: variables, invariants and events. States are described by typed variables. Invariants that state the guaranteed properties of the model express the functional refinement and safety property. Event-B can be given state machine semantics. Each event contains guard(s) and action(s). The event guards express the necessary conditions that enable the event to successfully and usefully trigger, and actions describe the state transitions over the variables. Proof Obligations (POs) are used to state that invariants are satisfied by every event.

Event-B Tool: RODIN [ABHV06] is an open tool platform based on Eclipse. This extensible tool was developed by the European Union ICT Project DEPLOY7(2008-2012). RODIN includes editors, a proof obligation generator (PO-generator), graphical front ends, theorem provers and the ProB8 animator and model checker.

7 see DEPLOY - Industrial deployment of system engineering methods providing high dependability and productivity: FP7 Project 214158 http://www.event-b.org
8 see ProB - http://www.stups.uni-duesseldorf.de/ProB/

162 A Formal Co-Simulation Approach for Wireless Sensor Network Development
Refinement: refinement is a method that allows software engineers to manage the complexity of the development by layering the abstraction of the models. A simple abstract view of essential requirements is implemented first. More requirement or design detail is added at each refinement step until implementation, data structure and algorithm are added to the concrete model in order to bring the model to become close to the real implementation. Refinement POs state that concrete refining events must correctly implement their counterpart abstract refined events.

Shared-Event Model Decomposition: the complexity of large system development is managed by breaking a single machine into sub-machines [But09, SPHB11]. These sub-machines interact by exchanging messages via shared events. The shared-event decomposition Rodin plug-in\(^9\) is applied to this work to decompose a software controller from a sensor environment.

5.2 WSN Development in Event-B

The benefit of using the F-style for WSN development described in Section 1 is presented in this section. Event-B is used to create a WSN specification and its verification. The development approach can be shown in Figure 3. The two upper layers: Application and Network layers are implemented. The Event-B refinement technique is used to layer the model which corresponds to each layer of protocol stack. We apply our MintRoute models proposed in [IPM13]. The refinement structure and some events are adjusted to support our co-simulation framework. Six refinement models are created. This begins with a very simple abstract model (M0) in which the data packet is transmitted to a sink in one atomic step. Then, the first and second refinement models (M1-M2) fulfill the operation for SensorApp. These refinement models define the neighbour node to determine multi-hop network for broadcasting mechanism before they are refined down to implement MintRoute. Each step of MintRoute protocol is layered (as in M3-M5, Figure 3) from neighbourhood discover to parent selection as described in Section 3. The models for the C-style of co-simulation development are also prepared by separating the software controller from the environment for both layer protocols. Shared event decomposition technique is applied to decompose the second and fifth refinement models corresponding to application and routing layers respectively. Note that separated environment models ENV2 and ENV5 are used for the controller verification and validation before decomposition. In the C-style development, they are replaced with the concrete simulation environment implemented in MiXiM.

For the route tree construction verification, we create safety invariants as shown in Figure 4. This shows us the benefit of proof to indicate no-loop property as a necessary invariant proved after simulation revealed it. We apply the definition of transitive closure (tcl) and the no-loop property proposed by Abrial[Abr10] and applied in [DBA08, HKBA09]. We define the route tree in @inv3_2. The initialisation of variable nodes contains only a sink ({Sink}) and the route tree (cRouteTree) is initialised to be an empty set. As soon as a node is explored to discover its parent, it is recorded in nodes with a pair between this node and its parent put in cRouteTree. Flag completedRoute in invariant @inv3_3 indicates the completion of route tree construction. Thus, invariants @inv3_2 and @inv3_3 represents that when the route tree is finished, each sensor node must have its own parent. Furthermore, safety invariant @saf3_1 illustrates the no-loop property. This can guarantee that there are no loops in the routing tree. Theorem @mth3_3 confirms every

---

node must have a route to a sink. Theorems @mth3_1 and @mth3_2 are introduced to help the proof of @mth3_3.

Figure 4: Safety invariants regarding the route tree in the third refinement model.

5.3 Model Decomposition, Verification and Validation

As WSN is a distributed system in which each node exchanges data via a channel interface, shared event decomposition [But09, SPHB11] is used to separate software controller from environment as shown in Figure 5. The software controller contains the necessary variables and events expressing communication networking protocols and algorithms such as SensorApp and MintRoute whereas environment consists of variables and events regarding the connectivity, channel and sensing environment. We design these two subcomponents to exchange messages via shared events. The steps of communication can be described as follows:

(P1) The environment activates each sensor node in the controller to sense and create a data packet via a shared event (sensing).
(P2) Each sensor node transmits a data packet down to the channel. The shared event (send_down) passes the forwarder node id and packet information as channel parameters (chnPar).
(P3) The channel indicates the neighbour of the current forwarder.
(P4) The channel returns the neighbour list and the forwarded packet to every neighbour controller via the shared event (send_up) to indicate the operation transmitting a packet up to specific receivers.
(P5) Each neighbour node (including a sink, e.g. nodes 0,1,4,3 of sender node 2, Figure 5) receives a forwarded packet (receive_pkt and sink_recv_pkt) or detects a duplicated packet (receive_dup_pkt).
Model Verification by POs: we verify the properties of a machine by proving that every event still satisfies invariants. 330 POs were discharged, in which 85 percent of the total number of POs were proved automatically by RODIN. This includes the invariant to guarantee that no packets are lost under the perfect network. Furthermore, the safety invariant regarding the absence of loop problem was discharged automatically. However, the remaining were proved interactively. This is because invariants and tcl properties include quantified predicates and graph properties.

Model Animation and Validation: we create the testing scenarios affecting the different link quality ratios that satisfy all desired requirements and strategies. These are used to animate and trace the list of operations to validate the formal model on ProB.

6 C-style Co-modelling

6.1 FoCoSim-WSN Framework for WSNs

We develop a prototype FoCoSim-WSN framework of node controller models on the Event-B simulation and sensor environment models on MiXiM simulation as shown in Figure 6. Event-B layers each communication protocol and algorithm to create and verify node controller models through refinement steps, whereas each protocol layer in MiXiM environments only contains gates (without any protocol algorithms) for communicating with the adjacent layer. In order to co-simulate these models, a master is implemented in Groovy language of the ProB tool. Here we describe the implementation of our master.

To schedule the event in the Event-B model, we implement multiple threads in the master. Each thread creates the instance of the Event-B model representing the controller of each sensor node which corresponds to each virtual node in the MiXiM Environment. MiXiM provides periodic timers such as a sensing timer and route broadcast timer, which allows the Event-B controller and MiXiM Environment models to exchange input/output periodically. TCP sockets are implemented as data exchange interfaces on both sides. We mock-up our own interfaces for MiXiM. FMInterface is the front-end interface which contains the synchronization event which corresponds to the shared-event defined in Event-B. This event is scheduled at fixed intervals and dedicated to maintain the data synchronization between the Event-B controller and MiXiM environment. Our protocol algorithms communicate with a sensor environment by exchanging the packet information (chnPar) and the receiving neighbour lists (nbrLst) together with the forwarded packet via the socket program implemented in the master. SimManager is the back-end interface where the parameters passed from FMInterface are transited down to/up from the channel via the protocol stack.
During the initialization phase, MiXiM initializes the physical environment such as interval time, ratio propagation, max transmission power and path loss coefficient alpha from the configuration file. Virtual nodes are created and the connection between them is established by method `updateNicConnection()` in MiXiM’s `ConnectionManager`. Only nodes that are placed within the maximal interference distance of each other can be connected. This forms the network topology. Then, `FMInterface` starts requesting a connection to the Event-B controller via sockets implemented in the master. Once the connection is established, `FMInterface` sends the necessary information extracted from the configuration file such as the number of nodes (`numNodes`) and identified sink id (`SinkId`) to a master. The master creates multiple threads corresponding to the number of virtual nodes generated in the MiXiM environment before it starts creating, initialising and loading the instance of Event-B controller model into each thread. Furthermore, each thread contains a `TimerTask` which is used to schedule the sequence of the events in the Event-B model. Then, the master relays the completion of the Event-B model initialisation to `FMInterface` in MiXiM via a socket. The example of the above mentioned initialization phase can be expressed in Figure 8.

At the simulation phase, the MiXiM begins to simulate at time 0. Every periodic time, MiXiM sends the information to tell each controller node to start performing the operation such as sensing and creating a packet. MiXiM generates sense data to the required random distribution for each node controller. When each node creates a packet, the channel information (`chnPar`) including the initial source of packet, the sequence number, the forwarder and the destination node id (setting to -1 for broadcasting mechanism) is synchronized between the output share-event in Event-B controller (e.g. `event send_down`, Figure 5) and the synchronization event in `FMInterface`. The virtual packet containing this correspondent input parameter is created and transmitted to the virtual neighbour node via the protocol stack. The receiving neighbour lists in MiXiM of the forwarder node are synchronized back from the synchronization event in `FMInterface` to the input share-event in Event-B controller (e.g. `event send_up`, Figure 5) of each thread of node controller. After receiving the neighbour lists, each node controller will perform the next operation (e.g the receiving events described in step P5, Figure 5). Then, the receiving node that is not the destination forwards/rebroadcasts a received packet to its neighbours. The steps of this communication are the same as described for packet creation and transmission mechanism.
6.2 Co-simulation Case Study Modelling

This section describes how to use our general \textit{FoCoSim-WSN} framework to implement the specific models in our case study.

![Co-simulation models](image)

Figure 7: Co-simulation models.

The C-style model for mixed (Event-B/MiXiM) co-simulation is developed by reuse of F-style formal-based and S-style simulation-based developments. The controller from the F-style development is reused to implement that in this mixed co-simulation. The communication steps for this co-simulation style correspond to the steps in Figure 5. We exercise the C-style co-simulation at two levels of abstraction, M2(\textit{SensorApp}) and M5(\textit{MintRoute}) as shown in Figure 7. In the MiXiM environment, two protocol layers, \textit{SimpleApp} and \textit{SimpleNet} extend from the base modules in the same way that we implement in S-style. These are implemented as a standard module for application and network layer containing packet encapsulation and decapsulation functions and gates to enable the corresponding Event-B controller representing the upper layer model to be able to communicate with the lower layer in MiXiM environment via \textit{FMInterface} and \textit{SimManager}.

Figure 8 shows the master algorithm interaction for \textit{SensorApp} corresponding to Figure 7a for one sensing cycle. As we use the same Event-B controller, we design steps R1, R2 and R4 to be the same as steps P1, P2, P4. The input/output parameters are passed from/to shared event (\texttt{send\_down/recv\_up}) in order to exchange information between the controller and environment. Then, these parameters are passed by a socket sending and receiving mechanisms (methods \texttt{send} and \texttt{recv}) as shown in steps R2 and R4. Step R5 is implemented for a receiving node to receive the forwarded data packet, method \texttt{anyEvent} provided by ProB Groovy is used to create alternative choices among events \texttt{receive\_pkt}, \texttt{receive\_dup\_pkt} or \texttt{sink\_recv\_pkt}. Note that we do not describe the master algorithm for the \textit{MintRoute} protocol. This is because it has the co-simulation interaction step to exchange the same input/output parameters for beacon and route packet transmission.

Considering step R3 which corresponds to step P3 in Figure 5, after each node in MiXiM environment receives channel information from \textit{FMInterface} via \textit{SimManager}, the packet is generated based on this information and transmitted down to lower layers and finally a channel. Note that virtual nodes in the MiXiM environment are generated by module \textit{SimpleApp} (the
same as SimpleNet for the network layer) containing only gates for transmitting/receiving a data packet to/from lower layers. The forwarding node uses method sendDown to relay a current packet down to the lower layer and finally to the channel. Method handleLowerMsg is used by receiving nodes for receiving a forwarded packet from the lower layer. SIMManager collects information from receiving nodes to generate neighbour list. Finally, this neighbour list is sent to FMInterface and relayed to the Event-B controller via the master.

6.3 Co-simulation Prototype Validation

In order to validate our C-style prototype, we ran our C-style co-models within 1400 seconds of simulation time with different network topologies varying from 1-hop to 3-hop network containing 4 to 7 nodes respectively. These simulation networks were generated from the parameters configured in the MiXiM configuration file. This performs eleven cycles of sensing periods, together with three cycles of parent selection. We compared the performance analysis results in C-style with that of S-style to identify the difference and weakness of C-style and S-style. However, we only found the randomness of the routing algorithm to discover the route tree. Figure 9, for example, shows one of our co-simulation results from the network containing the topology illustrated in Figure 2a. As shown in Figure 9a for both styles, the latency of node 2 is always low as it always chooses a sink to be a parent. However, in Figure 9b, at two-thirds of running simulation time, the latency of node 5 in S-style simulation is considerably high. This is because its parent had the highest number of forwarded data packets compared to that of C-style.
6.4 Engineering Process for C-style Development

Based on our experience, our prototype FoCoSim-WSN enables three flexible modes of working in which formal and simulation can be integrated easily.

This can start from the pure Event-B model (F-style) development in which the protocol algorithm can be modelled through the refinement steps. System engineers can develop either an early model for some high level or a refined model for more detailed level of functional abstraction. Then, the software controller illustrating the protocol algorithm can be separated from a sensor environment by using the decomposition technique. This separated controller is also prepared for C-style co-simulation.

On the other hand, the developed controller algorithm in pure S-style simulation model can be separated from the environment and implemented in the Event-B model in the same way. The separated environment, which will be used by C-style co-simulation, still retains only the communication gates (no algorithm) together with the standard function in MiXiM such as packet encapsulation and decapsulation. However, instead of developing the sensor environment from scratch, reusing our standard module for communicating between two upper layers (such as SimpleApp and SimpleNet) is another alternative way to prepare simulation environment.

The next step is C-style co-simulation in FoCoSim-WSN framework. The Event-B controller model from F-style and the separated sensor environment from S-style are co-simulated via a master. To support the development with reuse, our master is modularized into the different interfaces such as WSNSocket, TimerTask and EventBCtl for encapsulating the functionality of the socket program, scheduling events for multiple threads and Event-B controller model interface respectively. In order to develop a specific master algorithm, the system engineers only customize the EventBCtl to be compatible with their Event-B model (as we have done for our proposed co-simulation models for different layers). During implementing a master algorithm, the synchronization (shared) events for communicating between these two models are needed to be identified. However, the automatic master code generation is still left for future work. This work has accomplished the generic modules for only two upper layers: application and network. The lower standard modules for the lower layer such as MAC is still required and is work in progress.

7 Conclusion and Future Work

This paper has demonstrated the co-simulation approach to the extension of current SE for WSN development. Our prototype development shows that the framework integration for (F)ormal, (S)imulation and (C)osimulation can combine the benefits between two modelling approaches. The complexity to manage in communication protocol and algorithm development can be reduced by the refinement approach provided by Event-B. Furthermore, Event-B offers strong V&V in which the absence of certain classes of faults such as the loop problem in the route tree can be guaranteed by POs. Whereas a stochastic environment framework of wireless channel, radio and analogue models provided by simulation can help engineers to analyse and evaluate the performance of the network such as network latency and congestion. Our prototype FoCoSim-WSN framework provides an iterative interworking scheme through multiple refinement levels. System engineers also can work either F- or S-style development before the separated controller
and environment are combined and co-simulated into our prototype framework. This framework can be flexible and utilized to integrate between F-, S- and C-style modelling. System engineers can implement their models cross over into these three flexible modes of working easily.

In the future, we will perform the experimentation of this preliminary prototype framework through long running testing scenarios. The real network deployment problem will be addressed by this prototype framework as expressed in [IPM13]. Node failure, unreliable connection and buffer overflow scenarios will be injected into our co-simulation. The bottleneck problem will be tackled by limiting the queue size through long running simulation. “Killer” traces will be sought to validate formal Event-B models. More dense network models will be considered in order to evaluate the reliability of this framework and explore the network congestion problem. Code generation experiments from the node controller for the real node is still promising for our future work. Open research issues are the node level co-simulation development and the extension for multi-cosimulation.

Bibliography


Adaptive Task Automata with Earliest-Deadline-First Scheduling

Leo Hatvani\textsuperscript{1}, Alexandre David\textsuperscript{2}, Cristina Seceleanu\textsuperscript{3} and Paul Pettersson\textsuperscript{4}

\textsuperscript{1}leo.hatvani@mdh.se
\textsuperscript{3}cristina.seceleanu@mdh.se
\textsuperscript{4}paul.pettersson@mdh.se
Mälardalen University
Västerås, Sweden

\textsuperscript{2}adavid@cs.aau.dk
Aalborg University
Aalborg, Denmark

Abstract: Adjusting to resource changes, dynamic environmental conditions, or new usage modes are some of the reasons why real-time embedded systems need to be adaptive. This requires a rigorous framework for designing such systems, to ensure that the adaptivity does not result in invalidating the system's real-time constraints.

To address this need, we have recently introduced adaptive task automata, a framework for modeling, verification, and schedulability analysis in adaptive, hard real-time embedded systems, assuming a fixed-priority scheduler.

In this work, we extend the adaptive task automata framework to incorporate the earliest-deadline-first scheduling policy, as well as enable implementation of any other dynamic scheduling policy. To prove the decidability of our model, and at the same time maintain a manageable degree of conciseness, we show an encoding of our model as a network of timed automata with clock updates. To support this, we also show that reachability in our class of timed automata with updates is decidable. Our contribution helps to streamline the process of designing safety critical adaptive embedded systems.

Keywords: model-checking, task automata, earliest-deadline-first scheduling

1 Introduction

One way to enable real-time embedded systems to cope with environment, application, or platform changes is to introduce adaptivity at the design phase of system development. Adaptivity lets the system adjust to a new situation, but at the same time may introduce new errors such as breached timing constraints or other extra-functional requirements. Our goal is to propose a way to streamline modeling and verification of adaptive embedded systems (AES) in order to minimize the introduction of such errors at the design stage.

In the framework of adaptive task automata (ATA) that we have recently proposed \cite{HPS12}, we have started to address this need by providing formal support for modeling the AES behavior, simulation of the system execution, and verification of the schedulability. By formally verifying the
system’s schedulability, we ensure that the system is going to meet its hard real-time specifications as well as satisfy any other extra-functional properties.

In our previous work on adaptive task automata, we have assumed fixed priority scheduling (FPS) policy. In this work we are extending the framework to support dynamic scheduling policies by incorporating the earliest-deadline-first (EDF) scheduling policy into the framework. Hereinafter we will refer to the specific variant of ATA with the EDF scheduling policy as ATAEDF.

The main contribution of this work is to find solutions to the challenges of verifying the EDF schedulability of hard real-time tasks, in ATA. To tackle this, we show that verification of schedulability in ATAEDF, described in Section 2, is decidable, by proposing an encoding of the framework as a network of timed automata with (clock) updates (Section 3). We present a summary of the proof of bisimilarity between the model and its encoding as well as decidability of reachability for our class of timed automata with updates (Section 4).

2 Adaptive Task Automata

The adaptive task automata framework builds on top of task automata [FKPY07] by providing predicates that influence task release patterns based on the content of the ready queue. The task automata framework, in turn, is based on timed automata [Alu99] extended with: tasks that can be released upon entering locations, a queue, and a scheduler to handle the released tasks and simulate their execution. Since the current work elaborates on ATA extensively, we refer the reader to the cited literature for more in-depth information.

In our model, we assume a uniprocessor system with independent, non-suspending tasks. For each task, computation time and relative deadline are known and are specified as natural numbers. At any point in time, there can be at most one task instance (job) per task in the queue and will be also referred to as task.

2.1 Introductory Example

As a simple example, consider the set of tasks in Figure 1. Each task is characterized by its execution time \(C\) and a relative deadline \(D\). Figure 1(a) models the release of the task \(t_1\) at time 0 by annotating the initial location (double concentric circle) with the task. Task \(t_2\) is released in the second location after 3 time units. The delay is modeled by adding a zero-initialized clock \((x)\) to the system, annotating the initial location with the invariant \(x \leq 3\) that models that the location will be exited after at most 3 time units, and adding a guard \(x \geq 3\) on the edge, denoting that the
edge will not be taken until at least 3 time units have passed.
If we schedule the model in Figure 1(a) using EDF, the deadline of the task \( t_1 \) will be reached before the task has a chance to complete. Assuming that we have \( t_2' \), a lower quality alternative to task \( t_2 \), having a lower computation time, we could release \( t_2' \) instead. To be able to chose the variant of the task to be released, we have introduced the following predicates in our previous work [HPS12]:

- \( inqueue(t_i) \) which is true iff the task \( t_i \) is waiting in the ready queue or currently executing.
- \( sched(t_i) \) evaluates whether the task \( t_i \) is going to complete its execution by the deadline.
- \( sched(t_i, t_j) \), assuming that the task \( t_i \) is already in the queue, evaluates whether it will complete in time if the task \( t_j \) is released into the queue.

By incorporating the predicate \( sched(t_i, t_j) \) into the model of Figure 1(a), we get the model presented in Figure 1(b). Here, task \( t_2 \) is released only if it will not disrupt task \( t_1 \), otherwise, task \( t_2' \) is released. With this modification, which can be seen as adaptive behavior, both tasks can successfully complete.

2.2 Overview of the Existing Framework

In ATA, the ready queue is a sequence of tasks ordered by the scheduling policy. Each task \( t_i \) in the ready queue is defined by two real values \( c_i \) and \( d_i \). They represent the remaining execution time until completion \((c_i)\) and the time until the task reaches its deadline \((d_i)\).

Let us denote by \( T \) the set of tasks, and by \( P(T) \), ranged over by \( p \), the set of various Boolean combinations of the above predicates over the set of tasks. Utilizing this notation, an adaptive task automaton can be defined as follows.

**Definition 1** [HPS12] An adaptive task automaton over actions \( Act \), clocks \( X \), invariants \( \Phi(X) \), guard constraints \( B(X) \), tasks \( T \), and predicates over tasks \( P(T) \) (Definition 3) is a tuple \( \langle Act, X, L, l_0, E, I, M \rangle \) where \( L \) is a finite set of locations, \( l_0 \in L \) is the initial location, \( E \subseteq L \times B(X) \times P(T) \times Act \times 2^X \times L \) is the set of edges, \( I : L \rightarrow \Phi(X) \) is a function assigning each location an invariant, and \( M : L \rightarrow T \) is a function annotating locations with tasks.

Guard constraints \( B(X) \) are a set of conjunctions of atomic constraints of the type \( x \sim C \) or \( x - y \sim \leq \) where \( x, y \in X \) are clocks, \( C \) is a natural number, and \( \sim \in \{<, \leq, =, \geq, >\} \). Invariants \( \Phi(X) \) are a set of conjunctions of atomic constraints of the type \( x \sim C \) where \( x \in X \) is a clock, \( C \) is a natural number, and \( \sim \in \{<, \leq\} \).

In the case of \((l, g, p, a, r, l') \in E\), we write \( l \xrightarrow{g,p,a,r} l' \), where \( g \in B(X) \) is a guard constraint, \( a \in Act \) is an action, and \( r \) is the subset of clocks that will be reset on taking the edge.

We can represent the state of an adaptive task automaton as a triple \( (l, u, q) \), where \( l \in L \) is the current location, \( u : \mathbb{R}_{\geq 0} \) is a function mapping clocks to non-negative real values, and \( q = [l_0(c_0, d_0),..., l_n(c_n, d_n)] \) is the current ready task queue. \( \text{Sch}(q) \) is a function that returns the ready queue sorted according to the scheduling policy, and \( \text{Run}_{\text{Sch}}(q, \delta) \) is a function that returns the ready queue after it was executed for \( \delta \) time units.
Definition 2 [HPS12] Given an adaptive task automaton $\langle Act, X, L, l_0, E, I, M \rangle$ with an initial state $\langle l_0, u_0, q_0 \rangle$, and a scheduling strategy $Sch$, its semantics is a transition system defined as:

$$\langle l, u, q \rangle \xrightarrow{\delta} Sch \langle l', r(u), Sch(M(l') :: q) \rangle \text{ if } l' \stackrel{g, p, r}{\rightarrow} l' \in E, q \models p, \text{ and } u \models g$$

$$\langle l, u, q \rangle \xrightarrow{\sigma} Sch \langle l, u \oplus \delta, Run_{Sch}(q, \delta) \rangle \text{ if } (u \oplus \delta) \models I(l)$$

where $r(u)$ is 0 for all $x_i \in r$ and $u(x_i)$ otherwise, $t :: q$ is the result of releasing $t$ into the queue $q$, and $u \oplus \delta$ is the result of adding $\delta \in \mathbb{R}_{\geq 0}$ to all clock values in $u$. If both transitions are enabled, the choice is non-deterministic.

Intuitively, in the context of tasks, transitions are possibilities to release new tasks, while delays in locations correspond to the execution of tasks.

Definition 3 [HPS12] Given a task automaton state $\langle l, u, q \rangle$, with $q = [l_0(c_0, d_0), \ldots, t_n(c_n, d_n)]$, a scheduling policy $Sch$, and two distinct tasks, $t_i$ and $t_j$, let $P$ be the set of predicates $\{ inqueue(t_i), sched(t_i), sched(t_i, t_j) \}$ satisfied as follows:

$$\langle l, u, q \rangle \models inqueue(t_i) \text{ if } t_i \in q$$

$$\langle l, u, q \rangle \models sched(t_i) \text{ if } inqueue(t_i) \land (c_i + \sum_{j \in HP(t_i)} c_j) \leq d_i \lor \neg inqueue(t_i) \land \langle l, u, Sch(t_i :: q) \rangle \models sched(t_i)$$

$$\langle l, u, q \rangle \models sched(t_i, t_j) \text{ if } inqueue(t_i) \land \langle l, u, Sch(t_j :: q) \rangle \models sched(t_i)$$

where $HP(t_i)$ is the set of all tasks that have higher priority than $t_i$, and $Sch(t_j :: q)$ is the queue ordered by the scheduling policy $Sch$ after the release of the task $t_j$.

Boolean combinations of the above predicates over a set of tasks $T$ give us the set of all possible combinations of predicates denoted by $P(T)$.

3 Encoding of $ATA_{EDF}$

In order to show the decidability of the $ATA_{EDF}$ framework, we have encoded the universal $ATA_{EDF}$ model as a network of timed automata with (clock) updates (TAU). First we present the framework of timed automata with updates. The framework was introduced previously by Bouyer et al. [BDFP04], yet we use a variant whose decidability has to be proven for our result to hold. Then the encoding itself is laid out in three steps. The first step shows a way to encode task releases, the second provides the intuition behind the encoding of the predicates used for adaptivity, and the third introduces the encoding of the scheduler. After we have encoded the system as timed automata with updates, we provide a proof that the reachability problem for our class of timed automata with updates is decidable and that the encoding is bisimilar to the original model. The $ATA_{EDF}$ is more challenging than ATA as the task priorities are decided online.

3.1 Timed Automata with Updates

The timed automata framework, as defined by Alur and Dill [AD94], has served as the basis for several modeling variations proposed in order to fit specific design purposes [FKPY07, BDFP04, LBB+01]. Along the same line, our approach also relies on a variant of timed automata.
To concisely encode the scheduler model as timed automata, we need to allow for “clock to clock” assignments. Although such clock assignments are already present in the updatable timed automata framework [BDFP04], they are defined on models without invariants on locations. Since our work depends on location invariants, let us define the extension of timed automata that supports clock to clock assignments as well as location invariants.

Definition 4 A timed automaton with updates (TAU) over clocks $X$ and actions $Act$ is a tuple $\langle Act, X, L, l_0, E, I \rangle$, where $L$ is a finite set of locations, $l_0$ is the initial location, $E \subseteq L \times B(X) \times Act \times 2^X \times 2^{X^2} \times L$ is the set of edges, and $I : L \rightarrow \Phi(X)$ assigns invariants to locations. In the set of edges $E$, $B(X)$ is the set of guard constraints, $2^X$ represents the set of clock resets, and $2^{X^2}$ represents the set of clock assignments of the form $x := y$, where $x, y \in X$.

The set of invariants $\Phi(X)$ is a set of conjunctions of atomic expressions of the type $x \sim C$ where $x \in X$ is a clock, $C$ is a natural number, and $\sim \in \{<, \leq\}$. The set of guard constraints $B(X)$ can be defined as a set of Boolean combinations of atomic expressions of the type $x \sim C$ or $x - y \sim C$ where $x, y \in X$ are clocks, and $\sim \in \{<, \leq, =, \geq, >\}$.

In the case of $(l, g, a, r, s, l') \in E$, we write $l \xrightarrow{g \cup r s} l'$, where $r$ is the subset of clocks that will be reset on taking the edge, and $s$ the set of clock assignments.

The semantics of TAU is defined in terms of a timed transition system over states of the form $(l, u)$, where $l$ is a location, $u : \mathbb{R}_{\geq 0}$ is an assignment of clocks to non-negative real values, and the initial state is $(l_0, u_0)$, where $u_0$ assigns all clocks in $X$ to 0.

Definition 5 Given a timed automaton with updates $\langle Act, X, L, l_0, E, I \rangle$ with an initial state $(l_0, u_0)$, its semantics is a transition system defined as:

- $(l, u) \xrightarrow{a} (l', r(s(u)))$ if $l \xrightarrow{g \cup r s} l' \in E$ and $u \models g$
- $(l, u) \xrightarrow{\delta} (l, u \oplus \delta)$ if $(u \oplus \delta) \models I(l)$

where $s(u)$ performs the assignments $x_i := x_j$ for every $(x_i, x_j) \in s$, $r(u)$ is 0 for all $x_i \in r$ and $u(x_i)$ otherwise, and $u \oplus \delta$ is the result of adding $\delta \in \mathbb{R}_{\geq 0}$ to all clock values in $u$. If both transitions are enabled, the choice is non-deterministic.

A timed trace $\sigma$ of a TAU, as is also the case with timed automata [AD94], is a sequence of delay and action transitions $\sigma = (l_0, u_0) \xrightarrow{a_1} (l_1, u_1) \xrightarrow{a_2} \ldots \xrightarrow{a_n} (l_n, u_n)$ where $a_i$ can be either action ($\xrightarrow{a}$) or delay ($\xrightarrow{\delta}$) transition, and a location $l$ is said to be reachable if there exists a timed trace ending in the state $(l, u)$.

A network of TAU, $A_1 || \ldots || A_n$ over $X$ and $Act$ is defined as the parallel composition of $n$ TAU over $X$ and $Act$. Semantically, a network of TAU again describes a timed transition system obtained from those components, by requiring action transitions to synchronize on complementary actions (i.e., $a?$ is complementary to $a!$) [BY04].

### 3.2 Earliest-Deadline-First Scheduling Policy

To encode the scheduler, we need to clearly define the EDF policy in the context of this paper. Since the strategy for choosing the next task between two or more tasks with equal deadlines does
not impact the optimality of the EDF algorithm [GD99], we can give the following definition of EDF with deterministic tie resolution.

**Definition 6** According to the EDF scheduling policy with deterministic tie resolution, the priority $P_i$ of task $t_i$ is greater than the priority $P_j$ of task $t_j$ if the time left until the absolute deadline $d_i$ of task $t_i$ is smaller than the time left until the absolute deadline $d_j$ of task $t_j$, or their absolute deadlines are equal and $i > j$ holds. This can be expressed as

$$P_i > P_j \iff d_i < d_j \lor (d_i = d_j \land i > j)$$

where $i$ and $j$ represent strictly ordered task indices.

### 3.3 Task Releases

In ATA, tasks are released on changing to locations that are annotated with sets of tasks. A straightforward method to realize instant task triggering upon entering a location is to use synchronization channels on the edges of the corresponding TAU representation. This is demonstrated in Figure 2.

![Figure 2: (a) task automaton, (b) (a)’s encoding, (c) part of (b)’s scheduler](image)

In Figure 2(a), we have a basic task automaton location with two disjunctive edges leading to it. Location $l_1$ is annotated with the task set $\{t_0\}$. By entering the location via any of the edges, the task $t_0$ should be released and handled by the scheduler.

Modeling this behavior in TAU requires annotating every edge entering the location $l_1$ with a synchronization channel that creates a network of timed automata between the observed automaton presented in Figure 2(b) and the corresponding edges in the scheduler automaton as seen in Figure 2(c). In some cases, additional committed locations [BGK+02] might be needed to accomplish this.

### 3.4 Schedulability Predicates

The ATA model implements adaptivity via a set of scheduling predicates that may restrict edge guards: $sched(t_i)$, $sched(t_i, t_j)$, and $inqueue(t_i)$. All predicates are evaluated within the context of the current ready queue.

To express the predicates in timed automata with updates, we need to define an adequate encoding of the relevant variables that describe tasks in ATA models. The task automata model and consequently the adaptive task automata model define the task $t_i$ in terms of remaining computation time $c_i$ and time left until the deadline $d_i$. We encode the remaining computation time as the difference between the response time $r_i$ and the computation time $c_i$: $c_i = r_i - c_i$. 
To illustrate this encoding, let us observe Figure 3. The left side of the figure presents a Gantt chart of task releases, while the right side presents a graph of the values of the variables $c$ and $r$ for the same set of tasks. Note that, in the graph, the tasks $t_2$ and $t_3$, as well as $t_1$ and $t_4$ are presented on the same level to conserve vertical space.

At time 0, task $t_1$ is released. A higher priority task $t_2$ preempts it at time 1. At the moment of preemption, the response time $r_1$ is increased by $C_2$, the computation time of $t_2$, while the response time of task $t_2$ is equal to its computation time. Both tasks complete when their computation time becomes equal to their response time, respectively.

Two time units after task $t_1$ completes, task $t_3$ is released. It is already executing when task $t_4$ is released. Although task $t_4$ has computation time of only 1 time unit, its response time already accounts for $t_3$. Due to the continuous nature of timed automata clocks, we cannot extract information on how much of the computation time of task $t_3$ has been already used, so we have to use the full response time of task $t_3$ increased by the response time of task $t_4$. In order for this response time to be in context, we also need to copy the clock value of $c_3$ to $c_4$, hence the clock $c_4$ starts from 1.

The time until deadline is encoded by simply comparing an increasing clock to the relative deadline, but it is not shown here.

### 3.5 Scheduler and Queue

Next, we encode the EDF scheduling policy together with the queue as a single automaton, which we will hereafter refer to as the scheduler automaton.

Our scheduler is created assuming the encoding of predicates outlined in the previous section and the EDF policy presented earlier. These two constraints, addressed at the same time, have significantly increased the complexity of the encoding. In Figure 4, we show the entire scheduler model encoded as a timed automaton with updates, using synchronization channels to release tasks.

To reduce the presentation complexity of the encoding and make it more accessible to human readers, we have used a number of shorthands. For example, the queue is encoded as the set $q$. Since this set is referenced in every location, we need to replicate each location for every possible value of $q$. Since the number of tasks in the system ($N$) is finite and known in advance, this means that there will be $2^N$ replications of every location to reflect the set $q$. Next, only those locations that imply values that satisfy the incoming guards are connected by the edges to the originating location. The same approach can be applied to all other integer variables and translate...
this representation into a pure TAU. The exception to this approach is the function $t_{\text{next}}^{\text{EDF}}()$ that will be addressed later.

The scheduler consists of three locations: Idle, Busy, and Error. The edges are classes of edges that are instantiated by iterating the variable $t_i$ over the set of tasks. Task identifiers such as $t_i$ and $i$ are used interchangeably to reduce the maximum subscript level.

Since a task can be in the queue or not, the queue is encoded as a set $q$. Tasks themselves are represented via a number of variables: $t_i$ represents the $i$-th task, $t_{\text{run}}$ keeps track of the currently running task, $c_i$ represents task computation clock explained in Subsection 3.4, $t_i$ contains the current response time of the task, and $c_i$ is compared to $t_i$ to evaluate if the task has completed its execution; $d_i$ is a clock that is reset when a task is released, and is compared to the natural $D_i$ to check if the task’s deadline has passed, $P_i$ is the current priority of the task. The priority $N$, equal to the number of tasks in the system, is the highest priority and it corresponds to the currently
executing task.

The scheduler starts in the location Idle. This location corresponds to an empty task queue and it will be reentered on any occasion when there are no tasks left in the queue.

The edge going out of the location Idle is First task release. This edge is taken whenever the encoding of the adaptive task automaton synchronizes on release_i channel without any additional constraints. Consequently, the task t_i is added to the queue, the currently running task is set to t_i, the response time is set to the computation time, the deadline clock is reset, and the task is assigned the highest priority.

In Busy location, there are four edges looping in the state, one returning to Idle and one leading to Error location. The invariant on Busy location, shown in dashed rectangle in Figure 4, ensures that, in the Busy location, the currently running task will not execute longer than its computation time, and that all of the tasks in the system have not missed their deadlines.

In case that a deadline is missed, the edge Deadline miss is taken. The deadline is considered missed when the task is in the queue, still has some execution left, and has reached or exceeded its deadline. In such case, the system enters the Error location and deadlocks.

To explain the looping edges on the Busy location, let us first define the selector r^{EDF}._next().

**Definition 7** The selector r^{EDF}_next(t_i) = t_j selects the task t_j that has the next higher priority in the queue relative to the task t_i, regardless of whether the task t_i is in the queue or not according to the deterministic EDF policy (Definition 6).

The selector returns the empty set if it is invoked for the highest priority task in the queue or any not-yet-released task that would become the highest priority task if it were released.

Due to the nature of the EDF algorithm, a pure TAU implementation of this selector requires replication of any edge annotated with this selector into several edges. For the current permutation of tasks in the queue (implied by the current pure TAU location, and expressed via P_i and q variables in the representation), edges are created to test whether the new task will fit into any of the given possible positions in the queue. During verification, due to the determinism outlined in Definition 7, only one of those edges will be enabled at any time.

The edge High priority task release employs this selector to check if the newly released task has higher priority than any of the tasks in the queue. The edge guard also checks whether the currently running task is still running. This check ensures that whenever a task completes it is removed from the queue before any further actions are taken. Since the newly released task has a higher priority than any other task in the queue, its response time is equal to its computation time. All of the other tasks’ response times need to be increased by the computation time of the newly released task. Priorities of other tasks are reduced and the newly released task acquires the highest priority.

On the other hand, when the newly released task has lower priority than the currently running task, it needs to be placed at the correct place in the queue via Low priority task release edge. This is where the determinism of our EDF implementation via r^{EDF}_next selector comes into play. We need to ensure that the tasks added to the queue via this edge will be executed in the same sequence as they are added to the queue. Otherwise, the computed response times would be invalidated. As with the previous edge, we add the task to the queue, but this time we need to copy the response time and computation clock from the higher priority task. Then, we increase the response time of
the new task, as well as any of lower priority, with the computation time of the released task.

As the time passes in Busy state, tasks are executing and will be removed from the queue when they complete, by one of the Task \( t_{\text{run}} \) done edges. The edge Task \( t_{\text{run}} \) done and \( q \neq \emptyset \) is taken if the task has completed its execution before the deadline and if the next task is present in the queue. To switch the currently running task, the latter is taken out of the queue, a new task is set to currently running task and all of the active tasks’ priorities are increased by one to keep priority values bound between 1 and \( N \).

If the task is the last task in the queue, the edge Task \( t_{\text{run}} \) done and \( q = \emptyset \) is enabled and removes the task from the queue while moving the automaton into the Idle location.

To keep all clocks and response times bound the edge Maintain \( c_i \leq C_x \), resets the clock \( c_i \) to 0 every time an active clock reaches the maximum clock value \( C_x \) and the corresponding response time is decreased by \( C_x \). While this edge alters the value of clocks, it does not influence the relevant difference \( r_i - c_i \). This mechanism resolves the potential unboundedness of the system caused by the inheritance of \( c_i \) and \( r_i \) values in Low priority task release. Without it, any system that repeatedly releases tasks of lower priority than the currently running task can become unbounded.

4 Decidability

The decidability of schedulability verification for our model depends on two things: decidability of reachability for our variant of timed automata with updates (Subsection 4.1) and that the encoding of ATA\(_{\text{EDF}}\) model into timed automata with updates represents the original model correctly (Subsection 4.2).

4.1 Decidability of Timed Automata with Updates

Alur and Dill [AD94] observe that we can partition the state space of a timed automaton into a finite number of discrete regions that can be exhaustively explored in a finite amount of time. Hence, the location reachability problem is decidable.

Our refined region equivalence relation is based on the relation given in [AD94] and extended by the region equivalence relation for timed automata with diagonal constraints presented by Bengtsson and Yi [BY04], and Fersman et al. [FKPY07].

**Definition 8** (Refined region equivalence \( \approx \) [FKPY07, AD94, BY04]) For a clock \( x \in X \), let \( C_x \) be a natural number. For a positive real number \( t \), let \( \{t\} \) denote the fractional part of \( t \), and \( \lceil t \rceil \) its integer part. Let \( u, v \in \mathcal{Y} \) be two regions, \( \mathcal{G} \) a finite set of diagonal constraints in the form \( x - y \approx Z_{\geq 0} \) where \( Z_{\geq 0} \) is the set of non-negative integers, and \( \approx \in \{<, =, >\} \).

We define \( u \approx v \), i.e. \( u \) and \( v \) are refined-region-equivalent iff

1. for each clock \( x \), either \( \lfloor u(x) \rfloor = \lfloor v(x) \rfloor \) or \( u(x) > C_x \) and \( v(x) > C_x \),
2. for each clock \( x \), if \( u(x) \leq C_x \), then \( \{u(x)\} = 0 \) iff \( \{v(x)\} = 0 \),
3. for all clocks \( x, y \), if \( u(x) \leq C_x \) and \( u(y) \leq C_y \) then \( \{u(x)\} \leq \{u(y)\} \) iff \( \{v(x)\} \leq \{v(y)\} \), and
4. \( u \models g \) iff \( v \models g \) for all \( g \in \mathcal{G} \).

182 Adaptive Task Automata with Earliest-Deadline-First Scheduling
Given Definition 8 of refined region equivalence, we can postulate that operations over regions will not disrupt the refined region equivalence relationship on TAU.

Lemma 1  Given a timed automaton with updates, let \( \mathcal{G} \) denote the set of diagonal constraints in the automaton and \( C \) be the maximum of \( M_x \) (the ceiling of \( x \)) and all constants appearing in the guards and invariants of the automaton involving clock \( x \). Let \( u, v \in \mathcal{V} \) and \( t, t' \in \mathbb{R}_{\geq 0} \). Then \( u \approx v \) implies

1. \( u + t \approx v + t' \) for some real number \( t' \) such that \( \lfloor t \rfloor = \lfloor t' \rfloor \),
2. \( u[x \mapsto 0] \approx v[x \mapsto 0] \) for a clock \( x \), and
3. \( u[x \mapsto y] \approx v[x \mapsto y] \) for all pairs of clocks \( x \) and \( y \).

Proof Outline. Lemma 1 can be trivially proven for the case when only one clock is assigned a new value. The case with multiple clocks being assigned new values can be proven by observing that we can reduce the problem to relative ordering of fractional parts of clocks which are consistent for all clocks between regions based on the third criterion of Definition 8. The full proof is given in [HDSP14].

Lemma 2  (Bisimulation of TAU) Let us assume a timed automaton with updates, a location \( l \) and clock assignments \( u \) and \( v \). Then \( u \approx v \) implies that:

1. when \((l, u) \rightarrow (l', u')\) then \((l, v) \rightarrow (l', v')\) for some \( v' \) such that \( u' \approx v' \), and
2. when \((l, v) \rightarrow (l', v')\) then \((l, u) \rightarrow (l', u')\) for some \( u' \) such that \( u' \approx v' \).

Proof Outline. The proof follows from Lemma 1. Assume a location \( l \) and clock assignments \( u \) and \( v \), such that \( u \approx v \). The refined region equivalence relation \( \approx \) defines that the guards will evaluate in both \( u \) and \( v \) to the same truth values. Therefore, the set of enabled transitions is equal in both valuations.

Lemma 3  (Location Reachability) The location reachability problem for timed automata with updates and invariants is decidable if the bound \( M_x \) for each clock \( x \) is known.

Proof. Lemma 1 shows that for each location \( l \) of the automaton, there is a finite number of equivalence classes derived from the bisimulation relation \( \approx \). Since the number of locations is finite, the entire state space of an automaton can be partitioned into a finite number of equivalence classes and these equivalence classes can be effectively generated and searched.

4.2 Model Bisimulation

Once we have encoded the entire \( \text{ATA}_{\text{EDF}} \) system as a network of TAU, we need to show that there exists a bisimulation between the original model and the encoding.

Our main result is described by Lemma 4 below, for which we outline the proof. In Definition 9, we first introduce the concept of schedulability as reachability.

Definition 9  (Schedulability) The adaptive task automaton \( A \) with initial state \((l_0, u_0, q_0)\) and scheduling strategy \( \text{Sch} \) is not schedulable iff there exists a trace \((l_0, u_0, q_0)(\longrightarrow_{\text{Sch}})^* (l', u', q')\)
such that in the state \((l', u', q')\) there is a task \(t_i\) with more than zero computation time left, \(c_i > 0\), and no more time to execute, that is \(d_i \leq 0\). The state \((l', u', q')\) is marked as \((l', u', Error)\).

**Lemma 4** Let \(A\) be an adaptive task automaton and \(Sch\) the EDF scheduling strategy presented in **Definition 6**. Assume that \((l_0, u_0, q_0)\) and \((\langle l_0, Idle\rangle, u_0 \cup v_0)\) are the initial states of \(A\), and the product automaton \(E(A)||E(Sch)\), respectively, where \(l_0\) is the initial location of \(A\), \(u_0\) and \(v_0\) are clock assignments assigning all clocks with 0, and \(q_0\) is the empty task queue. Then:

For all \(l\) and \(u\):
\[
(l_0, u_0, q_0) \rightarrow^* (l, u, Error) \text{ implies } \langle (l_0, Idle), u_0 \cup v_0 \rangle \rightarrow^* \langle (l, Error), u \cup v \rangle \text{ for some } v.
\]

For all \(l, u,\) and \(v\):
\[
(l_0, u_0, q_0) \rightarrow^* (l, u, Error) \text{ implies } \langle (l_0, Idle), u_0 \cup v_0 \rangle \rightarrow^* (l, Error, u \cup v).
\]

**Proof Outline.** The encoding of the \(ATA_{EDF}\) automaton differs from the automaton itself in two key aspects: invocations of tasks and adaptivity predicates.

Since the task releases in \(ATA_{EDF}\) can be said to be of a non-blocking nature, we essentially verify that the scheduler will be non-blocking as well. Indeed, the only situation when there are no enabled edges annotated with a synchronization channel is when the scheduler automaton enters the Error state.

Our encoding exposes the values required for the evaluation of the encoded schedulability predicates directly. In order to check that the \(ATA_{EDF}\) adaptivity predicates are going to evaluate to the same results, we establish a correlation between tasks’ parameters in \(ATA_{EDF}\) and the encoding. Once these mappings have been properly established, we check edge-by-edge that they are maintained. The full proof is given in [HDSP14].

Since we have proven that the reachability problem is decidable for TAU, stated by **Lemma 3**, also that every \(ATA_{EDF}\) can be translated into a bisimilar TAU, we can conclude that the problem of checking schedulability of \(ATA_{EDF}\) is decidable as well.

### 5 Related Work

Our work tries to unify schedulability analysis with modeling and analysis of adaptive embedded systems. At the same time, a number of works address problems in those two separate fields, as well as non-modeling methods for analysis of schedulability in adaptive contexts. While this is by no means an exhaustive list of the works in these areas, we will try to list those that are closest to ours.

In the following works, verification of adaptive embedded systems is done on a more coarse scale than in our approach. Most of these approaches could be used in synergy with ours to provide system level verification, while ours provides task level granularity. Adler et al. [ASSV07] use Kripke structures as the underlying presentation of the system and specify the system’s properties using LTL. Schneider et al. [SST06] have proposed a method to describe and analyze adaptation behavior in embedded systems in which the data flow is augmented with quality descriptions used by configuration rules to determine potential adaptations. Goldsby et al. [GCZ08] provide the AMOeba-RT model focused on run-time verification and monitoring.

In the area of adaptive scheduling, most work [JPG04, LRK03] was done to achieve a lower
energy consumption by exploiting dynamic voltage scaling features of modern CPUs. While such approaches can be used to analyze schedulability in some adaptive contexts, our approach makes it possible to model and analyze more precisely task release patterns of non-periodic tasks.

Finally, other works have approached verification of schedulability by means of timed automata for uniprocessors [DILS09, MLR+10], and multiprocessors [YLX10] without explicit inclusion of adaptive functionality.

6 Conclusion

In this work, we have shown that the verification of adaptive task automata with earliest-deadline-first scheduling policy is decidable. To support our claim, we have encoded our adaptive task automata model as timed automata with updates and presented that the model and its encoding are bisimilar, as well as given a proof that reachability in our variant of timed automata with updates is decidable.

Our main result is the proof of decidability of our ATA extensions. Using ATA, it is possible to model the environment of an embedded system as well as behavior of functional and extra-functional properties in response to internal or environmental changes. Thus we verify the behavior of specified properties throughout the execution of the system.

In this work, we have implemented the EDF scheduling policy. However, by replacing the selector \( r_{\text{next}} \), we can implement any other policy that is deterministic and does not change relative task priorities after their release into the queue. A non-deterministic selector would invalidate the schedulability testing predicates (\( \text{sched}() \)) since the response times predicted when testing a task would not necessarily correspond to the actual response times after the task is released.

During the encoding, we have faced a number of challenges. To support dynamic scheduling policies and schedulability predicates, we have required dynamic construction of task response times, which, in turn, have required a clock copying mechanism that had to be added as an extension of timed automata.

As future work, we plan to further explore removal of the assumptions, specifically extend the framework to support modeling of multi-core systems, smart handling of tasks with variable execution time, shared resources, as well as create a set of templates that correctly model the most commonly utilized task release patterns.

Acknowledgements: This research has been supported by the Swedish Research Council, which is gratefully acknowledged.

Bibliography

### References


Performance Analysis of Distributed and Asynchronous Systems using Probabilistic Timed Actors

Ali Jafari, Ehsan Khamespanah, Marjan Sirjani, and Holger Hermanns

1 Reykjavik University, School of Computer Science and CRESS
2 University of Tehran, School of ECE
3 University of Saarland, School of Computer Science

Abstract: Many real-time distributed applications exhibit probabilistic and non-deterministic behaviors. In this paper, we introduce Probabilistic Timed Rebeca (PTRebeca) as an actor-based language for modeling probabilistic distributed real-time systems with asynchronous message passing. We propose the semantics of PTRebeca model in Timed Markov Decision Process (TMDP), the integral semantics of probabilistic timed automaton (PTA) with one digital clock. To analyze PTRebeca models, we develop a tool set to automatically generate a TMDP model from a PTRebeca model in the form of the input language of PRISM model checker. We use PRISM for performance analysis of PTRebeca models against expected reachability and probabilistic reachability properties. We show the applicability of our approach using a few case studies and experimental results.

Keywords: Probabilistic real-time system, probabilistic Timed Automata, Timed Markov Decision Process, Probabilistic Timed Rebeca, Model checking, Performance analysis

1 Introduction

Our society more and more relies on computing systems that are distributed, consisting of concurrently executing components which communicate asynchronously over networks. Modeling and analyzing such systems is a nontrivial and intricate task, owed to their complex behavior. There is thus a need for modeling languages that match well with computational models and are supported by tools to analyze performance and dependability aspects of such systems.

The actor model was proposed for modeling distributed and asynchronous systems, advocating that software systems are built by composing concurrent objects. Actors are distributed, autonomous objects that interact by asynchronous messages. Building on an event-driven and message-based foundation, actors provide scalability and also less error-prone concurrent models. With the growth of cloud computing, web services, networks of embedded computers, and multicore architectures, programming using the actor model has become increasingly relevant.

Popular actor programming languages and frameworks include Erlang and the Scala / Akka family. Many projects in industry, e.g. at Google (like DART) and Microsoft (like
Asynchronous Agents Library), have explored the actor model. Large applications such as Twitter’s message queuing, image processing in MS Visual Studio 2010, as well as the Vendetta game engine have been designed on the basis of this model.

Rebeca [SM01, SMSB04] is an actor-based modeling language designed to enable formal verification of actor models and hence bridge the gap between formal methods and software engineering. Using Rebeca we can deploy a model-driven development approach with formal basis. Rebeca is supported by formal verification tools and techniques which are based on the formal semantics of the language [SJ11]. An extension of Rebeca [ACI*11] has been proposed to provide the ability of modeling and verification of distributed systems with real-time constraints. In this context, Floating Time Transition System (FTTS) were introduced to significantly reduce the state space generated for model checking of Timed Rebeca (TRebeca) models [KSS*14]. Deadlock freedom and schedulability analysis of TRebeca models can be performed using FTTS.

Since its introduction, TRebeca has been used in different areas. One example is in analyzing different routing algorithms and scheduling policies in NoC (Network on Chip) designs [SMMS13, SMS13]. Another example is schedulability analysis of distributed real-time sensor network applications [MKSA13], more specifically a real-time continuous sensing application for structural health monitoring in [LMS13]. An ongoing project is on evaluating different dispatching policies in clouds where we have priorities and deadlines in Mapreduce clusters, based on the work in [GCR*09]. In analyzing all the above mentioned applications, we observed the need for modeling probabilistic behavior. In an earlier work, pRebeca is proposed as an extension of Rebeca to model probabilistic systems [VK12], but pRebeca does not support the time features.

In this paper, we propose Probabilistic Timed Rebeca (PTRebeca) which benefits from modeling features of TRebeca and pRebeca, combining the syntax of pRebeca and TRebeca languages. This aims at enhancing our modeling ability in order to cover more properties, by performance evaluation of probabilistic real-time actors. Although the syntax of PTRebeca is a combination of TRebeca and pRebeca, their semantics and supporting tools are not applicable for PTRebeca. Consequently, we propose a semantics to support timing, probabilistic, and non-deterministic features. To the best of our knowledge, PTRebeca is the first actor-based language which supports time, probability, and non-determinism in modeling distributed systems with asynchronous message passing.

We propose PTRebeca on the basis of a study of different distributed and asynchronous applications, studied to identify what is needed for modeling and analysis of those applications, relative to different probabilistic and timed probabilistic models (discrete, continuous, stochastic) proposed in the literature. In PTRebeca, time is discrete, and discrete probability distributions are used. Using probabilistic and non-deterministic assignments, the computation outcomes and network delays can become probabilistic or non-deterministic. For performance evaluation of PTRebeca models we employ probabilistic model checking, as a single computational techniques for both functional verification and performance evaluation. The benefits of combining performance evaluation with functional verification is elaborated upon in [BHHK10].
The main contributions of this paper are as follows:

- **Modeling**: PTRebeca supports the modeling of non-deterministic and probabilistic behaviors which is widely required in distributed asynchronous real-time systems.

- **Semantics**: We propose Timed Markov Decision Process (TMDP) as semantics of PTRebeca. TMDP can be regarded as the discrete time semantics of probabilistic timed automata (PTA) [KNPS06].

- **Analysis**: We harvest probabilistic model checking algorithms developed for PTA and MDP for the analysis of probabilistic timed properties. For the analysis, we use PRISM [HKNP06] as a back-end model checker, so as to also support expected reachability and probabilistic reachability analysis for PTRebeca models.

- **Implementation**: We present a tool developed to generate the TMDP of PTRebeca models automatically. The generated TMDP is in the form of an XML file. The XML file is converted to the input language of PRISM.

- **Case Studies**: We present a ticket servicing and a sensor network application example to demonstrate the feasibility of the approach.

### Advantages of PTRebeca in digital time semantics.

In PTRebeca, time is discrete, and when generating state space, time can be modeled using a single integer-valued variable. The state space of a PTRebeca model is finite whenever the model represents a recurrent behavior. Additionally, the time-shift equivalency approach [KSS+14] can be used to bound state space when time progresses, and efficient model checking algorithms developed for untimed systems can be applied to integer-timed models [KNPS06, HH09].

Using this approach, PTRebeca models can be verified against properties specified in PCTL [KNSS02]. The model checker PRISM supports this and two other performance measures: expected reachability and probabilistic reachability properties [KNPS06]. This analysis trajectory can be applied to PTRebeca models because the TMDP semantics is equivalent to a diagonal-free and closed PTA. A closed PTA does not contain strict inequalities, a diagonal-free PTA does not compare values of different clocks, which is assured by construction since the semantics has a single clock.

The TMDP semantics unfolds parallel composition of PTRebeca components. There is an alternative approach for performance analysis of PTRebeca models where each component of the PTRebeca model is converted to a PTA. The parallel composition of PTA (of all components) represents the behavior of the PTRebeca model. This approach is explained in [JKS], and PRISM can be used for performance analysis of the PTRebeca model through model checking the resulting PTA. The apparent benefit of avoiding the state space explosion often caused by interleaving parallel composition does not manifest itself in the PTRebeca setting: in [JKS], we demonstrate that the state space generated via the TMDP semantics is much smaller than the state space generated from the parallel composition approach. Therefore we advocate the TMDP semantics as the basis for our mapping to PRISM and for performance analysis of PTRebeca models.
2 Probabilistic Timed Rebeca

In this section, we introduce Probabilistic Timed Rebeca (PTRebeca). We first present Rebeca [SM01, SMSB04], and then we show its extension with timing features to build TRebeca [ACI+11]. Finally we discuss how probability and time are added to Rebeca to build PTRebeca, enabling the modeling of probabilistic timed behaviors. The syntax of PTRebeca is presented in [JKS]. We model a simple ticket service example to explain the modeling features of PTRebeca.

Rebeca. Rebeca is an actor-based modeling language with formal semantics that is supported by model checking tools. A Rebeca model consists of the definition of reactive classes and the instantiation part which is called main. The main part defines instances of reactive classes, called rebecs. The behavior of the instances of a reactive class is determined by its message servers. The internal state of a reactive class is represented by the valuation of its state variables.

In Rebeca, computation is event-driven, where messages can be seen as events. Each rebec takes a message from its message queue and executes the corresponding message server. Execution of a message server body takes place atomically (non-preemptively). Communication takes place by asynchronous message passing, which is non-blocking for both sender and receiver. The sender rebec sends a message to the receiver rebec and continues its work. The message is put in the message queue of the receiver. The message stays in the queue until the receiver takes and serves it. Although in theory we define no boundary for the queue length, in the supporting tools we always have a queue length that is defined by the user. The operational semantics of Rebeca is introduced in [SMSB04], to which we refer for more details. The syntax of Rebeca is represented in [JKS].

Timed Rebeca. TRebeca was introduced as an extension of the Rebeca language to model real-time reactive systems. Just as with Rebeca, the formal semantics of TRebeca is defined using Structural Operational Semantics (SOS) [ACI+11]. In a TRebeca model, each rebec has its own local time, which can be considered as synchronized distributed clocks. Methods are executed atomically, but passing of time can be modeled while executing a method. Instead of a message queue for each rebec, there now exists a bag containing sent messages together with timing information, which are used to process the message in the intended order in time. Different timing primitives are added to Rebeca syntax to cover a variety of timing features that a modeler might need to address in a message-based, asynchronous and distributed setting. These timing primitives are delay, deadline and after, and detailed below. The syntax of timing primitives are shown in [JKS].

- **Delay**: delay(t) increases the value of the local time of the respective rebec by the amount of t.
- **Deadline**: r.m() deadline(t), after t units of time the message m of rebec r is not valid any more and is to be purged from the bag.
After: \( r.m() \) after(t), the message cannot be taken from the bag before \( t \) time units have passed.

Upon sending, a message is put in the message bag at the receiver together with its associated time tag and deadline tag. The time tag of a message is the value of local time of the sender when the message was sent, unless the message is augmented with an after primitive. In this case the value of the argument of after is added to the value of local time of the sender to build the time tag.

### 2.1 Probabilistic Timed Rebeca

PTRebeca language supports the modeling and verification of real-time systems with probabilistic behaviors. The syntax of PTRebeca is a combination of pRebeca and TRebeca. We propose the appropriate semantics for PTRebeca to be able to model and verify probabilistic properties. In [JKS], we show the extension made to the syntax of TRebeca to build PTRebeca. In a probabilistic assignment, a value is assigned to the variable with the specified probability. In probabilistic assignment, \( e_1, \ldots, e_n \) are real values between 0 and 1, and sum up to 1. Notably, by using probabilistic assignment, the value of the timing constructs (delay, after, and deadline) can also become probabilistic.

Different probabilistic behaviors can be modeled using PTRebeca, depending on the system under study. We present a simple ticket service system in Figure 1 to illustrate how PTRebeca can be applied. Each entity in the system is mapped to an actor in PTRebeca model. The ticket service model includes a customer, a ticket service, and an agent. The customer \( c \) sends a ticket request by sending the message sendRequest() to the agent \( a \) (line 27). The agent forwards the request to the ticket service \( ts \) by sending the message requestTicket() (line 17). The message requestTicket() has a deadline which is set non-deterministically (line 16). The ticket service issues a ticket and replies to the agent request by sending the message sendTicket() (line 6). The agent sends the message getTicket to the customer to complete the issuing process (line 20). The customer sends a new request after 10 or 30 units of time with probabilities of 0.25 or 0.75, respectively (lines 29-32).

### 3 Semantics of Probabilistic Timed Rebeca

In this section, we define the Timed Markov Decision Process (TMDP) semantics of a PTRebeca model. Formally, a TMDP is defined as follows [JLS07].

**Definition 1 (Timed Markov Decision Process)** A timed Markov decision process is a tuple of \((TMDP)T = (S, s_0, Act, \rightarrow, L)\) includes the following components:

- A set of states \( S \) with an initial state \( s_0 \in S \),
- A set of actions \( Act \),
Performance Analysis of Distributed and Asynchronous Systems using Probabilistic Timed Actors

1 reactclass TicketService(4) {
2 knownrebecs [Agent a] {
3 TicketService () {} { }
4 msgsrv requestTicket() {
5 delay(3);
6 a.sendTicket();
7 }
8 }
9 reactclass Agent(4) {
10 knownrebecs {
11 TicketService ts ;
12 Customer c ;
13 }
14 Agent() {}
15 msgsrv sendRequest() {
16 int a = ?(4,5) ;
17 ts.requestTicket() deadline(a);
18 }
19 msgsrv sendTicket() {
20 c.getTicket() ;
21 }
22 }
23 reactclass Customer(4) {
24 knownrebecs {
25 Agent a ;
26 }
27 Customer() {
28 self . try () ;
29 }
30 msgsrv try() {
31 a.sendRequest();
32 }
33 }
34 msgsrv getTicket() {
35 int b = ?(0.75:30,0.25:10) ;
36 self . try () after (b);
37 }
38 }
39 msgsrv try() {
40 }
41 main {
42 Agent a(ts, c) :() ;
43 TicketService ts (a) :() ;
44 Customer c(a):() ;
45 }

Figure 1: The model of the ticket service system.

• A timed probabilistic, non-deterministic transition relation \( \rightarrow \subseteq S \times Act \times \mathbb{N} \times \text{Dist}(S) \) such that, for each state \( s \in S \), there exists at least one tuple \((s, -, -, -) \in \rightarrow\).

• A labelling function \( L : S \rightarrow 2^{AP} \), where \( AP \) is the set of atomic propositions.

The transitions in a TMDP are performed in two steps: given that the current state is \( s \), the first step is a non-deterministic selection of \((s, act, d, \nu) \in \rightarrow\), where \( act \) denotes a possible action and \( d \) specifies the duration of the transition; in the second step, a probabilistic transition to state \( s' \) is made with probability \( \nu(s') \). Function \( \nu \in \text{Dist}(S) \) denotes a discrete probability distribution.

In the following, we define some concepts for PTrRebeca models before turning to the TMDP semantics of PTrRebeca.

Definition 2 (Probabilistic Timed Rebeca Model) A Probabilistic Timed Rebeca model \( M \) is the set of rebecs which are concurrently executing.

A computation of Probabilistic Timed Rebeca model \( M \) takes place by execution of all rebecs defined in the model according to the SOS-semantics in \([ACI^{+}11]\). For a Probabilistic Timed Rebeca model \( M \), the function \( O(M) \) returns all rebecs in the model \( M \).

Definition 3 (State of a PTrRebeca model in TMDP) A state of a PTrRebeca model \( M \) is a tuple \( s = (\prod_{r \in O(M)} (\text{state}(r_i) \times pc \times rt) \times \mathbb{T}) \times \mathbb{T} \), where \( \text{state}(r_i) \) is the state of rebec \( r_i \), \( \mathbb{T} \in \mathbb{N} \) is the current time of state, \( pc \in \mathbb{N} \) is the program counter of rebec \( r_i \), and \( rt \in \mathbb{N} \) is the resuming time of rebec \( r_i \).

Each rebec of \( M \) has a state which consists of the values of its state variables, its local time, and its message bag. Functions \( \text{sv}(s,r_i), \text{bag}(s,r_i), \) and \( \text{now}(s,r_i) \) return the state
variable valuation function, the content of message bag, and the local time of rebec $r_i$ in state $s$, respectively. In TMDP semantics of a PTRebeca model, the local times of rebecs have the same value. We define function $\text{now}(s)$ to access the time in state $s$.

The rebec program counter, $pc$ of rebec $r_i$ specifies the statement to be executed, and function $pc(s, r_i)$ returns the value of program counter of rebec $r_i$ in state $s$. The rebec resuming time, $rt$ of rebec $r_i$, determines the time when the statement of the message server of rebec $r_i$, pointed to by $pc$, is executed. Function $\text{rt}(s, r_i)$ returns the value of resuming time of rebec $r_i$ in state $s$.

In the initial state, the local time of all rebecs are set to zero, and the constructor of all rebecs are executed to initialize state variables and queues content. Initially, for all rebecs the value of program counter and the value of resuming time are supposed to be null.

**Definition 4 (The Content of a Message Bag)** A tuple $tmsg = (\text{msgsig}, \text{arrival}, \text{deadline})$ is a message where $\text{msgsig}$ is the message content, $\text{arrival}$ is the arrival time of the message, and $\text{deadline}$ is the deadline of the message. The arrival time of the message is computed based on the local time of the sender and the value of “after” of send message statement. The deadline of the message is also computed based on the local time of the sender.

For $tmsg \in \text{bag}(s, r_i)$, the functions $\text{sig}(tmsg)$, $\text{ar}(tmsg)$, and $\text{dl}(tmsg)$ return the $\text{msgsig}$, $\text{arrival}$, and deadline of the message $tmsg$, respectively. The message content $\text{msgsig}$ consists of the message name, the sender, the receiver, and its actual parameters and is shown as “sender $\rightarrow$ receiver.$\text{msgname}(\text{parameters})$”.

**Definition 5 (Possible Messages)** The set of messages $Tmsg = \{tmsg \mid \forall r_i, r_j \in O(M) \forall \text{ar, dl} \in \mathbb{N}, tmsg = (r_i \rightarrow r_j, \text{msgname}(\text{ar, dl}))\}$ is the set of all possible messages which can be sent by any rebec $r_i$ to another rebec $r_j$ at any arrival time $\text{ar}$ and deadline $\text{dl}$.

**Definition 6 (Rebec Enabled Messages)** Enabled messages of a rebec are messages whose arrival time is less than the time of state $s$: $\text{em}(s, r_i) = \{tmsg \in \text{bag}(s, r_i) \mid \text{ar}(tmsg) \leq \text{now}(s)\}$.

**Definition 7 (TMDP semantics of a PTRebeca model)** A TMDP of PTRebeca model $M$ is a tuple $(S, s_0, \text{Act}, \rightarrow, L)$, where:

- $S$ is the set of states according to Definition 3,
- $s_0 \in S$ is the initial state,
- $\text{Act}$ is a set of $Tmsg \cup \{\tau\} \cup \mathbb{T}$, where $Tmsg$ is the set of all possible messages which can be sent by any rebec to its known rebecs, $\tau$ is an internal action and $\mathbb{T} \in \mathbb{N}$ is the progress of time.
- $\rightarrow \subseteq S \times \text{Act} \times \mathbb{N} \times \text{Dist}(S)$ is the transition relation, where $(s, \text{act}, d, \nu) \in \rightarrow$ if and only if one of the following conditions hold for $s$. 

AVoCS 2014 195
1. **Taking a message for execution** If in state \( s \), there exists \( r_i \in O(M) \) such that \( pc(s,r_i) = \text{null} \) and \( em(s,r_i) \neq \emptyset \): The execution of \( tmsg \in em(s,r_i) \) results in \( s' \) with probability \( v(s') = 1 \) and \( d=0 \). In this case act is equal to \( tmsg \), \( tmsg \) is extracted from the message bag of the rebec \( r_i \), \( pc(s,r_i) \) is set to the first statement of message server \( tmsg \), and \( rt(s,r_i) \) is set to \( now(s) \).

2. **Internal action \( \tau \)** If in state \( s \), there exists \( r_i \in O(M) \) such that \( pc(s,r_i) \neq \text{null} \) and \( rt(s,r_i) = now(s) \): The statement of message server of \( r_i \) specified by \( pc(s,r_i) \) is executed and one of the following cases may occur based on the statement execution:

   (a) The statement is an ordinary statement: the execution of statement may change the value of some state variables of the rebec \( r_i \) or induce sending a message to a rebec. Then, \( pc(s,r_i) \) is increased by one, the act is \( \tau \), \( d=0 \), and the execution of \( \tau \) results in \( s' \) with probability \( v(s') = 1 \).

   (b) The statement is a non-deterministic assignment: the execution of non-deterministic assignment \( a =? (v_1,...,v_n) \) results in \( n \) different transitions from \( s \) to states \( s'_1,s'_2,...,s'_{n'} \) where \( a = v_i \) in state \( s'_i \). For each transition, the act is \( \tau \), \( d=0 \), and the execution of \( \tau \) results in \( s'_i \) (\( 1 \leq i \leq n \)) with probability \( v(s'_i) = 1 \).

   (c) The statement is a probabilistic assignment: the execution of probabilistic assignment \( a =? (p_1 : v_1,...,p_n : v_n) \) results in a transition from \( s \) to states \( s'_1,s'_2,...,s'_{n'} \) where \( a = v_i \) in state \( s'_i \). The act is \( \tau \), \( d=0 \), and the execution of \( \tau \) results in \( s'_i \) (\( 1 \leq i \leq n \)) with probability \( v(s'_i) = p_i \).

   (d) The statement is a delay statement with parameter \( t \in \mathbb{N} \): the execution of the delay statement does not change \( pc(s,r_i) \) (because the execution of delay statement is not yet complete), and \( rt(s,r_i) \) is set to \( now(s) + t \). (Note: the value of \( pc(s,r_i) \) will change to the next state after completing the execution of the delay which can be seen in item 3.) The act is \( \tau \), \( d=0 \), and the execution of \( \tau \) results in \( s' \) with probability \( v(s') = 1 \).

When the last statement of the message server of \( r_i \) is executed, the \( pc(s,r_i) \) is set to null.

3. **Progress of time** If in states \( s \), none of the aforementioned conditions in items 1 and 2 hold: this means \( \forall r_i \in O(M), ((pc(s,r_i) = \text{null} \land em(s,r_i) \neq \emptyset) \lor (pc(s,r_i) \neq \text{null} \land rt(s,r_i) = now(s))) \). In this case, \( now(s) \) is increased by the minimum amount of \( t_1 \in \mathbb{N} \) such that one of the aforementioned conditions becomes true. If \( pc(s,r_i) \neq \text{null} \) and \( rt(s,r_i) = now(s) \) (the current value of \( pc(s,r_i) \) points at a delay statement), \( pc(s,r_i) \) is increased by one. The act is set to time, \( d = t_1 \), and the execution of action time results in \( s' \) with probability \( v(s') = 1 \).

   - A labelling function \( L : S \rightarrow 2^{AP} \).

When more than one transition is enabled in state \( s \), a non-deterministic selection is made. □
4 Analysis of Probabilistic Timed Rebeca and Experimental Results

We have developed a tool set [Reb] in order to generate the TMDP semantics from PTRebeca models. This TMDP semantics can be exported to PRISM as a single MDP module with one integer-valued variable modeling the passage of time. In [JKS], we show the PRISM code generated for the ticket service example presented in Figure 1. Using a dedicated time action and the ability of assigning rewards to transitions in PRISM, we can analyze expected-time reachability and time-bounded probabilistic reachability properties.

In PTRebeca models, the capacity of message bags is bounded. The number of states in PTRebeca model can be finitely represented when the system shows recurrent behavior. We also use the time-shift equivalence approach proposed in [KSS+14] to avoid state space explosion otherwise induced by time progress. In this approach, two TMDP states $s$ and $t$ (in the sense of Definition 3) are time-shift equivalent if the values of all variables except timing variables, i.e. local time, arrival time, deadline, in states $s$ and $t$ are identical. Therefore, the two states can be identified by shifting time.

The PRISM modeling language is a state-based language while PTRebeca language benefits from high-level data structures and constructs which arguably makes modeling easier. PRISM models are thus closer to the underlying probabilistic models and therefore we bridge to PRISM on the semantics level.

4.1 Experimental Results

In this section, we present two examples demonstrating the applicability of the proposed approach for performance analysis of asynchronous systems.

Ticket Service. We extend the simple ticket service model shown in Figure 1 to a more complicated scenario detailed in [JKS]. In this case study, there are two customers, two ticket services, and one agent. Each customer sends a ticket issue request to the agent and the agent forwards the request to the first ticket service with probability of 0.6, and to the other one with the probability of 0.4. The ticket service issues a ticket and replies to the agent request. The agent sends the message to the customer to complete the issuing process. We analyze the following probabilistic reachability and expected reachability properties for the ticket service model:

- We check whether eventually tickets are issued for both customers. The property is specified as: $P \geq 1 \left[ F (c1\_issued = true) \& (c2\_issued = true) \right]$. The property is satisfied, which is to be expected according to the model.

- We check the maximum expected time until tickets are issued for both customers. Since we define rewards on transitions representing the passage of time, we are able to check such an expected time property. The property is specified by: $R["time"]_{max} =? F (c1\_issued = true) \& (c2\_issued = true)$. The result is 3.76 units of time, meaning tickets are issued for customers after 3.76 time units on average.
- We check the maximum expected number of requests until tickets are issued for both customers. To model check this property, we define rewards on transitions corresponding to requesting ticket. The property is formulated by:

\[ R\{\text{request}\} \]

\[ \text{max} = \mathcal{F}\{c1\text{issued} = \text{true} \& c2\text{issued} = \text{true}\} \].

The result is two requests, meaning tickets are issued for customers after two requests.

We also examine a variation of the ticket service model in which a system fault is injected to the model. The ticket service model is similar to the model in Figure 1 except that the ticket service only responds to the requests of customer \(c1\). We examine this case study to show that in the existence of a fault in the system, i.e. not issuing a ticket for customer \(c2\), it is detected through probabilistic model checking. The results of this scenario is reported in [JKS].

**Sensor Network.** The sensor network model is shown in [JKS]. There is a lab environment in which the toxic level changes periodically after an amount of time, specified by the value of variable \(\text{changingPeriod}\). The environment is safe at first. The toxic level is changed to a dangerous level with a probability of 0.02. If the toxic level of the environment reaches a dangerous level, the scientist, working inside the lab, will die after a specified amount of time, specified by the value of variable \(\text{scientistDeadline}\). One sensor in the lab environment is measuring the toxic level of the lab. The measured information is sent periodically by the sensor to the administrator. A sensor may fail to report the measurement to the administrator with a probability of 0.01. If the toxic level reported by the sensor reaches a predefined dangerous level, the administrator sends a message to the scientist, who is assumed to be working in the environment, to inform him to leave the lab and go to a safe place. The administrator waits for a while for an acknowledgement from the scientist. If the acknowledgement is not received by the admin on time, the admin orders a rescue team to get hold of the scientist.

In this model the value of probabilities are small enough to let the model converge to the optimum value; and consequently show the real behavior.

Figure 2(a) shows the maximum and the minimum probabilities of the scientist death when the value of variable \(\text{checkingPeriod}\) changes. If the sensor checks the environment with a high frequency, i.e. the value of variable \(\text{checkingPeriod}\) is low, the probability of sensor failure will increase, resulting in high probability of the scientist death. For example, when the sensor checks the environment once every units of time, the environment is checked five times before the first change in the environment. Therefore, the cost of the sensor use and consequently the probability of sensor failure increases. When the sensor frequency is low, the environment changes cannot be detected on time; resulting in a high probability of the scientist death. There is an optimum value for the variable \(\text{checkingPeriod}\), i.e. sensor frequency, which is five according to the obtained results reported in Figure 2(a).

As the results show, at times 5, 10, 15, 20, and 25, the maximum probability of the scientist death equals one. At these times because of concurrency between time related behaviors in the system, there is a scenario in which the dangerous level is reported too late to the administrator and the scientist will die. At these times, the execution
<table>
<thead>
<tr>
<th>Period</th>
<th>sensorFreq</th>
<th>Pmax</th>
<th>Pmin</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.006519</td>
<td>0.005525</td>
<td>1.007259</td>
</tr>
<tr>
<td>2</td>
<td>0.5675</td>
<td>0.5654</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.4668</td>
<td>0.4651</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.54</td>
<td>0.3957</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.3488</td>
<td>0.3422</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.2763</td>
<td>0.2753</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.6016</td>
<td>0.2112</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0.4501</td>
<td>0.3172</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0.7184</td>
<td>0.4424</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>0.5944</td>
<td>0.4975</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>0.8245</td>
<td>0.6525</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.7832</td>
<td>0.5708</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>0.6798</td>
<td>0.6041</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1.007259</td>
<td>0.7232</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.7876</td>
<td>0.5708</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>0.8788</td>
<td>0.8040</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>1.007259</td>
<td>0.7232</td>
<td></td>
</tr>
</tbody>
</table>

The value of variable **scientistDeadline** is 10.

(a) The value of variable **scientistDeadline** is 10.

(b) The value of variable **scientistDeadline** is 12.

**Figure 2:** The maximum and minimum probabilities that the scientist eventually dies, when the sensor frequency changes.

sequence of the following messages is important and causes the special behavior.

- Checking the sensor value by the administrator (it is repeated periodically after 5 units of time)
- Changing the toxic level of the environment to a dangerous level (period is 5)
- Checking the environment by the sensor (Figure 2(a) shows the probability of the scientist death for different value of this period)
- Sending a message **die** to the scientist (after 10 units of time) when the environment is dangerous

This experiment shows that the exceptional timing behavior can be revealed by probabilistic performance evaluation. It is not possible to find this special behavior by the tools and techniques developed for TRebeca language at this moment.

In Figure 2(b), the value of variable **scientistDeadline** equals 12; the scientist has more time to be saved before being killed by the toxic environment. The maximum probability of the scientist death is not equal to one at times 5, 10, 15, 20, and 25, but because of concurrency between time related behaviors, there is a scenario in which the dangerous level is reported too late and consequently the maximum probability of the scientist death increases. There is an optimum value for the variable **checkingPeriod**, i.e. sensor frequency, which is ten in this experiment.

### 5 Related Work

**PRISM.** PRISM is a well-established and powerful model checker with a state-based input language. An input model of PRISM is composed of a number of modules which can share variables and interact with each other. PRISM is well equipped with theories and reduction techniques [HKNP06], but lacks high-level programming constructs like loops, and primary data structures like arrays, which makes modeling hard.
In contrast, PTRebeca provides high-level object-based programming features and asynchronous message passing, which makes modeling easier. So, in modeling we benefit from capabilities of PTRebeca, and in analysis we use the capabilities of the PRISM model checker.

**Modest.** Modest [HHHK13] is a high-level and convenient language for describing stochastic timed and hybrid systems. It supports loop constructs, structs and arrays, exception handling, and other advanced programming constructs. For the probabilistic timed fragment of Modest, model checking can be performed using a digital time semantics [HH09] or by a direct mapping to timed automata. Both approaches use PRISM as a backend model checker.

In contrast to Modest, PTRebeca supports object-based programming features, and follows the asynchronous message passing paradigm of actors, while Modest relies on synchronous message passing. Otherwise, the spirit, especially with respect to the analysis via PRISM, is similar.

**ProbMela.** ProbMela is a probabilistic version of Promela [Hol97]. The operational semantics of ProbMela is defined as an MDP [BCG04]. In [CB06], ProbMela is used as input language for the MDP model checker LiQuor which provides qualitative and quantitative analysis of LTL properties. There is also a mapping from ProbMela to the PRISM language, which makes probabilistic analysis possible [CBGP08].

PTRebeca is an event-driven and actor-based language whereas ProbMela is process-based. Both languages are asynchronous in spirit. We proposed a semantics of PTRebeca as TMDP (or PTA with digital clocks), enabling the analysis of timing and probabilistic behaviors of asynchronous systems.

**PMaude.** PMaude extends standard rewriting theories of Maude with probability [AMS06]. There is an actor extension of probabilistic rewriting theories for PMaude which removes non-determinism. A statistical technique is provided to analyze quantitative aspects of systems using discrete-event simulation. In comparison with PMaude, modeling asynchronous systems is more straightforward in PTRebeca language as it is an actor-based language. Also PTRebeca supports non-determinism in the model and there is no need to resolve it by assuming distribution on different choices of non-determinism. It is because of the probabilistic model checking facilities which are provided by PRISM.

**Summary.** In PMaude, probability distribution functions (rates and stochastic functions) are provided for modeling probabilistic behaviors. Also, PMaude implements stochastic continuous-time. In ProbMela, probabilities are drawn from discrete probability distributions, and passage of time can be modeled using a timer process. Modest enables a direct high-level modelling of PTA and more complex models. In all aforementioned languages, non-deterministic behavior can be modeled. In analysis, PMaude resolves non-determinism, and uses statistical model checking to verify properties which
results in inaccurate results. In the analysis of ProbMela and Modest, non-determinism is not resolved. Modest also provides the option of a digital clock semantics, which, just like we do here, is handed over to PRISM for model checking.

Our focus in designing PTRebeca has been on ease of modeling and efficiency of analysis mainly for asynchronous applications. To this end, we use discrete time model and discrete probability distributions. These decisions showed to be effective in modeling different applications that we have targeted. Moreover, resolving non-determinism by a discrete probability distribution generates inaccurate estimations, so, we avoided that by choosing TMDP as the semantics of PTRebeca. We were able to formalize the advance of time in our model using a single integer-valued variable. The language design of PTRebeca and its analysis approach is closest to the Modest approach, apart from the latter not being object-oriented and not being asynchronous by design.

6 Conclusion

In this paper we introduced the syntax and semantics of Probabilistic Timed Rebeca (PTRebeca) for modeling and verification of probabilistic real-time actor systems. As the model of time in PTRebeca is discrete, we decided to use discrete-time TMDP with an integer-valued time variable for the semantics of PTRebeca. PTRebeca models can thus be analyzed against PCTL, expected reachability, and probabilistic reachability properties. Our proposed approach is implemented as a part of Afra toolset [Reb]. PRISM is used as a back-end model checker for analyzing PTRebeca models. This is similar to the approach put forward for Modest, and we are therefore considering a direct connection between PTRebeca and Modest.

In addition to the benefits of using TMDP semantics for analysis of PTRebeca models, our technique is based on the actor model of computation where the interaction is solely based on asynchronous message passing between the components. Hence, the proposed semantics is general enough to be applied to similar computation models where there is message-driven communication and autonomous objects as units of concurrency, and there exists discrete probabilistic behaviors in the model such as agent-based systems.

Acknowledgement

The work on this paper was supported by the project “Timed Asynchronous Reactive Objects in Distributed Systems: TARO” (nr. 110020021) of the Icelandic Research Fund.

Bibliography


Model Checking C++ with Exceptions

P. Ročkai*, J. Barnat and L. Brim

Faculty of Informatics, Masaryk University
Brno, Czech Republic
{xrockai,barnat,brim}@fi.muni.cz

Abstract: We present an extension of the DIVINE software model checker to support programs with exception handling. The extension consists of two parts, a language-neutral implementation of the LLVM exception-handling instructions, and an adaptation of the C++ runtime for the DIVINE/LLVM exception model. This constitutes an important step towards support of both the full C++ specification and towards verification of real-world C++ programs using a software model checker. Additionally, we show how these extensions can be used to elegantly implement other features with non-local control transfer, most importantly the longjmp function in C.

Keywords: model checking, C++, exception handling, LLVM

1 Introduction

Widespread and regular use of formal verification methods in the general software development is one of the major goals in computer science. As a matter of fact, recent formal method research trends put a strong emphasis on direct practical applicability of verification results. A current example of this trend is the activity in the program analysis community and the Software Verification Competition [Bey14]. The strong drive to make formal method applications approachable by the general software development and engineering community highlights the fact that the most important factor of using formal methods in practice is their ease of use. Hence, formal methods must be applied at a level that software engineers and developers naturally work at – that is, in an overwhelming majority of cases, at the source code level.

There are multiple reason for this: not only working with program source code is natural for a software developer, source code also constitutes very precise notation, which is a natural match for a formal system. While it is true that semantics of programming languages are usually not rigorously specified, they do often attain a very high level of precision in their specifications. The main inconvenience of specifications of languages like C++ lies in the large volume of text, and consequently, large amount of facts. Nevertheless, the complex natural-language specs have a formal counterpart: compilers. While a C++ compiler is a very complex software system, the fact is that real-world compilers achieve an enviable level of agreement in their semantics, despite numerous optimisation passes they all implement.

Consequently, there is a natural tendency to build model checkers that can be applied to programs written in commonly-used languages: most importantly C, C++ and Java. Clearly, there

* Petr Ročkai has been partially supported by Red Hat, Inc.
are limitations to what a model checker can do: the problem it is tackling is, in general, firmly undecidable. In theory, this is a huge red flag – we are trying to solve a problem that we know for a fact cannot be solved. Nevertheless, a partial solution can still be immensely useful: after all, a software engineer often has to argue about properties of programs that are in general undecidable. In this case, all that matters is whether the instance at hand can be solved.

There is however another limitation, which is usually more important in practice: conformance to programming language specifications. In order to derive substantial utility from a model checker, it should implement a full programming language specification: the programs that software developers write and that they can run should be also valid inputs to a model checker. This is especially critical if we expect a seamless integration of model checking tools into a development workflow. The brunt of the problem at hand is that programming languages as specified are already very constraining – engineers in pursuit of more elegant and more maintainable code already skirt the boundaries of what is allowed in a particular programming language. Introducing substantial constraints to enable model checking is, in many cases, a non-starter.

This is especially a concern with C++, which is a relatively high-level language, with a long development history and widespread use. Some of the features the language offers are rather unpalatable (especially so in the model checking community), usually because they exhibit very complex semantics. While some of the problematic aspects can be conjured away by targeting a suitable intermediate language and hijacking a good existing compiler frontend – such as LLVM [LA04] (the IR) and its companion CLang (the frontend) – this is not the case with all such the features. A particularly hairy example is exception handling, which necessarily finds its way into the intermediate representation.

Besides their complicated semantics, which are already a formidable problem, they bring an entirely new phenomenon to model checking: non-local transfer of control. While not unapproachable, it complicates everything – and a modern software model checker is already complicated enough. It is easy to see how tempting it is to constrain the input language of the model checker to disallow exceptions. However, for the reasons expounded earlier, we firmly believe that it is very important to provide full coverage of language features in a model checker. This paper primarily presents our experience in implementing exception handling in DIVINE, an explicit-state model checker for C and C++ programs based on LLVM.

2 Preliminaries

DIVINE [BBH+ 13] is a general-purpose explicit-state model checker for safety and LTL properties. For LTL model checking, it uses an automata-based approach [VW86], reducing the decision procedure to a graph problem – namely detection of an accepting cycle in the state space graph of a program under verification. In order to tackle large graphs, it implements efficient parallel algorithms for both reachability (for safety verification) and accepting cycle detection (for LTL

1 Clearly, there are specialised projects where programming language semantics need to be severely constrained, whether it is due to formal treatment – this is sometimes the case with mission-critical software – or due to limitations of the hardware platform, a situation most often encountered in the embedded systems space. Nevertheless, in the latter category, increases in hardware capabilities of embedded systems is apt to reduce this gap between embedded and mainstream general-purpose programming.
model checking). Implementations tailored for both shared-memory and distributed-memory parallel computers are available, along with an assortment of memory-saving techniques. For more recent results in the field of parallel and distributed model checking, see e.g. [ELPP12].

Among other input languages, DIVINE can handle programs written in LLVM intermediate representation (LLVM IR). The main use-case for explicit-state model checking, and especially LTL model checking in this area is for unit testing of parallel programs. While explicit-state model checking per se (without the aid of some form of abstraction) cannot handle arbitrary IO behaviour, this is something that software engineers deal with all the time – testing cannot do that either. Of course, an ideal solution would overcome this problem as well – but we contend that this is not a serious obstacle in pragmatic use. However, there are two interesting things that an explicit-state model checker can do (and where testing struggles): asynchronous lock-based parallelism (which is an ubiquitous concern in contemporary C++ programming) and liveness (LTL) checking. Moreover, since LLVM is quickly becoming the lingua franca of software analysis tools [MFS12, CDE08], it is not unconceivable that an explicit-state model checker would be integrated into a larger abstraction/CEGAR-based decision procedure, in order to tackle the open-world angle in program verification.

Now if we have a model checker that can handle LLVM IR as its input, and a compiler frontend that can translate C++ into LLVM IR (and there are at least two such compilers, CLang and GCC) – we can compile C++ programs to IR (bitcode) and run a model checker on it. This appears to be very easy on the surface of it, but there are hidden complexities. First, in order to actually verify a program, it needs to be fully defined: in basically all cases, programs make use of the C and C++ standard libraries. Especially the C++ standard library constitutes a very substantial amount of code, and we cannot reasonably argue about program correctness if we don’t include this code. The libraries in turn make use of system-level APIs (in the cases we are interested in, this is mostly POSIX), which are fortunately fairly constrained and small, compared to the standard libraries themselves.

As we have argued above in Section 1, constraining the language that can be used with the model checker is an option of last resort, as it has substantial impact on its usefulness. A good way to approach this problem is to compile the libraries themselves into bitcode and bundle them with the program’s own bitcode, to form a nearly fully-defined LLVM program. The missing system-level APIs can be either provided as stubs (a practice commonly used in testing), or in some cases – where they constitute important part of program functionality – implemented in terms of a small number of model-checker provided primitives. To this end, DIVINE provides an implementation of both the C and C++ standard libraries, including the small number of changes required for compatibility with the different system-level API implemented by DIVINE. The overall structure of the libraries, with emphasis on exception handling (see also Section 3 for more details on this) is illustrated in Figure 1.

3 Exception Handling

Exception handling is an area where code generation needs to co-operate in order to implement correct language semantics. Since code generators are part of LLVM, but LLVM itself is programming-language-agnostic, the LLVM code generators need to provide a sufficiently generic
Figure 1: The various components involved in exception handling, and their interaction with execution and verification. The source code is first compiled using a suitable C++ frontend (clang or gcc) into LLVM IR. When building a binary for execution, the IR code is fed to a code generator and combined with common components (the standard C and C++ libraries), and with execution-specific components: libunwind and execution-specific parts of the C++ runtime support library (the personality routine and the libunwind-based stack unwinder). For verification purposes, the LLVM IR is instead combined with those same common components that have been converted into intermediate representation, and with verification-specific runtime functions from the DIVINE C++ runtime. The resulting bitcode file is then fed to DIVINE, using its LLVM subsystem to generate the state space and execute a suitable verification algorithm on that state space.
interface to allow implementation of efficient exception handling.

In all modern C++ compilers, zero-cost exceptions are the norm: the exception handling machinery imposes no overhead at all unless an exception is actually thrown. This means that the code generator is not allowed to insert special instructions for calls or for saving context when entering try blocks. In order to allow this sort of behaviour, all exception handling logic needs to happen at an exception throw time, and for this to be possible, a stack unwinder is required. The unwinder is platform-specific, and needs to understand the particular ABI and most importantly the layout of the program stack and individual stack frames. LLVM itself does not provide an unwinder library: it is usually provided by the operating system.

Unfortunately, the interface of the unwinder library is not entirely specified, and as such, it is also somewhat platform-specific. There are two major surfaces of the unwinder, each exposed to different part of the compiler/standard library duo. On one hand, the unwinder needs unwind tables in order to correctly unwind the stack. These unwind tables are generated by LLVM, since they reflect the high-level structure of individual stack frames, which is itself generated by LLVM. These tables end up being a part of the program text, i.e. they are stored in the executable image, and are as such a static part of the program. On the other hand, there is the “dynamic”, or runtime, interface of the unwinder library, which is exposed to the language runtime instead: when an exception is raised, the language runtime uses the unwind library and the unwind tables generated by LLVM to guide the exception handling process.

While C++ is the primary target of the exception-handling mechanisms in LLVM, care has been taken to make it sufficiently general to accommodate other language runtimes, as long as their exception handling works along the same general principles. The main requirement for an exception system to be compatible with LLVM is that it can use the same unwinder interface, or at very least that it can process the unwind tables produced by LLVM. On many platforms (all modern UNIX systems based on the ELF executable format), these unwind tables are in a standardised format, mandated by the DW ARF specification [DWA10]. Other platforms use different unwind tables, though.

Besides information about the structure of a stack frame, unwind tables contain information about how exception handling should process this particular stack frame. In programming languages with lexical scoping, lexically scoped variables cease to exist when their scope terminates: normally, this happens when a function returns. However, exceptions create a new way in which a lexical scope can cease to exist, namely that an exception is propagated through this scope upwards. As long as lexically scoped (local) variables are sufficiently simple (plain old data in C++ terminology), this is not a major problem: the stack is unwound, so the storage associated with those variables is automatically reclaimed. However, C++ and a number of other languages allows scoped variables of complex types, with associated destructors: code that the runtime guarantees is executed just before the variable is deallocated. Particularly in C++, this is widely used to implement reliable, automatic resource acquisition and release². Even though similar schemes have been proposed for C [Tur], they are usually implemented using set jmp and longjmp primitives, do not use any compiler support and therefore do not map to the LLVM

---

2 In the C++ community, this design pattern is known as RAII: Resource Acquisition Is Instantiation. Among other things, it is used to safely hold mutual exclusion locks, dynamically allocated memory and other non-composable resources inside functions that could experience non-local loss of control due to exceptions.
exception handling mechanism.

Nevertheless, LLVM as such has no concept of destructors, nor does the unwinder library. The language compiler needs to generate cleanup handlers, i.e. blocks of code that take care of calling any appropriate destructors, or performing other language-specific cleanup when a stack frame is torn down because the stack is being unwound. Moreover, the same mechanism is used for exception handlers: the main difference is that an exception handler stops the propagation of an exception, and its role is to deal with the exceptional situation: exception handlers correspond to the catch blocks attached to a try block.

In order to improve efficiency (at the expense of simplicity) of the unwinder, it has a concept of exception type: different types of exceptions can happen, and a particular catch block can handle only a subset of those exception types. Each call-site in each call frame possibly contains a cleanup handler, and a list of exception handlers. Deciding whether a particular exception handler can handle a particular exception type is deferred to a personality function: a language-specific callback provided to the unwinder. This personality function helps the unwinder decide, among other things, which handler to invoke for a particular exception type.

3.1 Mapping Exceptions to LLVM

Now that we have established the basics of how exceptions are implemented in general, we will look at how those concepts map to LLVM. The machinery provided by LLVM to handle exceptions consists of 3 instructions: invoke, landingpad and resume. The invoke instruction is like a call instruction, but it provides extra provisions for exception propagation: unlike call, it is a terminator instruction, i.e. it is always last in a basic block. It is also a branching instruction: it takes two basic block addresses as parameters corresponding to two branches – the first is taken upon a normal return from the function, the other is taken if an exception has been raised in the callee.

The invoke instruction co-operates tightly with the landingpad instruction: the basic block that the exception branch of invoke points to must begin (after any possible ϕ instructions) with a landingpad instruction, and the entire basic block is called a landing block. The landingpad instruction then encodes the list of exception handlers and whether there is a cleanup handler present, and which personality function to invoke for the corresponding callsite (invoke instruction). The syntax of the landingpad instruction is following:

```
<r> = landingpad <rt> personality <t> <pers_fn> <clause>*
<r> = landingpad <rt> personality <t> <pers_fn> cleanup <clause>*
```

<clause> := catch <type> <value>
<clause> := filter <array constant type> <array constant>

If the landing block is a cleanup one, the stack unwinder always transfers control to the landing block during the unwinding process, regardless of any exception handlers. If the landing block is

---

3 In upstream LLVM documentation, what we call a “landing block” here is referred to as a “landing pad”. The reason for this departure is that the original terminology makes it easy to confuse “landingpad” as an instruction and “landing pad” as a basic block.
not a cleanup landing block, it is only executed if some catch clause in the landingpad instruction matches the exception type (as decided by the provided personality function).\(^4\)

Since each invoke instruction only has a single landing block associated, this landing block is responsible for handling any and all catch clauses of the higher-level programming language covering the particular callsite. The return value of the landingpad instruction is crucial in deciding what action to take when the landing block is entered, and corresponds to the return value of the personality function. In other words, when the unwinder executes the personality function (which is part of the language runtime), it stores its return value, and provides this return value in the result of the landingpad instruction. Since the personality function has access to the part of the unwind tables generated from the landingpad instruction, it can communicate information encoded in the unwind table to the landing block itself. In the libc++ runtime, the personality function returns a tuple consisting of a pointer to the exception object itself, and a “handler switch value”, an integer which corresponds to the index of a relevant “catch” clause of the landingpad instruction, or a special value (-1) when no catch clauses match but a cleanup needs to be performed.

The code generated for the landing block then checks the handler switch value computed by the personality function, and transfers control to a cleanup or handler block accordingly. Finally, if the selected handler is a cleanup handler, the exception propagation (stack unwinding) needs to be resumed after the cleanup is done. This is achieved by the resume instruction, which expects as a parameter the same value that was returned by the corresponding landingpad instruction which interrupted the exception propagation.

Interestingly, there are no LLVM instructions for raising (throwing) exceptions. This is left entirely in the management of the language runtime, which needs to closely co-operate with the stack unwinding library anyway (the interface of the personality function is mandated by the stack unwinder).

\section{C and C++ Runtime Support in DIVINE}

As we have argued in Section 2, in order to verify real-world code, we need to provide an implementation of standard libraries: DIVINE provides libc, in form of bitcode that can be linked to (incomplete) bitcode produced by the compiler from the C program itself. While the implementation of libc is mostly complete, in some respects, it behaves differently from traditional OS-provided versions. Since the program that is being verified is not allowed to actually interact with the world, such function calls are implemented either as “stubs” possibly using non-deterministic choice, or they interact with DIVINE using a private DIVINE-specific interface.

The case of C++ is slightly more complicated. While many language features require no special runtime support (i.e. the same as C), there are some that do, most notably Run-Time Type Identification (RTTI) and exception handling. Besides those areas where library support code is required for language features, like in C, most C++ programs make use of the standard

\(^4\) Additionally, the filter clauses restrict the types of exceptions that can be propagated through the invoke instruction corresponding to this landing block, akin to how exception specifiers work in C++. If an exception is thrown and it reaches a filter clause of the appropriate type, a language-specific action is invoked. In C++, this action is user-specified, and defaults to terminating the program.
Consequently, there are two libraries that are required by virtually all C++ programs: the runtime support library, and the standard library. Multiple implementations of both exist\cite{DIVINE} – DIVINE ships with `libc++abi` for the runtime portion and `libc++` for the stdlib portion.

As far as RTTI goes, there are no special considerations with regards to model checking. The upstream `libc++abi` code can be used verbatim with DIVINE. Exceptions are more complicated, and are, coincidentally, a feature that is most often neglected in analysis tools and model checkers that work with C++ programs. Exception handling in C++ consists of three major parts: unwind tables, landing pads and exception handlers which are all generated by the compiler based on the input code, using special (although language-neutral) LLVM instructions: `invoke` and `landingpad` being the two most notable. Additionally, the C++ runtime library uses a CPU- and platform-specific stack unwinder and contains a language-specific personality routine. The personality routine makes use of the unwind tables generated by the compiler to guide the stack unwinder during an exception (see Section 3 for details).

An LLVM interpreter hence needs to provide a stack unwinder and an API to access the unwind tables, for use by the personality routine. In DIVINE, the unwinder interface is extremely simple, consisting of a single trap, `divine_unwind`. The language runtime can use `__divine_unwind` to remove a number of topmost stack frames from the stack of the current thread, returning control to the topmost remaining frame. If the active instruction in the target frame is an `invoke` instruction, control is transferred to its alternate destination basic block (a landing block), and the value passed to `__divine_unwind` is passed on to the personality routine of the landing block.

\section{Implementation}

We have outlined the mechanisms used by LLVM to implement language-agnostic exception handling in Sections 3 and 3.1. There are multiple points where DIVINE has to hook into those mechanisms in order to support exception handling in a particular programming language. While a substantial part of that support is language-agnostic, crucial pieces of infrastructure are part of the language’s standard library: in case of C++, this is `libc++abi` as explained in Section 4.

In a native code generator in LLVM, the information from `landingpad` instructions generated in the frontend is used to construct unwind tables. The format of those tables is platform- and architecture-specific. To read those tables, `libc++abi` uses the `libunwind` interface (originally specified as part of the IA64 C++ ABI). This interface is semi-standard, but no actual standardising document exists. Since the `libunwind` implementation is tied to the binary format of the executable, via the in-memory image of the unwind tables, it cannot be directly used in DIVINE. Likewise, it is tied to a specific architecture/platform via its knowledge of stack and register layout – another disqualifying feature. Therefore, `libunwind` needed to be replaced with a new implementation for DIVINE.

\footnote{The GNU compilers ship with `libstdc++`, which contains, as a subproject a runtime support library `libsupc++`. Clang ships with `libc++`. Depending on platform, a choice of either `libc++abi` or `libcxxrt` is available for use with `libc++`. An independent implementation is available from Apache Software Foundation under the name `libcxx`. Multiple compilers ship yet different implementations.}
struct R1 { R1() { /* ... */ } ~R1() { /* ... */ } };
struct RC { R1 r1; int *resource;
RC() : r1() {
  resource = new int[32];
}
~RC() { delete[] resource; }
};

int main() {
  try {
    RC res;
    // work with the resource...
  } catch (...) {
    // handle exceptions
  }
}

Figure 2: Example source code, for illustrating exception handling mechanisms. See Figures 3 and 4.

5.1 The libunwind interface

There were two basic options: either replicate the portion of the libunwind interface used by libc++abi, making it possible to use unmodified source for libc++abi – which sits on a higher level than libunwind. Conceptually, this is a tempting solution – the more of the library code is left intact, the more faithful the verification. There is a major downside though: the interface between libunwind and libc++abi is complex and intricate. This is especially true of the interface between the unwinder and the personality function: the unwinder uses the personality function as a callback, invoking it once for each active frame on the stack at the moment an exception is raised. The personality function uses a pair of platform-specific registers to pass the handler switch value and the exception pointer to the exception handler: it cannot invoke the handler itself, as the stack has not been unwound yet and the handler would end up running in the wrong context. For this reason, libunwind provides an interface to splice register values into the context of the exception handler to be invoked.\footnote{This is clearly implemented in a platform-specific fashion. If the registers are always saved on the stack, their stack images will be rewritten. If they are clobber-type registers, they can be written to directly and the unwinder will take care not to clobber them before transferring control to the selected exception handler. Other options may be available depending on platform.}

It would be in principle possible to implement this interface in DIVINE system space: each thread would need two special thread-local variables to hold these values, and the landingpad instruction would simply read those values and copy them into appropriate LLVM registers. The downside is extra space overhead – 16 bytes per thread, allocated even if no exceptions are currently active.\footnote{Those 16 bytes could be compressed away in most cases to a single bit, at expense of code complexity. However, system-space complexity is very costly, and complexity involved in addressing the state vector even more so.}
Figure 3: Example of an exception-handling process as it happens in the DIVINE runtime (see Figure 2 for the source code). The situation at the top of the flowchart corresponds to an out-of-memory condition in the program. Constructor of class RC was trying to obtain dynamic memory (using operator new), but the allocation request has failed. As a result, operator new is throwing an exception – the throw statement in the C++ source code of the implementation is translated to a _cxa_throw call, which uses _cxa_throw_divine to unwind the stack. The unwinder first uses _divine_landingpad to find an exception handler (which it finds in the call frame of the main() function, and any intervening cleanup handlers (there is one in the RC constructor itself). The unwinder proceeds to call the personality routine to obtain a handler switch value and passes the result to _divine_unwind, along with the address of the first cleanup handler. _divine_unwind removes stack frames up to the cleanup handler, which takes control and calls a destructor of the locally constructed R1 instance. Finally, when done, the cleanup handler invokes the resume instruction which continues the propagation up the stack, to the exception handler (the catch block in main()).
Figure 4: Example of an exception-handling process as it happens in the standard libc++abi process on 64-bit Linux (see Figure 2 for the source code and Figure 3 for comparison with DIVINE). The situation at the top of the flowchart corresponds to an out-of-memory condition in the program – as a result, operator new is throwing an exception – the throw statement in the C++ source code of the implementation is translated to a cxa_throw call. The cxa_throw implementation then calls into libunwind – the UnwindRaiseException function in particular. At this point, libunwind takes over control, looping over active stack frames. Each frame is examined by calling the personality routine with a UA_SEARCH_PHASE flag, in the context of the throw statement. In this phase, an exception handler is identified, but the stack is not yet unwound. In the next phase, the stack is actually unwound, and again, each frame is examined by a call to the personality routine. If a cleanup handler or the selected exception handler is found, it is invoked by returning URC_INSTALL_CONTEXT to libunwind (otherwise, URC_CONTINUE_UNWIND indicates that unwinding should continue with the next frame). Cleanup handlers return control to libunwind by invoking Unwind Resume.
Another downside is that this limits flexibility: while the LLVM exception mechanism is made to play nice with libunwind, it is flexible enough, at least in theory, to admit another approach to stack unwinding. Using this approach would mean changing the DIVINE system space to accommodate a different landingpad return type.

While this API/ABI issue has reasonable solutions, there is a more important issue at play. While libunwind understands the platform-specific portions of unwind tables, it provides no support for parsing the language-specific chunks. This means that libc++abi code itself has ABI-specific knowledge of the unwind table layout, needed to extract the exception type info and switch values. All libunwind does here is provide a pointer to the lsda (language-specific data area) portion of the unwind table for a given stack frame. In order to support this libc++abi code in its literal form, DIVINE would have to synthesise DWARF-formatted8 lsda areas from landingpad instructions. This is unpleasant, because it is a complex format designed for space efficiency, and the encoded tables are completely C++ specific, even specific to C++ on a particular platform. The only reasonable way to provide such tables would be to leverage pieces of the existing x86 (or x86-64) code generator to synthesise the lsda tables. LLVM, however, does not provide an interface to this functionality.

5.2 DIVINE-specific unwinding API

Both these issues in mind, we have chosen a different approach, which requires modifications to libc++abi, but can be implemented with just 2 new system-space builtins – one for querying metadata encoded in landingpad instructions, based on a stack frame reference (_divine_landingpad) and another for actually unwinding the stack (_divine_unwind).

This clearly requires some changes in libc++abi: one is the personality function, and the other is the actual _cxa_throw implementation: a call to this function is inserted by the C++ compiler at the site of a throw statement (along with some support code). While in the original libc++abi implementation, the personality function bears most of the burden (since libunwind does the stack search, calling out to the personality function as needed), this is reversed in the DIVINE implementation. Here, the personality function merely extracts the correct items from the exception header to pass on to the exception handler. The _cxa_throw implementation, on the other hand (and unlike in the libunwind version) unwinds the stack itself using _divine_landingpad. This builtin does not change anything, but provides the caller with landingpad metadata, using a simple integer indexing of stack frames. Negative indices start at the top of the stack, non-negative at the bottom. This makes it easy for the unwinder to walk through the stack one frame at a time, looking for an appropriate handler. When the handler is found, it can call the personality function (pointer to which is part of the landingpad metadata) and pass it to _divine_unwind along with the frame address it obtained from calling _divine_landingpad. The job of _divine_unwind is then simple enough: destroy all the frames above the one addressed and transfer control to the landing block associated with the active invoke instruction in the now-topmost stack frame. _divine_unwind also takes care of copying the value it obtained from its caller (in this case the return value of the personality

---

8 DWARF is a companion format to encode debug and other metadata in ELF executable images. A backronym “Debugging With Attributed Record Format” has been invented for it.
function) into the result of the corresponding landingpad instruction.

The implementation of \texttt{\_\_\_divine\_landingpad} takes advantage of the implicit garbage collection done by DIVINE, as it allocates the metadata block on heap. Since the block is neither flagged as a result of an \texttt{alloca} instruction, nor as a result of \texttt{\_\_\_divine\_malloc}, it is transparently retained as long as necessary without being flagged by the interpreter as a memory leak.

5.3 \texttt{setjmp} and \texttt{longjmp}

The C functions \texttt{setjmp} and \texttt{longjmp} can be used for non-local transfer of control, in a way somewhat similar to C++ exceptions. In fact, some C programs use those two semi-standard functions to implement somewhat crude exception handling in C. The purpose of the \texttt{setjmp} function is to save enough of the machine state to allow non-local transfer of control to the point in program where \texttt{setjmp} was called. The \texttt{longjmp} partner then, using a context saved by the \texttt{setjmp} call, restores the corresponding machine state. The state is exactly the same as it was right after \texttt{setjmp} call returned for the first time, with one exception: the return value of the \texttt{setjmp} call is altered in its second return, to make it possible to detect whether the return was a “normal” return or a \texttt{longjmp} return.

Clearly, exception handling based on \texttt{setjmp/longjmp} cannot be “zero-cost” – state has to be explicitly saved at the start of every \texttt{try} block, and possibly before any resource acquisition. The latter problem can be side-stepped by maintaining a separate “resource” stack \cite{Turb}, but even then, entering \texttt{try} blocks is fairly expensive. Nevertheless, robust C programs may choose this style of exception handling, since the runtime overhead can be outweighed by the programming benefits – especially due to fewer and simpler error paths to write, maintain and test. Finally, there are other uses for \texttt{longjmp} in programs, besides exceptional situations.

While \texttt{longjmp} is not nearly as widely used as C++ exceptions are, the reasons for supporting this primitive are similar, even if somewhat weaker. Fortunately, the primitives we have designed for C++ exception handling can be easily re-used in implementing \texttt{setjmp} and \texttt{longjmp} – since \texttt{\_\_\_divine\_unwind} can just as easily stop at a call instruction as it can on an \texttt{invoke} instruction, we only need minor extensions to the \texttt{\_\_\_divine\_landingpad/\_\_\_divine\_unwind} mechanism. The main difference between exceptions and \texttt{longjmp} is how the control flow at the point of \texttt{setjmp} is handled. The DIVINE-specific implementation of \texttt{setjmp} needs to be able to find out the program counter value of its enclosing frame, corresponding to the call instruction. This can be done by slightly extending \texttt{\_\_\_divine\_landingpad}, to provide the program counter value for call instructions in the stack (this does not alter the semantics of \texttt{\_\_\_divine\_landingpad} for \texttt{invoke} instructions in any way).

Finally, \texttt{\_\_\_divine\_unwind} needs to be extended as well, to allow the caller to specify where to restart the execution in the target frame – since \texttt{longjmp} is not above the corresponding \texttt{setjmp} in the call stack, a successful \texttt{longjmp} needs to change the program counter in the target frame, in addition to unwinding. Luckily, this is fairly easy, since the \texttt{\_\_\_divine\_unwind} caller can specify the program counter corresponding to the call site to unwind to. For normal C++ exceptions, the caller just puts in a 0, meaning no program counter adjustment (i.e. the semantics stay exactly the same) and \texttt{longjmp} passes in the program counter value obtained from a \texttt{\_\_\_divine\_landingpad} call done by the \texttt{setjmp} function.
6 Case Studies

Besides the simple fact of making model checking possible on a substantially wider class of programs, exceptions themselves are an interesting subject for model checkers: error paths are notoriously hard to test. With a model checker, however, it is easy to insert non-deterministic failures and check that the program behaves sensibly under all sorts of error conditions. Resource leaks are among the most common errors encountered in error paths, which makes the problem even harder to debug – resource leaks, especially memory leaks, require special tools to diagnose in a test, such as valgrind.

Since DIVINE can already diagnose memory leaks in LLVM inputs, checking error paths involving exceptions becomes a fairly easy task. However, error paths can contain more serious errors as well – especially in multi-threaded programs, where threads are not isolated from the effects of other threads failing to handle an exception, and the entire program may crash. Among the first issues that we have found using our new exception support in DIVINE is such a crash, in std::thread implementation in libc++, under out-of memory conditions.\footnote{The proposed patch that fixes the problem can be found in http://llvm.org/bugs/show_bug.cgi?id=15638 and the relevant source code in the file libc++/std/thread.} When a new thread is created using this standard C++ interface, most of its state is allocated in the newly-created thread, before user code is executed. Since this allocation can fail with an exception, and the libc++ implementation fails to install an exception handler in the context of this newly-created thread, the exception cannot be caught. In such cases, the C++ standard document requires the runtime library to call an “unexpected exception handler”, which, unless overridden by the user, terminates the application.

In order to fix this problem, we have moved the memory allocation code into the calling thread. To avoid synchronisation problems and possible resource leaks, this happens before the new thread is created – the calling thread allocates all the dynamic state for the new thread and passes it down as a parameter. This way, any exceptions related to resource exhaustion happen in the calling thread, in a context where users can control the scope and propagation of exceptions by wrapping the call to the thread constructor in a suitable catch block.

7 Conclusions

We have shown how to extend an explicit-state software model checker based on LLVM with support for exception handling, with focus on C++ exceptions. To ensure the viability of the approach described in the paper, we have created an implementation as part of the DIVINE model checker. Additionally, we have used this implementation to verify properties of C++ programs that make use of exceptions (either directly or via the standard library) and in the process found an exception-related bug in the libc++ implementation of std::thread.\footnote{According to [RFS’13], the ESBMC++ tool also supports C++ exceptions. Unfortunately, it appears that this support is so far largely theoretical: the current version (1.23) produces a spurious counterexample on a simple test case, taking a path through the code which disregards the fact that an exception has been raised.}

To the best of our knowledge, this makes DIVINE the first model checker to be able to verify C++ code with exceptions\footnote{The proposed patch that fixes the problem can be found in http://llvm.org/bugs/show_bug.cgi?id=15638 and the relevant source code in the file libc++/std/thread.}. The main contributions of the paper are twofold: first, the description of C++ and LLVM exception handling mechanisms in the context of model checking, and...
second the implementation derived from it: all relevant source code is freely available as part of the current DIVINE distribution.

Bibliography


A Constraint-Solving Approach for Achieving Minimal-Reset Transition Coverage of Smartcard Behaviour

Renaud De Landtsheer, Christophe Ponsard, Nicolas Devos
{rdl,cp,nd}@ctic.be, CETIC Research Center, Belgium

Abstract: Smartcards are security critical devices requiring a high assurance verification approach. Although formal techniques can be used at design or even at development stages, such systems have to undergo a traditional hardware-in-the-loop testing phase. This phase is subject to two key requirements: achieving exhaustive transition coverage of the behavior of the system under test, and minimizing the testing time. In this context, testing time is highly bound to a specific hardware reset operation. Model-based testing is the adequate approach given the availability of a precise model of the system behavior and its ability to produce high quality coverage while optimizing some cost criterion. This paper presents an original algorithm addressing this problem by reformulating it as an integer programming problem to make a graph Eulerian. The associated cost criterion captures both the number of resets and the total length of the test suite, as an auxiliary objective. The algorithm ensures transition coverage. An implementation of the algorithm was developed, benchmarked, and integrated into an industrial smartcard testing framework. A validation case study from this domain is also presented. The approach can of course be applied to any other domains with similar reset-related testing constraints.

Keywords: Model-based testing, Constraint Solving for Verification, Hardware in the loop

1 Introduction

Testing remains the major industrial approach for software validation and verification. However, with the development of model-based engineering and the use of Domain Specific Languages, testing is not any more disconnected from modeling activities. In this context, Model-Based Testing (MBT) has appeared as way to automate test generation from models [UL06, DSVT07]. Based on a model of the system under test, it provides the advantage of automated generation, ensuring complex coverage criteria, evolution of the test suites as the specification evolves. However, there are also a number of shortcomings: the effort required to build a model is also quite high and requires a deep change in the design process. Moreover, additional effort is also required to manage a translation from an abstract test suite to concrete test cases. There are also still language and tool limitations to cope with, especially when managing large models and generating tests from them [UL06].

In the context of this paper, we take a more focused approach which does not try to capture the behavior of the whole system but only specific behavioral aspects expressed in the widely industrially adopted formalism of state machines and which is part of the UML standard [Har87, Fow03].
Several coverage criteria for state machines have been proposed and are now considered as standard: transition coverage, state coverage, pair of transition coverage, pair of state coverage, path coverage, etc. They provide relevant trade-off points between the level of assurance provided by a successful test campaign, and the cost and time devoted to the creation and execution of a test suite. The overall length of the test suite and the individual cost of some steps of testing are the two factors influencing the time necessary for executing a test suite.

In this paper, we focus on state machine transition coverage which is quite popular. Our contribution is to take into account a frequent problem of the cost of managing transition under tests by driving the system into a well-defined state, for example by applying some "reset" procedure. The cost of this system reset can be quite prohibitive and should be minimized. We present an algorithmic approach to automatically generate test suites that minimize the number of resets that must be performed to execute a test suite. We consider that a reset must be performed between each test case. Our algorithm also minimizes the overall length of the test cases as an auxiliary cost criterion. So far, our algorithm exclusively supports the transition coverage criterion.

This problem was encountered in the context of smartcard testing. In this context, the cost of resets arises because it is performed either by flashing the program memory, or by switching to a new and fresh device in case the hardware flashing is disabled. Hardware flashing might be disabled for security reasons, as it might give access to the innards of the card. Those operations are an order of magnitude more time consuming than performing the software actions listed by the state machine implemented in the smartcard. However, this problem can be encountered in many other domains not only in control systems but also in information systems, for example in business workflow management systems where going to a known state can require complex database operations.

In this paper we will consider a running example from the smartcard domain: a fragment of the public Eurocard-Visa-Mastercard (EMV) specification [EMV08]. It is described in a relatively small state machine presented in Figure 1 driving the card lifecycle between initialization to more operational states. This system runs on smart card commonly encountered for debit and credit cards.

![Figure 1: The EMV state machine that governs smart cards](image)

Our proposed approach can be summarized by the following reasoning:

1. We consider that a test suite is a path in the state machine. Each test case is a fragment
of this path, and test cases are separated by hardware resets. The hardware reset is modeled as an extra transition in the state machine, each state but the initial one has a reset transition that leads to the initial state. Hardware resets are thus added to the purely software state machine in the beginning of our algorithm.

2. A transition coverage criterion is very close to the definition of Eulerian path. An Eulerian path in a graph is a path that takes each transition exactly once. Finding an Eulerian path in a graph has complexity $O(\#\text{Edges})$. The search for the test suite is therefore nothing else but searching for an Eulerian path, assuming such a path exists. The length is of course optimal because it is equal to the number of edges.

3. There can only be an Eulerian if the graph is Eulerian itself and this can be checked by simple conditions, mainly involving the degree of the nodes. There are two possibilities to cope with this, either to develop a specific algorithm derived from the one that finds an Eulerian path or to transform the graph into an Eulerian one. We have opted for the second approach because it seemed computationally more efficient, much more elegant, and less expensive to implement.

4. Transforming an ordinary graph into an Eulerian graph can be achieved by adding some edges to enforce the conditions that make a graph Eulerian. This operation allows for degrees of liberty. However, we can only add reset edges (from any state to the initial state) or duplicate existing transitions. We cannot add any other new transitions.

5. All added transitions do not have the same impact, especially adding reset transitions is very costly and one can prefer to add many ordinary transitions instead, this can be formulated as an optimization problem. The optimization problem is expressed as finding the set of transitions making a graph Eulerian while minimizing the cost criteria which is expressed as the weighted sum of the added transitions, with reset transitions having a much larger costs than normal ones. The optimal test suite is then simply the Eulerian path cut at the reset transitions.

6. On the technical side, this problem can be easily expressed as an Integer Programming (IP) problem, which can be solved efficiently using Integer Programming techniques (MIP solver).

The remainder of this paper is structured as follows. Section 1 formally defines the problem that we are solving in this paper and the notation used throughout the paper. Section 2 introduces the specific technological background of our approach, notably Eulerian graphs and integer programming. Section 3 presents our algorithm in full details. Section 4 illustrates the result of running our algorithm on our EMV case study as well as on a full set of benchmarks showing how it scales. Section 5 presents some related work. Finally, section 6 summarizes the paper and gives open issues and possible extensions of this approach.

2 Problem Statement

The problem solved in this paper is formally defined as follows:
Given a state machine,

- described by its states $S$ and its transitions $T$, $T \subseteq S^2$
- with initial state $\text{init}$, $\text{init} \in S$
- ensuring reachability (i.e. each state is reachable from $\text{init}$)

Find a set of paths, each path starting at $\text{init}$, such that:

- each transition of the state machine is taken at least once by one of the paths
- the number of paths is minimal
- the overall length of these paths is minimal

The two objective functions are strictly prioritized: the number of resets is minimized first, and then, keeping the same number of resets, the overall length of the test paths is minimized.

3 Technical Background

This section presents our approach. It first recalls necessary background on graph theory, Eulerian paths, and integer programming. It then presents some inspiring problems from graph theory and explains the intuition behind our algorithm. Throughout the paper, the concepts of state machine and graph are undistinguished, and denoted by any of these two terms.

3.1 Graph Theory and Eulerian Path

An Eulerian path in a graph is a path that takes each transition exactly once. An Eulerian cycle is an Eulerian path such that the end node of the path is the same as the start node of the path. An Eulerian graph is a graph with an Eulerian path. There is a classical theorem that links the Eulerian property to simple properties of a graph. It can be found in lecture books [Tru93]. We only recall the property itself and not its proof. A directed graph is Eulerian if:

- it is strongly connected: for each pair of node, there is a path leading from one to the another
- for each node, its in-degree is equal to its out-degree

As exception to the rule above: a pair of nodes called $\text{init}$ and $\text{end}$ might exist:

- the out-degree of $\text{init}$ is equal to the in-degree of $\text{init}$ plus one
- the in-degree of $\text{end}$ is equal to the out-degree of $\text{end}$ plus one
- there might not be a path from $\text{end}$ to $\text{init}$
Hierolzer’s algorithm finds an Eulerian path in \(O(#\text{Edges})\) in an Eulerian graph [Hie73]. The principle of the algorithm is to build a path starting from the init node such that the path takes edges at most once. The path does not necessarily cover all edges of the graph. The algorithm then iteratively completes the path with edges that were not taken. It proceeds by walking on the path until reaching a node that has edges leaving the node and that are not in the path yet. The algorithm explores a path starting from this new edge, and only takes edges that were not taken so far. This additional path is necessarily a cycle that comes back to the node where it started. This secondary path is then added to the initial path. The walk then proceeds on the completed path by first following the added section, and iteratively enriches the path with other missing edges. When the walk is finished, the path has been enriched with all the edges of the graph, it is therefore Eulerian.

3.2 Integer Programming

Integer programming is about solving problems of finding \(x\) such that:

- \(x \cdot c\) is minimal
- \(A \cdot x \leq b\)
- all values of \(x\) are integers

Where \(x\) is a vector of unknowns, \(b\) and \(c\) are vectors of constants, \(A\) is a matrix of constants, \(x \cdot c\) is the Cartesian product of \(x\) and \(c\), \(A \cdot x\) is the matrix product of \(A\) and \(x\), the inequality \(\leq\) holds for each row of the two vectors.

Efficient solvers are available for such problems [GLP, SCI, LPS, IBM]. Notice that the above IP problem only includes an inequality but can easily be extended to handle equality as well: equality \(p = q\) can be encoded by the conjunction of two constraints \(p \leq q\) and \(-p \leq -q\).

3.3 Inspiring Problems

Related problems are presented in [EJ73, Thi03]. Among them, the approach is notably inspired from the Chinese postman problem. The Chinese postman must take each street of the city at least once to deliver the mail, and wants to minimize his overall walking distance. This problem is efficiently solved by an IP-based approach, to duplicate the necessary edges in order to make the graph Eulerian. Out of the Eulerian graph, an Eulerian path can be computed efficiently.

Two different classes of edges appear in a related problem called the rural postman problem. This problem is similar to the Chinese postman problem except that some streets are actually rural pathways where no mail must be delivered. They can be used as shortcuts and might not be taken in the final path.

In our case, the graph is directed, and there are two classes of edges: reset transitions and other transitions. Reset transitions must be minimized first, and might not be taken in the final path. Other edges must also be minimized, as a secondary objective.
4 Algorithm

The global view of the algorithm is depicted in Figure 2.

- The first step transforms the initial state machine into an Eulerian state machine by inserting reset transitions representing hardware resets, and duplicating some transitions of the state machine. The state machine resulting from the transform is designated as the Eulerian machine, with transitions $T_e$ and states $S_e$.

- The second step finds an Eulerian path in the Eulerian state machine by applying Hierholzer's algorithm. It delivers a path that takes each transition at least once, and includes both transitions from the initial state machine and reset transitions. The algorithm is called so that the path starts at the init node.

- The third step cuts the Eulerian path at reset transitions. Each fragment is a path that starts at init and ends anywhere in the state machine.

We will focus here only on the first step as the two others are trivial. This first step enriches the initial state machine with reset transitions. The set of reset transitions added to the initial state machine is represented by $R$.

In order to express the duplication of the edges present in the initial state machine, each edge $e$ of the initial state machine gets an associated unknown $x_e$ that represents the number of time it will be present in the Eulerian state machine. Similarly, reset transitions also get an associated unknown $x_i$ that will be resolved.

The main constraint that we want to enforce on the $x$ variable is the one of degree equality: for each node, its in-degree should be equal to its out-degree. This constraint, taken from the definition of the Eulerian graph in Section 3.1 has an acceptable exception regarding the degree of nodes: a pair of nodes called the init and end node, respectively must enforce a slightly different equality: the init node has one more out-edge and the end node has one more in-edge. The init node of the exception is fixed to be the init node of the state machine, while the end node of the transition is not identified yet at this stage. To elegantly cope with this exception, we introduce a set of additional transitions to the initial state machine, called fictive transitions. An example of such a fictive transition is presented in Figure 3.

![Figure 2: Architecture of our test generation approach](image-url)
These fictive transitions will enable us to post the same equality constraint on each node, independently of the possible exception on this constraint, because we will take the fictive transitions into account in the equality constraints. In the Eulerian state machine, if the exception turns out to happen, one fictive transition will link the end node to the init node, and ensure that the degree constraint will be enforced. If the exception does not happen, no fictive transition will be present in the Eulerian state machine. Each node but the init node has a fictive transition that links it to the \textit{init} node. We represent the set of fictive transition by the set $F$. Each fictive transition $f$ gets also an associated unknown variable $x_f$.

We now review each condition that must be enforced by these unknowns from \textbf{Section 3.1} and from the problem statement in \textbf{Section 2} and show how they are translated into constraints of an IP problem:

- **Constraint C1** - \textbf{For each node, its in-degree is equal to its out-degree}. This translates neatly into an IP problem. Considering a node $n$ of the graph, we can find all transitions reaching (resp. leaving) the node, including the fictive and reset ones, they are denoted as $I_n$ (resp. $O_n$), we just need to post that $\sum_{e \in I_n} x_e = \sum_{e \in O_n} x_e$. This constraint is posted for each node $n$.

- **Constraint C2** - \textbf{There is at most one fictive transition in the state machine}. It is is posted on the set of fictive transitions $F$: $\sum_{f \in F} x_f \leq 1$.

- **Constraint C3** - \textbf{Each transition from the original machine is taken at least once}. It is to be posted for each transition of the original state machine, thus not for reset or fictive transition. For each transition $t$ in the initial state machine, we have to post the following constraint: $x_e \geq 1$.

- **Constraint C4** - \textbf{No transition can be taken a negative number of times}. This is straightforwardly translated into a MIP: for each transition $e$ in the state machine, we have to post that $x_e \geq 0$. This is partially redundant with the previous constraint, so that it need only be posted on reset and fictive transitions.

- **Constraint C5** - \textbf{The graph must be strongly connected}. This constraint is more complex. From \textbf{Section 2} we know that the initial state machine is quasi-strongly connected, as there is a path from init to each node. We only need to ensure that there is a path from init
each node to init. This is implemented by adding the necessary reset transitions. Consider
a state $s$ of the state machine. It must be connected to init. We thus consider $Z_s$ the set of
states reachable from $s$. If $\text{init} \in Z_s$, the constraint is entailed for $s$. If $\text{init} \notin Z_s$, at least
one reset transition must be added from one of the set if $Z_s$ to establish the connection to
init. Considering $R_{Z_s}$, the set of reset transitions from $R$ that start in a state of $Z_s$, the
corresponding constraint is as follows: $\sum_{r \in R_{Z_s}} x_r \geq 1$. Computing all the $Z_s$ can be performed
at once through the Floyd-Warshall algorithm [Flo62].

The cost criterion is composite and is expressed in terms of the IP problem as follows:

- The first cost criterion is that the number of reset transitions in the Eulerian path should be
  minimal. The associated objective function to minimize is: $\sum_{e \in R} x_e$.
- The second cost criterion is that the overall length of the test case should be minimal. The
  associated objective function to minimize is: $\sum_{e \in T} x_e$.

Combining the two objective functions into a single one is performed by summing them with
a high weighting on the first criterion. This weighting is computed to be strictly bigger than
the maximal value of the second criterion. We selected the value $\#\text{Edge}^2$, where $\text{Edge}$ is the
set of transitions in the initial graph. This is higher than the number of edges in the Eulerian
path because the worst case would be that taking one more transition of the state machine re-
quires crossing the whole state machine again. Other values can be used, but the outcome of the
optimization will be the same, for the $x$ variables.

The result of the IP problem is the number of times each transitions must be represented in the
transformed state machine to make it Eulerian with minimal cost.

The Eulerian state machine is elaborated from the initial one, the reset transitions and the
computed $x$ values: the states of the Eulerian state machine are the ones of the initial one, and
each transition $t$ of the initial state machine, or from the set of reset transition is present exactly $x_t$
times in the Eulerian one. The potentially identified fictive transition is not added to the Eulerian
state machine because it was only a modeling artifact to ease the encoding of the problem into an
IP problem. It can be omitted in the Eulerian state machine without losing the Eulerian property
of this machine thanks to the exception of the definition.

5 Implementation and Validation

5.1 Implementation

An implementation of this approach has been developed in Python and relies on several open
source libraries such as NetworkX and Coopr [Net, San]. NetworkX is a Python software pack-
age for the creation, manipulation, and study of complex networks. It offers a lot of useful
features such as graph generators, network structure and analysis measures, and graph drawing.
Coopr is a collection of Python software packages that supports a diverse set of optimization ca-
pabilities for formulating and analyzing optimization models. In particular, we used the Pyomo
package which supports mathematical modeling of integer programs in Python. Pyomo enables
to define symbolic problems, create concrete problem instances, and solve these instances with
standard solvers. We used the GLPK solver for the MIP resolution [GLP].
The prototype is fully integrated in a test management toolset in a smart card context [DPD+12]. Figure 4 shows the tool with the FSM editor open as well as a generated test scripts. Test scripts can easily be generated from those FSM as each transition is directly referencing a command with parameters. Special scripts are also available (e.g. for the reset transition).

![Figure 4: Prototype integration inside the STMicroelectronics test tooling](image)

5.2 Illustration on the EMV Case Study

For the EMV model presented in Figure 1, our tool finds a single test case that alone ensures the transition coverage criterion. No reset transition is needed here because the state machine is strongly connected from the start. The run time was around a tenth of second on a 2GHz machine. The single test case has 16 transitions. It is described in Figure 5.

5.3 Benchmarking

We have performed some benchmarking of our algorithm, both to check its efficiency and scalability, and to try to measure how efficient it could be with respect to the cost criterion. Benchmarking the cost criterion is actually only aimed at checking that minimizing it gives some good result and we know that our algorithm is optimal with respect to it.

Our benchmarking is based on randomly generated state machines with specific characteristics. They are generated randomly so as to reach a given number of nodes and transitions. We have swept the two dimensional space of number of nodes and transitions so as to give a wide view of the efficiency of our algorithm. The number of nodes range from 4 to 84 while the number of transitions ranges from 4 to the maximal connectivity the size of these state machines. There are 2268 test cases. The machine used for the benchmarking is a dual-core Intel 2,53 GHz with 3,5Gb of RAM running Microsoft Windows.
We have measured a set of parameters, both related to the generated test state machine and to the behavior of our algorithms, namely:

- The number of nodes that have no access to the initial state (those will trigger the need for inserting reset transitions)
- The diameter of the graph; to compare it to the length of the generated test suite
- The run time and memory consumption of our algorithm
- The number of test cases found in the generated test suites which is equal to the number of resets plus one
- The overall length of the generated test suite: this is the summed length of the test cases

Figure 6 reports the run time and peak memory consumption with respect both to the number of transitions and the number of nodes. Surprisingly, the run time seems to be nearly insensitive to the number of nodes. It seems to be more or less linear with respect to the number of transitions of the state machine. There is an exception in the run time, which is a graph with 45 edges and around 100 transitions which can either be a very badly shaped test case, or an external factor (e.g.: antivirus) which could not be reproduced. The peak memory consumption seems to be also linear with respect to the number of transitions, and seems to be much less dependent on the number of nodes.

The main objective function is the number of reset transitions. The left graph of Figure 7 presents the number of reset transitions generated by our algorithm with respect to the number of nodes that have no path to the initial state. These are the nodes that potentially require the insertion of a reset transition in order to make the graph Eulerian. We have drawn the 1:1 line on the graph to enlighten two phenomena. First, the number of resets is often lower than the number of nodes that have no path to the initial state. This is because the algorithm manages to visit them in a row, as they form a chain towards some "dead end" node. Second, there can be more resets

Figure 5: Single test trace elicited for the EMV state machine
inserted than the number of such nodes because such dead end node can have several incoming arcs; hence it requires as many resets to be inserted to ensure that the graph is Eulerian.

The right graph of Figure 7 presents the length of the generated test suite with respect to the number of edges of the graph. We can see that the dependence is mostly linear, except for small numbers of edges, where it might be higher.

Figure 6: Run time and peak memory consumption of our algorithm on our benchmark suite

Figure 7: Number of generated resets wrt. number of nodes without path to init (left) and suite length wrt. number of edges (right)
Our graphs are generated randomly, so that, in the statistical sense, our benchmarks are only valid assuming a random set of graphs. Extreme cases can likely be found, such that the algorithm behaves very inefficiently. Nevertheless, these benchmarks give a good overview of the average behavior of the algorithm.

6 Related Work

Finite state machines are a formalism that is implicitly or explicitly present in a number of model-based testing approaches. We focus here on approaches with explicit FSM descriptions.

- The commercial QTronic tool suite [Ver] provides a graphical FSM editor. The FSM description can also be coupled with a richer JAVA-inspired language. Considering only the FSM part, it supports not only state and transition coverage but also more complex coverage criteria like transition pairs (all pairs of input/output transition to a state). However, it does not support the management of the cost of special transitions like resets. The underlying technology is based on algorithms, including symbolic state space analysis, constraint solving, and combinatorial optimization. A more complete evaluation of this tool can be found in [And10].

- SmartTesting relies on UML class and state diagrams to model the system under test [Sma, BGLP08]. More complex behaviors can be specified using UML OCL (Object Constraint Language). The test generation relies on a set-constraints based solver. It does not cope with reset costs.

- The UML-B plug-in for RODIN enables the specification of FSMs in graphical form with Event-B as underlying formalism [SB08, Abr10]. Model-based test generation is currently being developed based on the ProB model-checker which is implemented in SICStus Prolog and uses co-routining and finite domain constraint solving [LB08]. Transition costs are also currently not taken into account.

Other research has addressed the problem of generating reset-minimal test suites in the larger scope of generating a checking sequence and taking into account the notion of distinguishing sequences. A first work proposed some heuristics [UWZ97]. Hierons improved them based on optimizations of the state recognition sequences and their use to construct the test segments. This resulted in shorter checking sequences produced from minimal, completely specified, and deterministic finite state machines [HU02]. An improved algorithm was more recently proposed: it produces minimal length sequences in its class and does not require the FSM to be strongly connected [HU10]. The approach is totally different and does not rely on the notion of Eulerian path. Besides, we did not consider distinguishing sequences. The notion of distinguishing sequences is not relevant in the smartcard domain where a separate oracle is available (emulator implementation). However, in a more general context, guaranteeing distinguishing sequences would be relevant.

MIP-based tools have also been used in the context of test case generation. [NSZ07] presents how scheduling problems can be solved by means of a MIP and how the generated schedules can be used as test cases for real-time systems. They furthermore present how the randomization
introduced by some MIP solvers on the generated schedules can lead to test cases exhibiting similar randomness.

An MIP-based tool able to statically check meta-properties of the state machine such as disjointness and exhaustiveness of transitions is described in [OQL08]. This work relies on a state-machine model manipulating numerical data. Their solver is also able to identify sets of values to inject in such state machine to lead it towards some pre-defined states. This can be used to generate test data that should bring the state machine into some predefined state. Iteratively, one should generate a single test case for each state through this procedure. Our approach does not encompass the decoration of state machine by condition and numerical state variables, but it provides test suite generation approach enforcing a coverage criterion for the whole test suite.

Consequently, we believe that the work described in this paper is an original algorithm for generating a checking sequence minimizing the number of reset transitions used.

7 Conclusions and Perspectives

This paper has presented an original algorithm that generates a test suite for a state machine using a constraint solving approach. The test suite ensures that each transition of the state machine is taken at least once, and guarantees to include as few test cases as possible, to keep the number of resets as low as possible. It furthermore ensures that the overall length of the test suite is as small as possible. The approach is original because it is based not on an exploratory approach such as depth first search, but on a graph transforms approach. It first transforms the state machine into an Eulerian one where such a test suite can be found straightforwardly. The transformation is computed efficiently with the help of a MIP solver.

The insights of this approach are interesting for the following reasons: despite the apparent complexity of the problem, the proposed solution is elegant and can easily be implemented. Its general performance is excellent and scales well.

An apparent limitation of our approach is the use of flat FSM while real world examples are expressed as parallel and hierarchical decomposition. For example, smartcards are expressed as the combination of multiple agents with specific behavior. They can be either generic (e.g. the presented EMV agent for the lifecycle, a PIN management agent, ...) or application specific. However, several algorithms exist to translate such problems to a flat FSM [DPC+14]. The resulting FSM is more complex but our algorithm scales well.

Another area of extension would be to consider the notion of distinguishing sequences in order to detect invalid test executions without any external oracle. However, in our industrial case an oracle was available as the test sequence is being played both on a simulator and an implementation, including in hardware-in-the-loop mode which takes long reset time and which motivated this paper.

Acknowledgements: The research leading to the results presented in this paper has received funding from Wallonia through the project INOGRAMS (convention nr 7171). We also thank STMicroelectronics Belgium for providing us the case studies and allowing us to publish the result of this research.
Bibliography


Research Ideas
Towards the Verification of Bidirectional Railway Models in CSP

Phillip James1, Faron Moller1, Hoang Nga Nguyen2, Markus Roggenbach1, Steve Schneider3, Helen Treharne3

1Swansea University; 2University of Nottingham; 3University of Surrey

Abstract: We develop proof support for bi-directional railway modelling.

Keywords: Railway verification; CSP; model checking.

Formal verification of railway control software has been identified as one of the “Grand Challenges” of Computer Science [Jac04]. In a number of publications [MNR+12, MNR+13, JMN+14b, JMN+14a] we have – in cooperation with our industrial partner Siemens Rail Automation UK – developed a scheme-plan verification approach based on the formal method CSP-B. Characteristics of our approach include: a railway Domain Specific Language (DSL); model transformations between DSL representations and formal models; general abstractions on the DSL level proven to be correct relative to the formal models; and ongoing implementations in our OnTrack tool [JTT+13]. In this, uni-directional scheme-plans have been our main focus to ease both modelling and verification. In this paper we focus on how to extend our approach towards the verification of bi-directional scheme plans, for example, an end station as shown in Figure 1. Here, trains can move from the entry track with Signal 5 to the two platforms A and B; furthermore, trains can leave platform A and platform B towards the Exit track. In our example, track TC6 is used in opposing directions: one direction for moving into platform A, and one for moving out of platform A.

Uni-directional train control systems are considerably simpler than their bi-directional counterparts. Uni-directional systems have to prevent the setting of a route that is currently in use, i.e., a train is travelling on it. In Figure 1 an example of a route would be the sequence of units ⟨TC9, TC8, TC7, TC6, TC3⟩ from signal S5 to signal S7. Uni-directional systems further have to prevent conflicting routes from being set at the same time. Routes are said to be conflicting when they share a unit. In Figure 1, e.g., the route from signal S5 to signal S7 and the route from signal S5 to signal S8 are conflicting, as they for instance share track TC9. Bi-directional systems additionally have to deal with the direction of travel. This leads to the new challenge of preventing opposing routes from being set at the same time. Routes are called opposing when they share a unit but for use in opposite directions. In Figure 1, e.g., the route from signal S5 to signal S7 and the route from signal S2 to the exit track are conflicting, as they use TC6 in opposite directions. In interlocking design, this challenge is addressed by route locking and release of units in sequence behind the train (therefore sometimes also called sequential release).

Recently we have extended our modelling approach for bi-directional scheme plans [JMN+14a, JMN+14a] to include directional information which models sequential release. This increases code complexity (as events have to take care of directions). Also, in model checking, the state space becomes considerably larger. E.g., our single junction [MNR+13] using uni-directional modelling has 8,646 states, however has 196,284 states when using bi-directional modelling (both models include two trains), showing that our new bi-directional models are of a larger complexity. The single junction under discussion consists of 15 tracks, one point, and six signals. For
Towards the Verification of Bidirectional Railway Models in CSP

Figure 1: An end station with two platforms.

comparison: our bi-directional model of the end station shown in Figure 1 has 32,369 states. The end station consists of eight tracks, one crossing, four points, three route signals and two fixed signals.

In the unidirectional world, we developed three abstractions [JMN+14b] that were key for verification: finitisation, covering, and topological abstraction, without which model checking large models is not feasible. Here we focus on the finitisation abstraction for bi-directional rail systems.

Safety in our models is dependent on the number of trains that are introduced into the model. This motivates the following definition of safety.

**Definition** Let $\text{ERROR} = \{\text{collision, derailment, runthrough, grantedRouteOccupied, offRoute}\}$ be the set of errors of interest. We make the following definitions.

- A scheme plan $SP$ is $n$-free (for $n \in \mathbb{N}_{>0}$ and $e \in \text{ERROR}$) iff $e$ never occurs in a run of our bi-directional model of $SP$ involving at most $n$ trains.

- A scheme plan $SP$ is safe iff it is $n$-free for all $n \in \mathbb{N}_{>0}$ and $e \in \text{ERROR}.

The contribution of this paper is to provide the technique of finitisation for bi-directional railway models, i.e., to reduce the proof of safety to a proof of $2$-e-freedom for $e \in \text{ERROR}$. In particular, we show the following.

**Theorem 1** A scheme plan $SP$ is safe if it is $2$-collision-free, $1$-derailment-free, $1$-runthrough-free, $1$-grantedRouteOccupied-free and $1$-offRoute-free.

The proof of this theorem is by induction over the length of a run of our model. Here, we have to argue about each line of the code. Due to the higher complexity of the code, this proof is considerably longer – due to the necessity to argue about sequential release, it is also more difficult than its uni-directional counterpart.

**Theorem 2** If a scheme plan $SP$ is $l$-free then $SP$ is $k$-free for any $k < l$.

**Corollary** A scheme plan $SP$ is safe if it is $2$-free for all $e \in \text{ERROR}.

This corollary allows us to reduce the safety proof for a scheme plan to one model checking run based on two trains. Based on this technique, we successfully proved our single junction as...
well as the end station from Figure 1 to be safe (when equipped with appropriate control and release tables).

The bi-directional modelling is of interest for the wider community since it is a step towards modelling of the European Train Control System, which is also based on sequential release. Similarly, the theoretical result of finitisation is of interest, as it provides further justification for applying bounded model checking to rail control systems. It is future work to adapt covering and topological abstraction for bi-directional modelling as well.

Acknowledgement The authors would like to thank Simon Chadwick, Siemens Rail Automation, for helpful discussions on sequential release, and Erwin R. Castesbeiana (Jr.) for pointing out the right directions.

Bibliography


Efficacy Measurement of Early Intervention Techniques

Dave Donaghy$^1$ and Tom Crick$^2$

$^1$ dave.donaghy@hp.com
HP Bristol, UK

$^2$ tcrick@cardiffmet.ac.uk
Department of Computing
Cardiff Metropolitan University, UK

Abstract: Compiler technology has, for some considerable time, been sufficiently advanced that individual programmers are able to produce, in reasonably short periods of time, tools that might aid with the development process in novel ways: for example, one can easily produce a C compiler tool that will detect uncommon uses of integer arithmetic (such as the rare multiplication of values that are commonly only added) and flag such uses as potential errors.

However, there is currently no convenient way to measure the efficacy of such techniques: where one might assume that uncommon uses of integer arithmetic might be erroneous, we do not have a way of measuring the cost saving associated with the potential early detection of occurrences of such things.

We present a method of measuring the efficacy of a single early intervention, based on the replaying of previous executions of a compile-build-test cycle. This measurement process allows us to identify the software errors that were introduced during an original development and subsequently fixed; additionally, it allows us to identify the subset of such errors that would have been identified by the early intervention. By these means, we can take an existing historical record of a development, and extract from it meaningful information about the value of a proposed new early intervention technique.

Keywords: Verification, Software Engineering, Efficiency, Version Control Systems, Repository Mining

1 Introduction

It is possible for software developers to utilise all manner of tools that will analyse practically any aspect of their source code and report on it in any way they choose; while this freedom will allow arbitrary invention on the part of the software developer, it might also allow construction of analysis tools that seem effective, but in reality are not; additionally, individual developers might have different ideas of what constitutes an effective tool [ZVDv08]. It would be useful, therefore, to formalise two separate ideas:

1. What constitutes effectiveness in the realm of compilation tools?
2. Is my tool effective?
2 Robust Efficacy Measurement

Some notions of tool effectiveness during code development and compilation might be as follows:

1. Will my tool make my code better?

2. Since the beginning of my current project, how many errors would have been detected using my tool, which was not in use (or in fact conceived) at the time?

Clearly these two criteria have been written deliberately to highlight the difference between objective, measurable criteria and subjective, hard-to-measure ones. We thus propose (and suggest ways for answering) the following question: *how can we phrase objective questions about measuring the effectiveness of tools run at the time of software compilation?*

3 Retrospective Measurements

We often have access to massive amounts of historical data in the form of a source-code repository [BKPS97, ZZWD05], such as those used by Subversion or Git (among many other tools used for such purposes). Often, though, while these repositories provide the ability to take time-based snapshots of workspaces that “work” (in some loose sense) at a given point in time, we may not take full advantage of the information in them [BRB+09, SK12].

Imagine that we have access to a new compiler option that will allow us to simply prevent compilation of code where a certain kind of error is detected. Assuming we have access to full historical information in our source-code repository, perhaps from repository mining, we can simply re-run our compile-test cycle with the new tool in place, and see which errors would have been identified before their actual detection time. Again, with full historical information, we can identify *how much earlier* each would have been identified, and then make judgments about the benefits of the new tool.

The remaining question, then, is this: if we *do not* have full historical information (and it is likely that we do not), then how much information do we need to judge the efficacy of our new techniques?

4 Open Questions

The following questions may be answered by constructing suitable experiments with data from historical repositories:

1. What build system changes are necessary to allow the retrospective addition of compiler tools to the build cycle?

2. What artefacts must be stored in a repository in order to reproduce compile-build test cycles and identify all errors?
3. How often must snapshots be taken in order to identify the points in time where errors were identified and fixed?

4. What is the potential for false alarms raised by error-checking tools in this proposed system?

Bibliography


No-Test Classes in C through Restricted Types

Dave Donaghy\textsuperscript{1} and Tom Crick\textsuperscript{2}

\textsuperscript{1} dave.donaghy@hp.com  
HP Bristol, UK

\textsuperscript{2} tcrick@cardiffmet.ac.uk  
Department of Computing  
Cardiff Metropolitan University, UK

\textbf{Abstract}: Object-oriented programming (OOP) languages allow for the creation of rich new types through, for example, the class mechanism found in C++ and Python (among others).

These techniques, while certainly rich in the functionality they provide, additionally require users to develop and test new types; while resulting software can be elegant and easy to understand (and indeed these were some of the aspirations behind the OOP paradigm), there is a cost associated to the addition of the new code required to implement such new types. Such a cost will typically be at least linear in the number of new types introduced.

One potential alternative to the creation of new types through extension is the creation of new types through restriction; in appropriate circumstances, such types can provide the same elegance and ease of understanding, but without a corresponding linear development and maintenance cost.

\textbf{Keywords}: Verification, Restricted Types, Compilers, Plug-ins

\section{Introduction}

Object-oriented programming (OOP) languages allow for the creation of rich new types through, for example, the class mechanism found in C++ and Python. However, it might be possible to obtain some of the gains of such techniques without the associated overheads in cost.

\section{Development Cost of New Types}

In an object-oriented development environment, it can reasonably be said that all software is encapsulated as methods on various types; indeed, Java, for example, requires that all executable code be written as type methods, allowing for the notion that static methods are still a kind of type method.

At the very least, then, the development of new types has some cost (and in particular, some financial or resource cost) associated to it. While we do not intend to directly measure this cost, a fair starting assumption might be that is linear in the number of new types introduced.
3 Restricted Types

One potential alternative to the creation of new types through extension is the creation of new types through restriction [NSPG08]; in appropriate circumstances, such types can provide the same elegance and ease of understanding, but without a corresponding linear development and maintenance cost.

As an example, consider an integer counter, intended to represent the number of occurrences of a certain event: the operations one might like to have on such an entity can be described as follows:

1. Create a new counter, with a value of zero.
2. Increment the counter by one.
3. Compare the value of the counter against a given integer.

Note that we might want to describe such operations explicitly, with the assumption that all other operations (for example, multiplying the counter by 8, or setting bits 2, 3 and 7), are disallowed.

One could clearly create such an object simply (and elegantly) in C++ or Java using a class construct, but the point here is that creation of such a new type would involve new, deployable, testable software with a non-trivial associated cost; a counter such as this is, mathematically and naturally speaking, a special kind of integer, and therefore we already have all the required software (built into the hardware and run-time environment) that we need. In particular, what we really need is a constraint: we must promise not to use disallowed “non-counter” operations on counters.

4 Open Questions

We can ask the following questions to frame future work in this area:

1. What existing common (or indeed uncommon) types naturally present themselves as restrictions of existing types, either built-in/primitive types or other existing types?
2. What amount of software is involved in the definition of those types, for example appropriate compiler/toolchain support? (e.g. [ANMM06, NS07, MME+10, GCC10, LLVM14])
3. How can we ensure that these restrictions, especially as compiler plugins, are harmless? [Nys11]
4. (Harder) What financial cost has historically been involved in the creation and maintenance of those types?
5. What proportion of that cost might be saved by new techniques for developing restricted types?
Bibliography


Physical Type Tracking through Minimal Source-Code Annotation

Dave Donaghy\textsuperscript{1} and Tom Crick\textsuperscript{2}

\textsuperscript{1} dave.donaghy@hp.com
HP Bristol, UK

\textsuperscript{2} tcrick@cardiffmet.ac.uk
Department of Computing
Cardiff Metropolitan University, UK

Abstract: One of many common artefacts of complex software systems that often needs to be tracked through the entirety of the software system is the underlying type to which numerical variables refer. Commonly-used languages used in industry provide complex mechanisms through which general objects are associated to a given type: for example, the \texttt{class} (and \texttt{template}) mechanisms in Python (and C++) are extremely rich mechanisms for the construction of types with almost entirely arbitrary associated operation sets.

However, one often deals with software objects that ultimately represent numerical entities corresponding to real-world measurements, even through standardised SI units: metres per second, kilogram metres per second-squared, etc. In such situations, one can be left with insufficient and ineffective type-checking: for example, the C \texttt{double} type will not prevent the erroneous addition of values representing velocity (with SI units \textit{metre per second}) to values representing mass (SI unit \textit{kilogram}).

We present an addition to the C language, defined through the existing \texttt{attribute} mechanism, that allows automatic control of physical types at compile-time; the only requirement is that individual variables be identified at declaration time with appropriate SI (or similar) units.

Keywords: Verification, Software Engineering, Type-Checking, Units, Compilers, Plug-ins

1 Introduction

Large (and indeed small) software systems typically track data, stored in a variety of different types; in fact, “bytes in, bytes out” is a fairly accurate description of a massive portion of the functionality of large software systems. While this is clearly true, it is of course an extraordinary over-simplification: the nature of the data we track through software systems ultimately maps all of the data that humankind has, can, or ever will, encompass.

It is not much of an understatement to say that tracking the types and content of these data represents the whole job of software development. Indeed, whole paradigms (for example, object-oriented development) may be thought of as addressing this one significant issue. However, such paradigms, while rich and functional, can also be cumbersome.
2 Simple Techniques for Data Tracking

In certain scenarios, the nature of the data we track may make it amenable to simpler representation: from a mathematical point of view, while a C++ class representing a command-line instruction to be parsed and executed is not a mathematical object in any useful sense; whereas a C++ object representing the distance from a geographical point to another most assuredly is.

Nevertheless, it is common to either use the same complex, powerful techniques to track these mathematical objects as to track non-mathematical ones; or indeed not to effectively track them at all. To contrast that idea, it might be possible to track these fundamentally mathematical items in C-like languages in ways that allow minimal additional effort at development time, and no maintenance effort at all.

For example, there is no meaningful way in which one can (or should) add 10 metres to 20 kilograms [Fos13]; there is, however, a way in which we can multiply the two: the result has a value of 200, and a unit of “kilogram metre” (one has to understand slightly more physics that the authors to recall the precise physical meaning of a unit of “kilogram metre”, but one certainly exists) [Bur14]. Units-of-measure are to science what types are to programming; over the years, a number of attempts have been suggested to extend programming languages with support for static checking of units-of-measure [Ken97, EB02, Ken09], alongside bespoke languages (e.g. Frink [Eli08]).

3 Implicit Type Operation and Restricted Types

Imagine, then, that a variable declaration can be tagged with a unit; that two variables can be arbitrarily multiplied; that two variables can be added if and only if their underlying units match exactly. With these abilities in place, we could simply manipulate all physical types (kilograms, metres, farads etc) as built-in numerical types, without resorting to the complex class mechanisms that we might otherwise need.

We might call such a declaration a restricted type: being based on a built-in type (or indeed, any arbitrary type) it is then further restricted by constraining the operations allowed on it: no additional software is written to define the type, and the only distinction between it and the type from which it is defined is this: that with the new type, either the software compiled and behaves identically, or it does not compile at all [DC14]. While similar features exist for certain programming paradigms (for example, functional e.g. Haskell), there are significant opportunities for their implementation in the languages commonly used for industrial software development. Such a feature could improve the code quality for critical systems, as well as detecting errors earlier, lowering software development costs.

4 Open Questions

We can ask the following questions to frame future work in this area:

1. Which existing definitions of complex types (for example, C++ classes) can be replaced with restricted types?
2. By how much might software development effort be reduced using such techniques?
3. How much more effective might such techniques be at detecting software errors?

Bibliography


Towards a General Approach for Symbolic Model-Checker Prototyping

Edmundo López Bóbeda, Maximilien Colange, Didier Buchs

dedmundo.lopez@unige.ch, maximilien.colange@unige.ch, didier.buchs@unige.ch

Centre Universitaire d’Informatique
University of Geneva, Geneva, Switzerland

Abstract: We propose a novel approach to prototype and create symbolic model-checkers. Our approach focuses on providing a high-level abstraction above Decision Diagrams. It allows the model checker creator to start from a high-level formal semantics and to define an efficient Decision Diagram based model checker.

Keywords: symbolic model checking, rewriting, semantics

Introduction
We propose an approach to rapidly prototype symbolic model checkers. A model checker has to be able to take the semantics of a formalism, and translate it to an efficient representation, e.g., in our case Decision Diagrams (DDs), to perform the verification. This translation has two main difficulties. First, most DDs are only capable of handling at most integers, i.e., complex data structures in the original formalism have to be flattened to integer vectors. Second, the model checker developer has to be familiar with the subtleties of DDs, e.g., DDs handle sets of states, which implies the need for adapted algorithms and optimizations.

Our Proposition
We propose a language to describe the semantics of a system in an intuitive manner, and that is automatically translatable to efficient DD operations. Our language is based on rewriting rules and rewriting strategies. Rewriting strategies control the application of rewriting rules to make it more efficient. Our framework provides a set of predefined strategies which the model checker developer can combine to create custom defined strategies. Also, to describe the semantics in our approach it is not necessary to flatten complex data types to integers. Indeed, complex data types can be expressed naturally with terms and rewrite rules.

Creating a Specification
To specify a system the user needs to define what is a state and the transitions of the system. Suppose that we want to model the well known dining philosophers problem. Each philosopher is either thinking or eating, and he has one fork at each side. Here is a term that represents the initial state of such system, i.e., a table with three philosophers: philo(thinking, fork, philo(thinking, fork, philo(thinking, fork, empty))). We assume here that the first and the last philosophers are next to each other closing the circle. The syntax of terms is given using Abstract Data Types.

The basic pattern to express transitions is the following. Local modifications of the state are represented by rewrite rules. Global modifications are represented by strategies that combine rewrite rules and other strategies. For example, to represent a philosopher that takes the right fork and is then waiting for the left fork we would write the following rule: philo(thinking, fork, $p) \leadsto philo(waitingLF, busyFork, $p)$ ($p$ is a variable). In our framework, a rewrite rule is also a strategy, and all strategies are deterministic. Hence, rewrite rules in our framework are only applied at the root of the term to keep them completely deterministic. The predefined strategy
Towards a General Approach for Symbolic Model-Checker Prototyping

Onek(S) applies strategy S to the k\textsuperscript{th} subterm of a given term. Combining the One strategy with the Choice(S\textsubscript{1},S\textsubscript{2}) strategy, which applies S\textsubscript{1}, or S\textsubscript{2} if S\textsubscript{2} cannot be applied, we can create a strategy that applies some strategy S to the last philosophers in the term. The strategy looks like this: ApplyToLast(S) = Choice(One\textsubscript{3}(ApplyToLast(S)),S). ApplyToLast tries to apply itself recursively to the last subterm of \texttt{philo} and applies S only when the last One\textsubscript{3}(ApplyToLast(S)) cannot be applied. Other predefined strategies are Sequence and Identity. The former applies the strategy S\textsubscript{1}, and then S\textsubscript{2} to the result of S\textsubscript{1}; the latter just returns the same term it was applied to.

Our framework also allows to describe in the same language the DD algorithms, e.g., state space calculation and optimization. For this, all strategies are naturally extended to work on sets of terms instead of just terms. This enables the description of the complete state space calculation as a strategy. The Union(S\textsubscript{1},S\textsubscript{2}) strategy, which applies S\textsubscript{1} and S\textsubscript{2} and the does the union of the result, and Fixpoint(S), that applies S until a fixpoint is reached, are provided for this purpose.

Related Work

The usage of rewriting as a semantic framework is proposed in [MM93]. Term rewrite strategies were described in [BKK\textsuperscript{+}96]. We enriched them with some new strategies and enriched their semantics to handle sets of terms. As DD back end we use \SigmaDDs [HML\textsuperscript{+}11]. Their use in model checking was limited to the rewriting of large sets of terms for Algebraic Petri nets model checking. We are the first ones to use them to represent the whole system state.

Implementation and Preliminary Results

We implemented our approach in StrataGEM [LCB14], a tool downloadable at http://sourceforge.net/projects/stratagem-mc/. StrataGEM takes a textual domain specific language as input to describe a transition system using the strategies described above. It can also translate Petri nets to a transition system and compute their state space. The full philosopher problem example can be found at http://sourceforge.net/projects/stratagem-mc/files/. Compared to other symbolic model checkers our results are promising as StrataGEM can handle extremely large state spaces (> 10\textsuperscript{400} states). We have also created transformations from richer formalism such as DiVinE [BBR09].

Bibliography


Formal Verification of the Danish Railway Interlocking Systems

Linh H. Vu\textsuperscript{1}, Anne E. Haxthausen\textsuperscript{1}, and Jan Peleska\textsuperscript{2}

\textsuperscript{1} DTU Compute, Technical University of Denmark, Denmark
\textsuperscript{2} Department of Mathematics and Computer Science
University of Bremen, Germany

Abstract: In this paper, we present a method for formal verification of the new Danish railway interlocking systems. We made a generic and reconfigurable model of the behaviors and high-level safety properties of non-collision and non-derailment. This model accommodates \textit{sequential release} – a new feature in the new Danish interlocking systems. Instantiating the generic model with interlocking configuration data results in a concrete model and high-level safety properties. Using bounded model checking and inductive reasoning, we are able to verify safety properties for model instances corresponding to railway networks of industrial size.

Keywords: railway interlocking systems, formal verification, bounded model checking, k-induction, safety-critical software systems, RobustRails

Introduction. An interlocking system is responsible for guiding trains safely through a given railway network. It is a vital part of any railway signaling system and has the highest safety integrity level (SIL4) according to the CENELEC 50128 standard \cite{CEN11}. Conventionally, the development and verification process of interlocking systems is informal and mostly manual, hence time-consuming, costly, and error-prone. As part of the RobustRails research project\textsuperscript{1}, our work aims at establishing a holistic method supporting such a process. The method should be formal and facilitate automation in order to provide a better development and verification process compared to the conventional one. In Denmark, in the period of 2009–2021, new interlocking systems that are compatible with ETCS Level 2 \cite{ERT12} will be deployed in the entire country within the context of the Danish Signalling Programme\textsuperscript{2}. In the context of the RobustRail project accompanying the signalling programme on a scientific level, the proposed method will be applied to these new systems.

Figure 1 shows the verification process of our proposed method. The process begins with the configuration data \cite{VHP14} in XML of an interlocking system consisting of: (i) a railway network layout specifying the geographical information of the railway network, and (ii) an interlocking table specifying how routes must be set so that trains can travel safely through the network. This configuration data is then used to instantiate our generic behavioral model of the Danish interlocking systems, and its high-level safety properties of non-collision and non-derailment. The behavioral model accommodates \textit{sequential release}, a new feature that provides the potential to increase the capacity of the network. For verification and validation (V&V), we follow the typical two-step approach. In the first step, the data validation is performed on the configuration data to ensure the well-formedness of the model. In the second step, for a given

\textsuperscript{1} http://robustrails.man.dtu.dk
\textsuperscript{2} http://uk.bane.dk/visArtikel_eng.asp?artikelID=6090
configuration data, the concrete high-level safety properties are verified for the concrete model instance using bounded model checking (BMC) in combination with inductive reasoning. If the model instance does not satisfy the properties, counter-examples will be generated. The method has been implemented as a tool-chain for verifying interlocking systems using the RT-Tester toolbox [Pel13] and its state-of-the-art SMT solver – SONOLAR.

**Experiments.** We have successfully used the tool-chain to verify the safety properties for model instances of a number of railway networks, ranging from a trivial tiny toy network to a large station (Køge) extracted from the early deployment line of the new Danish signalling systems. The results demonstrate the advantages of the combined approach of BMC and inductive reasoning for verification of safety properties. First, it allows errors to be spotted quickly. Our experiences with BMC show that the counter-examples, if there are any, are found within a relatively short time of the model checking process. Second, the method can be scaled up to the size of real applications in practice, hence facilitating the automated V&V process of interlocking systems.

**Bibliography**


Verification of Fix Protocol Session Layer

Kamil Kuboň¹

¹ kamil.kubon@gmail.com
Faculty of Mathematics, Physics and Informatics
Comenius University in Bratislava, Slovakia
ION Trading, London United Kingdom

Abstract: This work describes a verification project for basic properties of Session layer of Fix Protocol using a bytecode transformation from an open source implementation QuickFix/J to Isabelle/HOL. Verification will be done using modal μ-calculus.

Keywords: Fix Protocol, Java, bytecode, Isabelle/HOL, modal μ-calculus

1 Introduction

One of the most widely used protocols in the domain of financial services is Fix Protocol [Fixa]. It is an open protocol with specifications available from the Fix Protocol organisation. It contains an application layer for business related messages and a session layer related to the reliable communication between counterparties [FPb]. It is used by major exchanges and investment banks as well as by many smaller companies to provide trading connectivity. As it participates in a transfer of large volumes of assets, there is a demand for correct and quick implementations of the protocol. As per knowledge of the author, there has been no formal verification of the protocol in a theorem prover performed so far, despite high volumes being transferred by it. The purpose of this work is to verify basic protocol properties of the Session layer [FPb] and an open source implementation called QuickFix/J [Qui]. Core parts of the QuickFix/J will be converted to process algebraic language implemented in Isabelle/HOL [ISA]. Logic for verification will be an embedding of modal μ-calculus [BS07] in Isabelle/HOL.

As an overview of related work, following developments can be mentioned focusing on imperative fragment of Java language: project Bali [ON99] or more recent [SP11]. For works related to Java concurrent and memory Model one can mention [Loc14] or [AS07] mechanised in Isabelle/HOL. For the mechanisation of process algebras in Isabelle/HOL, one can mention works of Jesper Bengston [Ben12]. For the logic embeddings in Isabelle/HOL, there is for example encoding of TLA* in [GM11]. For the modal μ-calculus formalisation, [Mic01] is available for the theorem prover COQ. For the Fix Protocol verification, there exist commercial certification services like from the company [Gre], for the [Qui] acceptance tests are available.

2 Transformation description

The transformation tool, which is being developed as part of the project should take the Java bytecode and transform selected methods into Isabelle/HOL. Formulas to be verified and abstracted
parts needed for the verification will also be specified in Isabelle/HOL code. For the purpose of QuickFix/J verification, the core area on which we focus is a class Session which is responsible for processing incoming messages. It contains references to other classes like SessionState, which holds a message queue and references MessageStore class which is responsible for storing Fix messages. Session class provides logic for Session layer logic involving the logon, logout process, validation and message recovery. Class directly calling Session is SessionConnector, which is interfacing network connection layer. The plan for the transformation is to translate classes and methods containing the core logic, like the classes Session and SessionState and for the related classes use abstractions instead of their implementation - using simpler processes that satisfy their interface. Such classes are for example Message and Header, representing Fix messages or SessionConnector with connections to network layer. The translation tool is being implemented in Java [JAV] using framework [ASM] for a bytecode handling.

In Isabelle/HOL, the target language being developed is based on process algebra, with the features needed for both translated and abstracted code, like successful termination, deadlock, nondeterministic choice, summation, process call, method invocation frames representation, return from the function call, exception handling, data types support, class hierarchy support, heap representation and Java concurrency features. The basic idea behind the transformation is to translate bytecode instruction into a process call, with parameters reflecting stack and local variables manipulation. Instance variables will be modelled as processes with actions for modifying and retrieving data.

Verification will be done using formulas being formulated in the language of modal μ-calculus embedding, being developed in Isabelle/HOL. Features to be verified include correctness of the message recovery process, message being sent will be eventually delivered, correctness of the individual logon modes, exclusion of deadlocks and for example an absence of race conditions. To help the verification, modal μ-calculus embedding should contain inference rules with respect to a term construction of the process algebra.

3 Summary

The implementation of the project stated above is currently in progress and the following contributions expected. The first one is verification of core properties of the Fix protocol Session layer using a theorem prover. This implies clarification of the protocol properties by formulating them in a formal logic. Next contribution is the process algebraic language implemented in Isabelle/HOL, which has transformation from a bytecode supporting abstraction and features used by the communication protocol from finance environment demanding reliable implementation. Contribution is also in providing modal μ-calculus embedding in Isabelle/HOL. Currently, the work related to transformation is in progress. Initial versions of the process algebraic language and modal μ-calculus embedding in Isabelle/HOL have been implemented.

Acknowledgements: I would like to express thanks for the support from my doctoral advisor doc. RNDr. Damas Gruska PhD as well as to Lucy and my family.
Bibliography


[ISA] Isabelle. Theorem prover Isabelle/HOL. http://www.cl.cam.ac.uk/research/hvg/Isabelle/


[Qui] QuickFix/J. http://www.quickfixj.org/

Delta-Oriented Testing for Finite State Machines
(Extended Abstract)

Mahsa Varshosaz and Harsh Beohar
Center for Research on Embedded Systems
Halmstad University, Sweden
{mahsa.varshosaz, harsh.beohar}@hh.se

Software Product Line (SPL) engineering is a promising approach to develop a set of related software systems (product families). This approach enables systematic reuse in the software life cycle by explicitly defining the commonalities and variabilities of the products of an SPL. The main advantage of adapting this engineering approach is to decrease the time to market and the development cost [6].

Just like conventional software, testing an SPL is crucial for validation and verification, and thus for quality assurance. Since the number of potential products in an SPL can be extremely large, testing each product individually can be very costly or even impossible. Thus, a number of testing approaches – which, unfortunately, lack mathematical rigour – have been proposed to fame the variability of such systems in testing (see [3, 4] for surveys of such approaches).

In this note, we take the first step in adapting the W-method [2] – a formal approach – to test SPLs by formulating a notion of reuse for test models and test cases associated with products in SPL. Roughly, W-method uses finite state machines (FSMs) as test models together with machine equivalence [1] as the conformance notion between a specification and its implementation.

**Definition 1.** An FSM $M$ is a 6-tuple $(s_0, S, I, O, \mu, \lambda)$, where $s_0$ is the initial state, $S$ is a finite set of states, $I$ and $O$ are, respectively, finite nonempty sets of input and output symbols, $\mu : S \times I \rightarrow S$ is the transition function and $\lambda : S \times I \rightarrow O$ is the output function.

Furthermore, two states are $X$-equivalent ($X \subseteq I^*$) if and only if the two states produce same output for every input sequence $\sigma \in X$. Lastly, two machines $M, M'$ are $X$-equivalent, denoted by $M \equiv_X M'$, if and only if for every state of $M$ there is an $X$-equivalent state of $M'$ and vice versa. Also, we have machine $M$ conforms to Machine $M'$ if and only if they are $I^*$-equivalent.

W-method indirectly establishes that an implementation confirms to a given specification by executing a set of input sequences on the implementation and then comparing the observed outputs with the expected outputs. In short, if there are no deviations from the expected behavior then the implementation complies to the specification. Note that the conformance is established indirectly, because it is assumed that the state-transition function of the implementation is not known (the well-known testing hypothesis, cf. [1]). To make this idea work, W-method assumes a set of restrictions on the test model; namely, all states in the specification are reachable from the initial state, the specification is fully specified, and both specification and its implementation are minimal.

To this end, and building upon the ideas from [5], we use delta-oriented modeling, where a core test model is assumed to be the starting point in the test process and the test model for each product is provided by addition and removal of some parts (deltas) to/from the ones generated for the core model. Since W-method assumes a certain restrictions on the test model, the definition of deltas should be such that each model generated upon applying a delta preserve these conditions. As a result, we define (Def. 2) a restricted form of delta which allows to add a set of states and transitions to the core test model and also removal of some of the existing transitions.

**Definition 2 (FSM Delta).** Given an FSM $M = (s_0, S, I, O, \mu, \lambda)$, a delta for $M$, denoted by $\Delta(M)$, is a quintuple $(S_\Delta, I_\Delta, O_\Delta, \mu_\Delta, \lambda_\Delta)$ where,

- $S_\Delta$ is a finite non-empty set of states such that $S \cap S_\Delta = \emptyset$.
- $I_\Delta$ and $O_\Delta$ are, respectively, finite non-empty sets of input and output symbols such that $I \subseteq I_\Delta$.
- $\lambda_\Delta : (S \cup S_\Delta) \times I_\Delta \rightarrow O_\Delta$ is the (total) output function.
- $\mu_\Delta : (S \cup S_\Delta) \times I_\Delta \rightarrow (S \cup S_\Delta)$ is the (total) transition function, satisfying the following condition:

$$\forall s \in S_\Delta \cdot s \in init_\Delta \lor \exists s' \in init_\Delta \cdot s \in Reach(s'),$$

where $init_\Delta = \{ s \in S_\Delta \mid \exists s' \in S, \exists a \in I_\Delta \cdot \mu_\Delta(s', a) = s \}$ and $Reach(s)$ denotes the set of reachable states from state $s$. The set $init_\Delta$ represents the set of states which are directly connected to the core model.

We define the application of a delta to the core model as follows.
Definition 3 (Delta Application). The application of the delta $\Delta(M)$ to the FSM $M = (s_0, S, I, O, \mu, \lambda)$ is denoted by $\text{apply}(M, \Delta(M))$ which results in another FSM $M' = (s'_0, S', I', O', \mu', \lambda')$ such that: $s_0$ is the initial state which is the same as the one in $M$, $S' = S \cup S_\Delta$, $I' = I_\Delta$ and $O' = O_\Delta$ and for all $s \in S'$ and $a \in I'$:

$$
\mu'(s, a) = \begin{cases}
\mu_\Delta(s, a) & \text{if } (s, a) \in \text{Dom}(\mu_\Delta) \\
\mu(s, a) & \text{if } a \in I \land (s, a) \notin \text{Dom}(\mu_\Delta) \\
s & \text{otherwise}
\end{cases}
\quad \lambda'(s, a) = \begin{cases}
\lambda_\Delta(s, a) & \text{if } (s, a) \in \text{Dom}(\lambda_\Delta) \\
\lambda(s, a) & \text{if } a \in I \land (s, a) \notin \text{Dom}(\lambda_\Delta) \\
\tau & \text{otherwise}
\end{cases}
$$

The third case in both definitions, handles the fully specified assumption and it specifies that if a new input symbol is added w.r.t. the core model, then a self loop is added together with a fresh output denoted by $\tau \notin O_\Delta$.

Next, we motivate two criteria for future research focussing on the reuse of test cases that are generated by W-method. The first criterion is soundness, i.e., when is it safe to derive the test suite (a set of test cases) for a product from the test suite of the core model. Let $M$, $M'_i$ and $\Delta(M)_i$, $0 \leq i \leq n$ denote the core test model, the implementation of a product $p_i$, and the delta applied to generate the test model of product $p_i$ in an SPL with $n$ products, respectively. We assume that $\text{DeltaWMethod}(\Delta(M)_i)$ denotes the set of test cases generated with regards to $\Delta(M)_i$, then the delta-oriented approach is sound if and only if $\forall 1 \leq i \leq n$:

$$
\text{apply}(\Delta(M)_i, M) \equiv \text{WMethod}(\text{apply}(\Delta(M)_i, M)) M'_i
\Leftrightarrow
\text{apply}(\Delta(M)_i, M) \equiv \text{DeltaWMethod}(\Delta(M)_i) M'_i \land \text{apply}(\Delta(M)_i, M) \equiv \text{WMethod}(M) M'_i
$$

The right hand side of the above condition indicates the conformance of an implementation of a product $p_i$ to its test model $\text{apply}(\Delta(M)_i, M)$, according to a set of test cases generated from the core model using W-method and also a set of test cases generated from the delta, using the delta-oriented approach, to check the parts added to the core model and also parts that have changed through application of the delta to the core model. This means that the behavior of the implementation should conform to the behavior of the core model in addition to the extended behavior regarding to the delta.

The second criterion to be considered is the efficiency of the new approach. The efficiency of the new approach can be compared with that of the W-method according to the length of test cases generated by each method. Assuming that $|\text{WMethod}(\text{apply}(\Delta(M)_i, M))|$, $|\text{WMethod}(M)|$, and $|\text{DeltaWMethod}(\Delta(M)_i)|$, respectively, denote the total length of the test cases generated by the W-method for a product $p_i$ and the core model and the total length of the test cases generated using delta-oriented approach for product $p_i$, then the new method is efficient if and only if (where $| - |$ denotes the size (number of states) of a model or delta):

$$
\sum_{i=1}^{n} \text{WMethod}(\text{apply}(\Delta(M)_i, M)) >> \sum_{i=1}^{n} \text{DeltaWMethod}(\Delta(M)_i) + \text{WMethod}(M)
$$

if $|M| + \sum_{i=1}^{n} |\Delta(M)_i| < \sum_{i=1}^{n} |\text{apply}(\Delta(M)_i, M)|$

This means that if the summation of size of the applied deltas and the core model is very smaller than the total size of the generated test models for all products, which is the case in software product lines with high number of commonalities, then the new approach is efficient if the sum of the length of the test cases generated by the delta oriented approach and the test cases generated for the core model by W-method is very less than that of the test cases generated by the W-method for all products in the SPL.

Lastly, in addition to reducing the length of test cases, the application of the delta-oriented testing approach can reuse the results of the test executions on the core test model. Since, the existence of transition faults and output faults should be considered only for that parts of the model that are changed or added by the applied delta.

References
Interactive Verification of Hybrid Systems

Xian Li, Klaus Schneider

Embedded Systems Group
Department of Computer Science
University of Kaiserslautern, Germany
Email: {xian.li,klaus.schneider}@cs.uni-kl.de

Abstract: We present our approach towards the interactive verification of hybrid systems which takes advantage of combining human intelligence with existing decision procedures and tools. In particular, we target the parametric verification problem of hybrid systems, where values of parameters of the system have to be determined to meet the given specifications. To this end, we allow linear and non-linear arithmetic with real, integer, and boolean variables. A prototypical interactive theorem prover AIFProver is developed as an interactive verification environment providing an effective interface between designers and backend verification techniques and tools for hybrid systems.

Keywords: Interactive Verification, Parametric Verification, Hybrid Systems

The parametric verification problem of hybrid systems consists in deriving constraints on the values of parameters to meet the given specifications. Numerous approaches and tools targeted for hybrid system verification have been proposed in the literature for verifying either concrete systems with given parameter instantiations, or a subset of hybrid systems in the theory of reals only. So, the parametric verification problem has not yet found good solutions. We work on an interactive verification approach combining human intelligence with the existing verification methods for parametric hybrid systems, where we allow linear and non-linear arithmetic with real, integer, and boolean variables. To support the proposed interactive verification approach, our Averest system (http://www.averest.org), a toolset for HW/SW co-design and formal verification, has been extended by AIFProver, a prototypical interactive theorem prover for discrete and hybrid systems [GS13, LBS13].

The idea of interactive verification of parametric hybrid programs is illustrated in Fig. 1. The Averest compiler computes for a given hybrid program an equivalent set of guarded actions [BS10]. Proof goals are given as pairs \((\mathcal{G}, \mathcal{L})\) that contain a set of guarded actions \(\mathcal{G}\) and the set of currently active control-flow variables \(\mathcal{L}\) [GS13]. The AIFProver consists of proof rules that split these proof goals into subgoals or simply solve these proof goals directly. The proposed approach allows us not only for establishing interoperability among a wide range of different modeling formalisms and tools, but also for exploring a verification approach that takes advantage of the verification engineers.

The basic proof rules have been implemented in AIFProver as F# functions so that these can be applied either in an interactive F# session or in a proof script (i.e., a F# script). In addition to the integrated SAT solver and a BDD package, there are interfaces to SAL [dOR+04] for verification of discrete systems, and to KeYmaera [Pla10] for verification of continuous systems. Also, we explore a new method by integrating the SAT solver with BONMIN [BBC+08] to solve a special

AVoCS 2014 265
SMT problem addressing boolean combinations of propositional-logic atoms and theory atoms from undecidable non-linear arithmetic theories over integers and reals.

In the interactive verification approach, our hybrid modelling language serves as a bridge between different verification techniques and tools. Therefore, we are evaluating other SMT solvers and mathematical computation tools to extend the backend tool interface. Meanwhile, we are evaluating and extending the set of basic proof rules by benchmark examples. Except for the tool supported techniques, we plan to integrate non-tool supported techniques for the interactive verification approach as well. We are exploring and implementing abstraction and counterexample guided abstraction refinement (CEGAR) techniques to obtain some practically relevant results. Also, we consider the simulation-based verification method. Simulating some system instances explicitly could give some useful hints for for the interactive verification itself.

Bibliography


