LNCS 7687

Shaz Qadeer
Serdar Tasiran (Eds.)

Runtime Verification

Third International Conference, RV 2012
Istanbul, Turkey, September 2012
Revised Selected Papers

Lecture Notes in Computer Science 7687

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Shaz Qadeer Serdar Tasiran (Eds.)

Runtime Verification

Third International Conference, RV 2012
Istanbul, Turkey, September 25-28, 2012
Revised Selected Papers

Springer

Volume Editors

Shaz Qadeer
Microsoft Research
One Microsoft Way
Redmond, WA 98052, USA
E-mail: qadeer@microsoft.com

Serdar Tasiran
Koc University
College of Engineering
Rumeli Feneri Yolu
Sariyer, 34450 Istanbul, Turkey
E-mail: stasiran@ku.edu.tr

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-35631-5 e-ISBN 978-3-642-35632-2
DOI 10.1007/978-3-642-35632-2
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012956063

CR Subject Classification (1998): D.2, F.2, D.2.4, D.1, F.3, D.3, C.2

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2013
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in ist current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This volume contains the proceedings of the 2012 International Conference on Runtime Verification (RV), third in a series dedicated to the advancement of monitoring and analysis techniques for software and hardware system executions. RV 2012 was held during September 25–28, 2012, in Istanbul, Turkey.

Runtime verification is concerned with monitoring and analysis of software and hardware system executions. Runtime verification techniques are crucial for system correctness and reliability; they are significantly more powerful and versatile than conventional testing, and more practical than exhaustive formal verification. Runtime verification can be used prior to deployment, for verification and debugging purposes, and after deployment for ensuring reliability, safety and security, and for providing fault containment and recovery.

The history of the RV conference series goes back to 2001 when it started as a workshop. It continued as an annual workshop series until 2009 and became a conference starting in 2010. RV proceedings were published in *Electronic Notes in Theoretical Computer Science (ENTCS)* from 2001 to 2005. Since 2006, the RV proceedings have been published in *Lecture Notes in Computer Science (LNCS)*.

The RV 2012 program consisted of a mix of tutorials, invited talks, accepted papers (regular, tool, and short), a poster session, and a tool demonstration session. We received 50 submissions, out of which the Program Committee accepted 25. Each paper received four reviews. We also invited leading researchers to present two tutorials and three invited talks.

The organizers would like to thank the Steering Committee for their advice, the Program Committee for their hard work in selecting the papers, and the tutorial and invited speakers for their valuable contribution to the conference program. Financial support for the conference was provided by Google, Microsoft, Koc University, and STM. Finally, we thank EasyChair for their help with the conference management and creation of the conference proceedings.

September 2012

Shaz Qadeer
Serdar Tasiran

Organization

Program Committee

Howard Barringer	University of Manchester, UK
Saddek Bensalem	VERIMAG, France
Eric Bodden	European Center for Security and Privacy by Design, Germany
Cristian Cadar	Imperial College London, UK
Ylies Falcone	Université Joseph Fourier, France
Bernd Finkbeiner	Saarland University, Germany
Stephen Freund	Williams College, USA
Ganesh Gopalakrishnan	University of Utah, USA
Wolfgang Grieskamp	Google Inc., USA
Sylvain Hallé	Université du Québec à Chicoutimi, Canada
Klaus Havelund	Jet Propulsion Laboratory, California Institute of Technology, USA
Suresh Jagannathan	Purdue University, USA
Sarfraz Khurshid	University of Texas at Austin, USA
Martin Leucker	University of Lübeck, Germany
Benjamin Livshits	Microsoft Research, Redmond, USA
Shan Lu	University of Wisconsin, Madison, USA
Rupak Majumdar	University of California, Los Angeles, USA
Oded Maler	VERIMAG, France
Sharad Malik	Princeton University, USA
Atif Memon	University of Maryland, USA
Peter Müller	ETH Zürich, Switzerland
Shaz Qadeer	Microsoft Research, Redmond, USA
Venkatesh Prasad Ranganath	Microsoft Research, Bangalore, India
Vivek Sarkar	Rice University, USA
Koushik Sen	University of California, Berkeley, USA
Oleg Sokolsky	University of Pennsylvania, USA
Serdar Tasiran	Koc University, Turkey
Stavros Tripakis	University of California, Berkeley, USA
Martin Vechev	ETH Zurich, Switzerland
Willem Visser	Stellenbosch University, South Africa
Zheng Zhang	Microsoft Research, Beijing, China

Additional Reviewers

Ayoub, Anaheed
Ayoub, Nouri
Bartel, Alexandre
Bensalem, Saddek
Chaudhuri, Swarat
Chiang, Wei-Fan
Decker, Normann
El-Dib, Hasan
Elmas, Tayfun
Follner, Andreas
Ganov, Svetoslav
Gligoric, Milos
Gopinath, Divya
Habermehl, Peter
Kasikci, Baris
Kim, Chang Hwan Peter
Kumar, Ashwin
Lebeltel, Olivier
Legay, Axel
Li, Guodong
Lu, Zhang
McDirmid, Sean
Monmege, Benjamin
Nickovic, Dejan
Niebert, Peter
Nokhbeh Zaeem, Razieh
Pekergin, Nihal
Poplavko, Petro
Purandare, Rahul
Reger, Giles
Rydeheard, David
Sankaranarayanan, Sriram
Sawaya, Geof
Schönfelder, René
Sharma, Subodh
Sinha, Arnab
Smolka, Scott
Thoma, Daniel
Wang, Shaohui
Weiss, Alexander
Wies, Thomas
Yang, Guowei
Yang, Zijiang
Zhang, Lingming

Table of Contents

Dynamic Analyses for Data-Race Detection 1
John Erickson, Stephen Freund, and Madanlal Musuvathi

Symbolic Execution 2
Cristian Cadar and Koushik Sen

Dynamic Livelock Analysis of Multi-threaded Programs 3
Malay K. Ganai

Scalable Dynamic Partial Order Reduction 19
Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey

ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level 35
Jan Fiedor and Tomáš Vojnar

PaRV: Parallelizing Runtime Detection and Prevention of Concurrency Errors 42
Ismail Kuru, Hassan Salehe Matar, Adrián Cristal, Gokcen Kestor, and Osman Unsal

It's the End of the World as We Know It (And I Feel Fine) 48
Jim R. Larus

Detecting Unread Memory Using Dynamic Binary Translation 49
Jon Eyolfson and Patrick Lam

Sparse Coding for Specification Mining and Error Localization 64
Wenchao Li and Sanjit A. Seshia

Sliding between Model Checking and Runtime Verification 82
Martin Leucker

Runtime Verification and Enforcement for Android Applications with RV-Droid 88
Yliès Falcone, Sebastian Currea, and Mohamad Jaber

Temporal Monitors for TinyOS 96
Doina Bucur

Real-Time Runtime Verification on Chip 110
Thomas Reinbacher, Matthias Függer, and Jörg Brauer

BabelTrace: A Collection of Transducers for Trace Validation 126
Aouatef Mrad, Samatar Ahmed, Sylvain Hallé, and Éric Beaudet

Quantitative Trace Analysis Using Extended Timing Diagrams 131
Andreas Richter and Klaus Kabitzsch

Maximal Causal Models for Sequentially Consistent Systems 136
Traian Florin Şerbănuţă, Feng Chen, and Grigore Roşu

Monitoring Compliance Policies over Incomplete and Disagreeing Logs 151
David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu

Adaptive Runtime Verification 168
Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster

Malware Riding Badware: Challenges in Analyzing (Malicious/Benign) Web Applications 183
Giovanni Vigna

MapReduce for Parallel Trace Validation of LTL Properties 184
Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hallé

Path-Aware Time-Triggered Runtime Verification 199
Samaneh Navabpour, Borzoo Bonakdarpour, and Sebastian Fischmeister

Fast-Forward Runtime Monitoring — An Industrial Case Study 214
Christian Colombo and Gordon J. Pace

Runtime Enforcement of Timed Properties 229
Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet, and Omer Landry Nguena Timo

Monitoring Dense-Time, Continuous-Semantics, Metric Temporal Logic 245
Kevin Baldor and Jianwei Niu

Rewrite-Based Statistical Model Checking of WMTL 260
Peter Bulychev, Alexandre David, Kim G. Larsen, Axel Legay, Guangyuan Li, and Danny Bøgsted Poulsen

From Runtime Verification to Runtime Intervention and Adaptation 276
Martin Rinard

Certifying Solutions for Numerical Constraints 277
Eva Darulova and Viktor Kuncak

Profiling Field Initialisation in Java 292
Stephen Nelson, David J. Pearce, and James Noble

Defense against Stack-Based Attacks Using Speculative Stack Layout Transformation 308
Benjamin D. Rodes, Anh Nguyen-Tuong, Jason D. Hiser, John C. Knight, Michele Co, and Jack W. Davidson

Incremental Runtime Verification of Probabilistic Systems 314
Vojtěch Forejt, Marta Kwiatkowska, David Parker, Hongyang Qu, and Mateusz Ujma

Author Index 321

Dynamic Analyses for Data-Race Detection

John Erickson[1], Stephen Freund[2], and Madanlal Musuvathi[3]

[1] Microsoft
[2] Williams College
[3] Microsoft Research

Abstract. Data races caused by unsynchronized accesses to shared data have long been the source of insidious errors in concurrent software. They are hard to identify during testing, reproduce, and debug. Recent advances in race detection tools show great promise for improving the situation, however, and can enable programmers to find and eliminate race conditions more effectively. This tutorial explores dynamic analysis techniques to efficiently find data races in large-scale software. It covers the theoretical underpinnings, implementation techniques, and reusable infrastructure used to build state-of-the-art data-race detectors (as well as analyses targeting other types of concurrency errors). The tutorial provides industrial case studies on finding data races and closes with a discussion of open research questions in this area.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, p. 1, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Symbolic Execution

Cristian Cadar[1] and Koushik Sen[2]

[1] Imperial College London
[2] University of California at Berkeley

Abstract. Recent years have witnessed a surge of interest in symbolic execution for software testing, due to its ability to generate high-coverage test suites and find deep errors in complex software applications. In this tutorial, we give an overview of modern symbolic execution techniques, discuss their key challenges in terms of path exploration, constraint solving, and memory modeling, and present several tools implementing these techniques.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, p. 2, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Dynamic Livelock Analysis of Multi-threaded Programs

Malay K. Ganai

NEC Labs America, Princeton, NJ, USA

Abstract. Compared to deadlocks, where one or more threads are blocked forever, livelocks are harder to detect as it is not easy to distinguish between a long and an infinite busy wait (i.e., no progress) cycle. We propose a dynamic livelock analysis for a multi-threaded program by examining its execution trace. From the observed trace events, our approach uncovers livelock potentials due to infinite executions where one or more threads in a group are acquiring and releasing resources in busy-wait cycles to avoid deadlocks. Furthermore, to confirm a livelock potential, we orchestrate a partial-order schedule to induce a livelock during a program re-execution. We implemented our proposed approach in a prototype tool CBuster, comprising a light-weight binary instrumentation framework for C/C++ programs to record events, and to replay partial-order schedules. We applied our approach to identify and confirm livelocks in a case study based on SQLite, a widely used embedded multi-threaded database engine.

1 Introduction

Multi-threaded programming is error prone. The complex interaction between threads through synchronization primitives such as mutex acquires and releases can lead to a situation when one or more threads in a group do not make any forward progress (such as towards program termination). Based on whether CPU cycles are consumed or not, a lack of forward progress can be categorized [1] as a *deadlock* or a *livelock*.

In a deadlock, one or more threads in a group are blocked forever without consuming CPU cycles. A *resource deadlock* occurs when each thread is waiting to acquire a mutex resource held exclusively by another in the group. A *communication deadlock* occurs when receiver threads are waiting for messages which are either lost and/or the sender threads are also waiting for some unavailable resources.

In a livelock, one or more threads continuously change their states (and hence consume CPU cycles) in response to changes in states of the other threads without doing any useful work [2]. It is a busy-waiting analog of deadlock. Livelocks can be classified [1] into one of the following: *starvation* (where a thread is denied resources perpetually due to new external events) [3], *infinite execution* (where a thread is executing in an infinite loop without making progress) [4], and breach of safety properties [5]. In a livelock, the threads may not be blocked forever, and it is harder to distinguish between a long running process, and a do-nothing busy-wait cycle. Not surprisingly, programmers/testers find livelocks very hard to detect and debug and often they mislabel them in bug reports (such as [6–8]). Although livelocks do not bring a system to a standstill, but they can lead to performance degradation due to useless busy-wait cycles.

Related Work. There are relatively a few articles on detecting livelocks, in comparison with the vast literature available for detecting deadlocks using both static and

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 3–18, 2013.
© Springer-Verlag Berlin Heidelberg 2013

dynamic analysis. Previous work on livelocks are based mostly on static analysis such as [9, 10] targeting concurrent programming languages such as Ada and CSP. Using patterns/rules, they identify conservatively a set of "safe" paths that will not contribute to livelocks, for e.g., those that do not communicate externally using I/O. There are model checking approaches such as SPIN [11] and JPF [12] which can be used to target both deadlocks and livelock properties but have inherent scalability issues. There also has been some hardware-based solution for runtime livelock detection such as [13].

Techniques for detecting deadlocks can be broadly classified as follows: static analysis [14–17], model checking [18], runtime-monitoring [19, 20], and dynamic analysis [21–28]. Static analysis tools work directly on source code, with potential of full coverage, but often result in large false positives. Runtime-monitoring systems detect deadlocks in the currently executing program path. Runtime deadlock prevention systems such as [19,20,29] provide mechanism (albeit at some runtime overhead) to prevent recurring deadlock situations detected previously.

Dynamic analysis tools for deadlock detection work in two phases: first, they observe the trace events such as synchronization events, and then use static analysis techniques such as model checking or cycle detection schemes to identify potential deadlocks. The state-of-the-art technique to detect deadlock cycle condition (DCC) was first introduced [18,30] as Goodlock algorithm, and then later was extended and improved [21–24] to handle gated locks, happens-before, semaphore, and condition variables to reduce false positives, and to replay the detected deadlock potentials. Many tools based on DCC such as DeadlockFuzzer [25], MulticoreSDK [27], and MagicFuzzer [28] are engineered to improve cycle detection, reduce graph size and replay deadlock potentials with higher probability. These techniques, as such, may not be directly applied to detect and induce livelocks, which involve much more subtle interaction between threads. Nevertheless, we were inspired by such techniques. We propose a generalization of DCC to detect both livelocks and deadlocks potentials.

Motivation and Overview. We mainly focus on finding livelock potentials due to infinite executions where one or more threads in a group are acquiring and releasing resources in busy-wait cycles to avoid deadlocks. Such a livelock would necessarily involve intra-thread cycles (such as while/for loops) and inter-thread resource request cycles (such as potential deadlock cycles). We identify these cycles in two steps. In the first step (*static*), we focus on finding inter-thread resource request cycles (similar to deadlock cycle detection) that will potentially induce intra-thread cycles. In the second step (*dynamic*), we induce a livelock by orchestrating a partial-order schedule (based on detected inter-thread cycle) such that the involved threads acquire and release resources without actually making any forward progress.

We motivate our readers with a small example, as shown in Figure 1, based on SQLite user posts [6, 8]. SQLite [31] is a popular embedded multi-threaded database engine used in many applications such as Adobe, iPhone, Dropbox, and Firefox. SQLite has the ability to detect and report potential deadlocks. It does not commit SQL queries in such circumstances and throws exception to the application for re-trials.

In the example shown, all the acquires and releases of mutexes (in the body of the `while` loop) are carried out inside the SQLite database engine in response to the user SQL queries. The `while` construct represents re-trials for failed SQL queries requested by the user application. The two threads t_1 and t_2 acquire mutexes m_1 and m_2 using `G_lock(.)` procedure. The atomicity of their acquisition is ensured using a guarded

mutex m_g. After a thread acquires a mutex, it tries to acquire the other mutex using G_trylock(.) procedure. A trylock is a non-blocking primitive that does not wait forever for the requested mutex to be available. If the mutex is available, it acquires the mutex; otherwise, it returns a failure status (instead of blocking forever). When the trylock operation fails, the corresponding thread releases the first mutex, and yields to the other thread. Clearly, this avoids a deadlock in the example. However, a rare situation can occur when the trylock fails for both the threads as each thread holds the mutex needed by the other. In that case, both thread release the acquired mutexes i.e., m_1, m_2, and then retry. If the above ordering of mutex acquires and releases occur repeatedly, we get a livelock situation. Although deadlock and starvation avoidance mechanism are built-in SQLite, the applications are prone to livelocks as the internals of the database logic often play an intricate role with application code, creating vicious cycles. Currently, SQLite does not provide capability to detect potential livelocks triggered by SQL queries. This could be very frustrating to debug, especially when the user is unfamiliar with the internals of the database engine [6–8].

(a) Example code (based on [6,8])

```
G_lock (m):                 G_trylock (m):
a1:  lock(m_g)              b1:  lock(m_g)
a2:  lock(m)                b2:  r := trylock(m)
     unlock(m_g)                 unlock(m_g)
                                 return r

Thread1:                    Thread2:
  while (true) {              while(true) {
c1:  G_lock(m1)             d1:  G_lock(m2)
c2:  r0 := G_trylock(m2)    d2:  r1 :=G_trylock(m1)
     if (r0=fail) {              if (r1=fail) {
       unlock(m1) //yield          unlock(m2) //yield
       continue                    continue
     }                           }
     ...                         ...
     unlock(m2)                  unlock(m1)
     unlock(m1)                  unlock(m2)
     break                       break
  }                           }
```

(b) A Trace

t_1, cxt_1:	$lock(m_g)$;
t_1, cxt_2:	$lock(m_1)$;
t_1:	$unlock(m_g)$;
t_1, cxt_3:	$lock(m_g)$;
t_1, cxt_4:	$trylock(m_2)$;
t_1:	$unlock(m_g)$;
	$\cdots$;
t_1:	$unlock(m_2)$;
t_1:	$unlock(m_1)$;
	$\cdots$;
t_2, cxt_5:	$lock(m_g)$;
t_2, cxt_6:	$lock(m_2)$;
t_2:	$unlock(m_g)$;
t_2, cxt_7:	$lock(m_g)$;
t_2, cxt_8:	$trylock(m_1)$;
t_2:	$unlock(m_g)$;
	$\cdots$;
t_2:	$unlock(m_1)$;
t_2:	$unlock(m_2)$;

Fig. 1. (a) An sample code with a potential livelock, (b) a total ordered observed trace events of acquisition/release of mutexes, with thread contexts cxt_i

We propose to identify livelock potentials using dynamic analysis in three phases.

Phase I. We observe and record various synchronization events such as mutex acquires/releases. A mutex acquire event $lock(m)$ by thread t with current thread context cxt is denoted as $\langle t, cxt : lock(m)\rangle$, where cxt is a stack of call-site labels. For e.g., cxt_2 is $[\texttt{Thread1} : c_1, \texttt{G_lock} : a_2]$.

Phase II. Given recorded trace events, we search for a cyclic dependency of a set of mutexes among a group of threads such that the following conditions are satisfied:

(i) each mutex in the set is acquired by only one thread (referred as *first acquire*),
(ii) each thread intend to acquire another mutex from the set (referred as *second acquire*),
(iii) at second acquires, no two threads hold a common mutex *that was last acquired before the first acquire*.

If only trylock primitives (i.e., non-blocking) are used in all second acquires, we refer to such a cyclic dependency as a *trylock cycle*. For e.g., (m_1, m_2) is a trylock cycle, where t_1 (t_2) currently holds m_1 (m_2) and will acquire m_2 (m_1), resp., using trylock. Such a cycle is a livelock potential, as each thread will fail in the second acquire, but instead of blocking, it releases the mutex of first acquire, and may retry in a loop.

Approaches for detecting of deadlock potentials such as [21, 22, 24–28] use conditions (i)-(ii), and a stronger version of (iii) where no threads hold any common mutex at second acquires. The stronger condition prohibits a common mutex that was last acquired after the first acquires. For e.g., the lockset (i.e., the set of mutexes held by a thread) at the time t_1 acquires m_2, and that at the time t_2 acquires m_1 have a common mutex i.e., m_g that was acquired after first acquires. Thus, these approaches fail to detect the cycle (m_1, m_2). However, these approaches detect cycle (m_1, m_g) where t_1 holds m_1 and wants to acquire m_g, and t_2 holds m_g (and m_2), and wants to acquire m_1. Although such a cycle is a deadlock potential, a deadlock replayer (or confirmer) such as [23, 25, 28], will fail to induce a real deadlock due to non-blocking nature of trylocks. Moreover, such a cycle may not even induce a livelock. Consider the situation where t_1 holds m_1, and t_2 holds m_g and m_2. Since t_1 is blocked on m_g, t_2 will proceed trylock-ing m_1, but it will fail, and consequently release m_g. If thread t_2 now proceeds and releases m_2, then t_1 succeeds to acquire m_g, followed by m_2. In that case, t_1 does not have to retry in a loop. Thus, a livelock situation does not arise.

We offer a precise deadlock detection condition. We refer a cycle, satisfying (i)-(iii), as a *deadlock* iff only lock primitives (i.e., blocking) are used in second acquires. Clearly, such a cycle truly represents a deadlock as each thread would block forever for a resource held by another thread. A cycle, which is neither a trylock nor a deadlock, is referred as a *mixed*. Our approach detects the cycle (m_1, m_g), but classify it as a mixed.

Phase III. Based on a detected trylock cycle, we induce a livelock by orchestrating a partial-ordered schedule of global states `Head`→`Body`→`Tail` which we repeat in a program re-execution. (Note, events in $\{.\}$ are not ordered).

`Head`: $\{\langle t_1, cxt_2 : lock(m_1)\rangle, \langle t_2, cxt_4 : lock(m_2)\rangle\}$
`Body`: $\{\langle t_1, cxt_6 : trylock(m_1)\rangle, \langle t_2, cxt_8 : trylock(m_2)\rangle\}$
`Tail`: $\{\langle t_1 : unlock(m_1)\rangle, \langle t_2 : unlock(m_2)\rangle\}$
A deadlock can be induced similarly, but is not the current focus.

Contributions. Our main contributions can be summarized as follows:
– We formalize the conditions to effectively identify various lock cycles: trylock, deadlock, and mixed, for a given trace. We show that these conditions capture all livelock and deadlock potentials, and subsumes those detected by previous approaches [21, 22, 24–28]. We consider general cases of mutex acquisition: nesting/ non-nesting, blocking/non-blocking, shared/exclusive ownership, and where a mutex can change state between shared and exclusive ownership directly.
– We confirm a livelock potential by generating a partial-order schedule from a trylock cycle, and orchestrate a controller to induce a livelock during a program re-execution. Such orchestration can be applied to confirm deadlock as well.

– We implemented our proposed cycle detection scheme in a prototype tool CBuster. We built a light-weight binary instrumentation framework using interposition library for C/C++ programs to observe and control the execution of a running program. We demonstrate the feasibility of our approach in identifying livelock situations in a case study using SQLite-based application.

Outline. The rest of the paper is outlined as follows. In Section 2, we provide necessary background and notations used. In Section 3, we generalize the conditions for detecting livelocks and deadlocks potentials. In Section 4, we focus on an orchestration to confirm livelock potentials. We discuss implementation of the tool CBuster and experimentation case study in Section 5, and conclusion/future work in Section 6.

2 Preliminaries

A multi-threaded program consists of a set of concurrently executing threads T, each thread with a unique identifier t. The threads communicate with shared objects, some of which are used for synchronization such as mutexes[1] and signals. A trace of a program π is a total ordered sequence of observed events corresponding to various thread operations on shared objects. Each event e of the sequence, i.e., $e \in \pi$ is carried out by some thread, denoted as $e.t$, at a thread context, denoted as $e.cxt$. A *thread context* of an event is an abstraction of an execution state of the thread. It may comprise just a simple statement label to a more refined state that may comprise a thread callstack and a thread state. Each event e is one of the following event types, denoted as $e.etype$:

- `lock(m)/trylock(m)`: acquires mutex m in an *exclusive* state using a *blocking/non-blocking* call, i.e., if m is currently unavailable, `lock` will wait forever until it is available, while *trylock*[2] will return `fail` if mutex is currently unavailable.
- `lockS(m)/trylockS(m)`: acquires mutex m in a *shared* state using blocking/non-blocking call, resp. A `trylockS` may return `fail` if m is currently unavailable.
- `unlock(m)`: releases held mutex m.
- `wait(s)/notify(s)`: waits on/notifies a signal s.
- `fork(t')/join(t')`: forks/joins a thread child t'.
- `thread_start()/thread_end()`: thread starts/ends.

There could be multiple owners holding a mutex if it is in shared state, but there could be only one owner holding a mutex if it is in exclusive state. The availability/unavailability of a mutex depends on its current state. A *reader/writer* lock (or `reader/writer` trylock) can be expressed equivalently using `lockS/lock` (or `trylockS/trylock`) primitives, resp. A owner holding a mutex m in a shared state can upgrade it to an exclusive state without unlocking it first, provided it is the only owner (for e.g., a file lock in Linux supports such upgrades). In *nested* locking, all mutexes follow a simple rule: a mutex that is first acquired is released the last; otherwise, the locking is termed as *non-nesting*. A *lockset* is a set of mutexes held by the thread at some instant.

Given a trace π of a program, and events $e, e' \in \pi$, we say e *happens-before* e', i.e., $e \preceq e'$, if e is *observed* before e' in the trace. We use $e \prec e'$, to denote that

[1] Our approach supports various types of lock objects such as POSIX locks and file locks. To differentiate a lock object and a locking primitive, we use mutex to refer a lock object.

[2] We use `trylock` to subsume a timed-lock primitive, which is also a non-blocking primitive but it waits for a preset amount of time for the requested mutex.

e *must-happen-before* e, as causally ordered by thread program order and inter-thread events such as `fork/thread_start`, `notify/wait`, and `thread_end/join`, and applied transitively, i.e., $\exists e_1 \in \pi.\ (e \prec e_1 \preceq e')$ or $\exists e_2 \in \pi.\ (e \preceq e_2 \prec e')$. A must happens-before relation can be maintained easily using vector clocks [32,33].

Lock dependency [21,22]: Given a trace π, a lock dependency is a tuple $\tau = \langle t, m, L \rangle$ of a thread $t \in T$, a mutex m, and a lockset L such that the thread t will acquire a mutex m while holding all the mutexes in the lockset L. A lock dependency relation D for π is a set of lock dependencies on π.

Lock graph [21, 22]: A lock graph corresponds to a lock dependency relation D where a node corresponds to a mutex, and for each lock dependency $\tau = \langle t, m, L \rangle \in D$, there is an edge from node $n \in L$ to node m with an attribute (F, τ, N), where F, N denote the events corresponding to acquisition of mutexes n and m, respectively.

Deadlock Cycle Condition (DCC) [21,22]: A finite sequence of edges of a lock graph $\langle (F_1, \langle t_1, m_1, L_1 \rangle, N_1) \cdots (F_k, \langle t_k, m_k, L_k \rangle, N_k) \rangle$ $(k > 1)$ corresponds to a *deadlock cycle* iff following conditions are satisfied:

- **(a)** $\forall_{i \neq j} i, j \in [1, k]$, $t_i \neq t_j$, i.e., all threads are distinct.
- **(b)** $\forall i \in [1, k]$, $m_i \in L_{i'}$ where $i' = (i + 1)\ mod\ k$, i.e., the mutex to be acquired is currently held by next thread in the cycle.
- **(c)** $\forall_{i \neq j} i, j \in [1, k]$, $\neg(N_i \prec F_j)$, i.e., each event N_i that will acquire a mutex m_i should not happen-before before an event F_j that acquired a mutex m_j.
- **(d)** $\forall_{i \neq j} i, j \in [1, k]$, $L_i \cap L_j = \emptyset$, i.e., pairwise locksets at N_i, N_j events are empty (and hence, all mutexes are distinct).

We refer the set of mutexes $\{m_1, \cdots, m_k\}$ as *cycle mutexes*. In such a deadlock cycle, for a given thread $t_{i'}$ where $i' = (i + 1)\ mod\ k$, we refer the acquire of m_i (i.e., $F_{i'}$ event) as the *first acquire*, and that of $m_{i'}$ (i.e., $N_{i'}$ event) as the *second acquire*. Also we refer m_i as the *first mutex*, and $m_{i'}$ as the *second mutex*, resp., w.r.t. $t_{i'}$.

For the trace in Fig. 1(b), we show the corresponding lock graph obtained in Fig. 2, where each edge is shown with attribute τ (for better readability). As per DCC, the cycle (m_1, m_2) is not allowed as the lockset has a common mutex m_g which violates condition (d). The detected cycles (m_g, m_1) and (m_g, m_2), however, do not cause deadlocks as the second acquires use `trylock` primitive.

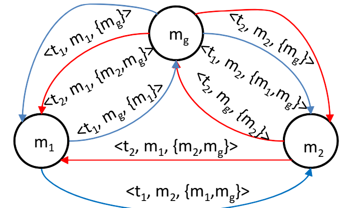


Fig. 2. A lock graph

3 Lock Cycles

We first focus on formalizing the conditions for inter-thread cycles, which we refer as *lock cycles*, that include deadlock and livelock potentials. We consider following cases (and their combination) of mutex acquisition: (i) nesting/non-nesting, (ii) blocking/non-blocking (such as using `lock/trylock`), (iii) exclusive/shared ownership (such as using `lock/lockS` or `trylock/trlockS`), and (iv) where a mutex can change state between shared and exclusive ownership directly without unlocking. These generalization are needed to handle real applications. First, we introduce a few definitions.

Each mutex object m held by a thread has following attributes: $m.id$ (id of the mutex), and $m.E$ (Boolean flag denoting if the mutex is held in exclusive state). A mutex

m' conflicts with m, denoted as $m \otimes m'$, iff they have same id, and at least one of them in exclusive state, i.e., $m'.id = m.id$, and $m'.E \vee m.E$.

A *lock-ordset*, denoted as $\overrightarrow{L}$, is an ordered set of mutexes currently held at a thread context, where (i) mutexes acquired earlier by the thread are ordered before the rest, (ii) no two mutexes have same id. A mutex m *conflicts* with $\overrightarrow{L}$, denoted as $m \otimes \overrightarrow{L}$, if there exists a mutex $m' \in \overrightarrow{L}$ s.t. m conflicts with m'. A lock-ordset $\overrightarrow{L}_1$ intersects with another $\overrightarrow{L}_2$, denoted as $\overrightarrow{L}_1 \cap \overrightarrow{L}_2$, iff there exists a conflicting pair of mutexes, i.e., $\exists m \in \overrightarrow{L}_1, m \otimes \overrightarrow{L}_2$. We use $m \cdot \overrightarrow{L}$ to denote a subset of lock-ordset that includes only those mutexes that were acquired before and including mutex $m' \in \overrightarrow{L}$ where $m'.id = m.id$. If such a mutex m' does not exist, them $m \cdot \overrightarrow{L}$ is an emptyset. Essentially, it excludes those mutexes that were acquired and held after m'.

Example: Consider threads t_0, t_1, t_2 acquiring/releasing mutexes m_0, m_1, m_2 in loops as shown in Fig. 3. The lock-ordset at context c_2 of t_0 is $\overrightarrow{L} = \{m_1, m_0\}$ as m_1 is acquired before m_0. Also, $m_0 \cdot \overrightarrow{L} = \{m_1, m_0\}$, and $m_1 \cdot \overrightarrow{L} = \{m_1\}$.

In general locking (such as non-nested), $m \cdot \overrightarrow{L}$ may not be the same as the lock-ordset at the time $m \in \overrightarrow{L}$ was acquired. Suppose $\overrightarrow{L} = \{m_1, m_2\}$ just after m_2 is acquired by a thread. If m_0 is acquired by the thread before m_1, but is released after acquiring m_1 but before m_2, then the lock-ordset at the time m_1 was acquired is $\{m_0, m_1\}$, but $m_1 \cdot \overrightarrow{L} = \{m_1\}$.

Lock order dependency: Given a trace π, a lock order dependency is a tuple $\tau = \langle t, m, \overrightarrow{L} \rangle$ of a thread $t \in T$, a mutex m, a lock-ordset $\overrightarrow{L}$ such that t will acquire m while holding all the mutexes in the lock-ordset $\overrightarrow{L}$. A lock order dependency relation $\overrightarrow{D}$ for π is a set of lock order dependencies on π. In Fig. 3, $\langle t_0, m_2, \{m_1, m_0\} \rangle$ denote a lock order dependency.

A *lock order graph* corresponds to a lock order dependency relation $\overrightarrow{D}$ where a node corresponds to a mutex, and for each lock order dependency $\tau = \langle t, m, \overrightarrow{L} \rangle \in \overrightarrow{D}$, there is an edge from a node $n \in \overrightarrow{L}$ to node m with an attribute (F, τ, N), where F, N denote the event corresponding to acquires of mutex n and m respectively. Two mutexes with same mutex id are mapped to the same node. In Fig. 3 each edge is annotated with an attribute τ and a thread-specific color for better readability.

Lock Cycle Condition (LCC): A finite sequence of edges of lock order graph $\langle (F_1, \langle t_1, m_1, \overrightarrow{L_1} \rangle, N_1) \cdots (F_k, \langle t_k, m_k, \overrightarrow{L_k} \rangle, N_k) \rangle$ $(k > 1)$ is a *lock cycle* iff following conditions are satisfied:

(a) $\forall_{i \neq j} i, j \in [1, k], t_i \neq t_j$, i.e., all threads are distinct.

(b) $\forall i \in [1, k], m_i \otimes \overrightarrow{L}_{i'}$ where $i' = (i+1) \; mod \; k$, i.e., mutex to be acquired conflicts with a mutex held by the next thread in the cycle.

(c) $\forall_{i \neq j} i, j \in [1, k], \neg (N_i \prec F_j)$, i.e., each event N_i that will acquire a mutex m_i should not happen-before before an event F_j acquiring m_j.

(d) $\forall_{i \neq j} i, j \in [1, k], m_i \cdot \overrightarrow{L}_{i'} \cap m_j \cdot \overrightarrow{L}_{j'} = \emptyset$, where $i' = (i+1) \; mod \; k$, $j' = (j + 1) \; mod \; k$, i.e., no two threads hold—at $N_{i'}, N_{j'}$ events—any common conflicting mutex *that was last acquired before $F_{i'}, F_{j'}$ events.*

Like in DCC, we refer the set of mutexes $\{m_1, \cdots, m_k\}$ as *cycle mutexes*. In a lock cycle, for a given thread $t_{i'}$ where $i' = (i+1)\ mod\ k$, we refer the acquire of m_i (i.e., $F_{i'}$) as the *first acquire*, and that of $m_{i'}$ (i.e., $N_{i'}$) as the *second acquire*. Also, we refer m_i as the *first mutex*, and $m_{i'}$ as the *second mutex*, resp., w.r.t. $t_{i'}$.

Example: Using LCC, we obtain the following lock cycles of length k, as shown in Fig. 3(a). Each cycle (of mutex nodes) is formed with edges of thread-specific colors.

k=2. (m_0, m_2), (m_0, m_3), (m_0, m_1)

k=3. (m_0, m_1, m_2), (m_0, m_3, m_1), (m_0, m_2, m_3), (m_1, m_2, m_3)

Consider the cycle (m_1, m_2, m_3) that corresponds to the sequence of lock order dependency: $\langle\langle t_0, m_2, \{m_1, m_0\}\rangle, \langle t_1, m_3, \{m_2, m_0\}\rangle, \langle t_2, m_1, \{m_3, m_0\}\rangle\rangle$. We obtain, $m_1 \cdot \{m_1, m_0\} = \{m_1\}$, $m_2 \cdot \{m_2, m_0\} = \{m_2\}$, and $m_3 \cdot \{m_3, m_0\} = \{m_3\}$. For the threads t_0, t_1, t_2, the first mutexes are m_1, m_2, m_3, respectively, and the second mutexes are m_2, m_3, m_1, respectively. The mutex m_0 held at the second acquire of each thread was last acquired after the first acquire. Clearly, condition LCC:(d) is satisfied (and so are LCC:(a)-(c)). However, DCC:(d) is not satisfied. Similarly, other cycles with length 3 are missed by DCC, although, it detects all cycles of $k = 2$.

Consider another example, shown in Fig. 3(b), where the acquisition of m_0 in threads t_1 and t_2 precede that of m_2 and m_3, respectively. The corresponding lock order graph is shown on the right. LCC detects only one lock cycle (m_0, m_1) (same as DCC). The cycle of mutexes (m_1, m_2, m_3) is not a lock cycle as $m_2 \cdot \{m_0, m_2\} \cap m_3 \cdot \{m_0, m_3\} = \{m_0\}$. Similarly, (m_1, m_0, m_3) is not a lock cycle.

Valid Cycle. We say a cycle is *valid* if all the threads in the cycle can acquire (respecting mutual exclusion and must happens-before orders) the corresponding first mutex before any thread can acquire the corresponding second mutex, and the second mutex of a thread conflicts with the corresponding first mutex acquired by another thread.

We now classify lock cycles into three distinct categories:

$deadlock$:	A lock cycle where all second acquires use `lock/lockS` (blocking)
$trylock$:	A lock cycle where all second acquires use `trylock/trylockS` (non-blocking)
$mixed$:	A lock cycle which is neither a trylock nor a deadlock

The only trylock cycle detected by LCC (in Fig. 3(a)) is (m_1, m_2, m_3). Other lock cycles detected (in Fig. 3(a)(b)) are mixed cycles, but no deadlock cycles.

Observation. A trylock cycle is an inter-thread cycle and a livelock potential as it can make all the threads (involved in the cycle) to fail in second acquires, release first mutexes, and repeat the acquires and releases in loops. In Section 4, we discuss a schedule orchestration technique to induce a livelock using a trylock cycle.

Note, a deadlock cycle has the potential to put the threads in a deadlock situation. A trylock/mixed cycle can not induce a deadlock situation, and a deadlock/mixed cycle can not induce a livelock situation.

3.1 Comparing LCC vs. DCC

To study the differences between LCC and DCC, we consider two cases based on whether `trylockS/lockS` primitives are *used*, or *not used at all*, resp.

Case A *(`trylockS/lockS` not used)*: LCC:(a)-(c) are essentially the same as DCC:(a)-(c), and DCC:(d) implies LCC:(d) conditions. Notably, (i) like DCC:(d), LCC:(d)

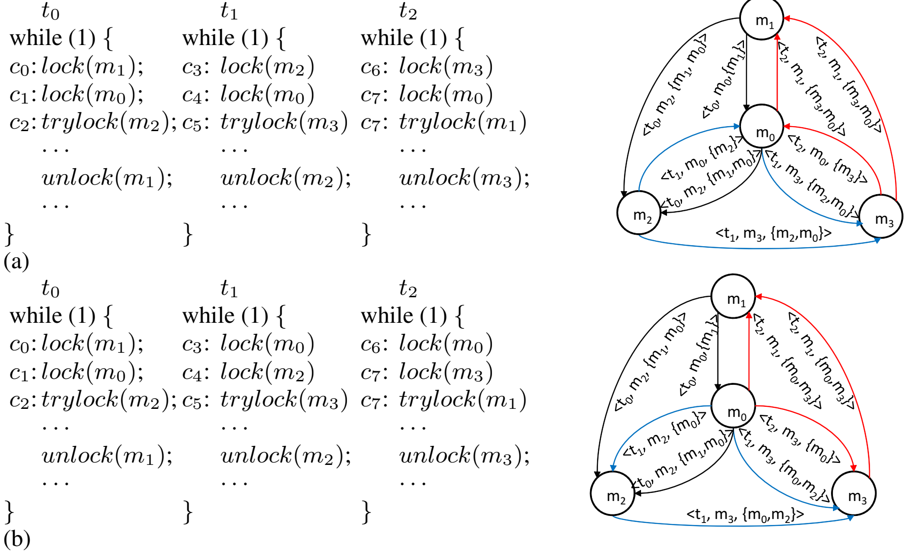


Fig. 3. Threads t_0 (black), t_1 (blue), t_2 (red) intra-thread loops (left), and lock-order graph (right). (a) A potential livelock involving lock cycle (m_1, m_2, m_3) is detected by LCC, but not by DCC. (b) The cycle involving (m_1, m_2, m_3) is not a lock cycle as LCC:(d) is not satisfied.

prohibits threads to have any common conflicting mutex that was acquired before first acquires and still held at second acquires (e.g., Fig. 3(b)), (ii) unlike DCC:(d), LCC(d) allows threads to have common conflicting mutexes that were acquired after first acquires but held at second acquires (e.g., Fig. 3(a). In other words, it allows intersecting lock-ordsets *at the time of second acquires*, i.e., for some i', j', $\overrightarrow{L_{i'}} \cap \overrightarrow{L_{j'}} \neq \emptyset$. With this relaxation, LCC can find more cycles than DCC. These cycles are also valid as per Lemma 2 (shown later). Given a trace π, let $cycle_{LCC}$ and $cycle_{DCC}$ denote a set of all cycles identified using LCC and DCC, respectively.

Lemma 1. *In the absence of `trylockS/lockS`, $cycle_{DCC} \subseteq cycle_{LCC}$.*

Proof. DCC:(d) implies LCC:(d). As the remaining conditions are identical in the absence of trylockS/lockS, proof follows. □

Case B *(`trylockS/lockS` used):* When mutexes are acquired in shared/exclusive states, DCC either detects a cycle that is spurious, or may miss valid cycles. Following two examples exemplify the limitation of DCC.

(DCC finds spurious cycles): Consider thread t_0 acquiring mutexes m_0, followed by m_1, and thread t_1 acquiring mutex m_1 followed by m_0. Assume all mutexes are acquired using `lockS` primitive. We *underline* a mutex to denote that it is acquired in a shared state. The corresponding lock dependency chain is $\langle\langle t_0, \underline{m_1}, \{\underline{m_0}\}\rangle, \langle t_1, \underline{m_0}, \{\underline{m_1}\}\rangle\rangle$. As per DCC, such a lock dependency chain is flagged as a deadlock cycle. However, as first and second mutexes are acquired in a shared state, they can not be conflicting, and hence, should not be flagged as a deadlock cycle. Since LCC:(b) is not satisfied, the cycle is not flagged as a lock cycle.

(DCC misses valid cycles): Consider two threads t_0, t_1 that acquired mutex m in shared state using `lockS`, and then both try to upgrade it to exclusive state using `lock`. The corresponding lock order dependency chain is $\langle\langle t_0, m_0, \{\underline{m}_0\}\rangle, \langle t_1, m_0, \{\underline{m}_0\}\rangle\rangle$. DCC will not flag it as a deadlock potential, as mutexes are not distinct, and locksets intersect (violating DCC:(d)). However, it is a valid deadlock cycle and is correctly flagged by LCC as lock-ordsets do not intersect, and acquire of second mutex in an exclusive state conflicts with the first mutex acquired in a shared state.

Lemma 2. *The set $cycle_{LCC}$ comprises only valid lock cycles in the presence of `lock/trylock/lockS/trylockS`primitives.*

Proof. As per LCC:(c)-(d), a thread $t_{i'}$ where $i' = (i+1)\ mod\ k$ ($i \in [1,k]$) should be able to acquire each mutex $m \in m_i \cdot \overrightarrow{L}_{i'}$ without waiting, However, it may block on a common conflicting mutex after the first acquire but before (or at the time of) the second acquire. Thus, all the cycles reported by LCC are valid lock cycles. □

4 CBuster: Livelock Analysis Tool

We discuss various phases of our tool CBuster (Cycle Buster), and focus primarily on inducing a livelock in an orchestrated execution. The tool CBuster comprises three phases: (I) collection of traces, (II) lock cycle detection, and (III) livelock confirmation.

In *phase I*, we instrument the program to collect various synchronization events. For occurrence of each synchronization event, we associate it with a calling thread, event type, a thread context, and a clock vector.

In *phase II*, from the collected trace events, we construct a set $\overrightarrow{D}$ of lock-order dependencies and a lock-order graph. We skip the exact implementation details, but we use many DCC-based optimization techniques [27,28]. These techniques reduce the graph size by removing nodes and edges that can not participate in any cycle as per DCC:(a)-(b). Since LCC:(a)-(b) are identical to DCC:(a)-(b) in the absence of `lockS/trylockS`, those optimization techniques are selectively applicable. On a reduced graph, we adapt techniques such as [28] to avoid identical cycles with the same set of lock order dependencies. Once we obtain cycles, we classify them into trylock, deadlock, and mixed cycles.

In *phase III*, we focus on inducing livelocks in a program re-execution (i.e., replay) using trylock cycles. From a trylock cycle $\langle(F_1, \langle t_1, m_1, \overrightarrow{L}_1\rangle, N_1)\cdots(F_k, \langle t_k, m_k, \overrightarrow{L_k}\rangle, N_k)\rangle$ ($k > 1$), we orchestrate a partial-order schedule of replay events which we repeat at least twice to expose and confirm a livelock situation. A replay event r has following attributes: $r.etype$ (event type), $r.cxt$ (thread context), and $r.m$ (mutex). For a given event e, we define a *matching* replay event r as follows: $r.etype := e.type$, and $r.cxt := e.cxt$. Due to dynamic allocation of mutex objects, we set $r.m$ during a mutex acquire, and use it to identify the corresponding mutex release event.

In our instrumentation framework, we insert a procedure, referred as `mutex_prehook` (not shown) which is invoked just before a mutex acquire/release operation. The procedure takes three arguments: the thread context of the current event (cxt), current event type $etype$, and current mutex object m. Instead of showing the pseudo-code (and nitty gritty details), we explain the schedule orchestration using local and global state transition diagram (shown in Figure 4) for better understanding.

Let each thread be in one of the following (abstract) states: head, body, tail, (as shown in Figure 4(a)). Let rh_t and rb_t denote matching replay events corresponding to events F_t and N_t of the thread t, and rt_t denote a replay event s.t. $rt_t.etype = unlock$. The guarded predicate $g_{i,j}$ and actions $a_{i,j}$ (shown in boxes) correspond to a local state transition from i to j, where $i, j \in${head, body, tail}.

The global (abstract) states of the system, shown in Figure 4(b), are Head, Body, Tail. Let $State$ denote a global variable that takes one of the state values: Head, Body, Tail, corresponding to a current global state. The global state transition occurs when the value of a global variable $Count$ is 0, at which point the value of $State$ is updated, and $Count$ is set to the length of the trylock, denoted as $CycleLen$ (=k). The variable $Count$ is used to synchronize the local transitions of threads. For e.g., a transition of a system from Head to Body occurs when all the threads (equal to $CycleLen$) have transitioned from head to body. Initially, all threads are in head state, and the global state ($State$) is Head. Also, $Count = CycleLen$.

Consider some thread t in head state, and $State$ = Head when mutex_hook is called. If the current event does not match rh_t (first acquire), i.e., $g_{head,head} = true$, t stays in head (self-loop, Box 1). However, if it matches rh_t i.e., $g_{head,body} = true$, the value of $Count$ is decreased by 1, $rt_t.m$ is set to m (needed to identify an unlock operation

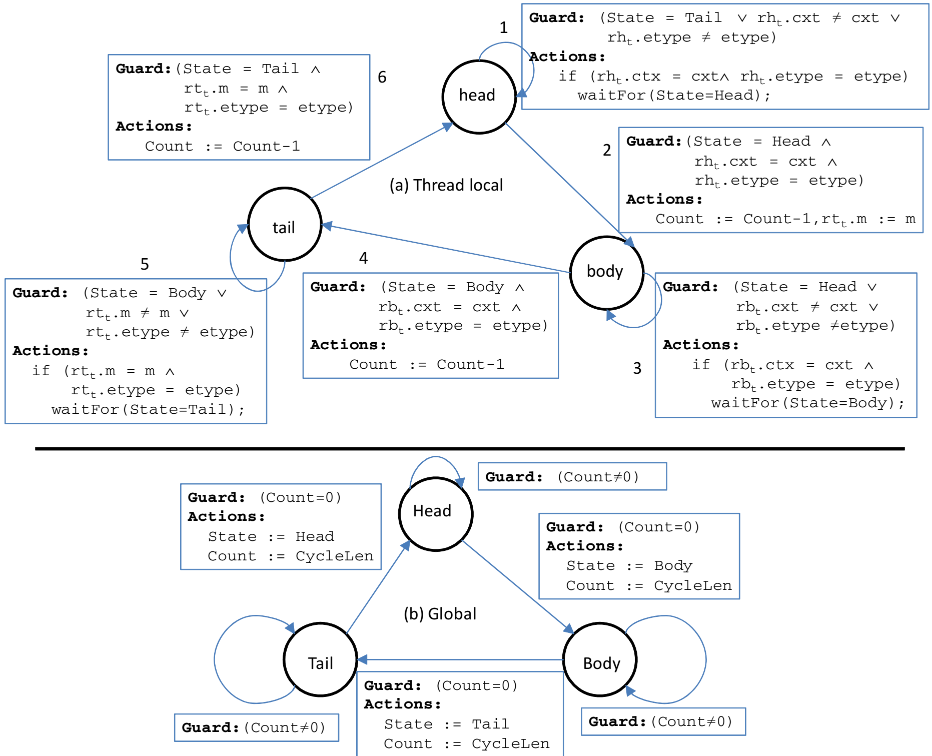


Fig. 4. Orchestrating schedules for inducing livelocks with intra-thread and inter-thread cycles. Illustrated with thread local (a) and global (b) state diagrams.

on m), and the thread transitions to body state (Box 2). In both scenarios, the current event is not blocked, i.e., mutex_prehook returns the control back to the event.

Consider a global state with a thread in body state, and $State$ = Head when mutex_hook is called. If the current event does not match rb_t (second acquire), then it is not blocked, and thread stays in body state (self-loop, Box 3). However, if it matches rb_t, then the thread waits in a loop inside mutex_prehook until $State$=Body (self-loop, Box 3). In other words, the current event is blocked. The value of $State$ is updated to Body (Figure 4(b)) only when all threads have transitioned to body state, i.e., threads were successful in executing events matching rh_t. Once the waiting is over i.e., $State$=Body, t transitions to tail state (Box 4) and mutex_prehook returns the control back.

Other state transitions can be similarly described. When we successfully induce the global state transition sequence Head→Body→Tail→Head at least twice, we report the confirmation of a livelock scenario. However, if the program takes a different path and the second occurrence of Head does not happen, then we don't report a livelock.

Other details: For maintaining same thread id during orchestration, we enforce the thread creation order as observed in the collected trace. We do so by ensuring atomicity—of fork invocation and the assignment of child thread id—to guarantee persistence of thread id across runs. During orchestration, each state transition—comprising $g_{i,j}$ and $a_{i,j}$—is made atomic using a global mutex. The context matching of the events are carried out, for e.g., by matching the respective call-site labels. The waitFor(c) primitive releases and re-acquires the global mutex in a loop until $c = true$.

5 Implementation and Experimentation

We implemented our approach in a light-weight instrumentation framework CBUSTER. We built an interposition library (in C/C++) and used LD_PRELOAD facility (available in Linux) to instrument and control the execution of the program during runtime. We also added necessary hooks such as mutex_prehook to confirm the livelock potentials. In our prototype implementation, we used a thread callstack (similar to [20, 28]) to represent a thread context.

We applied our approach to identify livelocks situations in a case study using SQLite-based application [8]. SQlite is a popular embedded light weight database engine (written in C) that supports ACID transactions. Unlike most other SQL databases, SQLite does not have a separate server process, and is built directly with client application as a single binary. SQLite reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. Two application processes (on same machine) can access same database file through a separate connection.

SQLite uses reader/writer locks to control access to the databases, allowing multiple process (or threads with separate connection to database) to do read query but only one to make changes to the database at any moment in time. On Unix system, the engine uses advisory file locks fcntl to implement the reader/writer locks. Moreover, the reader/writer locking are used in non-blocking style. When SQLite tries to access the database file that is (write) locked by another process, the default behavior is to return SQLITE_BUSY, thereby, preventing potential deadlocks. It also has a built-in mechanism to prevent write starvation, by disallowing new readers to acquire reader

locks from the moment the writer thread is waiting for the writer lock. However, the database does not prevent potential livelocks, which can happen when two competing and conflicting transactions repeatedly compete to access the database after getting SQLITE_BUSY errors.

SQLite supports two modes of atomic commit and rollback: rollback journal and Write-ahead-logging (WAL). In this case study, we consider the rollback journal mode. In this mode, the database is in one of the five lock states *unlock*, *shared*, *reserved*, *pending*, and *exclusive*. The database accesses are coordinated using three distinct flocks: `flock_P`, `flock_R`, and `flock_D` corresponding to pending, reserved, and database locks. Each flock is identified with the file-segment to lock. In unlock state, a thread cannot access a database. To get a read access, it has to first acquire the mutex `flock_P` in a shared state, followed by mutex `flock_D` in a shared state. It releases the mutex `flock_P` so that the mutex can be acquired in exclusive state when needed. Multiple reader threads can acquire `flock_D` in shared state similarly. If a thread wants to write, it has to acquire the mutex `flock_R` in exclusive state. If it succeeds, the database goes into reserved lock state. Only one thread can acquire mutex `flock_R`. While the database is in reserved lock state, new reader threads can still acquire mutex `flock_D` in shared state. The writer thread, while holding `flock_R` in exclusive state, acquires the mutex `flock_P` in exclusive state, and prevents new readers. This mechanism prevents writer starvation as no new shared lock will be granted. Once the database is in the pending state, the writer thread attempts to acquire mutex `flock_D` in exclusive state. Once the writer obtains the exclusive lock, it updates and commits the changes to database, releases all the mutexes i.e., `flock_D`, `flock_P`, `flock_R`. If the database is in shared state (i.e., `flock_D` is still held by some thread) while an exclusive lock `flock_D` is requested, SQLITE_BUSY error is returned to the application; in which the application has to rollback and retry the SQL request.

The flocks are acquired/released through `F_SETLK` command in the function call `fcntl`. The call with this command is non-blocking, and simply returns success/fail, like *trylock*. In our implementation, we extend our mutex handling ot flocks by uniquely identifying them. Further, we instrument the calls to `fnctl` to record the acquisition and release of flocks, and then classify them as `trylock/trylockS/unlock` events.

The application [8] that uses SQLite behaves as follows: There are two threads, `thread1` (t_1) and `thread2` (t_2), both access a common database file through separate connections. First thread "creates" a table object `scoreTbl`, and "inserts" two rows, before creating the second thread. After thread creation, both threads try to "update" the database concurrently. Each thread calls `sqlite3_exec` (a SQLite API) in a `while loop` with in a wrapper function to ensure that the updates succeeds eventually. In multiple runs of the code, we noticed that both threads succeed in 0-2 retrials most of the time.

In Figure 5(a), we show the sequence of relevant lock/trylock primitives in a good serialized trace of each thread (shown side-by-side) as collected by our tool CBUSTER. There are about 2K mutex acquire/release events. We searched for all lock cycles that satisfy LCC. The detection time is < 10 sec. Note, due to happens-before condition requirement LCC:(c), we do not generate any spurious cycles between "insert" and "update" queries. We detected a total of 10 lock cycles between the two "update" queries (which are not causally ordered). We group them if they have identical lock-order dependency chain with matching thread contexts of F_i and N_i events.

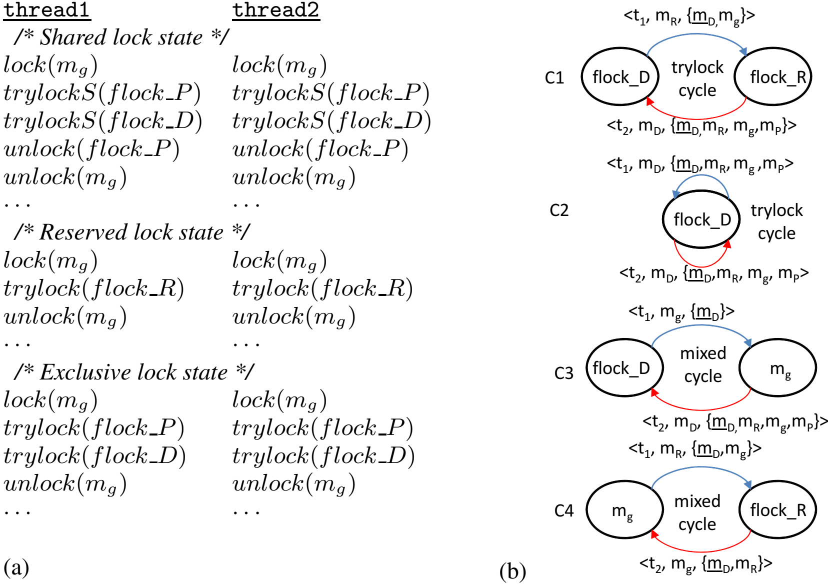


Fig. 5. (a) A trace of an SQLite application (based on [6,8]). Two threads eager to update database concurrently, but the database engine can potentially throw them in a livelock. In the serialized trace shown, however, both threads successfully updates database without retrying. (b) Lock cycle groups C1-C4 obtained using CBuster. C1-C2 are trylock cycles (confirmed livelocks), and C3-C4 are mixed cycles. Note, m_P is a shorthand for $flock_P$, and similarly others.

Out of 10 lock cycles, 4 of them are trylock cycles and are grouped into C1; another 4 of them are also trylock cycles and are grouped into C2; and the remaining are mixed cycles, C3 and C4. We show them in Figure 5(c). We use m_P as a shorthand notation for the mutex $flock_P$. Similarly, m_D, and m_R.

In C1 cycle, t_1 acquires `flock_D` in shared state, and then tries to get `flock_R` in exclusive state; while t_2 first acquires `flock_R` in exclusive state, and tries to get `flock_D` in exclusive state. The corresponding lock order dependency chain is $\langle\langle t_1, m_R, \{\underline{m_D}, m_g\}\rangle, \langle t_2, m_D, \{\underline{m_D}, m_g, m_R, m_P\}\rangle\rangle$. A mutex is underlined if it is acquired in shared state. The lock-ordsets $m_D \cdot \{\underline{m_D}, m_g\} = \{\underline{m_D}\}$ and $m_R \cdot \{\underline{m_D}, m_g, m_R, m_P\} = \{\underline{m_D}, m_g, m_R\}$ do not intersect. LCC:(a)-(d) are satisfied, and hence, it is a valid lock cycle. Furthermore, we were also able to induce livelocks for all the four trylock cycles in C1 group using our orchestrated scheduler, as discussed in Section 4.

In C2 cycle, t_1 and t_2 both acquire `flock_D` in shared states, then try to get `flock_D` in exclusive state. The corresponding lock order dependency chain is $\langle\langle t_1, m_D, \{\underline{m_D}, m_g, m_R, m_P\}\rangle, \langle t_2, m_D, \{\underline{m_D}, m_g, m_R, m_P\}\rangle\rangle$. All LCC conditions are satisfied, and therefore, it is a valid lock cycle. We were also able to induce livelocks for the four trylock cycles in this group.

For lack of space, we leave the discussion on mixed cycles C3, and C4. However, when we try our scheduler on them, we were able to force one thread to retry.

Our in-depth case study show the usefulness of the approach in detecting and confirming livelocks potentials in real application which can occur in intricate interactions between third-party library and user application.

6 Conclusion and Future Work

We presented dynamic livelock analysis framework for multi-thread programs, where we identify livelock potentials by examining a single execution trace of the program, and then induce a livelock by orchestrating a scheduler on a re-execution of the program. We also generalize cycle detection scheme to identify both deadlock and livelock potentials precisely in the presence various of mutex acquire schemes. We built a prototype tool and demonstrated its usefulness in a case study of SQLite-based application. We believe that similar livelock issues occur in the intricate interaction between application and third-party modules. In future, we would like to expand our case studies.

References

1. Ho, A., Smith, S., Hand, S.: On deadlock, livelock, and forward progress. Technical Report UCAM-CL-Tr-633, University of Cambridge, Computer Laboratory (2005)
2. Stallings, W.: Operating Systems: Internals and Design Principles. Prentice Hall (2001)
3. Mogul, J.C., Ramakrishnan, K.K.: Eliminating receive livelock in an interrupt-driven kemel. ACM Trans. Comput. Syst. 15(3), 217–252 (1997)
4. Tai, K.-C.: Definitions and detection of deadlock, livelock, and starvation in concurrent programs. In: ICPP, pp. 69–72 (1994)
5. Owicki, S.S., Lamport, L.: Proving liveness properties of concurrent programs. ACM Trans. Program. Lang. Syst. 4(3), 455–495 (1982)
6. Sqlite-Users, http://www.mail-archive.com/sqlite-users@sqlite.org/msg41725.html
7. Sqlite-Users, http://www.mail-archive.com/sqlite-users@sqlite.org/msg32658.html
8. Sqlite-Users, http://www.mail-archive.com/sqlite-users@sqlite.org/msg54618.html
9. Blieberger, J., Burgstaller, B., Mittermayr, R.: Static Detection of Livelocks in Ada Multitasking Programs. In: Abdennadher, N., Kordon, F. (eds.) Ada-Europe 2007. LNCS, vol. 4498, pp. 69–83. Springer, Heidelberg (2007)
10. Ouaknine, J., Palikareva, H., Roscoe, A.W., Worrell, J.: Static Livelock Analysis in CSP. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 389–403. Springer, Heidelberg (2011)
11. Holzmann, G.: The model checker spin. IEEE Transactions on Software Engineering (1997)
12. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: Proc. of ASE (2000)
13. Li, T., Lebeck, A.R., Sorin, D.J.: Spin detection hardware for improved management of multithreaded systems. IEEE Transactions on Parallel and Distrubuted Systems 17, 508–521 (2006)
14. Engler, D.R., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: SOSP, pp. 237–252 (2003)
15. Williams, A., Thies, W., Ernst, M.D.: Static Deadlock Detection for Java Libraries. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 602–629. Springer, Heidelberg (2005)
16. Shanbhag, V.K.: Deadlock-detection in Java-library using static-analysis. In: APSEC, pp. 361–368 (2008)

17. Naik, M., Park, C.-S., Sen, K., Gay, D.: Effective static deadlock detection. In: Proc. of ICSE, pp. 386–396 (2009)
18. Havelund, K.: Using Runtime Analysis to Guide Model Checking of Java Programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 245–264. Springer, Heidelberg (2000)
19. Wang, Y., Kelly, T., Kudlur, M., Lafortune, S., Mahlke, S.A.: Gadara: Dynamic deadlock avoidance for multithreaded programs. In: OSDI, pp. 281–294 (2008)
20. Jula, H., Tralamazza, D.M., Zamfir, C., Candea, G.: Deadlock immunity: Enabling systems to defend against deadlocks. In: OSDI (2008)
21. Bensalem, S., Havelund, K.: Dynamic Deadlock Analysis of Multi-threaded Programs. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 208–223. Springer, Heidelberg (2006)
22. Agarwal, R., Wang, L., Stoller, S.D.: Detecting Potential Deadlocks with Static Analysis and Run-Time Monitoring. In: Ur, S., Bin, E., Wolfsthal, Y. (eds.) HVC 2005. LNCS, vol. 3875, pp. 191–207. Springer, Heidelberg (2006)
23. Bensalem, S., Fernandez, J.-C., Havelund, K., Mounier, L.: Confirmation of deadlock potentials detected by runtime analysis. In: PADTAD, pp. 41–50 (2006)
24. Agarwal, R., Stoller, S.D.: Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In: PADTAD, pp. 51–60 (2006)
25. Joshi, P., Park, C.-S., Sen, K., Naik, M.: A randomized dynamic program analysis technique for detecting real deadlocks. In: Proc. of PLDI, pp. 110–120 (2009)
26. Joshi, P., Naik, M., Sen, K., Gay, D.: An effective dynamic analysis for detecting generalized deadlocks. In: FSE (2010)
27. Luo, Z.D., Das, R., Qi, Y.: Multicore SDK: A practical and efficient deadlock detector for real-world applications. In: ICST, pp. 309–318 (2011)
28. Cai, Y., Chan, W.K.: Magicfuzzer: Scalable deadlock detection for large-scale applications. In: Proc. of ICSE (2012)
29. Qin, F., Tucek, J., Sundaresan, J., Zhou, Y.: Rx: treating bugs as allergies - a safe method to survive software failures. In: SOSP, pp. 235–248 (2005)
30. Harrow, J.J.: Runtime Checking of Multithreaded Applications with Visual Threads. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 331–342. Springer, Heidelberg (2000)
31. SQLite home page, http://www.sqlite.org/
32. Mattern, F.: Virtual time and global states of distributed systems. In: Workshop on Parallel and Distributed Algorithms, France (1988)
33. Fidge, J.: Timestamps in message-passing systems that preserve the partial ordering. In: Australian Computer Science Conference (1988)

Scalable Dynamic Partial Order Reduction*

Jiri Simsa[1], Randy Bryant[1], Garth Gibson[1], and Jason Hickey[2]

[1] Carnegie Mellon University, Pittsburgh PA 15213, USA
[2] Google, Inc., Mountain View CA 94043, USA

Abstract. Systematic testing, first demonstrated in small, specialized cases 15 years ago, has matured sufficiently for large-scale systems developers to begin to put it into practice. With actual deployment come new, pragmatic challenges to the usefulness of the techniques. In this paper we are concerned with scaling dynamic partial order reduction, a key technique for mitigating the state space explosion problem, to very large clusters. In particular, we present a new approach for distributed dynamic partial order reduction. Unlike previous work, our approach is based on a novel exploration algorithm that 1) enables trading space complexity for parallelism, 2) achieves efficient load-balancing through time-slicing, 3) provides for fault tolerance, which we consider a mandatory aspect of scalability, 4) scales to more than a thousand parallel workers, and 5) is guaranteed to avoid redundant exploration of overlapping portions of the state space.

1 Introduction

Testing of concurrent programs is challenging because concurrency manifests as test non-determinism. A traditional approach to address this problem is stress testing, which repeatedly exercises concurrent operations of the program under test, hoping that eventually all concurrency scenarios of interest will be covered.

Unfortunately, as the scale of concurrent programs and the heterogeneity of environments in which these programs are deployed increases, the state space of possible scenarios explodes and stress testing stops being an effective mechanism for exercising all scenarios of interest.

To address the increasing complexity of software testing, researchers have turned their attention to systematic testing [8,12,14,18,19]. Similar to stress testing, systematic testing also repeatedly exercises concurrent operations of the program under test. However, unlike stress testing, systematic testing avoids test non-determinism by controlling the order in which concurrent operations happen, exercising different concurrency scenarios across different test executions.

* This research was sponsored by the U.S. Army Research Office under grant number W911NF0910273. The authors are also thankful to Google for providing its hardware and software infrastructure for the evaluation presented in this paper. Further, we also thank the members and companies of the PDL Consortium. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 19–34, 2013.
© Springer-Verlag Berlin Heidelberg 2013

To push the limits of systematic testing, existing tools combine of stateless exploration [8] with state space reduction [5,7,9] and parallel processing [21].

In this paper, we present a new method for distributed systematic testing of concurrent programs, which pushes the limits of systematic testing to an unprecedented scale. Unlike previous work [21], our approach is based on a novel exploration algorithm that 1) enables trading space complexity for parallelism, 2) achieves load-balancing through time-slicing, 3) provides for fault tolerance, which we consider a mandatory aspect of scalability, 4) scales to more than a thousand parallel workers, and 5) is guaranteed to avoid redundant exploration of overlapping portions of the state space.

The rest of the paper is organized as follows. Section 2 reviews stateless exploration, state space reduction, and parallel processing. Section 3 presents a novel exploration algorithm and details its use for distributed systematic testing at scale. Section 4 presents experimental evaluation of the implementation. Section 5 discusses related work and Section 6 presents the conclusions drawn from the results presented in this paper.

2 Background

In this section we give an overview of stateless exploration [8], dynamic partial order reduction (DPOR) [5], and distributed DPOR [21], which represent the state of the art in scalable systematic testing of concurrent programs.

2.1 Stateless Exploration

Stateless exploration is a technique that targets systematic testing of concurrent programs. The goal of stateless exploration is to explore the state space of different program states of a concurrent program by systematically enumerating different total orders in which concurrent events of the program can occur.

To keep track of the exploration progress, stateless exploration abstractly represents the state space of different program states using an *execution tree*. Nodes of the execution tree represent non-deterministic choice points and edges represent program state transitions. A path from the root of the tree to a leaf then uniquely encodes a program execution as a sequence of program state transitions.

Abstractly, enumeration of branches of the execution tree corresponds to enumeration of different sequences of program state transitions. Notably, the set of explored branches of a partially explored execution tree identifies which sequences of program state transitions have been explored. Further, assuming that concurrency is the only source of non-determinism in the program, the information collected by past executions can be used to generate schedules that describe in what order to sequence program state transitions of future executions in order to explore new parts of the execution tree.

Typically, stateless exploration uses depth-first search to explore the execution tree because of the space-efficient nature of its exploration, which is linear in the depth of the tree. Further, tools for stateless exploration such as VeriSoft [8] use

partial order reduction (POR) [7] to avoid exploration of equivalent sequences of program state transitions.

The pseudocode depicted in Algorithm 1 and 2 gives a high-level overview of stateless exploration. The ExplorePor algorithm maintains an exploration *frontier*, represented as a stack of sets of nodes, and uses depth-first search to explore the execution tree. The PersistentSet(*node*) function uses static analysis to identify what subtrees of the execution tree need to be explored. In particular, it inputs a node of the execution tree and outputs a subset of the children of this node that need to be explored in order to explore all *non-equivalent* sequences of program state transitions of the execution tree. The details behind the computation of PersistentSet(*node*) are beyond the scope of this paper and can be found in Godefroid's seminal treatment [7]. Note that our presentation of [8] omits the use of sleep sets [7]. This simplification is made to achieve consistency with other techniques [5,21] presented later in this section.

Algorithm 1. ExplorePor(*root*)

Require: A root node *root* of an execution tree.
Ensure: The execution tree rooted at the node *root* is explored.
1: *frontier* ← NewStack
2: Push({*root*}, *frontier*)
3: DFS-Por(*root*, *frontier*)

Algorithm 2. DFS-Por(*node*, *frontier*)

Require: A node *node* of an execution tree and a reference to a non-empty stack *frontier* of sets of nodes such that $node \in$ Top(*frontier*).
Ensure: The node *node* of the execution tree is explored and the exploration frontier *frontier* is updated according to the POR algorithm.
1: remove *node* from Top(*frontier*)
2: **if** PersistentSet(*node*) $\neq \emptyset$ **then**
3: Push(PersistentSet(*node*), *frontier*)
4: **for all** $child \in$ Top(*frontier*) **do**
5: navigate execution to *child*
6: DFS-Por(*child*, *frontier*)
7: **end for**
8: Pop(*frontier*)
9: **end if**

2.2 Dynamic Partial Order Reduction

Dynamic partial order reduction (DPOR) is a technique that targets efficient state space exploration [5,22]. The goal of DPOR is to further mitigate the combinatorial explosion of stateless exploration.

The stateless exploration discussed in the previous subsection uses static analysis to identify which subtrees of the execution tree need to be explored. However,

precise static analysis of complex programs is often costly or infeasible and results in larger than necessary persistent sets. To address this problem, DPOR computes persistent sets using dynamic analysis.

When stateless exploration explores an edge of the execution tree, DPOR computes the happens-before [13] and the independence [7] relations over the set of program state transitions. These two relations are then used to decide how to augment the existing exploration frontier.

The pseudocode depicted in Algorithm 3 and 4 gives a high-level overview of DPOR. The ExploreDpor algorithm maintains an exploration *frontier*, represented as a stack of sets of nodes, and uses depth-first search to explore the execution tree. The UpdateFrontier(*frontier*, *node*) function uses dynamic analysis to identify which subtrees of the execution tree need to be explored. In particular, the function inputs the current exploration frontier and the current node and computes the happens-before and independence relation between the transitions leading to the current node. This information is then used to infer which nodes need to be further added to the exploration frontier in order to explore all *non-equivalent* sequences of program state transitions. Importantly, the function modifies the exploration frontier in a *non-local* fashion as it can add nodes to an arbitrary set of the exploration frontier stack. The details behind the computation of UpdateFrontier(*frontier*, *node*) are beyond the scope of this paper and can be found in the original paper [5].

Algorithm 3. ExploreDpor(*root*)

Require: A root node *root* of an execution tree.
Ensure: The execution tree rooted at the node *root* is explored.

```
1: frontier ← NewStack
2: Push({root}, frontier)
3: Dfs-Dpor(root, frontier)
```

Algorithm 4. Dfs-Dpor(*node*, *frontier*)

Require: A node *node* of an execution tree and a reference to a non-empty stack *frontier* of sets of nodes such that *node* ∈ Top(*frontier*).
Ensure: The node *node* of the execution tree is explored and the exploration frontier *frontier* is updated according to the DPOR algorithm.

```
 1: remove node from Top(frontier)
 2: UpdateFrontier(frontier, node)
 3: if Children(node) ≠ ∅ then
 4:     child ← arbitrary element of Children(node)
 5:     Push({child}, frontier)
 6:     for all child ∈ Top(frontier) do
 7:         navigate execution to child
 8:         Dfs-Dpor(child, frontier)
 9:     end for
10:     Pop(frontier)
11: end if
```

2.3 Distributed Dynamic Partial Order Reduction

Distributed DPOR is a technique that targets concurrent stateless exploration. The goal of distributed DPOR is to offset the combinatorial explosion of possible permutations of concurrent events through parallel processing.

At a first glance, parallelization of DPOR seems straightforward: assign different parts of the execution tree to different *workers* and explore the execution tree concurrently. However, as pointed out by Yang et al. [21], such a parallelization suffers from two problems. First, due to the non-local nature in which DPOR updates the exploration frontier, different workers may end up exploring identical parts of the state space. Second, since the sizes of the different parts of the execution tree are not known in advance, the load-balancing needed to enable linear speedup is non-trivial.

To address these two problems, Yang et al. [21] proposed two heuristics. Their first heuristic modifies Flanagan and Godefroid's lazy addition of nodes to the exploration frontier [5] so that they add nodes to the exploration frontier eagerly. As evidenced by their experiments, replacing lazy addition with eager addition mitigates the problem of redundant exploration of identical parts of the execution tree by different workers. Their second heuristic assumes the existence of a centralized load-balancer that workers can contact in case they believe they have too much work on their hands and would like to offload some of the work. The centralized load-balancer keeps track of which workers are idle and which workers are active and facilitates offloading of work from active to idle workers.

3 Scalable Dynamic Partial Order Reduction

While scaling distributed DPOR to a large cluster at Google [15], we have identified several problems with previous work of Yang et al. [21].

First, at large scale, the algorithm must explicitly cope with the failure of worker processes or machines. Although Yang et al. suggest how fault tolerance could be implemented, they do not quantify how their design affects scalability. Second, although the out-of-band centralized load-balancer of Yang et al. renders the communication overhead negligible, it precludes features that are enabled by centralized collection of information such as support for fault tolerance or state space size estimation. Third, the load-balancing of Yang et al. uses a heuristic based on a threshold to offload work from active to idle workers. It is likely that for different programs and different number of workers, different threshold values should be used. However, Yang et al. provide no insight into the problem of selecting a good threshold. Fourth, their DPOR modification for avoiding redundant exploration is a heuristic does not guarantee zero redundancy.

In this section we present an alternative design for distributed DPOR. Our design is centralized and uses a single master and n workers to explore the execution tree. Despite its centralized nature, our experiments show that our design scales to more than a thousand workers. Unlike previous work, our design can tolerate worker faults, is guaranteed to avoid redundant exploration, and is based

on a novel exploration algorithm that allows 1) trading off space complexity for parallelism and 2) efficient load-balancing through time-slicing.

3.1 Novel Exploration Algorithm

The key advantage of using depth-first search for the purpose of DPOR is its favorable space complexity [8]. In fact, experience with systematic testing of concurrent programs based on stateless exploration [5,14,16,19] suggests that the bottleneck for stateless exploration is CPU power, and not memory size.

To enable parallel processing, Yang et al. [21] depart from the strict depth-first search nature of stateless exploration. Instead, the execution tree is explored using a collection of (possibly overlapping) depth-first searches and the exploration order is determined by a load-balancing heuristic.

Algorithm 5. ExploreDpor(n, *root*)

Require: A positive integer n and a root node *root* of an execution tree.
Ensure: The execution tree rooted at the node *root* is explored.

1: *frontier* $\leftarrow$ NewSet
2: Insert(Push({*root*}, NewStack), *frontier*)
3: **while** Size(*frontier*) > 0 **do**
4: Partition(*frontier*, n)
5: *fragment* $\leftarrow$ an arbitrary element of *frontier*
6: *node* $\leftarrow$ an arbitrary element of Top(*fragment*)
7: Pdfs-Dpor(*node*, *fragment*, *frontier*)
8: **if** Size(*fragment*) $= 0$ **then**
9: Remove(*fragment*, *frontier*)
10: **end if**
11: **end while**

Algorithm 6. Partition(*frontier*, n)

Require: A non-empty set *frontier* of non-empty stacks of sets of nodes and a positive integer n such that $n \geq$ Size(*frontier*).
Ensure: Size(*frontier*) $= n$ or $\forall$*fragment* $\in$ *frontier* : the number of nodes contained in *fragment* is 1.

1: **for all** *fragment* $\in$ *frontier* **do**
2: **if** Size(*frontier*) $= n$ **then**
3: **return**
4: **end if**
5: **while** the number of nodes contained in *fragment* is greater than 1 and Size(*frontier*) $< n$ **do**
6: *node* $\leftarrow$ an arbitrary element of a set contained in *fragment*
7: remove *node* from *fragment*
8: *new-fragment* $\leftarrow$ a new frontier fragment for *node*
9: Insert(*new-fragment*, *frontier*)
10: **end while**
11: **end for**

To overcome the limitations mentioned above, we have designed a novel exploration algorithm, called *n-partitioned depth-first search*, which relaxes the strict depth-first search nature of DPOR in a controlled manner and, unlike traditional depth-first search, is amenable to parallelization.

For the sake of the presentation, we first present a sequential version of DPOR based on the n-partitioned depth-first search. The main difference between depth-first search and n-partitioned depth-first search is that the exploration frontier of the new algorithm is partitioned into up to n frontier *fragments* and the new algorithm explores each fragment using a depth-first search interleaving exploration of different fragments.

The pseudocode depicted in Algorithm 5, 6, and 7 gives a high-level overview of DPOR algorithm based on the n-partitioned depth-first search. The algorithm maintains an exploration *frontier*, represented as a set of up to n stacks of sets of nodes. The elements of the exploration frontier are referred to as *fragments* and together they form a partitioning of the exploration frontier. The execution tree is explored by interleaving depth-first search exploration of frontier fragments. Algorithm 5 implements this idea by repeating two steps – PARTITION and PDFS-DPOR – until the execution tree is fully explored.

Algorithm 7. PDFS-DPOR(*node*, *fragment*, *frontier*)

Require: A node *node* of an execution tree, a reference to a non-empty stack *fragment* of sets of nodes such that *node* $\in$ TOP(*fragment*), and a reference to a set *frontier* of non-empty stacks of sets of nodes.

Ensure: The node *node* of the execution tree is explored and the fragment *fragment* of the exploration frontier is updated according to DPOR.

```
1: remove node from Top(fragment)
2: UpdateFrontier(frontier, fragment, node)
3: if Children(node) ≠ ∅ then
4:     child ← arbitrary element of Children(node)
5:     Push({child}, fragment)
6:     navigate execution to child
7: end if
8: pop empty sets from the fragment stack
```

The PARTITION step is detailed in Algorithm 6. During the PARTITION step, the current frontier is inspected to see whether existing frontier fragments should be and can be further partitioned. A new frontier fragment *should* be created in case there is less than n frontier fragments. A new frontier fragment *can* be created if there exists a frontier fragment with at least two nodes.

The PDFS-DPOR step is detailed in Algorithm 7. The PDFS-DPOR step is given one of the frontier fragments and uses depth-first search to explore the next edge of the subtree induced by the selected frontier fragment (the subtree that contains all ancestors and descendants of the nodes contained in the selected frontier fragment). The UPDATEFRONTIER(*frontier*, *fragment*, *node*) function operates in a similar fashion to the UPDATEFRONTIER(*frontier*, *node*) function described in the previous section. The main distinction is that after the function identifies

which nodes are to be added to the exploration frontier using Flanagan and Godefroid's algorithm [5], these nodes are added to the current frontier fragment only if they are not already present in some other fragment. This way, the set of sets of nodes contained in each fragment remains a partitioning of the exploration frontier – an invariant maintained throughout our exploration that which helps our design to avoid redundant exploration.

3.2 Parallelization

In this subsection we describe how to efficiently parallelize the above sequential DPOR design based on n-partitioned depth-first search.

First, observe that the presence or absence of the PARTITION step in the body of the main loop of the EXPLOREDPOR function of Algorithm 5 has no effect on the correctness of the algorithm. This allows us to sequence several PDFS-DPOR steps together, which hints at possible distribution of the exploration.

Namely, one could spawn concurrent workers and use them to carry out sequences of PDFS-DPOR steps over different frontier fragments. However, a straightforward implementation of this idea would require synchronization when concurrent workers access and update the exploration frontier, which is shared

---

Algorithm 8. EXPLOREDISTRIBUTEDDPOR(n, *budget*, *root*)

Require: A positive integer n, a time budget *budget* for worker exploration, and a root node *root* of an execution tree.
Ensure: The execution tree rooted at the node *root* is explored.
1: *frontier* ← NEWSET
2: INSERT(PUSH(*root*, NEWSTACK), *frontier*)
3: **while** SIZE(*frontier*) > 0 **do**
4: PARTITION(*frontier*, n)
5: **while** exists an idle worker and an unassigned frontier fragment **do**
6: *fragment* ← an arbitrary unassigned element of *frontier*
7: SPAWN(EXPLORELOOP, *fragment*, *budget*, EXPLORECALLBACK)
8: **end while**
9: wait until signaled by EXPLORECALLBACK
10: **end while**

Algorithm 9. EXPLORELOOP(*fragment*, *budget*)

Require: A non-empty stack *fragment* of sets of nodes.
Ensure: Explores previously unexplored branches of the subtree induced by the nodes of *fragment* until all branches are explored or the timeout expires.
1: *start-time* ← GETTIME
2: **repeat**
3: *node* ← an arbitrary element of TOP(*fragment*)
4: PDFS-DPOR(*node*, *fragment*)
5: *current-time* ← GETTIME
6: **until** *current-time* − *start-time* > *budget* or SIZE(*fragment*) $= 0$

---

by all workers. The trick to overcome this obstacle to efficient parallelization is to give each worker a private copy of the execution tree. As pointed out by Yang et al. [21], such a copy can be concisely represented using the state of the depth-first search stack of the frontier fragment to be explored.

A worker can then repeatedly invoke the PDFS-DPOR function over (a copy of) the assigned frontier fragment. Once the worker either completes the exploration of the assigned frontier fragment or it exceeds a time budget allocated for its exploration, it reports back with the results of the exploration. The exploration progress can be concisely represented using the original and the final state of the depth-first search stack of the assigned frontier fragment.

The pseudocode depicted in Algorithm 8 and 9 presents a high-level approximation of the actual implementation of our distributed DPOR. The implementation operates with the concept of "fragment assignment". When a frontier fragment is created, it is unassigned. Later, a fragment becomes assigned to a particular worker through the invocation of the SPAWN function. When the worker finishes its exploration, or exhausts the time budget assigned for exploration, it reports back the results, the fragment assigned to this worker becomes unassigned again. The results of worker exploration are mapped back to the "master" copy of the execution tree using the EXPLORECALLBACK callback function. The time budget for worker exploration is used to achieve load-balancing through time-slicing. The PARTITION function behaves identically to the original one, except for the fact that it partitions unassigned fragments only.

Algorithm 9 presents the pseudocode of the EXPLORELOOP function, which is executed by a worker. The PDFS-DPOR function is identical to the sequential version of the algorithm. The workers are started through the SPAWN function which creates a private copy of a part of the execution tree. Notably, the copy contains only the nodes that the worker needs to further the exploration of the assigned frontier fragment. Structuring the concurrent exploration in this fashion enables both multi-threaded and multi-process implementations.

Since our goal has been to scale the stateless exploration to thousands of workers, the scale of clusters available today, our implementation implements each worker as an RPC server running as a separate process. In such a setting, the SPAWN function issues an asynchronous RPC request that triggers invocation of the EXPLORELOOP function with the appropriate arguments at the RPC server of the worker. The response to the RPC request is then handled asynchronously by the EXPLORECALLBACK function, which maps the results of the worker exploration into the master copy of the execution tree and resumes execution of the main loop of Algorithm 8.

3.3 Fault Tolerance

As is commonly done in large distributed applications [4,6], failure of one out of thousands of nodes must be anticipated and handled gracefully, but failure of just one particular node is infrequent enough to be dealt with using re-execution.

In accordance with this practice, our design assumes that the master, which is running the ExploreDistributedDpor function, will not fail. The workers on the other hand are allowed to fail and the exploration can tolerate such events.

In particular, an RPC request issued by the master to a worker RPC server uses a deadline to decide whether the worker has failed. The value of the deadline is set commensurately to the value of the worker time budget.

When the deadline expires without an RPC response arriving, the master simply assumes that the worker has failed and makes no changes to the frontier fragment originally assigned to the failed worker. The fragment becomes unassigned again and other workers get a chance to further its exploration.

3.4 Load-Balancing

The key to high utilization of the worker fleet is effective load-balancing. To achieve load-balancing, our design time-slices frontier fragments among available workers. The availability of frontier fragments is impacted by two factors.

The first factor is the upper bound n on the number of frontier fragments that the ExploreDistributedDpor creates. This parameter determines the size of the pool of available work units. The higher this number, the higher the memory requirements of the master but the higher the opportunity for parallelism. In our experience, setting n to twice the number of workers worked fairly well. Future work on dynamic selection of the number of workers might be beneficial if the impact on the parallelism and memory use can be managed.

The second factor is the size of the time slice used for worker exploration. Smaller time slices lead to more frequent generation of new fragments but this elasticity comes at the cost of higher communication overhead. In our initial design we used a fixed time budget, choosing the value of 10 seconds as a good compromise for the elasticity vs. communication overhead trade-off. However, the initial evaluation of our prototype made us realized that a variable time budget improves worker fleet utilization at large scales.

In particular, we observed that as the number of workers increases, a gap between the realized and the ideal speed up opens up. Our investigation identified time periods during the exploration with insufficient number of frontier fragments to keep all workers busy.

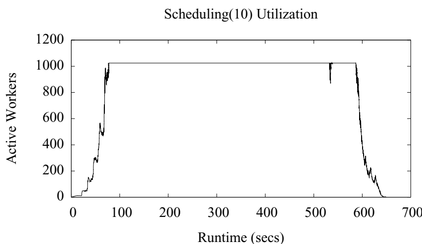


Fig. 1. Without Optimizations

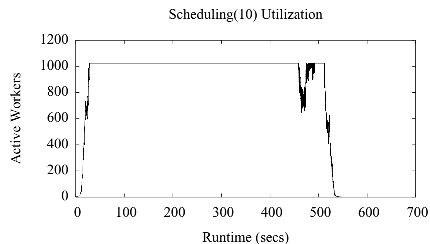


Fig. 2. With Optimizations

To study this problem, we recorded the number of active workers during the lifetime of a test. Figure 1 plots this information for one of our test programs on a configuration with $1,024$ workers and an upper bound of $2,048$ frontier fragments. The figure is representative of other measurements at such a scale.

One can identify three phases of the exploration. In the first phase, the number of active workers gradually increases over 100 seconds until there is enough frontier fragments to keep everyone busy. In the second phase, all workers are kept busy. In the third phase, the number of active workers gradually decreases to zero over 100 seconds. Ideally, the first and the third phase should be as short as possible in order to minimize the inefficiency resulting from not fully utilizing the available worker fleet.

To this aim, we have developed a technique based on a variable time budget. In particular, if the exploration is configured to use a time budget b, the master actually uses fractions of b proportional to the number of active workers. For example, the first worker will receive a budget of $\frac{b}{n}$, where n is the number of workers. When half of the workers are active, the next worker to be assigned work will receive a budget of $\frac{b}{2}$. The scaling of the time budget is intended to reduce the time before the master gets the opportunity to re-partition and load-balance and thus to reduce the duration of the first and the third phase.

We implemented this technique and re-ran the our scalability measurements. For comparison with Figure 1, Figure 2 plots the number of active workers over time for the optimized implementation. For this test program, the two techniques reduced the runtime from 655 seconds to 527 seconds. Similar runtime improvements have been achieved for other test programs.

3.5 Avoiding Redundant Exploration

For clarity of presentation, Algorithm 8 omits a provision that prevents concurrent workers from exploring overlapping portions of the execution tree. This could happen when two workers make concurrent UpdateFrontier calls and add identical nodes to their frontier fragment copies.

To avoid this problem, our implementation introduces the concept of "node ownership". A worker exclusively owns a node if it is contained in the original frontier fragment currently assigned to the worker, or is a descendant of a node that the worker owns. All other nodes are assumed to be shared with other workers and the node ownership restricts which nodes a worker may explore.

In particular, the depth-first search exploration of a worker is allowed to operate only over nodes that the worker owns. When it encounters a shared node during its exploration, the worker terminates its exploration and sends an RPC response to the master indicating which nodes of the final frontier fragment are shared. The ExploreCallback function checks which newly discovered shared nodes are already part of some other frontier fragment. If a newly discovered node is not part of some other fragment, the node is added to the master copy of the currently processed frontier fragment (ownership is claimed). Otherwise,

the ownership of the node has been already claimed and the node is not added to the master copy of the currently processed frontier fragment.

Although this provision could in theory lead to increased communication overhead and decreased worker fleet utilization, our experiments indicate that in practice the provision does not affect performance.

4 Evaluation

To evaluate our design, we implemented its prototype on top of ETA [15], a tool developed at Google used for systematic testing of multi-threaded components of a cluster management system. These components are written using a library based on the actors paradigm [1] and the ETA tool is used to systematically enumerate different total orders in which messages between actors can be delivered in order to exercise different concurrency scenarios.

4.1 Experimental Setup

For the purpose of evaluation of our implementation we have used instances of the three following tests. The Resource(x,y) test is representative of a class of actor program tests that evaluate interactions of x different users that acquire and release resources from a pool of y resources. The Store(x,y,z) test is representative of a class of actor program tests that evaluate interactions of x users of a distributed key-value store with y front-end nodes and z backend nodes. The Scheduling(x) test is representative of a class of actor program tests that evaluate interactions of x users issuing concurrent scheduling requests. These tests exercise fundamental functionality of core components of the cluster management system and are part of the unit test suite of the system.

Unless stated otherwise, each measurement presented in the remainder of this section presents a run of a complete exploration of the given test and the results report the mean and the standard deviation of three repetitions of a run. Lastly, all experiments were carried out on a Google data center using stock hardware and running each process on a separate virtual machine.

4.2 Faults

First, we evaluated the ability of the implementation to handle worker failures. Notably, we extended the ETA tool with an option to inject an RPC fault with a certain probability. When an RPC fault is injected, the master fails to receive the RPC response from a worker and waits for the RPC to timeout instead.

Our experiments with injected RPC faults have demonstrated that the runtime increases proportionally to the underlying geometric progression (of repeated RPC failures). For example, if each RPC had a 50% chance of failing, the runtime doubled. Since in actual deployments of ETA, RPC requests fail with probability well under 1%, our support for fault tolerance is practical.

4.3 Scalability

Next, to measure the scalability of the implementation, we compared the time needed to complete an exploration by a sequential implementation of DPOR against the time needed to complete the same exploration by our distributed implementation. We considered configurations with 32, 64, 128, 256, 512, and 1,024 workers and applied the algorithm to the Resource(6,6), Store(12,3,3), and Scheduling(10) actor program tests, parameters of which were chosen to stimulate interesting state space sizes.

These experiments were run inside of a dynamically shared cluster; that is, machines running worker processes are shared with other workloads. The time budget of each worker exploration was set to 10 seconds and the target number of frontier fragments was set to twice the number of workers.

The results of Resource(6,6), Store(12,3,3), and Scheduling(10) experiments are presented in Figure 3, Figure 4, and Figure 5 respectively. Due to the magnitude of the state spaces being explored, the runtime of the sequential algorithm was extrapolated using a partial run. The figures visualize the speedup over the extrapolated runtime of the sequential algorithm and compare it to the ideal speedup. Note that both axes of the graphs are in logarithmic scale.

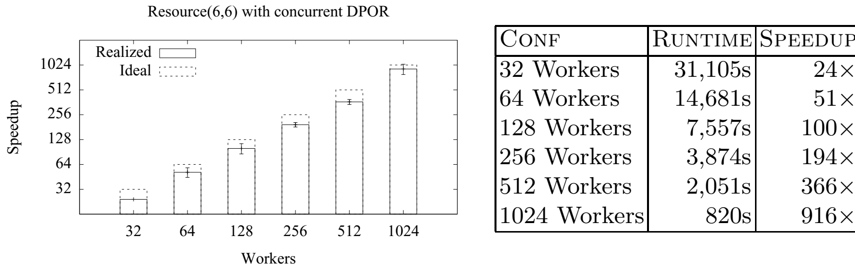
These results evidence the scalability of our implementation of DPOR at a large scale. The largest configuration uses 1,024 workers and our implementation achieves speedup that ranges between 760× and 920×.

4.4 Theoretical Limits

Finally, we carried out measurements that helped us to evaluate the theoretical scalability limits of our implementation. The purpose of the section is to project future bottlenecks. To this aim we focused on measuring memory and CPU requirements of the master.

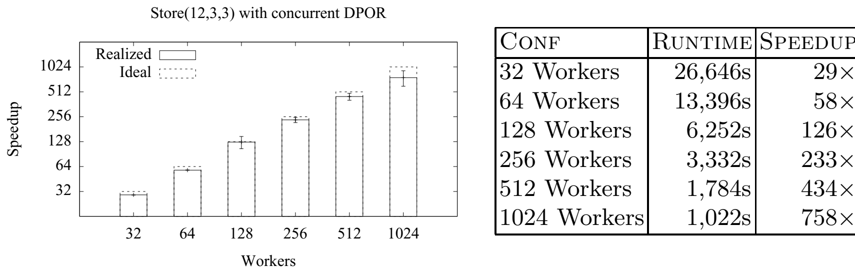
Memory Requirements: The memory overhead of our implementation is dominated by the cost to store the master copy of the exploration frontier. To estimate the overhead, we measured the amount of memory allocated for the explicitly stored nodes of the execution tree and the exploration frontier data structures over time. For the Scheduling(10) test on a configuration with 1,024 workers and an upper bound of 2,048 frontier fragments, the peak number of the allocated memory was less than 4MB. This number is representative of results for other tests at such a scale. Consequently, for the current computer architectures, the memory requirements scale to millions of workers.

CPU Requirements: With 1024 workers and a 10-second time budget, the master is expected to issue around 100 RPC requests and to process around 100 RPC responses every second. For such a load, the stock hardware running exclusively the master process experiences peak CPU utilization under 20%. Consequently, for the current computer architectures, the CPU requirements scale to around 5,000 workers. To scale our implementation beyond that, one can scale the time budget, hardware performance, or optimize the software stack.



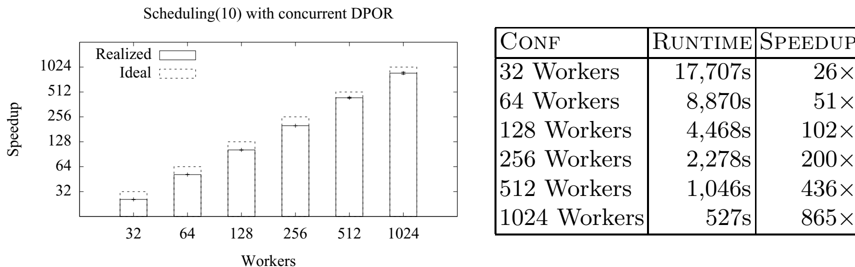
CONF	RUNTIME	SPEEDUP
32 Workers	31,105s	24×
64 Workers	14,681s	51×
128 Workers	7,557s	100×
256 Workers	3,874s	194×
512 Workers	2,051s	366×
1024 Workers	820s	916×

Fig. 3. For this example, DPOR explores on the order of 18.5 million branches and the sequential implementation is expected to require 209 hours to finish



CONF	RUNTIME	SPEEDUP
32 Workers	26,646s	29×
64 Workers	13,396s	58×
128 Workers	6,252s	126×
256 Workers	3,332s	233×
512 Workers	1,784s	434×
1024 Workers	1,022s	758×

Fig. 4. For this example, DPOR explores on the order of 21 million branches and the sequential implementation is expected to require 215 hours to finish



CONF	RUNTIME	SPEEDUP
32 Workers	17,707s	26×
64 Workers	8,870s	51×
128 Workers	4,468s	102×
256 Workers	2,278s	200×
512 Workers	1,046s	436×
1024 Workers	527s	865×

Fig. 5. For this example, DPOR explores on the order of 3.6 million branches and the sequential implementation is expected to require 126 hours to finish

For instance, one could replace the single master with a hierarchy of masters. The performance of our algorithm shows that hierarchical organization is not needed to scale to the size of state of the art cluster.

5 Related Work

Concurrent state space exploration have been previously studied in the context of several projects: **Inspect** [20] is a tool for systematic testing of `pthreads` C

programs that implements the distributed DPOR [21] discussed in Section 2. Unlike our work, the Inspect tool does not support fault tolerance, is not guaranteed to avoid redundant exploration, and has not been demonstrated to scale beyond 64 workers. **DeMeter** [10] provides a framework for extending existing sequential model checkers [12,19] with a parallel and distributed exploration engine. Similar to our work, the framework focuses on efficient state space exploration of concurrent programs. Unlike our work, the design has not been thoroughly described or analyzed and has been demonstrated to scale only up to 32 workers. **Cloud9** [3] is a parallel engine for symbolic execution of sequential programs. In comparison to our work, the state space being explored is the space of all possible programs inputs. Systematic enumeration of different program inputs is an orthogonal problem to the one addressed by this paper. Parallelization of software verification was also investigated in the context of explicit state space model checkers in tools such as **MurPhi** [17], **DiVinE** [2], or **SWARM** [11]. Stateful exploration is less common for implementation-level model checkers [8,15,19] where storing a program state explicitly becomes prohibitively expensive.

6 Conclusions

This paper presented a technique that improves the state of the art of scalable techniques for systematic testing of concurrent programs. Our design for distributed DPOR enables the exploitation of a large-scale cluster for the purpose of systematic testing. At the core of the design lies a novel exploration algorithm, n-partitioned depth-first search, which has proven to be essential for scaling our design to thousands of workers.

Unlike previous work [21], our design provides support for fault tolerance, a mandatory aspect of scalability, and is guaranteed to avoid redundant exploration of identical parts of the state space by different workers. Further, our implementation and deployment of a real-world system at scale has demonstrated that the design achieves almost linear speed up for up to $1,024$ workers. Lastly, we carried out a theoretical analysis of the design to identify scalability bottlenecks of the design.

References

1. Agha, G.: Actors: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge (1986)
2. Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: Parallel Distributed Model Checker (Tool paper). In: HiBi/PDMC 2010, pp. 4–7. IEEE (2010)
3. Bucur, S., Ureche, V., Zamfir, C., Candea, G.: Parallel symbolic execution for automated real-world software testing. In: EuroSys 2011, pp. 183–198 (2011)
4. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: BigTable: A distributed storage system for structured data. In: OSDI 2006, pp. 205–218 (2006)
5. Flanagan, C., Godefroid, P.: Dynamic Partial Order Reduction for Model Checking Software. SIGPLAN Not. 40(1), 110–121 (2005)

6. Ghemawat, S., Gobioff, H., Leung, S.: The Google file system. SIGOPS Oper. Syst. Rev. 37(5), 29–43 (2003)
7. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Springer (1996)
8. Godefroid, P.: Model Checking for Programming Languages using VeriSoft. In: POPL 1997, pp. 174–186. ACM (1997)
9. Gueta, G., Flanagan, C., Yahav, E., Sagiv, M.: Cartesian Partial-Order Reduction. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 95–112. Springer, Heidelberg (2007)
10. Guo, H., Wu, M., Zhou, L., Hu, G., Yang, J., Zhang, L.: Practical software model checking via dynamic interface reduction. In: SOSP 2011, pp. 265–278. ACM, New York (2011)
11. Holzmann, G.J., Joshi, R., Groce, A.: Swarm Verification Techniques. IEEE Transactions on Software Engineering 37, 845–857 (2011)
12. Killian, C.E., Anderson, J.W., Jhala, R., Vahdat, A.: Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In: NSDI 2007 (2007)
13. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM 21(7), 558–565 (1978)
14. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P.A., Neamtiu, I.: Finding and Reproducing Heisenbugs in Concurrent Programs. In: OSDI 2008, pp. 267–280 (2008)
15. Simsa, J., Bryant, R., Gibson, G., Hickey, J.: Efficient Exploratory Testing of Concurrent Systems. CMU-PDL Technical Report, 113 (November 2011)
16. Simsa, J., Gibson, G., Bryant, R.: dBug: Systematic Evaluation of Distributed Systems. In: SSV 2010 (2010)
17. Stern, U., Dill, D.L.: Parallelizing the MurPhi Verifier. Formal Methods in System Design 18(2), 117–129 (2001)
18. Vakkalanka, S.S., Sharma, S., Gopalakrishnan, G., Kirby, R.M.: ISP: A tool for model checking MPI programs. In: PPoPP 2008, pp. 285–286 (2008)
19. Yang, J., Chen, T., Wu, M., Xu, Z., Liu, X., Lin, H., Yang, M., Long, F., Zhang, L., Zhou, L.: MoDist: Transparent Model Checking of Unmodified Distributed Systems. In: NSDI 2009, pp. 213–228 (April 2009)
20. Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: A Runtime Model Checker for Multithreaded C Programs. University of Utah Tech. Report, UUCS-08-004 (2008)
21. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 58–75. Springer, Heidelberg (2007)
22. Yang, Y., Chen, X., Gopalakrishnan, G., Kirby, R.M.: Efficient Stateful Dynamic Partial Order Reduction. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 288–305. Springer, Heidelberg (2008)

ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level*

Jan Fiedor and Tomáš Vojnar

IT4Innovations Centre of Excellence, FIT, Brno University of Technology, Czech Republic

Abstract. This paper presents the ANaConDA framework that allows one to easily create dynamic analysers for analysing multi-threaded C/C++ programs on the binary level. ANaConDA also supports noise injection techniques to increase chances to find concurrency-related errors in testing runs. ANaConDA is built on top of the Intel's framework PIN for instrumenting binary code. ANaConDA can be instantiated for dealing with programs using various thread models. Currently, it has been instantiated for programs using the pthread library as well as the Win32 API for dealing with threads.

1 Introduction

Due to the arrival of multi-core processors to common computers, multi-threaded programming has become a standard in all widely used programming languages. Such programming, however, is more demanding and brings much more space for errors. Hence, adequate tools for discovering concurrency-related errors are highly needed.

One way to find errors in multi-threaded programs is *dynamic analysis* that monitors the execution of a program and tries to extrapolate the witnessed behaviour and issue warnings about possible errors even when no error is really witnessed in the given execution. However, monitoring the execution of a program can be quite challenging and programmers might spend more time writing the monitoring code than by writing the analysis code itself. In this paper, we present the ANaConDA framework which is a framework for adaptable native-code concurrency-focused dynamic analysis built on top of PIN [7]. The goal of the framework is to simplify the creation of dynamic analysers for analysing multi-threaded C/C++ programs on the binary level. The framework provides a monitoring layer offering notification about important events, such as thread synchronisation or memory accesses, so that developers of dynamic analysers can focus solely on writing the analysis code. In addition, the framework also supports noise injection techniques to increase the number of interleavings witnessed in testing runs and hence to increase chances to find concurrency-related errors.

The general ideas behind the framework and preliminary experiments with it have been presented in [2]. In the present paper, apart from mentioning some recent additions to the framework, we focus more on how to write an analyser using the framework, how

* This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the internal Brno University of Technology projects FIT-S-11-1 and FIT-S-12-1.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 35–41, 2013.
© Springer-Verlag Berlin Heidelberg 2013

to get some useful information which may help the user in locating an error, and how to use the tool. As the framework can be instantiated to support various multithreading libraries, we also describe some concrete instantiations, in particular, the instantiation for `pthreads`, already used for the experiments in [2], and a new instantiation for Win32 API. Finally, we discuss several real-life experiments done with the framework.

As for related tools, there exist many frameworks which may be used to simplify the creation of dynamic analysers for Java programs. The closest to ANaConDA is IBM ConTest [1] which inspired some parts of the design of ANaConDA. RoadRunner [3] is another framework very similar to ANaConDA. Both of these frameworks can monitor the execution of multi-threaded Java programs and provide notification about important events in their execution to dynamic analysers built on top of the frameworks. CalFuzzer [5] is an extensible framework for testing concurrent programs which can also be used to create new static and dynamic analysers and to combine them. Chord [8] is another extensible framework which might be used to productively design and implement a broad variety of static and dynamic analyses for Java programs. When dealing with C/C++ programs, the options are much poorer. One tool somewhat related to ANaConDA is Fjalar [4] which is a framework for creating dynamic analysers for C/C++ programs. However, Fjalar is primarily designed to simplify access to various compile-time and memory information. It does not provide any concurrency-related information. Moreover, it is build on top of Valgrind [9], which brings several disadvantages as discussed in Section 3.

2 Monitoring Multithreaded C/C++ Programs on the Binary Level

As was mentioned in the introduction, monitoring C/C++ programs can be quite difficult, especially when the monitoring is done on the binary level. One of the problems to be dealt with is monitoring of function execution. This is because the monitoring code has to cope with that the control can be passed among several functions by jumps. Hence, the control can return from a different function than the one that was called. Another problem is that the monitoring code must properly trigger notifications for various special types of instructions such as atomic instructions, which access several memory locations at once but in an atomic way, or conditional and repeatable instructions, which might be executed more than once or not at all. Further, some pieces of information about the execution of instructions or fuctions (such as the memory locations accessed by them), which are crucial for various analyses, may be lost once the instruction or function finishes its execution, and it is necessary to explicitly preserve this information for later use. Finally, in order to support various multithreading libraries, the analysers must be abstracted from the concrete library used. Possible solutions to the these problems were discussed in [2].

A problem that has not been considered in [2] is that the information needed for analysis is not the only information useful for the users. When the analyser detects an error, it should provide the users as much information as possible to help them localise the error. Retrieving information about the executed code, such as names of variables or locations in the source code, can give the users some information about the error. However, this information is often not sufficient since it may be difficult to know how

the program got to the variable or location where the error was detected. A much better help to the user is a backtrace to the erroneous part of the program.

ANaConDA currently supports backtraces equivalent to the ones given by the Linux `backtrace()` function, which contain the return addresses of the currently active function calls. The return addresses are stored on the call stack in the corresponding stack frames. The top stack frame's address can be obtained from the base pointer register, and each stack frame also contains the previous value of the base pointer, referring to the previous stack frame. By following the chain of base pointers, we can extract the return addresses and create a backtrace although we have to be careful when processing the stack frames as sometimes (e.g., during the initialisation of the program) the base pointer register may be used for other purposes and might point somewhere else than to a stack frame. The advantage of this approach is that we do not need to monitor every function call in the program and update the backtrace constantly. We are constructing the backtrace on demand, i.e., only when the analyser explicitly requests it, and we only need to know the value of the base pointer register, which can be retrieved with a negligible overhead. The only drawback is that the program must properly create the stack frames, which may sometimes not be true if some optimisations are used.

3 Implementation, Current Instantiations, and Usage

The ANaConDA[1] framework is an open-source framework written in C++ on top of PIN [7]. There are several reasons motivating the use of PIN as a binary instrumentation backend. First, PIN performs dynamic instrumentation, i.e., it instruments a program in the memory before it is executed. This means that the binary files of the program are left untouched. This is especially important when dealing with libraries as it allows one to transparently use an instrumented version of a library and simultaneously use the library as usual in other programs. PIN can also be used on both Linux and Windows, compared to Valgrind which is Linux-only, which allows a much wider range of programs to be analysed. Of course, PIN is primarily developed for use with Intel binaries. However, if the binary code does not contain any special AMD-only instructions, PIN works fine even for AMD binaries. Another advantage of PIN is that it preserves the parallel execution of threads of the analysed multi-threaded program. Valgrind, on the contrary, serialises thread execution [9], which may unnecessarily slow down the program and also the analysis as the analysis code usually runs in these threads too.

Instantiation. The ANaConDA framework abstracts analysers built on top of it from the specific multithreading library used, but it of course cannot do that without any information about the library. As explained in more detail in [2], the user must specify: (1) the names of the functions performing various thread-related operations, (2) the indices of parameters holding the synchronisation primitives the functions operate with, and (3) the `Mapper` objects used to abstract the synchronisation primitives to numbers uniquely identifying them. Abstraction of synchronisation primitives is necessary because their representation varies across various libraries, but analysers need to work with them in a uniform way.

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda`

```
pthread_mutex_lock 1 addr()
pthread_mutex_trylock 1 addr()
```

(a) Lock acquisitions (lock file)

```
__pthread_mutex_unlock_usercnt 1 addr()
```

(b) Lock releases (unlock file)

Fig. 1. An example of the configuration of monitoring lock operations in the pthread library

For example, if we use the pthread library and want to get notifications about lock acquisitions and releases, we have to specify that the pthread_mutex_lock and pthread_mutex_trylock functions are performing the lock acquisitions and the __pthread_mutex_unlock_usercnt function the lock releases. This is done by adding the names of these functions to the lock and unlock configuration files, respectively. All of these functions are taking the lock as the first parameter, and because locks are objects of the pthread_mutex_t structure, we can use the ANaConDA framework's built-in mapper object addr to convert the addresses of these objects into numbers uniquely representing them. To give this information to ANaConDA, we have to specify the index and the name of the Mapper object right after the name of the corresponding monitored function as can be seen in Fig. 1. The instantiation for signaling conditions and waiting on them is similar, we just have to instruct the framework to monitor the pthread_cond_signal, pthread_cond_broadcast, and pthread_cond_wait functions by inserting the appropriate information to the signal and wait configuration files.

As for the Win32 API, there is no function that performs purely lock acquisitions. Instead, the WaitForSingleObject function is used taking a generic HANDLE as the first parameter and performing a lock acquisition only if the HANDLE represents a lock (it may also represent, e.g., a thread or an event). In this case, we have an alternate way to tell ANaConDA when a function performs a lock acquisition. We can specify that the WaitForSingleObject function is a generic wait function whose behaviour depends on the type of the synchronisation primitive passed to it and then name a function which creates or initialises new locks. The framework then remembers which synchronisation primitives are locks because they were created by the user-identified lock creation/initialisation function. Subsequently, when a generic wait function (like WaitForSingleObject) is called, it will first determine what kind of synchronisation primitive its parameter represents. If it is a lock, it will properly trigger the lock acquisition notifications. In particular, in Win32 API, locks are created by the CreateMutex function which returns a HANDLE representing the lock. Configuring lock releases is much simpler as they are performed by a dedicated ReleaseMutex function which takes the lock (HANDLE) as the first parameter. As the HANDLE is in fact a generic pointer, we can also use the addr mapper object here to transform it into a unique number.

The Win32 API has no functions for signaling conditions and waiting on them. If such operations are needed, the users usually implement the operations themselves or use some libraries like pthread-win32 implementing them. However, as ensuring that the functions performing these operations will trigger the corresponding ANaConDA notifications is as easy as adding a few lines to the appropriate configuration files, the framework does not have any problems with the users using their own custom functions for these operations, which illustrates the generality of the framework.

Another problem with the Win32 API is that some of the functions that need to be monitored are jumping at the beginning of other monitored functions. In this case, PIN executes the monitoring code inserted before such functions, and if no special care was taken, the analyser would get a notification about a single event multiple times. The solution could seem to be easy as one could, e.g., think of simply specifying that one of the functions should not be monitored. However, the functions often have exactly the same names, so one cannot so easily differentiate between them. The framework solves this problem by checking if the stack pointer changed when a monitored function is about to be executed, and it does not issue a notification if its value remained the same as that means that nobody called the function, and the control must have jumped to it.

Usage of ANaConDA. To analyse a multi-threaded C/C++ program using ANaConDA, one first has to write (or get) an analyser to be used. The analyser must have the form of a shared object (in Linux) or a dynamic library (in Windows) which contains a set of functions that ANaConDA should call when a specific event, such as a lock acquisition, occurs in the program being analysed. The analyser has to register the callback functions for the events it needs to be notified about. This is done by calling the appropriate registration functions (provided by ANaConDA) in the `init()` function of the analyser, which ANaConDA executes once the analyser is loaded. For example, to be notified about lock acquisitions and releases, the analyser has to register its callback functions using the `SYNC_AfterLockAcquire` and `SYNC_BeforeLockRelease` functions, respectively.

Performing the actual analysis is then quite simple. One just needs to execute the PIN framework with ANaConDA as the *pintool*[2] to be used and specify the analyser which should perform the desired analysis together with the program which should be analysed. Noise injection can be enabled and configured in the `noise` section of the `anaconda.conf` configuration file. Currently, only the *sleep* and *yield* noise is supported, but the user may use different noise injection settings for the read and write accesses and also for each of the monitored functions. The slowdown of the execution of the analysed program is similar to Fjalar, i.e., around 100 times. Note, however, that the slowdown is mainly due to PIN and depends on many factors such as the amount of instrumentation inserted, the amount of information requested by the analyser, the amount of noise injected into the program, etc.

4 Experiments

A set of preliminary experiments with the framework was done in [2] where we analysed more than 100 student projects implementing a simple ticket algorithm (100–500 lines of code) under the `pthread` library. The projects passed all the tests originally used to mark them, but we still found errors in around 20 % of them using a simple data race detector called AtomRace [6], which we use in the tests discussed below too.

To test whether ANaConDA can handle really large and complex programs, we have used it to analyse the `Firefox` browser (more than 3 million lines of code) which uses

[2] A pintool can be thought of as a PIN plugin that can modify the code generation process inside PIN, i.e., it determines which code should be executed and where in the monitored program.

the pthread library. We did not find any severe or unknown errors. We did, however, find several data races which are left in the code since they are considered harmless. Considering the size of the program, the fact that it is thoroughly checked for data races regularly, and also that we used a very simple data race detector and performed only a very limited set of tests since we did not have any automatic test suite to use, we consider these results to still be quite promising.

We further analysed the unicap libraries for video processing, which also use the pthread library and are considerably smaller (about 40k lines of code) which allowed us to perform a larger number of tests. We have found several (previously unknown) data races in the libunicap and libunicapgtk libraries. Two of the data races can be considered severe as they may cause a crash of the program which uses these libraries. In both cases, one thread may reset a pointer to a callback function (i.e., set it to NULL) in between of the times when another thread checks the validity of this pointer and calls the function referenced by it, which can cause an immediate segmentation fault. We are currently preparing to report these errors to the developers using the ANaConDA's recently added backtrace support that can provide a rather detailed information where and why the error occurred.

Finally, we also successfully tested the framework on several Windows toy programs (100–500 lines of code). An application to larger programs is planned for the near future.

5 Conclusion

We have presented ANaConDA—a framework simplifying the creation of dynamic analysers for analysing multi-threaded C/C++ programs on the binary level. We have shown how to instantiate it for several widely used multithreading libraries and demonstrated on several case studies that it can handle even large real-life programs. With the help of the framework, we were able to write a simple analyser in a day and successively find several errors with it, which shows the usefulness of the framework.

References

1. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for Testing Multi-threaded Java Programs. Concurrency and Computation: Practice and Experience 15(3-5), 485–499 (2003)
2. Fiedor, J., Vojnar, T.: Noise-Based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level. In: Proc. of PADTAD 2012. ACM Press (2012)
3. Flanagan, C., Freund, S.N.: The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In: Proc. of PASTE 2010. ACM Press (2010)
4. Guo, P.J.: A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs. Master's thesis, Department of EECS, Cambridge, MA (May 5, 2006)
5. Joshi, P., Naik, M., Park, C.-S., Sen, K.: CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 675–681. Springer, Heidelberg (2009)
6. Letko, Z., Vojnar, T., Křena, B.: AtomRace: Data Race and Atomicity Violation Detector and Healer. In: Proc. of PADTAD 2008. ACM Press (2008)

7. Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In: Proc. of PLDI 2005. ACM Press (2005)
8. Naik, M.: Chord: A Static and Dynamic Program Analysis Platform for Java Bytecode, http://code.google.com/p/jchord
9. Nethercote, N., Seward, J.: Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In: Proc. of PLDI 2007. ACM Press (2007)

PaRV: Parallelizing Runtime Detection and Prevention of Concurrency Errors

Ismail Kuru[1], Hassan Salehe Matar[1], Adrián Cristal[2], Gokcen Kestor[2], and Osman Unsal[2]

[1] Koç University, İstanbul, Turkey
{ikuru,hmatar}@ku.edu.tr
[2] Barcelona Supercomputing Center, Barcelona, Spain
{adrian.cristal,gokcen.kestor,osman.unsal}@bsc.es

Abstract. We present the PaRV tool for runtime detection of and recovery from data races in multi-threaded C and C++ programs. PaRV uses transactional memory technology for parallelizing runtime verification and for buffering write accesses during race checking. Application threads are slowed down only due to instrumentation, but not due to the computation performed by runtime verification algorithms since the latter are run concurrently on different threads. Buffering writes allows us to recover from races and to safeguard against later ones.

1 Introduction

We present PaRV, a tool for runtime detection of and recovery from data races in multi-threaded C and C++ programs. We use components from transactional memory (TM) implementations in order to parallelize runtime verification and to buffer write accesses until they are determined to be free of races.

Concurrently with each application thread, a sibling thread in the style of [5] performs race detection using the Fasttrack algorithm[4]. This approach to parallelized runtime verification minimizes application slowdown. The application thread only experiences slowdown due to instrumentation. Once the sibling thread determines that the accesses within a block are free of races, the accesses in the buffer are committed to memory. If a race is detected, the block is rolled back, and extra synchronization is performed on variables experiencing races, which allows the execution to continue without race conditions. In its current form, our approach allows race-free execution of application binaries at a modest overhead even for legacy applications. With the availability of TM hardware in upcoming microprocessors and with a large number of cores expected to be available on processor chips, we expect our approach to have further reduced performance overhead and wide applicability for legacy applications.

2 Transactional Memory and Runtime Verification

Runtime verification slows down applications. For instance, race detection slows down C/C++ programs by 100 times or more. High overheads make post-deployment use of such runtime monitoring techniques infeasible. Even during

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 42–47, 2013.
© Springer-Verlag Berlin Heidelberg 2013

pre-deployment testing and runtime verification, such high overheads make it unlikely that runtime verification techniques will be used continuously during all runs.

Transactional memory implementations contain highly optimized mechanisms for logging and buffering events, and, in the case of parallelized implementations of transactions, for efficient inter-thread communication between threads working on the same transaction. Hardware vendors have started providing hardware support for transactional memory, which will make approaches using TM more efficient in the near future.

A related approach is that of log-based architectures (e.g., [3], [6]) which provide on-chip hardware resources for reducing the runtime overhead that comes from monitoring executions. Differently from log-based approaches, our tool does not need any additional hardware support but can benefit from it. Differently from how [6] makes use of hardware TM, we do not make use of conflict detection and version number management – parts of a TM implementation that incur significant computational overhead. We use the high-performance FastTrack algorithm instead.

One important way our approach will benefit from hardware support for TM announced or provided by processor vendors is by obviating the need for software-based synchronization to protect race checking metadata, such as vector clocks. With compiler support available for TM hardware, our approach will immediately enjoy the benefit of improved performance due to TM hardware.

STM^2 [5] is a novel, multi-threaded STM design, where each application thread has a dedicated auxiliary ("sibling") thread performing STM operations such as validation of read-sets, bookkeeping and conflict detection. The communication between application and auxiliary thread is provided by a communication channel and atomic status variables. The communication channel is implemented with a single-producer/single-consumer, circular, lock-free queue where the application thread (producer) posts read and write messages that the auxiliary thread (consumer) retrieves and processes them. We use STM^2's queue to communicate read, write and synchronization operations from the application thread to the sibling thread carrying out race detection. We also use STM^2's write buffering and transaction commit mechanisms to delay writing to memory of writes until they are shown to be bug-free. Specifics of these are explained in the next section.

3 Tool Architecture and Implementation

Figure 1 shows position of PaRV relative to DynamoRIO-Dynamic instrumentation tool. The high level organization of the tool is as follows. Instructions performed by each application thread are instrumented using the dynamic DynamoRIO binary instrumentation framework [2]. Between every application thread and its corresponding sibling thread, there is a FIFO queue (figure 2) in the style of the STM^2 circular buffer that the application thread writes to and the sibling thread reads from. On the application thread, read and write accesses and synchronization operations are instrumented such that for each of

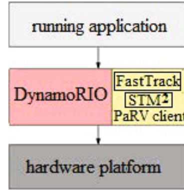


Fig. 1. Architecture of PaRV

these instructions executed, an *event* is placed on the FIFO queue. The sibling thread removes events from the queue and is able to carry out race detection for the sequence of instructions carried out by the application thread in this way.

The sequence of instructions performed by each thread are divided into non-overlapping portions called *consistency block*s using DynamoRIO binary instrumentation. Every synchronization event is in a consistency block by itself. The sequence of instructions performed by an application thread between two synchronization events constitute a block otherwise. The application thread and the sibling thread synchronize at consistency block boundaries. The application thread buffers all write accesses it performs. For consistency blocks that do not contain synchronization operations, when the sibling thread signals to the application thread that the processing of the block is complete, and detects no concurrency errors, the application thread commits the writes in the buffer to memory. For consistency blocks consisting of synchronization operations, the application waits for the sibling thread to complete processing the consistency block before it actually performs the synchronization operation. This is necessary for the runtime verification carried out by the sibling threads to have the same happens before relation as the execution produced by the application threads. Before using DynamoRIO to realize the implementation we tried our approach with PIN. However,with PIN we could not do some of the approaches discussed.

3.1 Runtime Instrumentation with DynamoRIO

Using DynamoRIO, the write and read accesses and synchronization operations performed by application threads are modified.

A write access (store instruction `ins`) is instrumented to implement the following steps. First, the address and the value to be written are extracted from `ins`. Then, an entry is written into the write buffer of the application thread, and an event corresponding to the write access is placed on the FIFO queue. The write instruction is then skipped. This is necessary, since we only want to commit to main memory writes determined to be free of concurrency errors. The write buffer implementation is borrowed from STM^2

```
addr = get_destination ( ins );    val = get_value ( ins );
write_to_buffer( addr, val);   enqueue_write_event( addr);
skip_instruction ( ins );
```

Fig. 2. Application-auxiliary thread interaction

A read access (load instruction `ins`) is instrumented so that a variable that was written to earlier by the current consistency block gets its value from the write buffer. Other reads get their value from the main memory.

```
read_from_local_write_buffer_or_mem ( ins );
addr = get_memory_operand ( ins );
enqueue_read_event ( addr );
```

When a consistency block ends, the application thread waits for the sibling thread to set an atomic signal to indicate completion of runtime verification for the curent consistency block. At that time, the write buffer is committed to main memory. Unlike the commit phase of an STM implementation, we do not need to acquire locks for the variables in the write buffer, since we are not carrying out conflict detection between consistency blocks in the sense of TMs. If no race has been detected by the sibling thread, then write buffers can safely be written to memory, since there are no concurrent racy writes. This is a significant factor in reducing the instrumentation overhead below what an STM would experience.

Before every synchronization operation, we end the ongoing consistency block if there is one, and then start a new one. We put on the FIFO queue an event representing the synchronization operation. When the sibling thread is done processing this event and notifies the application thread by setting an atomic variable, the application thread continues, performs the synchronization operation, and starts the new consistency block.

3.2 Detecting and Recovering from Races

Each sibling thread applies to the stream of events it receives from the event FIFO queue the FastTrack race detection algorithm [4]. FastTrack is an efficient, precise race detection algorithm. The algorithm is described by providing the updates and checks performed by each thread for each memory access or synchronization operation. In our tool, differently from the original FastTrack,

the application thread only records the events in the FIFO queue. The race detection computation is performed on the sibling thread for each event as it is removed from the FIFO queue. We implemented FastTrack in C based on the original implementation. The shared variables (e.g. vector clocks and epochs) used by FastTrack are protected by mutual exclusion locks. The sibling thread notifies the application thread of races or race-free completion of consistency blocks by setting atomic variables.

By buffering write accesses until the end of a consistency block, we are able to prevent racy writes from being written to memory, and racy reads from affecting later code. At the end of a consistency block, if the sibling thread signals a detected race condition, the consistency block is aborted (the write buffer discarded) and retried. The sibling thread notifies the application thread of the set of variables that experienced a race condition during the last execution of the consistency block. The application thread, when retrying the block, wraps each access to a racy variable x by an acquire and release of the lock that protects VC_x, the vector clock of x. Since the last access by another consistency block to x was followed by the sibling thread's access to VC_x, this ensures a happens-before relationship between the accesses and prevents a race condition. After a race is detected on x, all later accesses to x by application threads are protected by VC_x. By doing this for only variables that experience a race, we keep the performance overhead of our approach low.

4 Related Work

PaRV builds on research in the areas of transactional memory and dynamic race detection. It also bears similarities to approaches in the architecture literature for instrumenting and logging program executions, parallelizing dynamic monitoring, containing and recovering from errors encountered. In the following, we contrast PaRV with these approaches.

ParaLog [8] extends work on log-based architectures [3] provide hardware support for instrumenting, logging and monitoring executions of multithreaded programs. Techniques in ParaLog not only reduce the application slowdown due to instrumentation and logging, but also allow, similarly to PaRV, parallelized monitoring algorithms to be run on separate resources from the application, thus further reducing slowdown. ParaLog involves significant changes to processor and memory architecture. It accomplishes efficient tracking of ordering of events from different threads by monitoring cache coherence traffic. PaRV works on currently available, stock microprocessors, but If a platform provides LBA support, PaRV would incur much less slowdown as well.

Race-detection depends critically on, and almost entirely consists of tracking inter-thread dependencies precisely, and the multiple threads in the monitor accessing the per-address and per-thread metadata atomically. The hardware support in ParaLog directly targets efficient implementations of these operations. Taking an alternative approach, PaRV aims to reduce race-detection slowdown as much as possible in the absence of hardware support for monitoring. Differently

from ParaLog, PaRV uses TM technology to prevent races, and explicitly inserts extra synchronization into the program for avoiding later races.

The authors in [1] present the KUDA tool, which, similarly to PaRV, separates race detection from application execution threads using kernel threads in the GPU as helper threads. Differently from KUDA, PaRV synchronizes the application and helper threads so that race detection does not lag behind. This is essential for prevention of and recovery from races, two more features that distinguish PaRV from KUDA. KUDA also parallelizes race detection further than one helper thread per application thread in order to make use of the high degree of parallelism provided by the hundreds of cores on a GPU.

Veeraraghavan, et al in [7] present the Frost tool that addresses detection and prevention of data races by running multiple replicas of an application using complementary schedules. Races are detected by comparing states reached by different replicas, instead of processing event sequences. While providing significant reduction in slowdown, this approach suffers from two key weaknesses. First, for an application with faulty synchronization, it is quite possible that no schedule leads to race free execution. PaRV addresses this problem by adding synchronization to the program as needed. Second, race detection in Frost is imprecise. PaRV uses the FastTrack algorithm for precise detection of races.

References

1. Bekar, U.C., Elmas, T., Okur, S., Tasiran, S.: Kuda: Gpu accelerated split race checker. In: Workshop on Determinism and Correctness in Parallel Programming (WoDet), London, England, UK (March 2012)
2. Bruening, D.L.: Efficient, transparent and comprehensive runtime code manipulation. Technical report (2004)
3. Chen, S., Falsafi, B., Gibbons, P.B., Kozuch, M., Mowry, T.C., Teodorescu, R., Ailamaki, A., Fix, L., Ganger, G.R., Lin, B., Schlosser, S.W.: Log-based architectures for general-purpose monitoring of deployed code. In: Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability, ASID 2006, pp. 63–65. ACM, New York (2006)
4. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. SIGPLAN Not. 44, 121–133 (2009)
5. Kestor, G., Gioiosa, R., Harris, T., Unsal, O.S., Cristal, A., Hur, I., Valero, M.: Stm2: A parallel stm for high performance simultaneous multithreading systems. In: 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 221–231 (October 2011)
6. Sánchez, D., Aragón, J.L., García, J.M.: A log-based redundant architecture for reliable parallel computation. In: HiPC, pp. 1–10. IEEE (2010)
7. Veeraraghavan, K., Chen, P.M., Flinn, J., Narayanasamy, S.: Detecting and surviving data races using complementary schedules. In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP 2011, pp. 369–384. ACM, New York (2011)
8. Vlachos, E., Goodstein, M.L., Kozuch, M.A., Chen, S., Falsafi, B., Gibbons, P.B., Mowry, T.C.: Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In: Proceedings of the Fifteenth Edition of ASPLOS, ASPLOS 2010, pp. 271–284. ACM, New York (2010)

It's the End of the World as We Know It (And I Feel Fine)

James R. Larus

Microsoft Research

Abstract. The end of Dennard scaling and the imminent end of semiconductor feature scaling means that software systems and applications will no longer benefit from 40% per annum performance increases, a continually rising tide that lifted all boats. Future software developers will work harder to find the capability to support productive, high-level programming languages; richer, more natural models of human-computer interactions; and new, compute-intensive applications. This talk focuses on what software can do to find the performance headroom that we need. The solutions to this problem are more diverse and challenging than our previous path, and do not offer 40 years of uninterrupted progress. Some of these improvements are the performance engineering discipline that has only been necessary in cutting-edge systems, while others are opportunities to change the way in which software is developed. The new emphasis on performance, monitoring, adaptation and new ways of developing software should also lead the hardware and architecture communities to revisit the long-standing debate on the hardware-software interface.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, p. 48, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Detecting Unread Memory Using Dynamic Binary Translation

Jon Eyolfson and Patrick Lam

University of Waterloo

Abstract. Reading from uninitialized memory—that is, reading from memory before it has been written to—is a well-known memory usage error, and many static and dynamic tools verify that programs always write to memory before reading it. This work investigates the converse behaviour—writes that never get read, which we call "unread writes". Such writes are redundant—at best, they do not perform any useful work; furthermore, work done to compute the values to be written could corrupt the program state or cause a crash. We present a novel dynamic analysis, implemented on top of the Pin dynamic binary translation framework, which detects instances of unread writes at runtime. We have implemented our analysis and present experimental data about the prevalence of unread writes in a set of benchmark applications.

1 Introduction

Modern languages and compilers detect memory usage errors caused by reads from uninitialized memory: in Java, it is an error to read variable `x` before writing a value to it, and `gcc` warns about uses of uninitialized variables. Programs also contain the converse phenomenon: writes to memory which are never read. Such writes are redundant; at best, they don't perform any useful work. Computations that produce values used only in unread writes do not contribute to the goal of the program, gratuitously consume computational and memory resources, and may, in the worst case, crash the program—for example, the Ariane 5 crash was caused by an exception while computing an unused value[1].

Because compilers detect memory problems ahead of time, most compilers only report errors and warnings at an intraprocedural level, and only for local variables and private fields of classes. (`gcc` 4.6, for instance, reports warnings for unused but set variables.) Static approaches to memory error detection require detailed pointer information to detect memory errors on heap accesses: the compiler needs to know which heap references may and must alias, so that it can determine the access history of individual abstract memory locations. Must-alias analysis is critical for reducing the rate of false positives. However, implementations of whole-program must-alias analyses are rare.

Recently, Valgrind's Memcheck tool [1] has used dynamic binary translation to detect memory errors, including reads from uninitialized memory, at runtime.

[1] Section 2.1, `http://www.di.unito.it/~damiani/ariane5rep.html`

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 49–63, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Purify [2] detects a similar class of errors by inserting instrumentation code at compile time. In either case, runtime verification can ensure the absence of memory errors on an observed execution. Dynamic analyses need not reason about the heap, as a pointer comparison suffices to disambiguate heap addresses.

Our Tracerory tool implements a dynamic analysis to detect unread memory in realistic C and C++ applications. It supports multithreaded programs. We detect 1) *unread memory allocations* and 2) *unread writes* to the heap—writes with no corresponding read. When a developer runs their code under Tracerory, it reports instances of unread memory. Developers can use the report to manually inspect flagged program points and fix their code.

Figure 1 shows a high-level overview of our tool's operation. Tracerory takes two inputs: an executable to be monitored, and specifications about which parts of the program to monitor. While Tracerory executes the program, its runtime monitor processes the stream of memory allocations, reads, and writes, reporting unread writes and memory allocations.

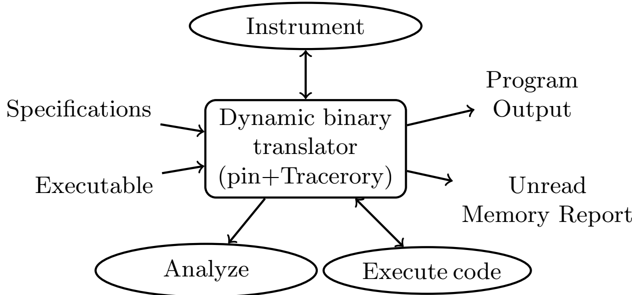


Fig. 1. Tracerory operation

The contributions of this paper are:

- the identification of unread memory as a source-level phenomenon of interest;
- a novel dynamic analysis to detect unread memory in programs at runtime;
- an implementation of our dynamic analysis in the Pin dynamic binary translation framework; and
- qualitative and quantitative results outlining the prevalence of unread memory in a collection of open-source benchmarks.

2 Overview

This section presents, using an example, the two suspicious memory usage patterns that our Tracerory dynamic monitoring tool detects. Section 5 presents additional instances of unread writes drawn from real-world programs.

```
1  int main(int argc, char *argv[])
2  {
3      X* x = new X();
4      Y* y = new Y();
5      for (int i = 0; i < 4; ++i) {
6          x->data = i;
7          y->data = i;
8      }
9      cout << x->data << endl;
10     delete x;
11     return 0;
12 }
```

Fig. 2. Example program with unread allocations and writes

```
Allocations unread and/or with unread writes: 2
Unique allocation sites unread and/or with unread writes: 2

Created: example.cpp:3 (1)
Destroyed: example.cpp:10 (1)
Unread: false (1)
Unread Writes (3):
    example.cpp:6 (3)

Created: example.cpp:4 (1)
Destroyed: [not destroyed] (1)
Unread: true (1)
Unread Writes (4):
    example.cpp:7 (4)
```

Fig. 3. Tracerory output for motivating example

The example program in Figure 2 allocates two objects, `x` and `y`. It then performs 4 writes to each object, reads from `x`, and finally deletes `x`.

Lines 3 and 4 allocate the memory objects, which we initially mark unread. Next, line 6 writes to `x` and line 7 writes to `y`. Our tool records the first two writes to `x` and `y` in the first iteration of the loop and marks the memory locations as having an active unread write. In the second iteration, the new writes overwrite the previously active unread writes. The tool marks the writes from the previous iteration as unread. At the end of the loop, there are 3 unread writes and 1 active unread write to each object.

Finally, the program reads from `x` on line 9 and deletes it on line 10. If an object is not deleted, we implicitly delete it when the program terminates (e.g. `y`). After an object is deleted, no further reads can be made to it. Therefore, we report the active unread write to `y` (in the last iteration of the loop). We also report the object itself as completely unread. Upon exit, our tool reports the unread object `y`, plus all unread writes to heap objects.

Tracerory outputs, for each "bad" object (with unread writes or itself unread): the location that created and destroyed the object; whether or not it is an unread object; and the number of unread writes to the object, along with the locations which performed the writes. If the tool observed a constructor call for the object,

it outputs the object's type. To minimize false positives, the tool only reports statement s as an unread write if all previous dynamic writes at s are unread. For instance, if s occurs in a loop, then our tool only reports s if all of its executions perform unread writes.

Figure 3 shows Tracerory's output for our example program. First, Tracerory reports `x` as an object of type `X` with 3 unread writes. Next, it reports `y` as an object of type `Y`, with 4 unread writes, which is completely unread. The unread writes on `x` indicate potentially-important information being ignored; writes to `y` may correspond to wasted memory and redundant, potentially harmful, work.

As is standard for dynamic analyses, we only report unread writes from a single program execution at a time. It is the responsibility of the developer to execute the program with enough test coverage to adequately explore its behaviour. A particular write may be unread for some, but not all, inputs. While such a write is most likely not problematic, we believe that the developer is best-placed to decide whether code changes are appropriate in such cases; perhaps the write could have been avoided on that input.

To help developers prioritize the generated reports, our tool coalesces and sorts its output. That is, it combines all objects allocated at the same static site, and displays a count of "bad" objects, unread objects, and unread writes for all objects allocated at that site. It lists the allocation sites which account for the most unread objects first. In the future, we hope to combine unread write reports from multiple executions, thus increasing the relevance of the reports.

3 Dynamic Analysis

To validate our design, we implemented the Tracerory tool atop the Pin dynamic recompilation toolkit [3]. Our tool works on x86-64 Linux binaries. Pin supports multithreaded programs and we have used appropriate data structures to ensure that Tracerory also supports multithreading. Because Pin's API provides an abstraction layer, Tracerory should also work on x86 binaries.

Generally, Pin tools run in two phases: a (slightly) ahead-of-time instrumentation phase, and an monitoring phase. Section 3.1 describes the instrumentation phase while Section 3.2 describes the analysis phase, which implements a runtime monitor to detect unread memory.

Our unread memory detection only monitors images (binaries and libraries) explicitly specified by the user. We call such images "watched images;" watching only specific images enables developers to focus their attention on memory usage which they are responsible for and can fix.

3.1 Instrumentation Phase

The instrumentation phase transforms the input executable to invoke our runtime monitor, which will be described in Section 3.2. Here, we describe our instrumentation points and the information they pass to the monitor.

Allocations and Deallocations. Our tool records all memory management calls by instrumenting standard C/C++ allocation and deallocation functions. For allocation functions (`malloc`, `calloc`, `realloc`), we insert a call to our monitor at the function entry and exit points. At the entry point, we pass the `size` argument to the monitor. At the exit point, we pass the returned pointer to the monitor. For deallocation functions (`free`), we instrument the function entry point and pass the pointer argument to the monitor.

Memory Accesses. Our tool instruments every memory read, plus memory writes from watched images. For both reads and writes, we pass all accessed memory addresses to the monitor. For writes, we also pass the instruction pointer.

Debugging Information. To help developers localize memory problems, our tool uses debug information to identify all program events (allocations, deallocations, reads, writes) by source code line number.

At each `call` instruction, our tool records, in thread-local storage, the debugging location and image name immediately before the call; this information is then available upon entry to the callee. A debugging location consists of a source line number, when available, or the procedure name and memory offset otherwise. It enables the tool to attribute memory allocations and deallocations to the code that requested or released the memory. Our tool uses the image name to ignore memory allocations from unwatched images.

Dynamic loaders introduce indirect calls, which may overwrite the debugging information as they call helper functions to load the function and resolve the function address. To prevent this, we ignore calls to the dynamic loader library. For 64-bit Linux, this library is `/lib64/ld-linux-x86-64.so.2`. We found that ignoring the dynamic loader does not negatively affect our tool, as most applications do not interact directly with it.

Virtual Functions. Our initial results included many unread writes to objects with virtual functions. These were writes to the virtual table, which developers do not control. Such writes should be ignored. We found that source locations for virtual table writes corresponded to definition points for member functions. We collect these source locations by inspecting every member function and recording its definition point; the monitor then ignores these locations.

3.2 Monitoring Phase

To enable the classification of memory accesses, our tool records immutable facts about each block of allocated memory (memory block), along with facts which change during the execution. Figure 4 illustrates the structure of a memory block. Our monitor stores memory blocks in an append-only list of unread memory blocks and a map of currently-allocated blocks keyed by base address.

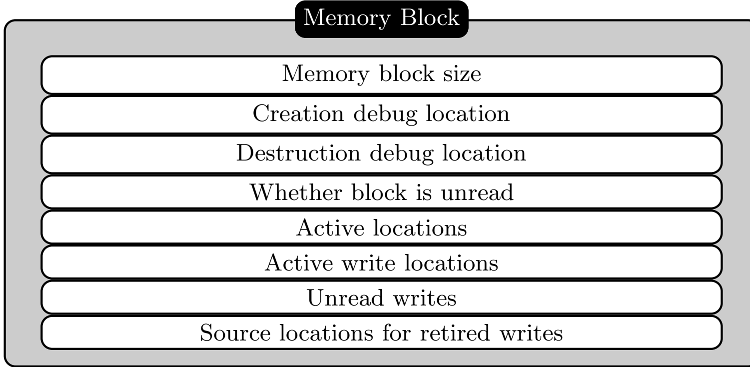


Fig. 4. Structure of a Memory Block

Allocations. At a call to an allocation function (e.g. `malloc`) originating from a watched image, the monitor records the requested memory size and calling source location in thread-local storage. Upon exit from the allocation, the monitor receives the returned pointer and creates a new currently-allocated memory block using that pointer as the base address. It also initializes the block's size and created location with the values recorded on entry and marks the block unread.

We take care to collect reliable source locations for custom allocator wrappers. Consider C++ object allocation; the `new` call is essentially a wrapper for `malloc`. Ordinarily, our tool would report a debugging location for `malloc` from within the `new` implementation. However, we would prefer to know `new`'s caller rather than `malloc`'s. To do this, we instrument the entry point to `new`, ensuring that the caller belongs to a watched image. If it does, we record the calling source location for the `new` and ignore the watched image check for the `malloc`. We support arbitrary user-specified custom wrappers in the same style.

Reads and Writes. To analyze a memory access, the tool looks up the requested memory address in the allocated-memory structure. It uses the `lower_bound` operation of the C++ STL `map`. If the address falls in the range `[address, address + size)` for any blocks, the tool then carries out the appropriate update on those blocks. The tool assumes that memory accesses occur at machine word granularity. The mutable part of the per-block structure includes a flag indicating whether the block has ever been read, as well as the following sets: active writes to the block; active locations (which have been accessed at least once after initialization); unread writes (a list); and source locations for retired writes (those which have been read). We ensure that concurrent threads do not simultaneously update the block structures.

The per-instruction updates are as follows:

– At a *read*: the monitor marks the containing memory block as read and removes any writes to the requested location from the active writes. It adds any removed writes to the set of retired writes.

To reduce the false positive rate, we only report unread writes from static program points from which no writes are ever read—equivalent to applying intersection to unread writes. Hence, if we ever observe a retired write from a given static program point, we filter out writes from that program point.
- At a *write*: if the write's destination already has an active write, then that value will be overwritten, hence unread. We thus add the previous write to the list of unread writes, if its source location belongs to a watched image. The tool also adds the current write to the list of active writes, if the source location is not retired.

At every memory access beyond the initial write, we mark the destination location as active. Our monitor uses this to omit initial values which are never subsequently accessed, when processing the block's deallocation.

Deallocations. Deallocations remove memory blocks from the set of currently-allocated blocks, moving them to the unread writes structure if appropriate. We only add active writes to unread writes if the memory location was active, i.e. accessed at least once after initialization. We move a block to the unread writes structure if the block has at least one unread write or is itself unread. We also set the block's deallocation location.

We do not need to instrument the `delete` function: an instrumented destructor call will always happen first. At a destructor, if `this` is an allocated memory block, our monitor follows the same process as for `free`.

Program Termination. Upon program exit, the tool simulates `free`s for all currently-allocated blocks, with a deallocation location of "[not destroyed]". It then traverses the heap structure abstraction and outputs summaries for all of the unread memory blocks and unread writes. To help the developer prioritize the most important program points, the tool sorts its output, putting blocks which account for more writes first. Within each block, the tool also sorts unread writes by descending order of unread write count. For each entry, the tool outputs the location where the memory block was created, the location where it was destroyed, whether or not it was unread, and its unread write locations.

False Positives. We summarize some false positives that Tracerory will report. Some false positives are due to analysis imprecision, and could be filtered out.

- Our analysis identifies many field initializers in C++ as unread writes; classes will often explicitly initialize all of their fields in the constructor and then re-initialize the fields later. The first initialization is unread.
- Data structure implementations in watched images often lead to unread writes. For instance, we found a implementation of linked list insertion:

```
*new_edge = edge; new_edge->next = stl->tail;
```

`next` is copied from an input and immediately overwritten, hence unread.

- Other false positives include idioms like resetting pointers to NULL after freeing them. Although this is good programming practice, it causes unread writes—freed pointers are never read.
- Our tool does not capture block memory accesses. Although we did not observe any instances of spurious unread writes caused by block accesses in our benchmarks, system calls like `read()` may potentially use DMA, which would cause our analysis to miss some reads.

Remark on Concurrency. In multithreaded programs, the execution depends on both the input and the scheduler. Our monitor only reports what happens on one execution; a write may be unread on an execution and read on another execution, even if the accesses are properly protected by locks. This is because mutual exclusion locks (without condition variables) do not impose an ordering. We believe that a write that is unread on any execution ought to be investigated as suspicious code; it is even more suspicious if it is only unread on some (but not all) executions. Such a program's results depend on the scheduler.

4 Formal Definitions

We continue by giving a precise definition of an unread write and formally stating the property that our runtime monitor enforces. The runtime monitor watches an execution trace on-line.

Definition 1. *An* execution trace *t is a sequence $t = t_1, \ldots, t_n$ of executed instructions $t_i = \langle pc_i, op_i \rangle$, where pc_i is a program counter value and op_i is an operation. Operations include allocations, reads (*$\mathsf{READ}\ addr_i$*) and writes (*$\mathsf{WRITE}\ addr_i$*), where $addr_i$ is of the form $base_i + off_i$. $base_i$ is a block's base address, returned from a previous allocation call. off_i is an offset into a block.*

Our definition of execution traces uses the scheduler's interleaving of instructions from different threads. A single input may give rise to multiple execution traces.

We can now define the notion of an unread write.

Definition 2. *An* unread write *is an executed instruction $t_u = \langle pc, \mathsf{WRITE}\ b + o \rangle$ with no subsequent* READ *from $b+o$ in that execution trace, such that 1) there is no preceding pair in the trace ($t_p = \langle pc, \mathsf{WRITE}\ b{+}o' \rangle$, $t_{p'} = \langle pc', \mathsf{READ}\ b{+}o' \rangle$), where $p < p' < u$, and 2) there exists some other access to the same location, $t_i = \langle pc'', op\ \ b + o \rangle$, where $i \neq u$.*

The definition primarily states that the memory location of the write must not subsequently be read. However, an otherwise-unread write should not be considered unread if any previous instruction at the same program counter value wrote to the same block and that value subsequently got read. Also, we do not report an unread write if it is the only access to a memory location; such writes are often one-time memory initializations.

Using Definition 2, we can state what our runtime monitor is looking for.

Proposition 1. *The runtime monitor described in Section 3 detects all unread writes in an execution trace.*

The proposition follows immediately from the design of our runtime monitor.

5 Experimental Results

In this section, we present the results of our unread memory analysis on a series of benchmark programs. We found that our tool successfully identified a number of instances of suspicious code as well as writes that were useless for a given execution. On our benchmarks, Tracerory caused a slowdown of 87× (geometric mean) over the original execution time, demonstrating its feasibility for occasional use on real codebases.

5.1 Qualitative Results

The main experimental results in this paper demonstrate the efficacy of our tool on five benchmarks: abiword, sqlite, crafty, ImageMagick, and Python. In all cases, our unread memory tool identified interesting code within the benchmarks; two of the benchmarks could be improved using the tool results, while the results illustrate some perplexing behaviour by ImageMagick.

abiword. AbiWord is a word processor written in C++. We used version 2.6.8 of AbiWord, which contains over 559,000 lines of code. Although we explored a number of workloads, we will present results from a run of AbiWord's command-line file-conversion mode which converts a 1.28M AbiWord file into plain text. Since AbiWord has not been tuned for performance, we expected to find a number of unread writes in its codebase. In addition to the base executable, we added `libabiword-2.8.so` and plugin libraries to our watched images.

The top sources of unread memory were utility routines, particularly string and vector implementations. For instance, AbiWord allocates 116,436 strings which it never reads. AbiWord also allocates 6,081 completely unread vectors. Note the role of watched images here: had AbiWord used the standard STL implementation, our tool would assume that the developers weren't interested in modifying the STL, and would therefore not report these writes. On the other hand, because the offending allocations and writes lie in AbiWord code, Tracerory reports these routines.

The remainder of the discussion presents domain-specific unread writes. We will ignore library-like unread writes.

- We found 11,336 unread writes to the private `m_leader` field in the `fp_TabRun` class. This field is never read on the file-conversion executions; it is only accessed by the `_draw()` method of `fp_TabRun`, which is never called on a file-conversion workload. There are no calls to the `getLeader()` method anywhere in the code.

  There are also 11,336 writes to the private `m_tabType` field, which is never read on this workload, or outside its defining class on any workload.

- We also found about 28,000 unread writes at each of the `fp_Run::setTmpLine`, `::setTmpX`, `::setTmpY`, and `::setTmpWidth` methods. These methods are only called by the `format()` method, and there are no other writes of the fields. The only reads of these fields are in the `clearIfNeeded()` method, which is only called by `format()`. It appears that `clearIfNeeded()` only executes when the document is reflowed, which never occurs on the file-conversion workload. We investigated the underlying fields and found that they were used to store the previous metrics of the run, allowing AbiWord to decide whether it actually needs to reflow the text. The text-conversion workflows only reflow the text once.
- Finally, we found 7,085 unread writes of a private field `m_iDrawWidth`. The write follows a discussion, in the comments, about the proper value for this field. It appears that the value does not matter on this workload, at least. It is, however, read in the `_clearScreen()` and `_draw()` methods, which are invoked in other workloads.

These examples illustrate how our tool correctly identifies writes which are redundant on a given execution.

sqlite. SQLite is a ubiquitous SQL database engine. We examined version 3.7.13 of SQLite (138,797 lines of C code) under its provided "zerodamage" workload and found a number of unread writes. We will describe the first three unread writes that our tool found.

- The first two unread writes are both to the `CellInfo` structure and enable SQLite to handle cases where an SQLite cell overflows a page. Tracerory reported 1,999 unread writes to the `nPayload` and `iOverflow` fields of `CellInfo`. The `nPayload` field is seldom read; two of the reads occur in assertions and a third is compiled in conditionally. The only regular read of this field is in the `clearCell()` function, which must not have been called on this execution. Our tool could help in ensuring good test coverage—it seems that it would be worthwhile to specifically craft a test to verify that the field value is correct. The `iOverflow` field is read more often than `nPayload`, with 6 static instances of reads in the code. (One of these reads is never compiled and belongs to a function annotated with the comment "This function does not contribute anything to the operation of SQLite.") The other reads occur in functions like `clearCell()` and `fillInCell()`.
- The third unread write is to the `validNKey` field of the `BtCursor` structure, a cursor over sqlite's central b-tree data structure. This field only has one read, which occurs in the static function `sqlite3BtreeMovetoUnpacked()`. It is written to 11 times across different parts of the SQLite code. Our tool suggests that it might be worthwhile to closely inspect these writes to ensure that they are correct, as they are not often used on this workload.

crafty. Crafty is a chess program and one of the SPECCPU benchmarks; version 23.4, which we examined, contains 34,792 lines of C code. We ran crafty's

included "bench" command after editing the code to evaluate only the first position (for performance reasons). Because Crafty is tuned for chess competitions, we did not expect to find inefficiencies in its main loop; however, all three unread memory reports from Tracerory were instances of suspicious or buggy code.

We manually investigated each of the unread memory blocks and found a number of code idioms which could be improved:

- We learned that crafty contains code to parse its command-line options—it does not use a library. The `main()` function allocates space for 512 potential arguments, each of maximum length 128, and calls `ReadParse()` to copy the arguments into its buffer. Since our test run does not use any command-line arguments, the allocation for the arguments is unread memory, and Tracerory lists it in its output, as we would expect.

  Furthermore, inspecting the code, we found a buffer overflow: it does not check that the command-line arguments are shorter than the buffer.
- We found 973,169 unread writes in one of the memory blocks allocated in the `InitializeHashTables()` function. The accompanying comment indicates that this function is supposed to completely clear the `pawn_hash_table` between test positions. The code itself iterates through an array and sets all but one field to 0; the remaining field gets -1.

  Calling `memset()` would be a more efficient way to clear the memory, and would be less likely to leave forgotten state around (especially in the context of program maintenance, where a developer might add a new field to the struct stored in the hash table.)
  We were surprised that our tool reported this code, since it appears to be initialization code. However, on our test run, `InitializeHashTables()` executes twice; our tool reports the second set of writes as unread writes.
- The final unread memory block points out code marked as a kludge in the comments. When crafty is asked to log its commands, it searches for the first nonexistent or small file named `log.NNN`. It uses `fstat` to identify small files if they already exist, but unconditionally allocates (and does not deallocate) the memory block for the `stat *` return information from `fstat`.

Although the problems in this benchmark were not directly caused by unread writes, we believe that it was useful to run Tracerory on crafty—inspecting unread memory in crafty pointed us to bugs and inefficiencies in the code.

ImageMagick. ImageMagick is a collection of tools for manipulating images, which consists of over 400,000 lines of code as of version 6.7.4-9. This benchmark uses many external libraries to open and process images, such as `libjpeg`; in this section, we report only the behaviour of the `convert` binary while watching ImageMagick-6.7.4-9 with its libraries `libMagickCore` and `libMagickWand`. ImageMagick is highly tuned for performance, and we did not expect to find many inefficiencies in its code. In addition, the README reports that the maintainers

perform a "comprehensive security assessment that includes memory and thread error detection to prevent security vulnerabilities" before each release.

We watched ImageMagick resize a picture from its original size of 1404×625 to a new size of 1280×720. We manually investigated some of the reported results from the unread memory tool, and present our findings below.

- Our tool reported 729,600 calls (somewhat, but not exactly, related to the number of pixels in the output) to `SetPixelOpacity` originating from the source file `resize.c`. The offending line is

  ```
  SetPixelOpacity(q, ClampToQuantum(pixel.opacity));
  ```

  We found that JPEG does not encode opacity—ImageMagick manufactures `OpaqueOpacity` for each pixel from JPEG input and discards it upon write. The writes to opacity are therefore redundant work on this workload.
- The largest offender, accounting for 2,632,500 writes, was `jpeg.c`. Unfortunately, this appears to be a false positive; ImageMagick converters write data to a temporary buffer, `QueueAuthenticPixels`, with the values of the pixels' red, blue, green and opacity channels. `ImageMagick` seems to write the data 1 byte at a time, and read 4 bytes at a time (which accounts for the number of unread writes $1404 \times 625 \times 3$). There are no reported unread writes for the blue channel—the blue channel is the offset that gets read.
- The third-largest source of unread writes, accounting for 2,944 writes, is apparently also a spurious report. These writes copy the ICC colour profile. The code is rather opaque and worth examining in detail for possible bugs, since the effect of the code is not obvious at all (Figure 5).

  ```
  p=GetStringInfoDatum(profile);
  for (i=(ssize_t)GetStringInfoLength(profile)-1;
       i >= 0; i--)
    *p++=(unsigned char) GetCharacter(jpeg_info);
  ```

  Fig. 5. ImageMagick code showing an unread write

  We verifi0ed that commenting out the writes does change the program output (although not visibly, as colour profiles are only used in internal calculations).

Tracerory pointed out a number of interesting idioms in the ImageMagick code. Our false positives tell us that ImageMagick produces output by batching up reads to its buffer, inconsistent with the original per-byte buffer write mode.

CPython. CPython is the default bytecode interpreter for the Python programming language. This application is multithreaded. It consists of over 350,000 lines of C code as of version 3.3.0a3. We ran the 366 individual tests shipped with CPython and watched the CPython executable, the `libpython` library, and all other Python libraries built in the standard configuration.

- Our tool reported 55,343 unread writes to the `overflowed` field of the `PyThreadState` object. Deeper inspection revealed that the implementation for protecting the stack from overflowing is in a haphazard state. The code also referenced a mailing list discussion which pointed out potential problems with the implementation. Our tool adds to the discussion by pointing out that the write which clears the overflowed flag is often unread.
- Our tool also reported unread writes to other parts of the interpreter state, including the line number `f_lineno`; and the previous instruction `f_lasti`, on `LOAD_FAST` and `STORE_FAST` instructions. The code revealed that the interpreter reads `f_lineno` only in tracing mode, which we were not using; those writes are therefore unnecessary in the interpreter's normal operation. The writes to `f_lasti` are generated by a macro. That field is used to report the current line number (e.g. when generating a stack trace) and during tracing.

Summary. Our tool illustrates the additional complexity added by unused modes of operation, as with Python and its tracing function. The extra state for unused modes are unused in normal operation. Such rarely-accessed state is likely to be less reliable than state which is regularly used. Programs with fewer modes will certainly be simpler than programs with more modes.

Internal library (vectors and strings) usage accounted for many of the unread memory reports. Some string implementations, e.g. AbiWord and python, track the string length and zero-terminate the string; we observed 152952 unread writes of the final 0 on one of our test cases. AbiWord's string implementation also includes the buffer length, which is unread for any string which is never grown. We observed thousands of unread writes of AbiWord vector elements.

5.2 Performance

To establish that our tool's performance is adequate, we timed it on a number of benchmarks, including the qualitative benchmarks above. Our test system is an Intel Core i7-3930K at 3.20GHz running ArchLinux GNU/Linux, and our tool runs on top of Pin 2.11 (release 49306). We compared the base runtime (without Pin) to the runtime with Pin alone and with our tool. Reported times are an average over three runs. Figure 6 presents our analysis times. The geometric mean of our slowdown compared to the raw execution time is 87×, while we add a geometric mean of 14× slowdown over pin with no instrumentation.

6 Related Work

We discuss two areas of related work. First, we summarize past work on investigating writes to memory; the related work in that area attempts to reduce memory bandwidth, while we are advocating the use of unread writes to improve code quality. Next, we discuss alternatives in the dynamic binary translation space, including other memory checkers which also use dynamic translation, and other applications of dynamic binary translation tools.

	raw (s)	pin (s)	pin slowdown	tracerory (s)	tracerory slowdown/raw	tracerory slowdown/pin
imagemagick	0.22	2.78	12.6	37.76	172	13.6
python	170.63	208.24	1.22	224.63	1.32	1.08
abiword	28.92	63.45	2.19	7116.43	239	112
ffmpeg	2.31	6.76	2.93	446.46	193	66.0
crafty	15.57	22.36	1.44	906.49	58.2	40.5
sqlite	0.01	3.47	347	7.21	721	2.08

Fig. 6. Unread Memory analysis times

6.1 Optimizing Memory Writes

The program transformation most closely related to the present research is the store elimination transformation proposed by Ding and Kennedy [4]. Their work generally attempts to reduce applications' memory bandwidth usage. They propose a loop-based transformation, store elimination, which eliminates redundant writes inside loop bodies. In store elimination, some loops write values back to an array while performing a computation of some summary information (e.g. a sum) over the array. If the code never reads the final values written to the array, then store elimination will eliminate the unread writes. Because our goal is to examine all of the program code for potential bugs, our dynamic analysis does not focus on loops and arrays, but rather considers all writes to the heap.

Trace optimizers like Dynamo [5] and rePLay [6] eliminate unread writes at runtime. A pass through a trace that is about to execute suffices for removing some unread writes, and such an optimization is therefore standard in the trace optimizer context. A more-powerful dynamic analysis could eliminate most unread writes. Our work, however, aims to help developers improve program quality by enabling them to remove unread writes, rather than to improve performance.

Arnold et al [7] have implemented Virtual-Machine level runtime monitors which detect a subset of our memory properties—their QVM can detect *idle objects*, i.e. objects on which only the constructor is called. We would report such objects as unread as long as the constructor only initializes the object.

6.2 Dynamic Binary Translators

We chose to build our monitor on top of the Pin engine [3]. Other dynamic binary translation engines would have been as effective for monitoring. DynamoRIO [8] and Valgrind [1] would also support unread memory detection.

Valgrind's Memcheck tool performs runtime verification for the following memory errors: accesses to unallocated memory, uninitialized memory, memory leaks, double frees and overlapping memory. Our tool analyzes unread writes, which are not detected by Valgrind. While not as serious as memory errors (they don't cause crashes), unread writes may lead to bugs or at least wasted resources—they should qualify as a novel "code smell" [9].

Another approach is to statically rewrite the program source by inserting monitoring calls. Purify [2] follows this approach; it transforms the program at

link time and inserts instrumentation code to detect memory errors. A program rewriting approach could potentially be equally effective for detecting unread memory; however, this approach requires recompilation of the program and libraries, which our current scheme does not need.

7 Conclusion

We have presented a novel dynamic analysis, unread memory, that investigates the converse of the standard memory safety property "all reads to memory must have previously been written". Our analysis instead identifies writes to memory that never get read. We explained the design and implementation of our analysis, using dynamic binary translation, and presented experimental results from a set of benchmarks. We found that unread writes often indicate something interesting in the code; a number of the writes that we found could be eliminated or improved without affecting the program semantics.

Acknowledgments. This research was supported in part by Canada's Natural Science and Engineering Research Council and an Ontario Graduate Scholarship. We'd like to thank Emina Torlak for helpful comments on a draft of this paper.

References

1. Nethercote, N., Seward, J.: Valgrind: a framework for heavyweight dynamic binary instrumentation. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007), pp. 89–100. ACM Press, San Diego (2007)
2. Hastings, R., Joyce, B.: Purify: Fast detection of memory leaks and access errors. In: Proc. of the Winter 1992 USENIX Conference, pp. 125–138 (1991)
3. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: Building customized program analysis tools with dynamic instrumentation. In: PLDI 2005, Chicago, IL, USA, pp. 190–200 (June 2005)
4. Ding, C., Kennedy, K.: The memory bandwidth bottleneck and its amelioration by a compiler. In: IPDPS, pp. 181–190. IEEE Computer Society (2000)
5. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: a transparent dynamic optimization system. In: PLDI 2000, pp. 1–12. ACM, New York (2000)
6. Fahs, B., Bose, S., Crum, M., Slechta, B., Spadini, F., Tung, T., Patel, S.J., Lumetta, S.S.: Performance characterization of a hardware mechanism for dynamic optimization. In: MICRO 34, pp. 16–27. IEEE Computer Society, Washington, DC (2001)
7. Arnold, M., Vechev, M.T., Yahav, E.: QVM: An efficient runtime for detecting defects in deployed systems. ACM Trans. Softw. Eng. Methodol. 21(1), 2 (2011)
8. Bruening, D., Garnett, T., Amarasinghe, S.P.: An infrastructure for adaptive dynamic optimization. In: CGO 2003, San Francisco, CA, pp. 265–275 (March 2003)
9. Fowler, M., Beck, K.: Refactoring: improving the design of existing code. Addison-Wesley Professional (1999)

Sparse Coding for Specification Mining and Error Localization

Wenchao Li and Sanjit A. Seshia

University of California at Berkeley

Abstract. Formal specifications play a central role in the design, verification, and debugging of systems. This paper presents a new viewpoint to the problem of mining specifications from simulation or execution traces of reactive systems. The main application of interest is to localize faults to sections of an error trace we term *subtraces*, with a particular focus on digital circuits. We propose a novel sparse coding method that extracts specifications in the form of *basis subtraces*. For a set of finite subtraces each of length p, each subtrace is decomposed into a *sparse* Boolean combination of only a small number of basis subtraces of the same dimension. We formally define this decomposition as the *sparse Boolean matrix factorization problem* and give a graph-theoretic algorithm to solve it. We formalize a sufficient condition under which our approach is sound for error localization. Additionally, we give experimental results demonstrating that (1) we can mine useful specifications using our sparse coding method, and (2) the computed bases can be used to do simultaneous error localization and error explanation.

1 Introduction

Formal specifications play a central role in system design. They can serve as high-level requirements from which a system is to be synthesized. They can encode key properties that the system must exhibit, finding use in formal verification, testing and simulation. Formal specifications are also valuable as contracts for use in code maintenance and compositional design. Finally, they are also useful in debugging and error localization, in the following way: if several local properties are written for a system, covering each of its components, then a failing property can provide information about the location of the bug. It is this last application of formal specifications — for error localization and debugging in *reactive systems* — that is the main focus of this paper.

Unfortunately, in practice, comprehensive formal specifications are rarely written by human designers. It is more common to have instead a comprehensive test suite used during simulation or testing. There has therefore been much interest in automatically deriving specifications from simulation or execution traces (e.g. [8,2]). It is important to note that, until they are formally verified, the properties generated from traces are only *likely* specifications or *behavioral signatures* of a design.

Different kinds of formal specifications provide different tradeoffs in terms of ease of generation from traces, generality, and usefulness for error localization. Büchi automata [4] provide a very general formalism, and are typically inferred by learning a finite automaton from finite-length traces and interpreting it over infinite traces. However, such automata tend to overfit the traces they are mined from, and do not generalize well to unseen traces — i.e., they are very sensitive to the choice of traces $\mathcal{T}$ they are

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 64–81, 2013.
© Springer-Verlag Berlin Heidelberg 2013

mined from and can easily exclude valid executions outside of the set $\mathcal{T}$. Linear temporal logic (LTL) formulas [18] are an alternative. One typically starts with templates for common temporal logic formulas and learns LTL formulas that are consistent with a set of traces. If the templates are chosen carefully, such formulas can generalize well to unseen traces. However, the biggest challenge is in coming up with a suitable set of templates that capture all relevant behaviors.

In this paper, we introduce a third kind of formal specification, which we term as *basis subtraces*. To understand the idea of a subtrace, consider the view of a trace as a two-dimensional table, where one dimension is the space of system variables and the other dimension is time. A *subtrace* is a finite window, or a snapshot, of a trace. Thus, just as a movie is a sequence of overlapping images, a trace is a sequence of overlapping subtraces. Restricting ourselves to Boolean variables, each subtrace can be viewed as a binary matrix. Given a set of finite-length traces, and an integer p, the traces can be divided into subtraces of time-length p. The set of all such subtraces constitutes a set of binary matrices. The basis subtraces are simply a set of subtraces that form a basis of the set of subtraces, in that every subtrace can be expressed as a superposition of the basis subtraces.

The form of superposition depends on the type of system being analyzed. In this paper, we focus on digital systems, and more concretely on digital circuits. In this context, one can define superposition as a "linear" combination over the semi-ring with Boolean OR as the additive operator and Boolean AND as the multiplicative operator. The coefficients in the resulting linear combination are either 0 or 1. The problem of computing a basis of a set of subtraces is equivalent to a Boolean matrix factorization problem, in which a Boolean matrix must be decomposed into the product of two other Boolean matrices. If we seek the basis of the smallest size, the problem is equivalent to finding the *ambiguous rank* [9] of the Boolean matrix, which is known to be NP-complete [24].

Given a set of subtraces, several bases are possible. Following Occam's Razor principle, we seek to compute a "simple" basis that generalizes well to unseen traces. More concretely, we seek to find a basis that is minimal in that each subtrace is a linear combination of only a small number of basis subtraces. This yields the *sparse basis problem*. In this paper, we formally define this problem in the context of Boolean matrix factorization and propose a graph-theoretic algorithm to solve the sparse-version of the problem. Such a problem is often referred to as a *sparse coding* problem in the machine learning literature, since it involves encoding a data set with a "code" in a sparse manner using few non-zero coefficients.

We apply the generated basis subtraces to the problem of error localization. In digital circuits, an especially vexing problem today is that of post-silicon debugging, where, given an error trace with potentially only a subset of signals observable and no way to reproduce the trace, one must localize the problem in space (to a small collection of error modules) and time (to a small window within the trace). Similar problems arise in debugging distributed systems. In addition, error localization is very relevant to "pre-silicon" verification as well. Our approach is to attempt to reconstruct windows of an error trace using a basis computed from slicing a set of good traces into subtraces of length p. The hypothesis is that the earliest windows that cannot be reconstructed are likely to indicate the time of the error, and the portions that cannot be reconstructed

are likely to indicate the signals (variables) that are the source of the problem. The technique can thus be applied for *simultaneous* error localization and explanation. We apply this technique to representative digital circuits.

To summarize, the main contributions of the paper are:

- We introduce the idea of *basis subtraces* as a formal way of capturing behavior of a design as exhibited by a set of traces;
- We formally define the *sparsity-constrained Boolean matrix factorization problem* and propose a graph-theoretic algorithm to solve it;
- We demonstrate with experimental results that we can mine useful specifications using our sparse coding method, and
- We show that the computed bases can be effective for simultaneous error localization and error explanation, even for transient errors, such as bit flips, that arise not just due to logical errors but also from electrical effects.

Organization. We begin in Sec. 2 with basic terminology and preliminaries. Sec. 3 introduces our approach to finding a sparse basis. In Sec. 4, we show how we can use our approach for performing error localization. Experimental results are presented in Sec. 5. Related work is surveyed in Sec. 6 and we conclude in Sec. 7.

2 Preliminaries

In this section, we introduce the basic notation used in the rest of the paper. Sec. 2.1 introduces notation representing traces of a reactive system as matrices, and Sec. 2.2 connects the matrix representation with a graph representation.

2.1 Traces and Subtraces

We model a reactive system as a transition system (V, Σ_0, δ) where V is a finite set of Boolean variables, Σ_0 is a set of initial states of the system, and δ is the transition relation. In general, V contains input, output and (internal) state variables. A state of the system σ is a Boolean vector comprising valuations to each variable in V. For clarity, we restrict ourselves in this paper to synchronous systems in which transitions occur at the tick of a clock, such as digital circuits, although the ideas can be applied in other settings as well.

Let the state of the system at the ith cycle (step) be denoted by σ_i. A *complete trace* of the system of length l is a sequence of states $\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_{l-1}$ where $\sigma_0 \in \Sigma_0$, and $(\sigma_{i-1}, \sigma_i) \in \delta$ for $1 \leq i < l$. Note however that the full system state and/or inputs might not be observed or recorded during execution. We therefore define a *trace* τ as a sequence of valuations to an observable subset of the variables in V; i.e., $\tau = \sigma'_0, \sigma'_1, \sigma'_2, \ldots, \sigma'_{l-1}$ where $\sigma'_i \subseteq \sigma_i$. A *subtrace* $\tau_{i,j}$ of length j in τ is defined as the segment of τ starting at cycle i and ending at cycle $i + j - 1$, such that $i \geq 0$, $j > 1$ and $i + j \leq l$, i.e. $\tau_{i,j} = \sigma'_i, \sigma'_{i+1}, \ldots, \sigma'_{i+j-1}$. We consider subtraces of length at least 2; i.e., containing at least one transition.

For example, Equation 1 shows a trace τ of length 4 where each state comprises a valuation to two Boolean variables. We depict the trace in matrix form, where the rows correspond to variables and the columns to cycles.

$$\begin{matrix} 1\ 0\ 1\ 1 \\ 1\ 0\ 1\ 1 \end{matrix} \tag{1}$$

The subtrace $\tau_{0,2}$ of τ is

$$\begin{matrix} 1\ 0 \\ 1\ 0 \end{matrix}$$

Let $\mathcal{T}_p$ be the set of all subtraces of length p in τ, i.e. $\mathcal{T}_p = \{\tau_{i,p} | 0 \leq i \leq l - p\}$. For any $\tau_{i,p} \in \mathcal{T}_p$, we can view it as a Boolean matrix of dimension $|V| \times p$. We can also represent it using a vector $v_i^p \in \mathbb{B}^{|V| \times p}$ by stacking the columns in $\tau_{i,p}$ (i.e., using a column-major representation). For example, v_0^2 as shown below represents the subtrace $\tau_{0,2}$.

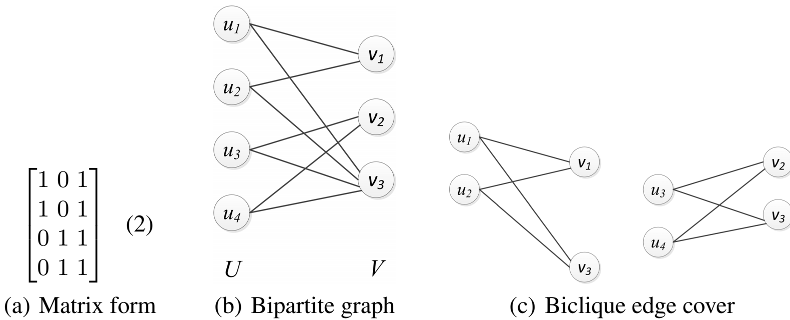
$$v_0^2 = \begin{bmatrix} 1\ 1\ 0\ 0 \end{bmatrix}^T$$

For brevity, we use v_i for v_i^p when the length of each subtrace p is obvious from the context. Hence, we can represent $\mathcal{T}_p$ as a Boolean matrix with $|V| \times p$ rows and $l - p + 1$ columns. For example, we can represent all the subtraces of length 2 for the trace in Equation 1 as the matrix in Equation 2 in Fig. 1(a).

2.2 Boolean Matrices and Bipartite Graphs

A Boolean matrix can be viewed as an adjacency matrix for a *bipartite graph* (*bigraph*, for short). Recall that a bipartite graph $G = \langle U, V, E \rangle$ is a graph with two disjoint non-empty sets of vertices U and V and such that every edge in $E \subseteq U \times V$ connects one vertex in U and one in V. For a Boolean matrix $M \in \mathbb{B}^{k_1 \times k_2}$, denote $M_{i,j}$ as the entry in the ith row and j^{th} column of M. Then, M can be represented by a bigraph G_M with $U = \{u_1, u_2, \ldots, u_{k_1}\}$ and $V = \{v_1, v_2, \ldots, v_{k_2}\}$, such that there is an edge connecting $u_i \in U$ and $v_j \in V$ if and only if $M_{i,j} = 1$. For example, the matrix X in Equation 2 (Fig. 1(a)) can be represented by the bigraph G_X in shown in Fig. 1(b).

A *biclique* is a complete bipartite graph; i.e., a bipartite graph $G' = \langle U', V', E' \rangle$ where $E' = U' \times V'$. Given a bigraph G, a *maximal edge biclique* of G is a biclique $B_1 = \langle U_1 \subseteq U, V_1 \subseteq V, E_1 = U_1 \times V_1 \rangle$ if it is not contained in another biclique of G, that is, there does not exist another biclique $B_2 = \langle U_2 \subseteq U, V_2 \subseteq V, E_2 = U_2 \times V_2 \rangle$ and



(a) Matrix form (b) Bipartite graph (c) Biclique edge cover

Fig. 1. Subtraces in matrix and bigraph form, and corresponding biclique edge cover

either $U_1 \subset U_2$ or $V_1 \subset V_2$. In the rest of the paper, we use the pair of vertices (U_1, V_1) to denote the maximal edge biclique B_1. For a set of bicliques Cov and a bigraph G, denote E_{Cov} as the set of edges in G covered by Cov, i.e. $\forall\, e \in E_{Cov}, \exists\, G' = \langle U' \subseteq U, V' \subseteq V, E' \rangle \in Cov,$ s.t. $e \in E'$. Cov is a *biclique edge cover* of G if and only if all the edges E in G are covered by the set, i.e. $E_{Cov} = E$. Abusing notation a little, we use E_v to denote the set of edges connected to vertex v. The smallest number of bicliques needed is called the *bipartite dimension* of G. For example, a biclique cover for the bigrah in Figure 1(b) is shown in Figure 1(c).

The view of Boolean matrices as bigraphs is relevant for decomposing a set of traces into a set of basis subtraces. The following problem is important in this context.

Definition 1. *Consider a Boolean matrix $X \in \mathbb{B}^{m\times n}$, the* Boolean matrix factorization problem *is to find k and Boolean matrices $B \in \mathbb{B}^{m\times k}$ and $S \in \mathbb{B}^{k\times n}$ such that*

$$X = B \circ S \tag{3}$$

That is, X is decomposed into a Boolean combination (denoted by the operator $\circ$) of two other Boolean matrices, in which scalar multiplication is the Boolean AND operator $\wedge$, and scalar addition ("$+$") is the Boolean OR operator $\vee$. In other words, we perform matrix/vector operations over the Boolean semi-ring with $\wedge$ as the multiplicative operator and $\vee$ as the additive operator. For example, the matrix in Equation 2 (Fig. 1(a)) can be factorized in the following way.

$$\begin{bmatrix} 1\,0\,1 \\ 1\,0\,1 \\ 0\,1\,1 \\ 0\,1\,1 \end{bmatrix} = \begin{bmatrix} 1\,0 \\ 1\,0 \\ 0\,1 \\ 0\,1 \end{bmatrix} \circ \begin{bmatrix} 1\,0\,1 \\ 0\,1\,1 \end{bmatrix}$$

We use $M_{\cdot,i}$ to denote the ith column vector of a matrix M, and $M_{i,\cdot}$ to denote the ith row vector of M. Thus, the columns of matrix X are $X_{\cdot,1}, X_{\cdot,2}, \ldots, X_{\cdot,n}$. We will refer to X as the *data matrix* since it represents the traces which are the input data. We call the matrix B the *basis matrix* because each $B_{\cdot,i}$ can be viewed as some basis vector in $\mathbb{B}^m$. We call the matrix S the *coefficient matrix*. Each $S_{\cdot,i}$ is a Boolean vector in which a 1 in the jth entry indicates that the jth basis vector is used in the decomposition and 0 otherwise.

We can also rewrite the factorization in the following way as a Boolean sum of the matrices formed by taking the tensor (outer) product of the ith column in B and the ith row in S.

$$\begin{bmatrix} 1\,0\,1 \\ 1\,0\,1 \\ 0\,1\,1 \\ 0\,1\,1 \end{bmatrix} = \begin{bmatrix} 1\,0\,1 \\ 1\,0\,1 \\ 0\,0\,0 \\ 0\,0\,0 \end{bmatrix} + \begin{bmatrix} 0\,0\,0 \\ 0\,0\,0 \\ 0\,1\,1 \\ 0\,1\,1 \end{bmatrix}$$

Notice that the two matrices on the right hand side represent the bicliques in Fig. 1(c).

Remark 1. *Clearly, a solution always exists for the problem in Definition 1. This is because one can always pick $k = n$ such that $B = I(B = X)$ and $S = X(S = I)$ (where I is the identity matrix). However, this is not particularly revealing in terms of*

the behaviors which each $X_{\cdot,i}$ is composed of. One alternative is to minimize k. The smallest k for which such a decomposition exists is called the ambiguous rank [9] of the Boolean matrix X. It is also equal to the bipartite dimension of the bigraph G_X corresponding to matrix X. The problem of finding a Boolean factorization of X with the smallest k is equivalent to finding a biclique edge cover of G_X with the minimum number of bicliques. Both problems are NP-hard [24]. On the other hand, one can choose to find an overcomplete basis ($k > n$) such that each $X_{\cdot,i}$ can be expressed as a Boolean sum of only a few basis vectors. We discuss this formulation in detail in Section 3.

3 Specification Mining via Sparse Coding

In this section, we describe how specifications are mined via sparse Boolean matrix factorization. The specifications we mine, *basis subtraces*, can be viewed as temporal patterns over a finite time window.

3.1 Formulation as Sparse Coding Problem

The notion of sparsity is borrowed from the wealth of literature in machine learning such as sparse coding [15] and sparse principal component analysis (PCA) [30]. *The key insight is that a sparsity constraint often generates a better interpretation of the data in terms of the underlying concepts.* In the setting of mining specifications from a trace, we argue that each subtrace of a trace can be viewed as a superposition of patterns, and a potential specification is a pattern that is commonly shared by multiple subtraces. These patterns are the so-called *basis subtraces*.

We present the *sparse Boolean matrix factorization problem* for computing basis subtraces below. A few different options are presented for formulating the problem and we pick one with a notion of sparsity that seems well-suited to our context.

Definition 2. *Given $X \in \mathbb{B}^{m \times n}$ and a positive integer C, the* sparsity-constrained Boolean matrix factorization problem *is to find k, $B \in \mathbb{B}^{m \times k}$, and $S \in \mathbb{B}^{k \times n}$ such that*

$$\begin{aligned} X &= B \circ S \\ \text{and} \quad \|S_{\cdot,i}\|_1 &\le C, \ \forall_i \end{aligned} \tag{4}$$

Let us reflect on the above problem formulation. The constraint $X = B \circ S$ imposes the requirement that the input data (subtraces) represented by X must be reconstructed as a superposition of the subtraces represented by B, with S encoding the coefficients in the superposition. The second constraint $\|S_{\cdot,i}\|_1 \le C, \forall_i$ encodes the sparsity constraint, which ensures that each subtrace in X is a sparse superposition of the subtraces in B.

More precisely, the definition above imposes a constraint on the number of 1s per column of S. Similar to the Boolean matrix factorization problem in Definition 1, a trivial solution is to set $B = X$ and $S = I$ (and $k = n$). However, this solution does not produce any sharing of patterns amongst the different subtraces and hence is

useless for specification mining. The optimization objective is thus the following, which maximizes the number of 1s in S.

$$\text{maximize} \sum_i \sum_j S_{i,j} \tag{5}$$

We describe how we solve this problem in Section 3.2.

One might also consider defining sparsity in a somewhat different manner. Instead of imposing a L_1-norm constraint on the columns of the coefficient matrix S, we can seek B and S such that the total sparsity is minimized.

Definition 3. *Given $X \in \mathbb{B}^{m\times n}$ and a positive integer k, the* sparsity-optimized Boolean matrix factorization problem *is the following optimization problem.*

$$\begin{aligned} \underset{B,S}{\text{minimize}} \quad & \sum_i^n \|S_{\cdot,i}\|_1 \\ \text{subject to} \quad & X = B \circ S \end{aligned} \tag{6}$$

The main issue with this problem definition is that k is fixed; in other words, one has to "guess" a suitable k for which B and S can be computed. While modifications of this problem that restrict or minimize k could potentially be useful, we leave the investigation of these variants of 6 to future work.

3.2 Solving the Sparse Coding Problem

In this section, we describe an algorithm that solves the *sparsity-constrained Boolean matrix factorization problem*, as formalized in Equations 4 and 5. Our solution is guaranteed to satisfy the sparsity constraint and tries to maximize the objective in Equation 5. The algorithm exploits the connection between the matrix factorization problem and the biclique edge cover problem described in Sec. 2. Specifically, it is based on growing a biclique edge cover Cov for the bigraph $G_X = \langle U, V, E\rangle$ corresponding to matrix X. At each step, a maximal edge biclique that covers some number of previously uncovered edges is added to Cov until Cov covers all the edges. The sparsity constraint is then a constraint on the number of maximal bicliques that can be used to cover the edges that connect each vertex in V. (Recall that each vertex in V corresponds to a column $S_{\cdot,i}$ of S.)

Notice that this algorithm relies on a way to generate maximal edge bliciques of a bigraph. Computing these bicliques is not easy: for instance, the closely-related problem of finding a maximum (not maximal) edge biclique in a bigraph is NP-complete [23]. Additionally, the number of maximal bicliques in a bigraph can be exponential in the number of vertices [11].

However, there exist enumeration algorithms that are polynomial in the combined input and output size, such as the Consensus algorithm in [1]. In addition, this algorithm runs in incremental polynomial time.

Algorithm 1 solves the sparsity-constrained Boolean matrix factorization problem by building upon some key concepts in the Consensus algorithm and adapting them for our problem context. These concepts are described below.

- **Consensus:** For two bicliques $B_1 = (U_1, V_1)$ and $B_2 = (U_2, V_2)$, the consensus of B_1 and B_2 is $B_3 = (U_3, V_3)$ where $U_3 = U_1 \cap U_2$ and $V_3 = V_1 \cup V_2$.
- **Extend to a maximal biclique:** For a consensus biclique $B_1 = (U_1, V_1)$, we can extend it to a maximal biclique $B_2 = (U_2, V_2)$ where $U_2 = U_1$ and $V_2 = \{v \mid \forall\, u \in U_1, (u, v) \in E\}$
  (V_2 is the set of vertices in V that are connected to every vertex in U_1).
- **v-rooted star biclique:** A v-rooted star biclique is the biclique formed by the node $v \in V$ and all the nodes connected to v (and the edges), i.e. $(\{u \mid (u, v) \in E\}, \{v\})$

The main idea of Algorithm 1 is the following. We try to cover the edges in the bigraph with as many maximal bicliques as possible, until we are about to violate the sparsity constraint at some vertex $v \in V$. In that case, we cover the remaining edges of v with the v-rooted star biclique. If there is still some $v \in V$ with uncovered edges at the end of the iteration, then we just cover it with the v-rooted star biclique as well. The final cover will be the union of the set of maximal bicliques added in the consensus steps $Cov_1 \setminus Cov_0$ with the set of star bicliques Cov_2.

4 Application to Error Localization

The key idea in our approach is to localize errors by attempting to reconstruct the error trace from basis subtraces generated from correct traces. Our hypothesis is that the earliest section (subtrace) of the error trace that cannot be reconstructed contains the likely cause of the error. Our localization algorithm is presented in this section, along with some theoretical guarantees. We begin with the problem definition.

Algorithm 1. Sparsity-constrained cover

1: **Input:** the set of v-rooted star bicliques Cov_0 and sparsity constraint C.
2: **Initialize:** $Cov_1 := Cov_0$, $Cov_2 := \emptyset$, $\alpha_v := C$, $\forall\, v \in V$, and $V_{cov} := \emptyset$.
3: **repeat**
4: Pick a new pair of bicliques $B_1 = (U_1, V_1)$ from Cov_1 and $B_2 = (U_2, V_2)$ from Cov_0, form the consensus B_3.
5: Extend B_3 to a maximal biclique $B_4 = (U_4, V_4)$.
6: **if** $(B_4 \notin Cov_1) \wedge (V_4 \cap V_{cov} = \emptyset)$ **then**
7: Add B_4 to Cov_1.
8: **for** $v \in V_4 \setminus V_{cov}$ **do**
9: $\alpha_v := \alpha_v - 1$
10: **if** $\alpha_v = 1$ **then** Add the v-rooted star biclique to Cov_2 and add v to V_{cov} **end if**
11: **end for**
12: **end if**
13: **until** $E_{(Cov_1 \setminus Cov_0) \cup Cov_2} = E$ or cannot find a new pair of bicliques B_1 and B_2
14: **for** $v \in V \setminus V_{cov}$ **do**
15: Add the v-rooted star biclique to Cov_2.
16: **end for**
17: **Output:** the sparsity-constrained cover $(Cov_1 \setminus Cov_0) \cup Cov_2$

4.1 Problem Definition

Consider the problem of localizing an error given a set of correct traces and a single error trace. Our goal is to identify a small interval of the timeline at which the error occurred. What makes the problem especially challenging is that the input sequence that generated the error trace is either unknown (or only partially known) or it is extremely slow to re-simulate the input sequence (if known) on the correct design (also sometimes referred to as a "golden model"). This means that a simple anomaly detection technique which checks the first divergence of the error trace and the correct trace obtained by simulating the golden model on the same input sequence does not work. One has to use the set of correct traces to help localize the bug in the error trace. This setting is especially applicable to post-silicon debugging where the bugs are often difficult to diagnose due to limited observability, limited reproducibility and susceptibility to environmental variations.

More formally, the error localization problem we address in this section can be defined as follows.

Definition 4. *Given an error trace of length l and an integer p, partition the trace into non-overlapping subtraces each of length p (w.l.o.g. assume l is an integer multiple of p; otherwise, the last subtrace can be treated specially).*

Then, the error localization problem *is to identify the subtrace containing the first point of deviation of the error trace from the correct trace on the same input sequence.*

One might note that the problem we define is not the only form of error localization that is desirable. For instance, one might also want to narrow down the fault to the signals/variables that were incorrectly updated.

Also, there might be more than one source of an error, in which case one might want to identify all of the sources.

While these goals are important, we contend that our algorithm to address the problem defined above can also be used to achieve these additional objectives. For example, the error explanation technique we present below can be used to identify which variables were incorrectly updated and how. Similarly, one can apply our reconstruction-based localization algorithm iteratively to identify multiple subtraces that cannot be reconstructed from the basis subtraces, and could potentially be used to identify multiple causes of an error.

4.2 Localization by Reconstruction

As described above, the key hypothesis underlying our approach is that the earliest section (subtrace) of the error trace that cannot be reconstructed contains the likely cause of the error.

Our error localization algorithm operates in the following steps:

1. Given a set of correct traces $\mathcal{T}$, first obtain the set of all *unique* subtraces of length p in $\mathcal{T}$. Denote this set by $\mathcal{T}_p$. Using the approach described in Section 2, convert the set $\mathcal{T}_p$ to a data matrix X.
2. Solve the *sparsity-constrained Boolean matrix factorization problem* for X for a given constant C.

3. Given an error trace τ', partition it into an ordered set of q subtraces of length p. Denote this set by $\mathcal{T}'_p$. The elements in $\mathcal{T}'_p$ are ordered by their positions in τ'. Convert $\mathcal{T}'_p$ to a data matrix X'.
4. Starting from $X'_{\cdot,0}$, try to reconstruct $X'_{\cdot,i}$ using the basis computed above with the same sparsity constraint C. Return i as the location of the bug if the reconstruction fails. In case all reconstructions succeed, return $\perp$ indicating inability to localize the error.

Algorithm 2 describes the above approach in more detail using pseudo-code. It uses the following subroutines:

- **dataMatrix** is the procedure that converts a set of subtraces to the corresponding data matrix described in Section 2.
- **sparseBasis** solves the *sparsity-constrained Boolean matrix factorization problem* using the graph-theoretic algorithm presented in Section 3 for X with a given C, and returns the computed basis B.
- **reconstructTrace** solves the following minimization problem.

$$\begin{aligned} \underset{S_i}{\text{minimize}} \quad & \|X'_{\cdot,i} \oplus (B \circ S_{\cdot,i})\|_1 \\ \text{subject to} \quad & \|S_{\cdot,i}\|_1 \leq C \end{aligned} \tag{7}$$

  where $\oplus$ is the bit-wise Boolean XOR operator, and is interpreted to apply entry-wise on matrices.

  Notice that for fixed C, this problem is *fixed-parameter tractable* because we can use a brute-force algorithm that enumerates all the $\sum_{1\leq i\leq C}\binom{k}{i}$ possible $S_{\cdot,i}$. It can also be solved using a pseudo-Boolean optimization formulation, where the Boolean variables in the optimization problem are the entries in $S_{\cdot,i}$.

Error Explanation. Denote $S^*_{\cdot,i}$ as the optimal solution to the minimization problem in Equation 7. If the minimum value is non-zero, then $E = X'_{\cdot,i} \oplus (B \circ S^*_{\cdot,i})$ is the minimum difference between the error subtrace $X'_{\cdot,i}$ and the reconstructed subtrace $B \circ S^*_{\cdot,i}$. Notice that E is also a subtrace, and can be interpreted as a finite sequence of assignments to system variables. In our experience, E is a pattern that explains the error; we expand further on this point using our experiments in Sec. 5.

Algorithm 2. Error localization in time

```
Input: Set of subtraces T_p from set of correct traces T, T'_p from error trace τ'
Input: Constant C > 0
  X = dataMatrix(T_p);  X' = dataMatrix(T'_p);  B = sparseBasis(X, C)
  for i := 0 → q − 1 do
    E = reconstructTrace(X'_{·,i}, B, C)
    if E ≠ 0 then  return i  end if
  end for
  return ⊥
```

4.3 Theoretical Guarantees

We now give conditions under which our error localization approach is *sound*. By sound, we mean that when our algorithm reports a subtrace as the cause of an error, it is really an erroneous subtrace that deviates from correct behavior.

Since our approach mines specifications from traces, its effectiveness fundamentally depends on the quality of those traces. Specifically, our soundness guarantee relies on the set of traces $\mathcal{T}$ satisfying the following *coverage metrics* defined over the transition system (V, Σ_0, δ) of the golden model:

1. *Initial State Coverage:* For every initial state $\sigma_0 \in \Sigma_0$, there exists some trace in $\mathcal{T}$ in which σ_0 is the initial state.
2. *Transition Coverage:* For every transition $(\sigma, \sigma') \in \delta$, there exists some trace in $\mathcal{T}$ in which the transition (σ, σ') occurs.

While full transition coverage can be difficult to achieve for large designs, there is significant work in the simulation-driven hardware verification community on achieving a high degree of transition coverage [25]. If achieving transition coverage is challenging for a design, one could consider slicing the traces based on smaller module boundaries and computing tests that ensure full transition coverage within modules, at the potential cost of missing cross-module patterns.

Our soundness theorem relates test coverage with effectiveness of error localization.

Theorem 1. *Given a transition system Z for the golden model and a set of finite-length traces $\mathcal{T}$ of Z satisfying initial state and transition coverage, if Algorithm 2 is invoked on $\mathcal{T}$ and an arbitrary error trace τ', then Algorithm 2 is sound; viz., if it reports a subtrace of τ' as an error location, that subtrace cannot be exhibited by Z.*

Proof. (sketch) The proof proceeds by contradiction. Suppose Algorithm 2 reports a subtrace of τ' as the location of the error. Recall that a subtrace must be of length at least 2. Thus, if we compute basis subtraces of length 2, any transition of the golden model Z can be expressed as a superposition of these basis subtraces and hence reconstructed from the basis subtraces B, since $\mathcal{T}$ contains all transitions of Z. A subtrace reported as an error location, in contrast, is one that cannot be expressed as a superposition of the basis subtraces and hence **reconstructTrace** will report that it cannot be reconstructed. Thus, any subtrace reported as an error location by Algorithm 2 cannot be a valid transition of the golden model Z. □

We also note that, in theory, it is possible for Algorithm 2 to miss reporting a subtrace that is an error location, if that subtrace is expressible as a superposition of basis subtraces. However, experiments indicate that it is usually accurate in pinpointing the location of the error. Details of our experiments are provided in Sec. 5.

5 Experimental Results

In this section, we evaluate our sparse coding approach to generate specifications and localize errors based on the following criteria.

(1) Are the computed "basis subtraces" meaningful? That is, do they correspond to some interesting specifications of the test circuit?

(2) Do the “basis subtraces” capture sufficient underlying structure of a trace? That is, can they be used to reconstruct traces that are generated from unseen input sequences?
(3) How accurately can we localize an error in an unseen trace (generated by unseen input sequences)?
(4) How good are the error explanations?

5.1 Arbiter

We first use a 2-port arbiter as an illustrative example to evaluate our approach. The 2-port arbiter is a circuit that takes two Boolean inputs corresponding to two potentially competing requests, and produces two Boolean outputs corresponding to the two grants. It implements a round-robin scheme such that it will give priority to the port at which a request has not been most recently granted. Let r_0, r_1 denote the input requests and g_0, g_1 denote the corresponding output grants. If a request r_i is granted, g_i goes high in the same cycle. Figure 2 shows part of a trace of the arbiter over the request and grant signals. The input requests were randomly generated and the trace was 100 cycles long.

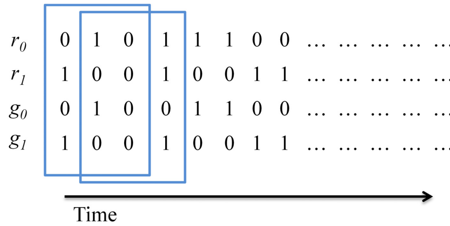


Fig. 2. A normal trace of a 2-port round-robin arbiter

We used a sliding window of length 3 to collect a set of subtraces. We then applied our sparse coding algorithm described in Section 3.2 to extract a set of “basis subtraces”. We set the sparsity constraint to 4 for this experiment, which is only one third of the total number of entries in a subtrace. We now evaluate our approach with respect to the four criteria stated at the start of this section:

(1) Figure 3 shows some of the basis subtraces computed. We can observe that basis (a) and (b) correspond to the correct behavior of the arbiter granting a request at the same cycle when there is no competing request. Basis (c) shows that when there are two competing requests at the same cycle, the arbiter first grants one of the requests and the ungranted request will stay asserted the next cycle and then gets granted.
(2) We further simulated the arbiter with random inputs another 100 times each for 100 cycles. For each of these traces, we also use a sliding window to partition them into subtraces of length 3. Using the basis computed from the trace depicted in Figure 2, we tried to reconstruct these subtraces and succeeded in every attempt. This was because all the sub-behaviors were fully covered in the trace from which the bases were computed, even though unseen subtraces exist in the new traces.

0	0	0
1	0	0
0	0	0
1	0	0

(a)

0	1	0
0	0	0
0	1	0
0	0	0

(b)

0	1	1
0	1	0
0	0	1
0	1	0

(c)

Fig. 3. Three basis subtraces computed via sparse coding

(3) For each of the 100 traces in (2), we randomly injected a single bit error (flipping its value) at a random cycle to one of the four signals in the trace. Our task was to test if we could localize the error to a subtrace of length 3 that contained it.
The following example illustrates one of the experiments. Figure 4(a) shows a snapshot of the trace.

	95	96	97	98	99
r_0	0	0	0	1	0
r_1	0	0	1	0	0
g_0	0	0	0	1	0
g_1	0	0	0	0	0

(a) Bit flip at r_1 at cycle 97

	96	97	98
r_0	0	0	1
r_1	0	1	0
g_0	0	0	1
g_1	0	0	0

(b) Error subtrace as identified

	96	97	98
r_0	0	0	0
r_1	0	1	0
g_0	0	0	0
g_1	0	0	0

(c) Error explanation subtrace

	96	97	98
r_0	0	0	0
r_1	0	0	0
g_0	0	0	0
g_1	0	1	0

(d) Alternative error explanation subtrace

Fig. 4. Error trace and explanation subtrace

Using the approach described in Algorithm 2, the subtrace containing the error was correctly identified. Among the 100 traces, we successfully identified the window at which the error was injected for 84 of them. Figure 4(b) shows the error subtrace. Following Equation 7, Figure 4(c) shows the (differential) subtrace $X_i' \oplus (B \circ S_i)$ that minimizes $|X_i' \oplus (B \circ S_i)|_1$ and serves as an error explanation.

Clearly, this subtrace reveals the injected error. While no fault model is assumed, this approach still pinpoints the bug behaviorally – a grant was not produced at g_1 at cycle 97 even when the corresponding request was made at r_1. Note that multiple error explanations (solutions to the minimization problem in Equation 7) can exist. Figure 4(d) shows an alternative error explanation subtrace for this example where g_1 was asserted but r_1 was not asserted at cycle 97.

(4) In Section 4.2, we argue that the *minimum difference* between an error subtrace and any possible reconstructed subtrace using the computed basis can serve as an explanation for the error. In the 84 traces for which the error was correctly localized, the injected bit error was also uncovered by solving the optimization problem in Equation 7.

5.2 Chip Multiprocessor (CMP) Router

Our second, larger case study is a router for on-chip networks. The main goal of this case study was to explore how the technique scales to a larger design, and how effective it is for error localization.

Fig. 5 illustrates the high-level design of the router, as a composition of four high-level modules. The input controller comprises a set of FIFOs buffering incoming flits and interacting with the arbiter. When the arbiter grants access to a particular output port, a signal is sent to the input controller to release the flits from the buffers, and at the same time, an allocation signal is sent to the encoder which in turn configures the crossbar to route the flits to the appropriate output port.

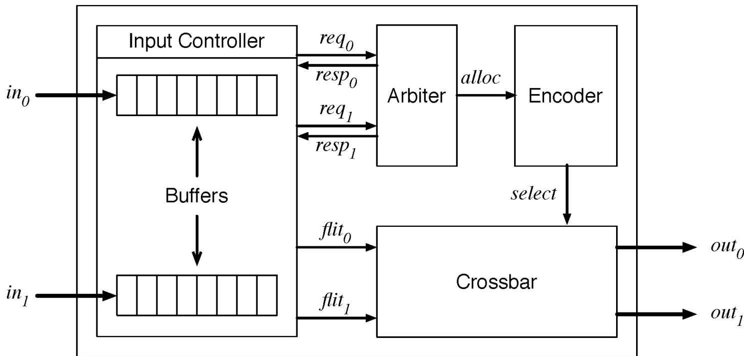


Fig. 5. CMP Router comprising four high-level modules

The router was simulated with two flit generating modules that each issued random data packets (each consists of a head, some body and a tail flit) to the respective input ports of the router. We observed 14 Boolean control signals in the router and a trace was generated for these 14 signals with a simulation length of 1000 cycles. We used a subtrace width of 2 cycles and obtained 93 distinct subtraces each with 14 signals over 2 cycles. A basis was computed from these 93 distinct subtraces subject to a sparsity constraint of 52 (see explanation for the choice of this number at the end of this section). It took 0.243 seconds to obtain this basis which contained 189 basis subtraces.

The router was simulated 100 times with different inputs. We used the first simultion trace to obtain the basis as described in the previous paragraph and the rest 99 traces for error localization. For each of these 99 traces, a single bit flip was injected to a random signal at a random cycle. The goal of experiment is to localize this bit error to a subtrace of 2 cycles (among the 999 subtraces for each trace) in which the error was introduced.

Following the localization approach described in Section 4.2 of the paper, 55 out of 99 of the errors were correctly localized. The remaining 44 errors were not localized (all the subtraces including error subtrace were reconstructed using the computed basis). The overall accuracy of the error localization procedure in this experiment was 55.6%.

Why is this error localization approach useful? Imagine you are given a good trace (or a collection of good traces) and then an error trace (that cannot be reproduced),

and you are asked to localize the error without knowing very much about the underlying system that generates these traces. (This situation arises when dealing with legacy systems, for example.) Here are two plausible alternative options to our sparse coding approach and the corresponding results:

(a) Hash all the distinct subtraces of 2 cycles in length in the good trace. For each of the subtraces of the same dimension in the bad trace, check if it is contained in the hash, and report an error if it is not contained. For the same traces used above, an error was reported for each of the 99 traces even before any bit flip was injected.
(b) Use a basis that spans the entire space of subtraces of 2 cycles, e.g. 14×2 subtraces where each contains only a single 1 in its entries and is orthogonal to the others. However, it is obvious that we cannot localize any error using this basis since it spans all possible subtraces.

Our method can be viewed as something in between (a) and (b). It finds a subspace that not only contains all the good subtraces but also generalizes well to unseen good subtraces from the basis. The generalization is a sparse composition of some key patterns in the good subtraces. An error is reported if a subtrace lies outside this subspace. The number 52 for the sparsity constraint was determined as a result of the minimization of sparsity such that the computed basis was just sufficient to reconstruct all the other 99 traces before error injection. This limits the size of the subspace spanned by the basis and hence increases the ability to detect an error.

6 Related Work

We survey related work along three dimensions: Boolean matrix factorization, mining specifications from traces, and error localization techniques.

6.1 Boolean Matrix Factorization

Matrix factorization or factor analysis methods are prevalent in the data mining community, with a common goal to discover the latent structures in the input data. While most of these methods are focusing on real-valued matrices, there have been several works recently that target Boolean matrices, for applications such as role mining [26]. Miettinen et al. [20] introduced the discrete basis problem (DBP). DBP is similar to our definition of the Boolean matrix factorization problem in which k is fixed and the objective is to minimize the reconstruction error. They showed that DBP is NP-hard and gave a simple greedy algorithm for solving it. In terms of sparse decomposition, Miettinen [19] showed the existence of sparse factor matrices for a sparse data matrix. Our paper describes a different notion of sparsity – we seek to express each data vector as a combination of only a few basis vectors, which can be dense themselves.

6.2 Specification Mining

Approaches to mine specifications can be largely categorized into static and dynamic methods. We restrict ourselves here to the dynamic methods that mine specifications from traces. Daikon [8] is one of the earliest tools that mine single-state invariants or

pre-/post-conditions in programs. In contrast, we focus on mining (temporal) properties over a finite window for reactive (hardware) designs. Some existing tools produce temporal properties in the form of automata. Automata-based techniques generally fall into two categories. The first class of methods learn a single complex specification (usually as a finite automaton) over a specific alphabet, and then extract simpler properties from it. For instance, Ammons et al. [2] first produce a probabilistic automaton that accepts the trace and then extract from it likely properties. However, learning a single finite state machine from traces is NP-hard [12]. To achieve better scalability, an alternative is to first learn multiple small specifications and then post-process them to form more complex state machines. Engler et al. [7] first introduce the idea of mining simple alternating patterns. Several subsequent efforts [27,28,10] built upon this work. In previous work, we proposed a specification mining approach similar to Javert that focuses on patterns relevant for digital circuits [16] and showed how this can be applied to error localization. However, such approaches are limited by the set of patterns. The present work seeks to remove this limitation by inferring design-specific patterns in the form of basis subtraces.

6.3 Error Localization

The problem of error localization and explanation has been much studied in literature in several communities: software testing, model checking, and electronic design automation. In model checking, Groce et al. [13] present an approach based on distance metrics which, given a counterexample (error trace), finds a correct trace as "close" as possible to the error trace according to the distance metrics. Ball et al. [3] present an approach to localizing errors in sequential programs. They use a model checker as a subroutine, with the core idea to identify transitions of an error trace that are not in any correct trace of the program, and use this for error localization. Both of these approaches operate on error traces generated by model checking, and thus have full observability of the inputs and state variables. In contrast, in our context, the trace includes only-partially observed state and is not reproducible.

In the software testing community, researchers have attempted to use predicates and mined specifications to localize errors [17,6]; however, these rely on human insight in choosing a good set of predicates/templates. In contrast, our approach automatically derives specifications in the form of basis subtraces, which can be seen as temporal properties over a finite window. Program spectra [14], which include computing profiles of program behavior such as summaries of the branches or paths traversed, have also been proposed as ways to separate good traces from error traces; however, these techniques are of limited use for digital circuits since they rely on the path structure of sequential programs and give no guarantees on soundness.

In the area of post-silicon debugging (see [21] for a recent survey), the problem of error localization has received wide attention. The IFRA approach [22], which is largely specialized for processor cores, is based on adding on-chip recorders to a design to collect "instruction footprints" which are analyzed offline with some input from human experts. Li et al. [16] have proposed the use of mined specifications to perform error localization; however, this approach relies on human insight in supplying the right templates to mine temporal logic specifications. Zhu et al. [29] propose a SAT-based

technique for post-silicon fault localization, where *backbones* are used to propagate information across sliding windows of an error trace. This additional information helps make the approach more scalable and addresses the problem of limited observability. Backspace [5] addresses the problem of reproducibility by attempting to reconstruct one or more “likely” error traces by performing backwards reachability guided by recorded signatures of system state; such a system is complementary to the techniques proposed herein for error localization.

7 Conclusion and Future Work

In this paper, we have presented *basis subtraces*, a new formalism to capture system behavior from simulation or execution traces. We showed how to compute a *sparse* basis from a set of traces using a graph-based algorithm. We further demonstrated that the generated basis subtraces can be effectively used for error localization and explanation.

In terms of future work, we envisage two broad directions: improving scalability and applying the ideas to other domains. Since the Boolean matrix factorization problem and its sparse variants can be computationally expensive to solve, the scalability of the approach must be improved. In this context, it would be interesting to use slightly different definitions of a basis (for example, using the field of rationals rather than the semi-ring we consider) so that the problem of computing a sparse basis is polynomial-time solvable. Moreover, the ideas introduced in this paper can be extended beyond digital circuits to software, distributed systems, analog/mixed-signal circuits, and other domains, providing many interesting directions for future work.

Acknowledgement. The authors acknowledge the support of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. This work was also supported in part by an Alfred P. Sloan Research Fellowship and a Hellman Family Faculty Fund Award.

References

1. Alexe, G., Alexe, S., Crama, Y., Foldes, S., Hammer, P.L., Simeone, B.: Consensus algorithms for the generation of all maximal bicliques. Discrete Appl. Math. 145, 11–21 (2004)
2. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL, pp. 4–16 (2002)
3. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: localizing errors in counterexample traces. In: POPL, pp. 97–105 (2003)
4. Buechi, J.R.: On a Decision Method in Restricted Second-Order Arithmetic. In: International Congress on Logic, Methodology, and Philosophy of Science, pp. 1–11. Stanford University Press (1962)
5. de Paula, F.M., Gort, M., Hu, A.J., Wilton, S.J.E., Yang, J.: Backspace: Formal analysis for post-silicon debug. In: FMCAD, pp. 1–10 (2008)
6. Dodoo, N., Lin, L., Ernst, M.D.: Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, MIT Laboratory for Computer Science (2003)
7. Engler, D., et al.: Bugs as deviant behavior: a general approach to inferring errors in systems code. In: SOSP, pp. 57–72 (2001)

8. Ernst, M., et al.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. 69(1-3), 35–45 (2007)
9. Froidure, V.: Rangs des relations binaires, semigrollpes de relations non ambigues. PhD thesis (June 1995)
10. Gabel, M., Su, Z.: Javert: fully automatic mining of general temporal properties from dynamic traces. In: FSE, pp. 339–349 (2008)
11. Gaspers, S., Kratsch, D., Liedloff, M.: On Independent Sets and Bicliques in Graphs. In: Broersma, H., Erlebach, T., Friedetzky, T., Paulusma, D. (eds.) WG 2008. LNCS, vol. 5344, pp. 171–182. Springer, Heidelberg (2008)
12. Gold, E.M.: Complexity of automatic identification from given data 37, 302–320 (1978)
13. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error explanation with distance metrics. Software Tools for Technology Transfer (STTT) 8(3), 229–247 (2006)
14. Harrold, M.J., Rothermel, G., Sayre, K., Wu, R., Yi, L.: An empirical investigation of the relationship between spectra differences and regression faults. Softw. Test., Verif. Reliab. 10(3), 171–194 (2000)
15. Lee, H., Battle, A., Raina, R., Ng, A.Y.: Efficient sparse coding algorithms. In: NIPS, pp. 801–808 (2007)
16. Li, W., Forin, A., Seshia, S.A.: Scalable specification mining for verification and diagnosis. In: Design Automation Conference (DAC), pp. 755–760 (June 2010)
17. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I.: Bug isolation via remote program sampling. In: PLDI, pp. 141–154 (2003)
18. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer-Verlag New York, Inc., New York (1992)
19. Miettinen, P.: Sparse boolean matrix factorizations. In: Proceedings of the 2010 IEEE International Conference on Data Mining, ICDM 2010, pp. 935–940. IEEE Computer Society, Washington, DC (2010)
20. Miettinen, P., Mielikäinen, T., Gionis, A., Das, G., Mannila, H.: The Discrete Basis Problem. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, pp. 335–346. Springer, Heidelberg (2006)
21. Mitra, S., Seshia, S.A., Nicolici, N.: Post-silicon validation: Opportunities, challenges and recent advances. In: Proceedings of the Design Automation Conference (DAC), pp. 12–17 (June 2010)
22. Park, S.B., Bracy, A., Wang, H., Mitra, S.: Blog: Post-silicon bug localization in processors using bug localization graphs. In: DAC (2010)
23. Peeters, R.: The maximum edge biclique problem is NP-complete. Discrete Applied Mathematics 131(3), 651–654 (2003)
24. Siewert, D.J.: Biclique covers and partitions of bipartite graphs and digraphs and related matrix ranks of 0,1 matrices. PhD thesis (2000)
25. Tasiran, S., Keutzer, K.: Coverage metrics for functional validation of hardware designs. IEEE Design & Test of Computers 18(4), 36–45 (2001)
26. Vaidya, J., Atluri, V., Guo, Q.: The role mining problem: Finding a minimal descriptive set of roles. In: Symposium on Access Control Models and Technologies (SACMAT), pp. 175–184 (2007)
27. Weimer, W., Necula, G.C.: Mining Temporal Specifications for Error Detection. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 461–476. Springer, Heidelberg (2005)
28. Yang, J., et al.: Perracotta: mining temporal api rules from imperfect traces. In: ICSE, pp. 282–291 (2006)
29. Zhu, C.S., Weissenbacher, G., Malik, S.: Post-silicon fault localisation using maximum satisfiability and backbones. In: FMCAD (2011)
30. Zou, H., Hastie, T., Tibshirani, R.: Sparse principal component analysis. Journal of Computational and Graphical Statistics 15, 2006 (2004)

Sliding between Model Checking and Runtime Verification

Martin Leucker

Universtity of Lübeck
Institute for Software Engineering and Programming Languages

Abstract. We present a unified semantics for linear temporal logic capturing model checking and runtime verification. Moreover, we present the main ingredients of a corresponding monitor synthesis procedure.

1 Introduction

One of the main research problems in runtime verification (RV) is the automatic synthesis of monitors from high-level specifications. A typical high-level specification language used in RV is linear temporal logic (LTL), for which different RV-specific semantical adaptions have been proposed in recent years. In this paper we propose a further semantics for LTL, a *predictive semantics*, which unifies ideas from model checking and runtime verification. Using abstraction, it allows to either concentrate on a more model checking like analysis of the underlying system, or a rather runtime verification oriented view on the system under scrutiny.

The main object to study in RV is the current *run* of the underlying system. Such a run may be finite or, at least from a theoretical point of view, infinite, for example when the execution of a reactive system like a web server is considered. However, when the underlying system runs, we can only observe a finite part of a potentially infinite run. We call the finite, observed part of the run an *execution*. Thus, in RV, we observe executions of an underlying system and want to assess the correctness of a high-level specification with respect to the run of the system. Now, one can come up with different semantics for LTL depending on how we understand executions and runs in detail.

In the basic case an execution and run coincide. Here, we think of a system that executed for a finite amount of time and the execution has terminated. This is the case for example when analyzing log files or when dealing with classical input/output oriented computations. If we want to analyze a correctness property for such an execution, an LTL semantics on finite words is most appropriate. A bunch of different variations of LTL semantics on finite words have been proposed in the literature, implicitly in Kamp's work [1] and more directly in Manna and Pnueli's work [2] or more recently by Eisner et al. in the context of LTL^+ and LTL^- [3] (see [4] for a comparison).

For reactive systems, however, the typical view on a computation is no longer the input/output behavior but the interaction of the system with its environment. In an ideal case such a run is infinite. For example, in the setting of a

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 82–87, 2013.
© Springer-Verlag Berlin Heidelberg 2013

web server we are not interested in a kind of final result of the server but deal with questions such as whether the web server follows the underlying protocol. Here, we consider an execution to be a *prefix of a potentially infinite run*. An appropriate semantics for such a setting with respect to RV was given in [5].

The idea is that a correctness property is evaluated on the current execution u with respect to all possible further extensions of the current execution. The rational is that it is fair to evaluate u with respect to all possible extensions as we know that u will extend somehow, but we do not really know how. If u together with all possible extensions satisfies the correctness property the runtime verification semantics of u with respect to the property is *true*. When all extensions of u violate the given correctness property the RV semantics of u with respect to the correctness property yields *false* while in all other cases the RV semantics yields ? meaning that no conclusive answer could be given. In other words, we give a three-valued semantics to LTL properties based on all possible extensions of the current execution.

In this paper we build on the previous idea, however, we extend the approach towards a *predictive* semantics by the following observations. Why do we check all possible extensions of the current execution u? Given a program $\mathcal{P}$ it seems to be more interesting to consider only the possible executions of the program $\mathcal{P}$. If we follow this idea we get *true* and *false* for the underlying property in more cases. In a sense, such a semantics would be more *precise*.

However, consider RV with such an idea right at the start for the empty word. We then have to check whether all the extensions of the empty word following the program $\mathcal{P}$ would satisfy our correctness property. Thus, we check whether all runs of our program $\mathcal{P}$ satisfy our correctness property and hence answer the model checking question. In consequence, we have to deal with the so-called state-space explosion also in RV.

The situation changes when we look at an abstraction $\hat{\mathcal{P}}$ of the underlying program $\mathcal{P}$ that has more runs than the original program $\mathcal{P}$. Then we can look at all extensions of an execution u with respect to the abstract system $\hat{\mathcal{P}}$. This may yield a more precise assessment than the original three-valued semantics but may be easier to check than model checking. Moreover depending on the level of abstraction one can focus more on the runtime verification aspects or more on the model checking ideas. In one of the extreme cases $\hat{\mathcal{P}}$ and $\mathcal{P}$ coincide and we solve the model checking problem while in the other extreme case $\hat{\mathcal{P}}$ just contains all possible executions over a given alphabet and we are in the traditional setting of three-valued LTL.

In this paper we further show that $\hat{\mathcal{P}}$ can actually be combined with a previous monitor synthesis procedure for three-valued LTL so that a monitor for the resulting predictive semantics is obtained. More precisely, the resulting monitor checks the semantics for a given execution u and a correctness property with respect to an abstraction $\hat{\mathcal{P}}$ of the underlying program $\mathcal{P}$.

In the remainder of this paper we make the previous ideas precise.

2 Preliminaries

For the remainder of this paper, let AP be a finite set of atomic propositions and $\Sigma = 2^{\mathrm{AP}}$ a finite alphabet. We write a_i for any single element of Σ. Finite traces over Σ are elements of Σ^*, and are usually denoted by $u, u', u_1, u_2, \ldots$, whereas infinite traces are elements of Σ^ω, usually denoted by $w, w', w_1, w_2, \ldots$.

The set of LTL formulae is inductively defined by the following grammar:

$$\varphi ::= \mathit{true} \mid p \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \; U \; \varphi \mid X\varphi \quad (p \in \mathrm{AP})$$

Let $i \in \mathbb{N}$ be a position. The semantics of LTL formulae is defined inductively over infinite sequences $w = a_0 a_1 \ldots \in \Sigma^\omega$ as follows: $w, i \models \mathit{true}$, $w, i \models \neg\varphi$ iff $w, i \not\models \varphi$, $w, i \models p$ iff $p \in a_i$, $w, i \models \varphi_1 \vee \varphi_2$ iff $w, i \models \varphi_1$ or $w, i \models \varphi_2$, $w, i \models \varphi_1 U \varphi_2$ iff there exists $k \geq i$ with $w, k \models \varphi_2$ and for all l with $i \leq l < k$, $w, l \models \varphi_1$, and $w, i \models X\varphi$ iff $w, i+1 \models \varphi$. Further, let $w \models \varphi$, iff $w, 0 \models \varphi$. For every LTL formula φ, its set of models, denoted by $\mathcal{L}(\varphi)$, is a regular set of infinite traces and can be described by a corresponding Büchi automaton.

A (nondeterministic) Büchi automaton (NBA) is a tuple $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$, where Σ is a finite alphabet, Q is a finite non-empty set of states, $Q_0 \subseteq Q$ is a set of initial states, $\delta : Q \times \Sigma \to 2^Q$ is the transition function, and $F \subseteq Q$ is a set of accepting states. We extend the transition function $\delta : Q \times \Sigma \to 2^Q$ to sets of states and (input) words as usual. A *run* of an automaton $\mathcal{A}$ on a word $w = a_1 \ldots \in \Sigma^\omega$ is a sequence of states and actions $\rho = q_0 a_1 q_1 \ldots$, where q_0 is an initial state of $\mathcal{A}$ and for all $i \in \mathbb{N}$ we have $q_{i+1} \in \delta(q_i, a_i)$. For a run ρ, let $\mathrm{Inf}(\rho)$ denote the states visited infinitely often. ρ is called *accepting* iff $\mathrm{Inf}(\rho) \cap F \neq \emptyset$.

A nondeterministic *finite automaton* (NFA) $\mathcal{A} = (\Sigma, Q, Q_0, \delta, F)$, where Σ, Q, Q_0, δ, and F are defined as for a Büchi automaton, operates on finite words. A *run* of $\mathcal{A}$ on a word $u = a_1 \ldots a_n \in \Sigma^*$ is a sequence of states and actions $\rho = q_0 a_1 q_1 \ldots q_n$, where q_0 is an initial state of $\mathcal{A}$ and for all $i \in \mathbb{N}$ we have $q_{i+1} \in \delta(q_i, a_i)$. The run is called accepting if $q_n \in F$. A NFA is called *deterministic* and denoted DFA, iff for all $q \in Q$, $a \in \Sigma$, $|\delta(q, a)| = 1$, and $|Q_0| = 1$.

As usual, the language accepted by an automaton (NBA/NFA/DFA), denoted by $\mathcal{L}(\mathcal{A})$, is given by its set of accepted words.

A *Moore machine* (also *finite-state machine*, FSM) is a finite state automaton enriched with output, formally denoted by a tuple $(\Sigma, Q, Q_0, \delta, \Delta, \lambda)$, where Σ, Q, $Q_0 \subseteq Q$, δ is as before and Δ is the output alphabet, $\lambda : Q \to \Delta$ the output function. As before, δ extends to the domain of words as expected. Moreover, we denote by λ also the function that applied to a word u yields the output in the state reached by u rather than the sequence of outputs.

In this paper, a (finite-state) *program* is given as a non-deterministic Büchi automaton for which all states are final. Runs of a program coincide with the runs of the Büchi automaton. The product of a program $\mathcal{P} = (\Sigma, Q, Q_0, \delta, Q)$ and an NBA $\mathcal{A} = (\Sigma, Q', Q'_0, \delta', F')$ is the NBA $\mathcal{B} = (\Sigma, Q \times Q', Q_0 \times Q'_0, \delta'', Q \times F')$ where $\delta''((q, q'), a) = \delta(q, a) \times \delta'(q', a)$, for all $q \in Q$, $q' \in Q'$ and $a \in \Sigma$. *Model checking* answers the question whether for a given program $\mathcal{P}$ and an LTL property φ, $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\varphi)$.

3 A Predictive Semantics for LTL

Let us recall our 3-valued semantics, denoted by LTL_3, over the set of truth values $\mathbb{B}_3 = \{\bot, ?, \top\}$ from [5]: Let $u \in \Sigma^*$ denote a finite trace. The *truth value* of a LTL_3 formula φ wrt. u, denoted by $[u \models \varphi]$, is an element of $\mathbb{B}_3$ defined by

$$[u \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega : u\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases}$$

In the previous definition, one might ask why not only consider extensions of u that altogether yield runs of an underlying program $\mathcal{P}$. Thus, one might be tempted to define a *predictive* semantics for a finite word u and a property φ with respect to a program $\mathcal{P}$, for example for the case $\top$, by $[u \models \varphi]_{\mathcal{P}} = \top$ iff $\forall \sigma \in \Sigma^\omega$ with $u\sigma \in \mathcal{P} : u\sigma \models \varphi$. However, for the empty word this means $[\epsilon \models \varphi]_{\mathcal{P}} = \top$ iff $\forall \sigma \in \Sigma^\omega$ with $\epsilon\sigma \in \mathcal{P} : \epsilon\sigma \models \varphi$ iff $\mathcal{L}(\mathcal{P}) \models \varphi$. Thus, any runtime verification approach following this idea implicitly answers the model checking question even before monitoring. Then runtime verification is at least as expensive as model checking.

We can follow a similar idea yet having control over the overall complexity using abstractions of the underlying program. An *over-abstraction* or and *over-approximation* of a program $\mathcal{P}$ is a program $\hat{\mathcal{P}}$ such that $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\hat{\mathcal{P}}) \subseteq \Sigma^\omega$.

Definition 1 (Predictive semantics of LTL). *Let $\mathcal{P}$ be a program and let $\hat{\mathcal{P}}$ be an over-approximation of $\mathcal{P}$. Let $u \in \Sigma^*$ denote a finite trace. The* truth value *of u and an LTL_3 formula φ wrt. $\hat{\mathcal{P}}$, denoted by $[u \models_{\hat{\mathcal{P}}} \varphi]$, is an element of $\mathbb{B}_3$ and defined as follows:*

$$[u \models_{\hat{\mathcal{P}}} \varphi] = \begin{cases} \top & \textit{if } \forall \sigma \in \Sigma^\omega \textit{ with } u\sigma \in \hat{\mathcal{P}} : u\sigma \models \varphi \\ \bot & \textit{if } \forall \sigma \in \Sigma^\omega \textit{ with } u\sigma \in \hat{\mathcal{P}} : u\sigma \not\models \varphi \\ ? & \textit{else} \end{cases}$$

We write $\text{LTL}_{\mathcal{P}}$ whenever we consider LTL formulas with a predictive semantics.

Remark 1. Let $\hat{\mathcal{P}}$ be an over-approximation of a program $\mathcal{P}$ over Σ, $u \in \Sigma^*$, and $\varphi \in \text{LTL}$.

- Model checking is more precise than RV with the predictive semantics:

$$\mathcal{P} \models \varphi \text{ implies } [u \models_{\hat{\mathcal{P}}} \varphi] \in \{\top, ?\}$$

- RV has no false negatives: $[u \models_{\hat{\mathcal{P}}} \varphi] = \bot$ implies $\mathcal{P} \not\models \varphi$
- The predictive semantics of an LTL formula is more precise than LTL_3:

$$\begin{aligned} [u \models \varphi] = \top \ \text{ implies } \ [u \models_{\hat{\mathcal{P}}} \varphi] = \top \\ [u \models \varphi] = \bot \ \text{ implies } \ [u \models_{\hat{\mathcal{P}}} \varphi] = \bot \end{aligned}$$

The reverse directions are in general not true. Thus, it is possible that a property is violated in the model checking sense but not spotted by RV with predictive semantics.

4 A Monitor Procedure for $\text{LTL}_\mathcal{P}$

Now, we develop an automata-based monitor procedure for $\text{LTL}_\mathcal{P}$. More specifically, for a given over-approximation $\hat{\mathcal{P}}$ of a program $\mathcal{P}$ and formula $\varphi \in \text{LTL}$, we construct a finite Moore machine, $\mathcal{B}^{\varphi}_{\hat{\mathcal{P}}}$ that reads finite traces $u \in \Sigma^*$ and outputs $[u \models_{\hat{\mathcal{P}}} \varphi] \in \mathbb{B}_3$.

For an NBA $\mathcal{A}$, we denote by $\mathcal{A}(q)$ the NBA that coincides with $\mathcal{A}$ except for Q_0, which is defined as $Q_0 = \{q\}$. Fix $\varphi \in \text{LTL}$ for the rest of this section and let $\mathcal{A}^\varphi$ denote the NBA, which accepts all models of φ, and let $\mathcal{A}^{\neg\varphi}$ denote the NBA, which accepts all counter examples of φ. The corresponding construction is standard [6].

Fix an over-approximation of a program $\hat{\mathcal{P}}$ for the remainder of this section and let $\mathcal{B}^\varphi$ and $\mathcal{B}^{\neg\varphi}$ be the product of the over-approximation with $\mathcal{A}^\varphi$ and $\mathcal{A}^{\neg\varphi}$, respectively, i.e., $\mathcal{B}^\varphi = \hat{\mathcal{P}} \times \mathcal{A}^\varphi$ and $\mathcal{B}^{\neg\varphi} = \hat{\mathcal{P}} \times \mathcal{A}^{\neg\varphi}$. For these automata, we easily observe that for $u \in \Sigma^*$ and $\delta(Q_0^\varphi, u) = \{q_1, \ldots, q_l\}$, we have $[u \models_{\hat{\mathcal{P}}} \varphi] \neq \bot$ iff $\exists q \in \{q_1, \ldots, q_l\}$ *such that* $\mathcal{L}(\mathcal{B}^\varphi(q)) \neq \emptyset$. Likewise, we have for the NBA $\mathcal{B}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, F^{\neg\varphi})$ as defined above, for $u \in \Sigma^*$, and $\delta(Q_0^{\neg\varphi}, u) = \{q_1, \ldots, q_l\}$ that $[u \models_{\hat{\mathcal{P}}} \varphi] \neq \top$ iff $\exists q \in \{q_1, \ldots, q_l\}$ *such that* $\mathcal{L}(\mathcal{B}^{\neg\varphi}(q)) \neq \emptyset$.

Following [5], for $\mathcal{B}^\varphi$ and $\mathcal{B}^{\neg\varphi}$, we now define a function $\mathcal{F}^\varphi : Q^\varphi \to \mathbb{B}$ respectively $\mathcal{F}^{\neg\varphi} : Q^{\neg\varphi} \to \mathbb{B}$ (where $\mathbb{B} = \{\top, \bot\}$), assigning to each state q whether the language of the respective automaton starting in state q is not empty. Thus, if $\mathcal{F}^\varphi(q) = \top$ holds, then the automaton $\mathcal{B}^\varphi$ starting at state q accepts a non-empty language and each finite prefix u leading to state q can be extended to a run of the over-approximation to satisfy φ.

Using $\mathcal{F}^\varphi$ and $\mathcal{F}^{\neg\varphi}$, we define two NFAs $\hat{\mathcal{B}}^\varphi = (\Sigma, Q^\varphi, Q_0^\varphi, \delta^\varphi, \hat{F}^\varphi)$ and $\hat{\mathcal{B}}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, Q_0^{\neg\varphi}, \delta^{\neg\varphi}, \hat{F}^{\neg\varphi})$ where $\hat{F}^\varphi = \{q \in Q^\varphi \mid \mathcal{F}^\varphi(q) = \top\}$ and $\hat{F}^{\neg\varphi} = \{q \in Q^{\neg\varphi} \mid \mathcal{F}^{\neg\varphi}(q) = \top\}$. Then, we have for all $u \in \Sigma^*$:

$$u \in \mathcal{L}(\hat{\mathcal{B}}^\varphi) \text{ iff } [u \models_{\hat{\mathcal{P}}} \varphi] \neq \bot \quad \text{and} \quad u \in \mathcal{L}(\hat{\mathcal{B}}^{\neg\varphi}) \text{ iff } [u \models_{\hat{\mathcal{P}}} \varphi] \neq \top$$

Hence, we can evaluate $[u \models_{\hat{\mathcal{P}}} \varphi]$ as follows: We have $[u \models_{\hat{\mathcal{P}}} \varphi] = \top$ if $u \notin \mathcal{L}(\hat{\mathcal{B}}^{\neg\varphi})$, $[u \models_{\hat{\mathcal{P}}} \varphi] = \bot$ if $u \notin \mathcal{L}(\hat{\mathcal{B}}^\varphi)$, and $[u \models_{\hat{\mathcal{P}}} \varphi] =?$ if $u \in \mathcal{L}(\hat{\mathcal{B}}^\varphi)$ and $u \in \mathcal{L}(\hat{\mathcal{B}}^{\neg\varphi})$.

As a final step, we now define a (deterministic) FSM $\mathcal{B}^\varphi$ that outputs for each finite string u and formula φ its associated predictive semantics wrt. the over-approximation $\hat{\mathcal{P}}$. Let $\tilde{\mathcal{B}}^\varphi$ and $\tilde{\mathcal{B}}^{\neg\varphi}$ be the deterministic versions of $\hat{\mathcal{B}}^\varphi$ and $\hat{\mathcal{B}}^{\neg\varphi}$, which can be computed in the standard manner by power-set construction. Now, we define the FSM in question as a product of $\tilde{\mathcal{B}}^\varphi$ and $\tilde{\mathcal{B}}^{\neg\varphi}$:

Definition 2 (Predictive Monitor $\mathcal{B}^\varphi$ for LTL-formula φ). *Let $\hat{\mathcal{P}}$ be an over-approximation of a program $\mathcal{P}$. Let $\tilde{\mathcal{B}}^\varphi = (\Sigma, Q^\varphi, \{q_0^\varphi\}, \delta^\varphi, \tilde{F}^\varphi)$ and $\tilde{\mathcal{B}}^{\neg\varphi} = (\Sigma, Q^{\neg\varphi}, \{q_0^{\neg\varphi}\}, \delta^{\neg\varphi}, \tilde{F}^{\neg\varphi})$ be the DFAs which correspond to the two NFAs $\hat{\mathcal{B}}^\varphi$ and $\hat{\mathcal{B}}^{\neg\varphi}$ as defined before. Then we define the* predictive monitor $\mathcal{B}^\varphi = \tilde{\mathcal{B}}^\varphi \times \tilde{\mathcal{B}}^{\neg\varphi}$ *for φ with respect to $\hat{\mathcal{P}}$ as the minimized version of the FSM $(\Sigma, \bar{Q}, \bar{q}_0, \bar{\delta}, \bar{\lambda})$, where $\bar{Q} = Q^\varphi \times Q^{\neg\varphi}$, $\bar{q}_0 = (q_0^\varphi, q_0^{\neg\varphi})$, $\bar{\delta}((q, q'), a) = (\delta^\varphi(q, a), \delta^{\neg\varphi}(q', a))$, and $\bar{\lambda} : \bar{Q} \to \mathbb{B}_3$ is defined by*

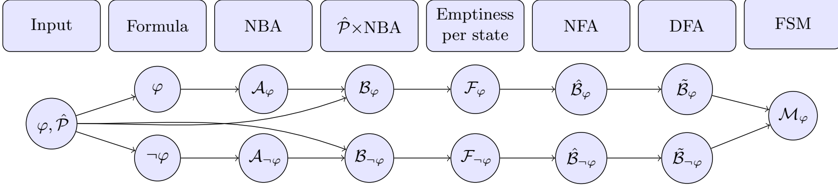


Fig. 1. The procedure for getting $[u \models_{\hat{\mathcal{P}}} \varphi]$ for a given φ and over-approximation $\hat{\mathcal{P}}$

$$\bar{\lambda}((q, q')) = \begin{cases} \top & \textit{if } q' \notin \tilde{F}^{\neg\varphi} \\ \bot & \textit{if } q \notin \tilde{F}^{\varphi} \\ ? & \textit{if } q \in \tilde{F}^{\varphi} \textit{ and } q' \in \tilde{F}^{\neg\varphi}. \end{cases}$$

We sum up our entire construction in Fig. 1 and conclude with the following correctness theorem.

Theorem 1. *Let $\hat{\mathcal{P}}$ be an over-approximation of a program $\mathcal{P}$, $\varphi \in LTL$, and let $\mathcal{B}^{\varphi} = (\Sigma, \bar{Q}, \bar{q}_0, \bar{\delta}, \bar{\lambda})$ be the corresponding monitor. Then, for all $u \in \Sigma^*$: $[u \models_{\hat{\mathcal{P}}} \varphi] = \bar{\lambda}(\bar{\delta}(\bar{q}_0, u))$.*

Complexity. Consider Fig. 1: Given φ, step 1 requires us to replicate φ and to negate it, i.e., it is linear in the original size. Step 2, the construction of the NBAs, causes an exponential blow-up in the worst-case. Step 3 multiplies the size of the automaton with the size of the over-approximation $\hat{\mathcal{P}}$. Steps 4 and 5, leading to $\hat{\mathcal{B}}^{\varphi}$ and $\hat{\mathcal{B}}^{\neg\varphi}$, do not change the size of the original automata. Then, computing the deterministic automata of step 6, might again require an exponential blow-up in size. In total the FSM of step 7 will have double exponential size with respect to $|\varphi|$ and single exponential size with respect to $\hat{\mathcal{P}}$. Note that steps 6 and 7 can easily be done on-the-fly.

References

1. Kamp, H.W.: Tense Logic and the Theory of Linear Order. PhD thesis, University of California, Los Angeles (1968)
2. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
3. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with Temporal Logic on Truncated Paths. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 27–39. Springer, Heidelberg (2003)
4. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. Journal of Logic and Computation 20(3), 651–674 (2010)
5. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM TOSEM 20(4) (July 2011)
6. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS 1986, pp. 332–345. IEEE Computer Society Press (1986)

Runtime Verification and Enforcement for Android Applications with RV-Droid*

Yliès Falcone, Sebastian Currea, and Mohamad Jaber

Laboratoire d'Informatique de Grenoble, UJF Université Grenoble 1, France
`FirstName.LastName@ujf-grenoble.fr`

Abstract. RV-Droid is an implemented framework dedicated to runtime verification (RV) and runtime enforcement (RE) of Android applications. RV-Droid consists of an Android application that interacts closely with a cloud. Running RV-Droid on their devices, users can select targeted Android applications from Google Play (or a dedicated repository) and a property. The cloud hosts third-party RV tools that are used to synthesize AspectJ aspects from the property. According to the chosen RV tool and the specification, some appropriate monitoring code, the original application and the instrumentation aspect are woven together. Weaving can occur either on the user's device or in the dedicated cloud. The woven application is then retrieved and executed on the user's device and the property is runtime verified. RV-Droid is generic and currently works with two existing runtime verification frameworks for (pure) Java programs: with Java-MOP and (partially) with RuleR. RV-Droid does not require any modification to the Android kernel and targeted applications can be retrieved off-the-shelf. We carried out several experiments that demonstrated the effectiveness of RV-Droid on monitoring (security) properties.

1 Introduction

Android [1] has risen as one of the most popular mobile operating systems. As the popularity of Android increases so is the need for validation techniques. A huge number of applications is available and an exhaustive/satisfactory validation process is missing. With this success has emerged bugged applications (because of complex life-cycle) and malwares that could seriously hinder devices' integrity and users' privacy [2].

Monitoring the behavior of Android applications appear as a candidate solution to circumvent these problems [3,4]. Runtime verification (RV) and enforcement (RE) are increasingly popular and effective dynamic validation techniques aiming at checking and ensuring the correct behavior of systems, respectively. These techniques consist in synthesizing a *monitor* from a high-level specification language, instrument the system and then integrate the monitor at relevant locations. At runtime, the monitor observes and possibly corrects the system's execution. In most of the runtime verification frameworks, instrumentation is automatic and relies on efficient and effective frameworks, e.g., aspect-oriented programming [5] and AspectJ (www.eclipse.org/aspectj/) its implementation for Java.

* This work was funded in part by the French-government Single Inter-Ministry Fund (FUI) through the IO32 project.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 88–95, 2013.
© Springer-Verlag Berlin Heidelberg 2013

However, for the Android platform such an effective instrumentation technique did not exist until quite recently [6]. Consequently, previously proposed approaches [3,4] had to modify, to different extents, the Android system to be able to log security-sensitive events. The downsides are limited portability between platforms and incompatibility with future releases of the operating system.

Contributions. We propose a framework that gets closer to the principles of runtime verification. Based on an in-house version of the AspectJ compiler [6] for the Android platform, we propose RV-Droid, a framework for "traditional" and user-friendly RV and RE that is compatible with state-of-the-art tools. RV-Droid is a stand alone Android application that does not require any modification to any part of Android devices. RV-Droid takes credit from Java-MOP [7] and RuleR [8] hence allowing efficient monitoring of various expressive specification formalisms. Because Android applications use a unified API where most of the sensitive operations go through a clearly identified set of methods, it becomes easy to write properties and monitors that work with any application. We propose several examples of such requirements. Finally, the architecture behind RV-Droid can be seen as a basis that can be further extended into more specialized implementations.

Paper Organization. Due to space reason, we do not provide an overview of Android, as literature abound on the subject and we believe that the architecture of RV-Droid and monitored properties are self-intelligible. Section 2 presents RV-Droid and its architecture. Experimentation and evaluation of monitoring properties with RV-Droid is done in Section 3. Related work is discussed in Section 4, while Section 5 draws some conclusions and present future developments.

2 An Overview of RV-Droid

RV-Droid is an Android application that interacts closely with a dedicated cloud (see Fig. 1). We provide a description of its features and some insights about its internal architecture. It represents approximately 4,200 LLOC (libraries and third-party tools excluded): 3,000 for the Android application and 1,200 for the cloud. RV-Droid allows user-friendly runtime verification of Android applications. RV-Droid takes as input an existing Android application with a property and then:

1. it synthesizes a monitor for the property, and
2. it integrates the monitor inside the application in a transparent way for the user.

RV-Droid works with Android Froyo 2.2 or higher, and does not require any modification to neither the Android kernel nor any part of the targeted device. Applications can be retrieved *off-the-shelf* from a personal (local or remote) repository or Google Play. Properties are available in a repository and are selected by the user according to an abstract description (informal requirement), the events involved in the property, and the formalization of the requirement. Monitor synthesis and runtime monitoring rely on third-party RV tools such as (for now) Java-MOP [7] and RuleR[1] [8]. Mainly, two operations are performed by RV-Droid: monitor synthesis and monitor integration. Monitor

[1] Courtesy of Howard Barringer and Klaus Havelund who offered a pre-release version.

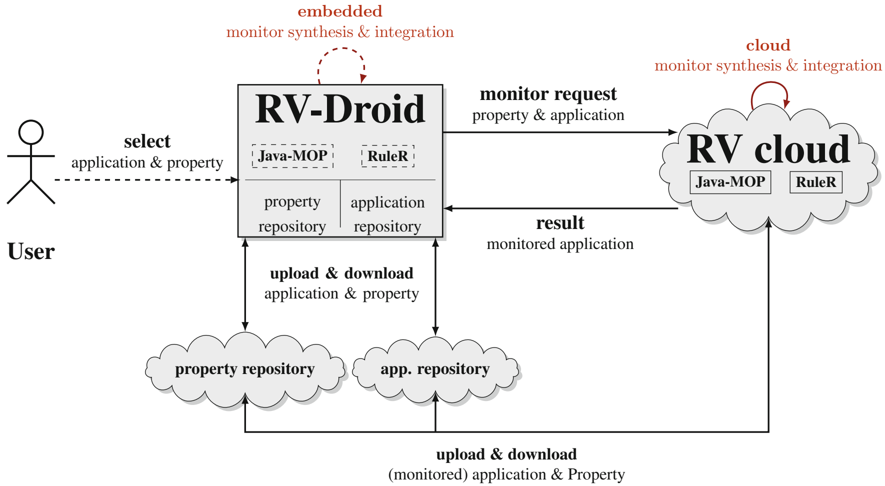


Fig. 1. RV-Droid context

synthesis consists in taking as input a property and generate some monitoring code, i.e., a decision procedure for this property. Monitor integration consists in instrumenting the target application to observe the relevant events that will trigger the monitoring code. For this purpose, RV-Droid relies on the aspect technology and an in-house version of the AspectJ compiler.

Monitor integration and aspect-oriented programming. RV-Droid supports two monitor integration (and aspect weaving) modes: embedded or in the cloud. Support of Aspect-Oriented Programming (AOP) on Android is ensured by Weave Droid, an in-house version of the AspectJ compiler [6]. One of the challenges faced by RV-Droid is to circumvent the current limitations to use AOP on Android applications that seriously hinder the mobility of the device and forbids "standard" runtime verification. For a description of the previously existing issues in using AOP on Android, the reader is referred to [6]. In a nutshell, the issues stem from the incompatibility of existing aspect-compilers with the Android `.apk` files (Android target binary file format). From an abstract point of view, our weaving process is achieved in several stages that mainly are: de-compile the application, weave the classes, convert the classes again in a format that Android can execute, and sign the application. These steps rely partly on third-party tools such as dex2jar (http://code.google.com/p/dex2jar/), Android dx tool (http://developer.android.com), and Zipsigner (http://code.google.com/p/zip-signer/).

Some code is shared between Weave Droid and RV-Droid, but Weave Droid has been re-implemented since then to make it more generic, and, to use indifferently aspects or specifications used by runtime verification tools.

Using third-party runtime verification tools. Based on the previously described process, monitor synthesis and monitor integration become possible using third-party runtime verification tools. Java-MOP provides facilities for monitor synthesis by generating aspects that query monitoring code in a library. RuleR does not provide aspect synthesis

facilities and one has to provide a specification together with the suitable aspect that will query the RuleR engine in a third-party library. We had to modify the third-party monitoring libraries because of some initial incompatibility with the Android system. From an abstract point of view, these libraries call some Java classes that are not provided by the Android runtime. Thus, we had to redirect these calls to a customized version of the Java runtime library. Note that the aforementioned modifications are transparent to the user who, in all cases, has only to download and install an Android application.

The remote processes are implemented as a web service queried by RV-Droid using the Simple Object Access Protocol (SOAP) and the web service client library kSoap (http://ksoap2.sourceforge.net/). The web service is deployed in the Glassfish application server. The two repositories execute on an SSH file transfer protocol server (SFTP).

3 Experimentation and Evaluation

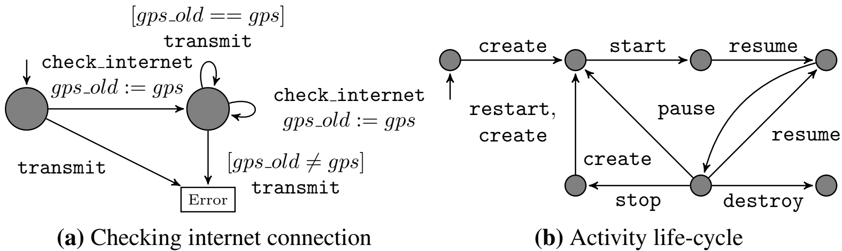
Verifying correct usage of Java data structures To evaluate RV-Droid, and assess the performance of state-of-the-art RV tools on recent Android devices, we carried out performance evaluation of Java-MOP monitors on three benchmarks with usual properties (available at Java-Mop's website). The first device is a Samsung Galaxy Tab 10.1, a tablet, with processor NVIDIA Tegra 2 dual core 1GHz and 1GB of RAM running on Honeycomb 3.1. The second device is a Samsung Galaxy Gio S5660, a mobile phone, with a 800 MHz processor and 278MB of RAM, running on Froyo 2.2.1. The considered benchmarks were Linpack (http://www.greenecomputing.com, B1), BenchmarkPi (http://androidbenchmark.com, B2), and DaCapo-xalan (http://dacapobench.org, B3). Linpack provides a general evaluation of the performance of the Dalvik virtual machine. BenchmarkPi provides an evaluation of the processor power of the device. DaCapo is the traditional benchmark used in RV that makes intensive use of Java data structures. Linpack and BenchmarkPi were taken off-the-shelf. However, using DaCapo required tweaking the original code, and, based on code analysis, we discovered that it is possible for only 6 of the 14 applications inside the benchmark.

Table 1. Benchmarks for properties over Java data structures – Galaxy Tab 10.1

Property	B1 (3.134) (s)		B2 (524.4) (ms)		B3 (489.3) (ms)	
	mon (s)	ovhd (%)	mon (ms)	ovhd (%)	mon (ms)	ovhd (%)
HasNext	3.439	9.732	552.1	5.282	547.8	11.956
UnsafeIterator	3.182	1.532	568.3	8.371	498.7	1.921
SafeEnum	3.189	1.755	591.3	12.757	512.3	4.701
SafeFileWriter	4.171	33.089	632.0	20.519	540.1	10.382
SafeSyncColl.	3.141	0.223	544.9	3.909	525.7	7.439
HashSet	3.142	0.255	574.8	9.611	549.8	12.365
UnsafeMapIterator	3.251	3.733	563.1	7.380	548	11.997
SafeSyncMap	3.152	0.574	540.4	3.051	553.7	13.162

Table 2. Benchmarks for properties over Java data structures – Galaxy Gio S5660

Property	B1 (19.65) (s)		B2 (1346) (ms)		B3 (2092) (ms)	
	mon (s)	ovhd (%)	mon (ms)	ovhd (%)	mon (ms)	ovhd (%)
HasNext	20.898	6.315	1567	16.359	3013	43.99
UnsafeIterator	21.115	7.419	2462	82.87	3121	49.15
SafeEnum	19.966	1.570	2476	83.894	2989	42.84
SafeFileWriter	20.532	4.454	2399	78.169	3569	70.56
SafeSyncColl.	20.623	4.939	2305	71.189	3035	45.04
HashSet	21.512	9.440	2292	70.2	2842	35.82
UnsafeMapIterator	20.775	5.689	2431	80.575	2895	38.35
SafeSyncMap	20.25	3.015	2416	79.46	3254	55.50



(a) Checking internet connection **(b)** Activity life-cycle

Fig. 2. Some properties inspired from the developer's guide

Performance results are shown in Tables 1 and 2 for the tablet and mobile phone, respectively. On the first line, for each benchmark, the execution time without monitor is indicated. For each property, the entries mon and ovhd indicate the average time for 10 executions of the monitored application and the induced overhead, respectively.

Verifying Android programming good practices. We monitored properties indicating whether Android's programming guidelines [1] are respected on some of the most popular games. Due to space reasons, an abstract monitor is given only for P1 and P2.

P1 *Before transmitting any data, it must be ensured that the device is connected to internet. And, it should be checked again each time the device is moved.* An abstract representation of the monitor used for this property is represented in Fig. 2a.

P2 *All methods involved in the activity life-cycle should be overridden.* To check whether the developer has followed this requirement, we can write an aspect that instruments those methods and tracks the (simplified) application life-cycle represented in Fig. 2b. If the method has been overridden by the developer, an event (corresponding to the method name) will be emitted by the monitored program. If, in a state, an unexpected event is emitted, it means that there is at least one method not overridden by the developer.

P3 *The device rotation facility should not be disabled.*
P4 *Only one dialogue window should be poped-up.*
P5 *In the restricted-memory mode, an application should start at most one service and end it, and not let the Dalvik virtual machine kill it.*

Preventing security issues through runtime enforcement. Among the 27 security findings discovered in [9], we wrote a monitor for 19 of them to either detect the vulnerability or even prevent it by disabling malicious method calls. The 8 remaining findings were related to too general concepts (e.g., "some developers toolkits probe for permissions through customized methods"). Being able to write a monitor to prevent security issues mostly depends on whether the referred sensitive data is retrieved through method calls. Method calls are caught by monitors and the data (passed as parameter) is then analyzed (e.g., an URI or string containing a premium-rate phone number).

4 Related Work

Both static and dynamic methods already exist to validate Android applications.

Static validation techniques. Verification of Android applications has been mostly investigated in relation with Android permissions [10]. At installation time, the user is asked whether the downloaded application is allowed to access security-relevant parts of the API. Stonaway [10] is a static analysis tool that determine whether applications disobey the principle of least privilege. Stonaway compares the permissions required by the calls made to Android's API to the permissions requested by the application. ComDroid [11] similarly analyses inter-application communication by examining emissions and receptions of intents (i.e., more or less messages) between applications to prevent information disclosure.

Dynamic analysis of Android applications. TaintDroid [3] is a framework for information-flow analysis of Android applications. It is based on information tainting and log collecting to determine whether sensitive information flows between applications.

Even closer to our work is a framework where a monitor runs on an Android device as a stand-alone application [4]. The "light" version of this approach modifies two files of the Android system to get notifications about security-sensitive events. This mild modification comes at the price of not being able to observe some low-level, potentially security-sensitive, operations. To circumvent this problem and get information about more events, the authors propose an in-house kernel module that has to load during boot. It is thus an *out-line* monitoring approach based on permission requests seen as events. Moreover, monitored properties are specified in an LTL variant and monitored using progression (i.e., formula rewriting).

Comparison with our approach. RV-Droid falls in the category of dynamic-analysis approaches. In contrast with existing approaches, RV-Droid is based on aspect-oriented programming for instrumentation. RV-Droid performs *in-line/on-line* monitoring, and, it features the following novelties and advantages. RV-Droid does not modify Android architecture, which, in our opinion, greatly favors usability, portability, and compatibility with next releases of Android. Monitors can be expressed using any event observable

through AspectJ. RV-Droid is not restricted to security properties, and, more general properties (e.g., correct implementation, debugging, statistics, etc) can be considered. Moreover, RV-Droid takes credit from Java-MOP and RuleR which are complementary in terms of expressiveness and efficiency and offer several input formalisms. While [4] is based on progression that can cause the size of the monitored formula to augment with the length of the trace, our monitors have been tested by running monitored applications for more than an hour without noticeable overhead. Also, RV-Droid permits runtime enforcement by e.g., disabling dangerous method calls. Finally, with a reasonable effort, RV-Droid can be extended to also support off-line monitoring.

5 Conclusion and Future Work and Developments

RV-Droid widens the interest of runtime verification to a large set of potential applications on mobile devices. Our framework is in the line of traditional of RV frameworks: (i) applications are seen as black boxes, (ii) applications are taken off-the-shelf, and (iii) the execution platform does not need to be instrumented. Our tool is still a prototype but will be released soon on Google code and Google Play.

We plan several conceptual extensions. Enforcement on method calls as presented in this paper can be extended to ensure the good usage of interfaces [12]. We also plan to design more elaborated aspects to be able to prevent intent-based attack surfaces [11] that requires to analyze the manifest data.

In the roadmap of RV-Droid, we plan to propose i) repositories of debugging and security monitors (aspects synthesized from properties), ii) integration with complementary RV tools (e.g., LARVA [13]), iii) customized application installer and donwloader where applications are automatically augmented with monitors after download, iv) repositories with sanitized (monitored) applications.

References

1. Google Inc.: Android developer site (2012), http://developer.android.com
2. Nouveau, T.: The Rise of Android Malware, TG Daily (November 2011)
3. Enck, W., Gilbert, P., Gon Chun, B., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.: TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Arpaci-Dusseau, R.H., Chen, B. (eds.) OSDI, pp. 393–407. USENIX Association (2010)
4. Bauer, A., Küster, J.-C., Vegliach, G.: Runtime Verification Meets Android Security. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 174–180. Springer, Heidelberg (2012)
5. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented Programming. In: Aksit, M., Auletta, V. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
6. Falcone, Y., Currea, S.: Weave Droid: Aspect-Oriented Programming on Android Devices – Fully Embedded or in the Cloud. In: The 27th IEEE/ACM International Conference on Automated Software Engineering (to appear, 2012), ASE 2012: preprint available online
7. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Rosu, G.: An overview of the MOP runtime verification framework. STTT 14, 249–289 (2012)
8. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: from Eagle to RuleR. J. Log. Comput. 20, 675–706 (2010)

9. Enck, W., Octeau, D., McDaniel, P., Chaudhuri, S.: A study of android application security. In: Proceedings of the 20th USENIX conference on Security, SEC 2011, p. 21. USENIX Association, Berkeley (2011)
10. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) ACM CCS, pp. 627–638. ACM (2011)
11. Chin, E., Felt, A.P., Greenwood, K., Wagner, D.: Analyzing inter-application communication in Android. In: MobiSys 2011, pp. 239–252. ACM (2011)
12. Hallé, S., Villemaire, R.: Browser-Based Enforcement of Interface Contracts in Web Applications with BeepBeep. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 648–653. Springer, Heidelberg (2009)
13. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time Java programs (tool paper). In: Hung, D.V., Krishnan, P. (eds.) SEFM, pp. 33–37. IEEE Computer Society (2009)

Temporal Monitors for TinyOS

Doina Bucur

Innovation Centre for Advanced Sensors and Sensor Systems (INCAS^3),
The Netherlands
doinabucur@incas3.eu

Abstract. Networked embedded systems generally have extremely low visibility of system faults. In this paper, we report on experimenting with online, node-local temporal monitors for networked embedded nodes running the TinyOS operating system and programmed in the nesC language. We instrument the original node software to signal asynchronous atomic events to a local nesC component running a runtime verification algorithm; this checks LTL properties automatically translated into deterministic state-machine monitors and encoded in nesC. We focus on quantifying the added (i) memory and (ii) computational overhead of this embedded checker and identify practical upper bounds with runtime checking on mainstream embedded platforms.

Keywords: Runtime verification, embedded software, LTL, automata, TinyOS, nesC.

1 Introduction

Embedded systems have become ubiquitous outside static, controlled industrial settings. They include both mobile embedded communication systems such as smartphones, and either static or mobile sensors and actuators operating in highly dynamic environments, such as *networks of wireless sensors* monitoring a natural or urban setting. Three features are common to such systems: (i) extreme *reactiveness*, in that system operation is driven by asynchronous external events—the inherent difficulty of writing correct software for asynchronous operation effectively increasing the probability of system failure; (ii) a tight bound on computational and memory *resources* of the hardware platform, necessary to ensure energy efficiency in operation; (iii) a need for *autonomous* operation. In this, we look at wireless sensor systems (WSNs), "the volatility of [which] is always in tension with ambitious application goals, including long-term deployments of several years, large scale networks of thousands of nodes, and highly reliable data delivery" [15]. We take the application case of the mainstream cross-platform operating system for wireless sensor nodes, TinyOS [14], its programming language, the event-based network embedded systems C (nesC) [12,11], and Telos [19]—an ultra-low power, highly ROM- and RAM-constrained wireless sensor module developed at the University of California, Berkeley.

Many failures of a WSN reportedly are rooted in faults at a single node. These faults are then rarely recovered from, due to the impracticability of reaching a

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 96–109, 2013.
© Springer-Verlag Berlin Heidelberg 2013

faulty node at its deployment location, and to the scarcity of mechanisms for self-diagnosis and repair of WSN nodes, e.g., generic built-in error checkers. In general, node-local failure modes include buffer and stack overflows, deadlocked, livelocked software and data races [16], no next-hop destination for data routing in a multihop network, unexpected outliers or gradient in sensed data, degradation of the battery [15], incorrect temporal use of the OS kernel's API [1], and any number of programmer-written invariants and qualitative or quantitative temporal properties.

We contribute an automata-theoretic runtime checker for node-local properties, running natively on embedded nodes running TinyOS. We support future-time LTL properties over system events. Relatively few efforts were made with regard to formal verification against failures in embedded software for WSNs. *Static checking* methods targeted at nesC or C for TinyOS, both for a single networked node [2] and a network [22,17,21,18,25] have met with mixed success in what regards the degree of *coverage* feasibly achieved, particularly with regard to the external *context* the node reacts to: the main difficulty of statically verifying a WSN is in modelling and checking a given node's software exhaustively against all possible networking and environmental settings.

On the other hand, *runtime checking* has become moderately accepted; naturally, checking at runtime precludes the need to exhaustively model the software's context. This is the case of SafeTinyOS [3], a node-local program-analysis-based checker for memory safety, now part of TinyOS; this has demonstrated that TinyOS kernel memory safety at runtime incurs a 13% ROM (code) and 5.2% CPU overhead. While other runtime-monitoring tools were also contributed, few fulfill, like our method, the crucial autonomy requirement that the runtime checking reside on the embedded nodes themselves. None of these checkers allow properties other than invariants (in the case of SafeTinyOS) and small handcrafted state machines as interface contracts for nesC software components in [1].

Standard temporal property languages such as LTL are suitable to express specifications for nesC software. For this, we construct the set of boolean atomic propositions to include (i) boolean conditions over program variables, e.g., `(data > 0x10)`, and (ii) program checkpoints, e.g., the entry point of the nesC function `Timer.fired()`. Using these additions to encode system events, memory safety properties may be written as LTL invariants, and interface contracts as LTL *Precedence* patterns. We allow all LTL properties which can be violated in finite time.

To generate runtime monitors for such temporal properties, we use a formal automata-theoretic algorithm to translate LTL into deterministic Büchi automata over finite words. The question whether a property currently holds thus translates into the question whether the current finite trace of execution events leads the monitor on an accepting transition; each verification step is done on-the-fly. We give a native nesC template implementation of a runtime monitor, together with an automatic translation from a deterministic automaton over finite words into this template. We then evaluate the overhead introduced by the monitoring and verification in terms of code (stored on-board in ROM

memory), volatile memory (RAM) and CPU load. We find that absolute overhead per monitor is negligible in what regards RAM and CPU load, but is up to 2.67KB of ROM for a basic LTL pattern, which amounts to 5.55% of the ROM integrated on a Telos revision B platform.

In what follows, Section 2 briefly overviews background matters in what regards both the theory of runtime checking and the TinyOS system; Sections 3 and 4 describe and evaluate, respectively, our monitoring framework, and Section 5 covers the related work and concludes.

2 Background

2.1 Theoretical Background

In the future fragment of propositional linear-time temporal logic (LTL), formulas are composed out of atomic propositions from a finite set AP, boolean logic operators, and the temporal operators **X** ("next time") and **U** ("until"); other temporal operators are defined in terms of these, e.g., **G** (the invariant "globally") and **F** ("eventually"). Whether a word over AP satisfies a LTL formula ϕ is defined inductively over the formula structure.

Given the set AP over which a temporal property is written, $\Sigma := 2^{AP}$ is a finite language over AP. A (nondeterministic) transition-based generalized Büchi automaton (TGBA) is a tuple $A := (\Sigma, Q, T, q_0, F)$, where Σ is a finite language as above, Q is a finite set of states, $q_0 \in Q$ is an initial state, F is a finite set of acceptance conditions, and T is the transition relation $T \subseteq Q \times \Sigma \times F \times 2^Q$, i.e., each transition is labelled and has attached acceptance conditions. General verification of LTL properties against a given system model is traditionally automata-theoretic: the LTL property is translated into a Büchi automaton A, such that a word σ over AP correctly described by ϕ allows a sequence of transitions such that each letter in σ matches a transition label, and the sequences satisfy each acceptance condition in A (infinitely many times, in the case of an infinite σ); i.e., a TGBA can be constructed for a given LTL property ϕ such that it accepts exactly the temporal words described in ϕ. Building a monitor for LTL only requires the particular case of *finite* words σ. The model-checking library SPOT [7,6][1] implements a number of LTL-to-TGBA translations, of which we used Couvreur/FM based on [4] and further formula and automaton simplifications; the resulting automaton compares well in terms of size with other translation algorithms, including that of LTL2BA [10].

Any nondeterministic automaton over finite words may be translated into an equivalent deterministic one; this *deterministic monitor* then has constant computational complexity, instead of linear in the size of the automaton. To generate a deterministic monitor from a nondeterministic Büchi automaton, [5] prunes all states which cannot start an accepting run, restricts the transition relation to the new set of states, and modifies all states to be accepting. SPOT

[1] An online engine is at `http://spot.lip6.fr/ltl2tgba.html` (last access: August 2012).

implements a similar pruning method, with resulting acceptance conditions on transitions.

2.2 Practical Background

A long line of low-power, integrated hardware platforms have been developed as WSN nodes in the past 15 years. The Telos [19] general design integrates computation, communication, storage, and sensing: an 8MHz, 16-bit MSP430-model microcontroller runs with 10KB of RAM, 48KB of ROM code storage, and, when active and with the radio on, draws 19mA of current from the battery pack. The equally mainstream alternative platform from the Mica family, MicaZ, runs on an 8-bit RISC-based ATmega128L microcontroller with 4KB of RAM. In our evaluation, we focus our experimentation on the popular Telos revision B platform (i.e., TelosB), with occasional comparisons to MicaZ.

Software for these platforms is constructed on the low-duty-cycle principle: the processor and radio transceiver are asleep for most of a duty interval, and periodically awaken for sensing, computation and communication duties. TinyOS [14] is a relatively young open-source operating system for WSN nodes, intended to allow such low-power duty cycling. TinyOS itself is written in a novel language, network embedded systems C (nesC), which extends C with code *components* (which may be either *configurations* or *modules*) wired through *interfaces*. Hardware Presentation Layer (HPL) components form the lowest level and interact directly with the hardware; higher-level system logic (e.g., device drivers or networking protocols) consists simply of one or more newly programmed components wired together with the relevant lower-level system components. A programmer's TinyOS application (i.e., the highest level of software abstraction) is programmed in the same way.

This component-based system design comes with advantages when implementing a monitor: (i) variables global to the entire OS are few, and (ii) a given, e.g., system peripheral or data structure pertaining to a network protocol is (each) de facto controlled from a single nesC system component. These facts simplify the task of code instrumentation for logging system events.

As usual for low-level networked systems, the associated programming languages and compilers support *asynchrony* natively: nesC interfaces are a bundle of asynchronous *events* and/or synchronous *commands*. For example, a hardware interrupt itself is an event to a low-level nesC component; this component's event handler may *signal* further events to higher-level components, triggering in effect an asynchronous event chain. All event handlers are non-interruptible and must be kept brief by the programmer, while any deferred computation in the form of synchronous TinyOS *tasks* should instead be *post*ed to the system's task queue.

We also capitalize on these OS features related to timeliness and asynchrony when designing our runtime logging and trace checking to be *real-time*: all system events of interest are notified through asynchronous nesC events, and every checking step is an `atomic` deferred task.

Fig. 1 depicts a schematic architecture of TinyOS, with our added runtime checker.

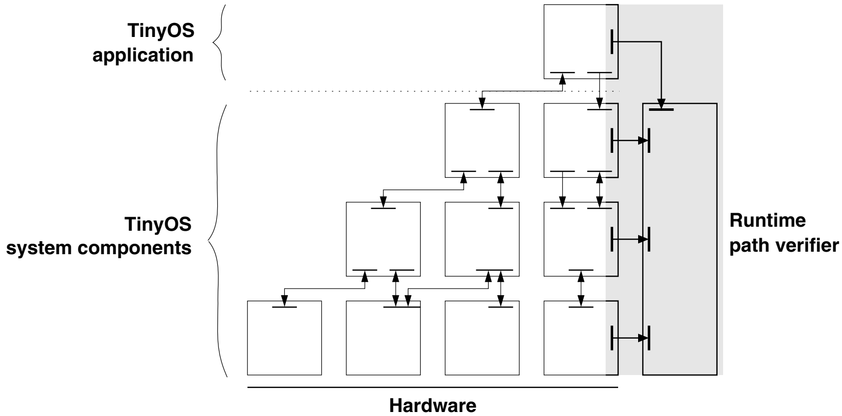


Fig. 1. An abstract schematic of TinyOS showing the system being implemented as nesC components wired through interfaces. A TinyOS application is effectively a set of nesC components, wired together in a graph-like structure. Our components and interfaces implementing runtime verification are emphasized (on the right). The runtime checker wires to any existing nesC system components of interest, and is thus notified of relevant system events.

3 TinyOS System Events, State, and Monitor Synthesis

System Events and Representing Atomic Propositions

As introduced in Section 1, system events may include, syntactically, boolean conditions over nesC variables and program checkpoints. Logging these system events is a relatively simple task, due to TinyOS's component-based design and the ensuing separation of concerns (described in Section 2.2). Furthermore, each nesC component of interest (such as a device driver or a protocol implementation) generally has a well-defined set of "important" variables and program checkpoints, such as the routing table (in the case of a network protocol) or the entry point of the event signalling the acquisition of new sensor data (in the case of a sensor's device driver). Thus, we found that exhaustively instrumenting a component once is sufficient for checking a large set of realistic temporal properties over the entire OS.

As testbed, we consider a representative sample TinyOS application from the existing codebase[2], Oscilloscope. This application is effectively a wiring together of a set of most-used TinyOS system components, including the drivers for the on-board sensors and the basic wireless networking stack. The top-level application logic then simply adds a duty cycle of 250ms, for which interval a timer is programmed to signal a periodic alarm event. In each of these cycles, a sensor is sampled with a call/signal command/event pair; when the number of successful

[2] The source repository for TinyOS is at `http://code.google.com/p/tinyos-main/` (last access: August 2012).

readings has filled a small buffer, the buffer is transmitted wirelessly to a fixed address.

We instrument for logging some of these crucial TinyOS system components, i.e., part of the implementation of the Hardware Presentation Layer (which effectively means that the state of any of the microcontroller's I/O peripherals is logged), and the high-level application logic itself. In what regards HPL, we consider as relevant system events the change in state of each bit in the microcontroller's peripheral registers; given the memory model of this platform, the peripherals on a TelosB platform form a set of 56 event types. For the high-level logic, we mark 10 conditions over variables and checkpoints. AP then equals this union set of system events.

Representing System State, and Matching Transitions

Only a subset of these atomic propositions in AP need monitoring for a given (set of) LTL properties ϕ. It is thus somewhat memory-inefficient to statically index each of the 66 event types in AP by a non-negative integer, and then statically encode the system state as a bit vector of 66, where each bit i is assigned the truth value of the corresponding $p_i \in AP$. We improve on this by instead dynamically generating the encoding for the system state per (set of) LTL properties, together with the monitor generation: for each atomic proposition p in ϕ, the lowest available index greater than zero is assigned. The trivial propositions *true* and *false* are treated in the same way. The resulting system state is thus a minimal bit vector of the size of ϕ (instead of the size of AP). Matching a transition in the automaton is then simply checking the required bits in the state bit vector.

Notifying System Events

Fig. 1 shows the components and interfaces added to the original TinyOS codebase for runtime checking. The *checker* itself is implemented as a new nesC component, automatically generated from a given LTL formula ϕ. This component `provides` a nesC event for any other logged component to `signal`; we list the header of this nesC checker in Fig. 2.

The *logging* of system events is then implemented as follows: each nesC component instrumented with logging simply *wires* to the checker (introduced above)

```
configuration PaxLTLC {
   provides async event void notify(uint16_t ap, bool val);
}
module PaxLTLP {
   // implements runtime checking for a given LTL property
}
```

Fig. 2. The header of our nesC runtime checker

through a nesC `notify` event, which is then `signalled` by the instrumented component at particular program checkpoints or variable writes.

We list the wiring for an HPL component in Fig. 3. We note that we did this additional nesC wiring and instrumentation manually. While a suitable tool implementing program analysis would automatize this process, we found that the instrumentation overhead is acceptably low: we only needed three new `signal` calls to log any change in the state of 56 microcontroller pins in the `HplMsp430GeneralIOP` modules in Fig. 3, due to the fact that all I/O ports are generated from a single "generic" module.

```
configuration HplMsp430GeneralIOC { [..]
}
implementation {
   [..]
   components PaxLTLC;
   [..]

   PaxLTLC.notify <- P10.pax_notify;
   PaxLTLC.notify <- P11.pax_notify;
   PaxLTLC.notify <- P12.pax_notify;
   // where P10, P11, etc are instantiations of HplMsp430GeneralIOP
   [..]
}

generic module HplMsp430GeneralIOP([..]) {
   [..]
   uses async event void notify(uint16_t ap, bool val);
}
implementation
{
   [..]
   async command void IO.set() {
      [..]
      signal pax_notify((PORTx*10+pin), TRUE);}
   async command void IO.clr() {
      [..]
      signal pax_notify((PORTx*10+pin), FALSE);}
   [..]
}
```

Fig. 3. Wiring and instrumentation added to the HPL components which control the microcontroller pins on the TelosB; the logging is done by signalling the `notify` event.

Monitor Synthesis

Take the LTL formula $\mathbf{F}\, r \rightarrow (\,!p\,\mathbf{U}\,(\,s \,|\, r\,))$ (the *Precedence* pattern with a *Before* scope from the KSU LTL pattern repository [8], meaning that atomic proposition s being true precedes p being true, and all before r is true). SPOT generates the equivalent deterministic monitor, as in Fig. 4.

As monitor encoding, we adopt a nesC version of the C++ `front_det_ifelse` encoding in [23], proven fairly efficient over experimentation for System C. The

encoding (of the Precedence property above, in Fig. 5) keeps track of the execution of the automaton with the integer variables `current` and `next`. At most one `if` branch on each of the two levels of conditionals is taken, for each notification event. The verification ends, and the property is proven violated, when at the arrival of an event notification, no transition is enabled. We generate these monitor implementations *automatically* from LTL properties, based on a nesC template.

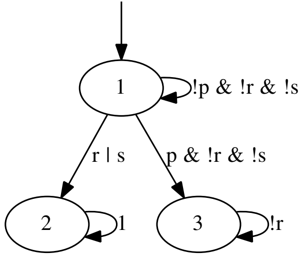


Fig. 4. Deterministic monitor generated for the specification $\mathbf{F}\, r \rightarrow (\,!p\, \mathbf{U}\, (\, s \,|\, r\,))$. State 1 is the initial state, and each transition is accepting.

```
implementation {
   async event void notify(uint16_t ap,
                           bool val) {
      // store (ap, val)
      if (!finished_checking)
         post step();
   }

   task void step() {
      atomic {
      // calculate new state with (ap, val)
      current_checking_steps++;
      current = next; next = -1;

      if (current == 1) {
         if ((call stateBV.get(r)) ||
            (call stateBV.get(s)))
            next = 2;
         else if ((call stateBV.get(p)) &&
                 !(call stateBV.get(r)) &&
                 !(call stateBV.get(s)))
            next = 3;
         else if (!(call stateBV.get(p)) &&
                  !(call stateBV.get(r)) &&
                  !(call stateBV.get(s)))
            next = 1;
      }
      else if (current == 2) {
         next = 2;
      }
      else if (current == 3) {
         if (!(call stateBV.get(r)))
            next = 3;
      }
      finished_checking = (next == -1);
}}}
```

Fig. 5. NesC monitor encoding for the monitor in Fig. 4. This forms most of the implementation for the `PaxLTLC` component introduced in Fig. 2.

4 Evaluation

Properties

Our test suite includes the basic KSU collection of future-time LTL property *patterns* and *scopes* [8]. The six property patterns are those of *Universality* (p is true), *Absence* (p is false), *Existence* (p eventually becomes true) and the related *Bounded Existence* (p becomes true at most twice), *Precedence* (s precedes p) and *Response* (after p, s eventually follows). To form more complex properties, any pattern is composed with any of five scopes: *Globally*, *Before r*, *After q*, *Between q and r* and *After q before r*. Thus, for the Oscilloscope application, any invariant over the values of sensed data is written as a Universality pattern in a Global scope (which we abbreviate by U-G); a specification requiring at least one successful sensing operation before a packet is sent may be written as an Existence-Before, abbreviated E-B.

Five of the ensuing thirty combined property types have trivial monitors, as they cannot be violated in finite time; these are the Existence-Globally (E-G), Existence-After (E-A), etc, and we omit them from the evaluation results. To these, we add two composite properties which are practically useful:

$$\bigvee_{i=1}^{k} \mathbf{G} p_i \quad \text{and a generic event-sequence chain} \quad p_1 \mathbf{U}(p_2 \mathbf{U}(\ldots \mathbf{U} p_k))$$

and also multiple basic monitors checking the same application.

Into these property types, we *randomly* input combinations of atomic propositions from our list of relevant system events. The resulting specifications are either violated or satisfied by the system software; our monitors will report whether the checking has finished (thus a violation was encountered) in real time, at the end of each monitoring step—variable `finished_checking` in the monitor implementation from Fig. 5 records the verification status, and is reported to the system users.

Metrics, Experimental Setup, and Results

As our runtime checker shares all on-board resources with the original application, we evaluate the monitor's performance in terms of computational (CPU overhead) and memory (RAM and ROM overhead) when running one or more checkers in the `PaxLTLC` component.

We measure the difference in the size of the binary application code between the monitored and the original, uninstrumented and unchecked, version used as a baseline (i.e., the ROM overhead). In Fig. 6, we evaluate the ROM overhead for example formulas of the basic LTL patterns composed with scopes, minus the cases with trivial monitors; patterns and scopes are shown as pairs X-Y of their abbreviations. In general, the difference we observed between the code size of a monitor for a property pattern, and that of the same monitor over different atomic propositions, is due only to the difference in the code instrumentation

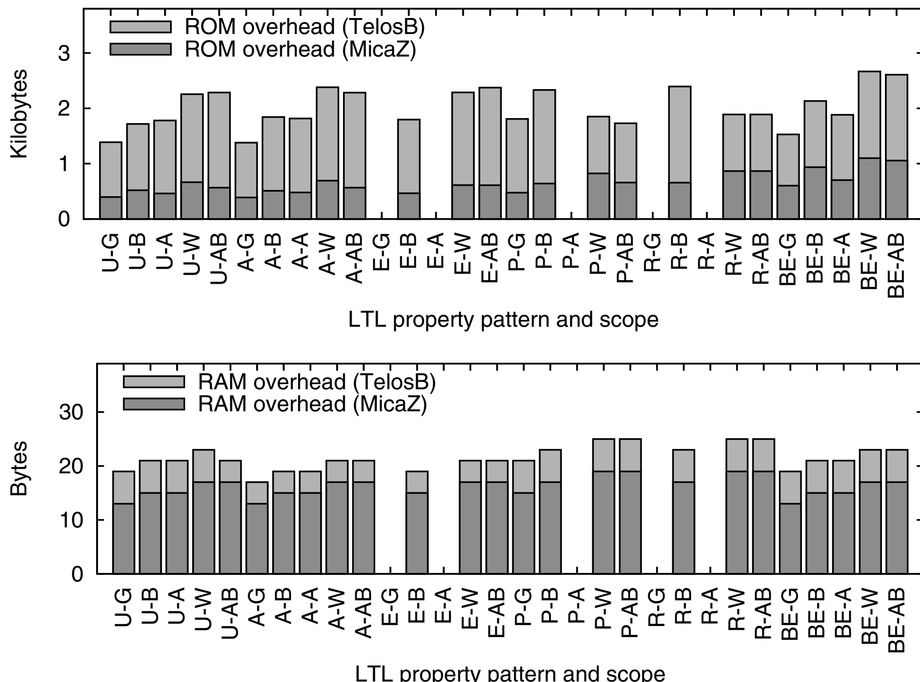


Fig. 6. Maximum code (ROM) and RAM overhead (in the latter case, we include data, uninitialized data, and maximum stack use) encountered by varying the subset of AP in each formula. Results as compiled by the platform compilers of the TelosB and MicaZ platforms out of the C software generated from nesC by the TinyOS compiler. The *original* application takes 19KB (ROM, TelosB), 13.7KB (ROM, MicaZ), 483B (RAM, TelosB), and 927B (RAM, MicaZ).

needed to notify of the occurrence of corresponding system events; the checker component is otherwise identical. In Fig. 6, we plot only the maximum values we encountered per property type.

A similar method is used to evaluate the RAM overhead, with the exception of the fact that only the `data` and `bss` (i.e., uninitialized data) binary segments are immediately readable from the compiled binary. TinyOS implements no dynamic memory allocation, which means that we only need to calculate the *maximum stack use* (an intrinsic part of the RAM metric). For this, we used actual execution runs of the monitored application in an experimental setup for TelosB; for MicaZ, we used `tos-ramsize`, a platform-specific static analysis tool for soundly assessing this metric, integrated in TinyOS. Fig. 6 also gives the final evaluation of the RAM overhead.

It is to note that for both ROM and RAM overheads, the microcontroller features (e.g., 16- versus 8-bit RISC) are crucial to the outcome; also, the RAM overhead is low in general, while ROM overhead averages around 2KB per LTL

property. To add some perspective, the maximum automaton size among these properties is 133 (where we take the size of an automaton to be the number of states times the number of transitions).

As for monitoring composite properties on a TelosB, the upper limit on a feasible k in $\bigvee_{i=1}^{k} \mathbf{G} p_i$ was $k = 5$; for $k = 6$, the code size (now including the original 19KB of code in ROM) exceeded the 48KB available on-board the platform; this automaton has 63 states and 665 transitions. For the event-sequence chain $p_1 \mathbf{U}(p_2 \mathbf{U}(\ldots \mathbf{U} p_k))$, we reached up to and including $k = 10$. For multiple basic monitors, the overhead is expectedly upper-bounded by the sum of the overheads in the single-monitor case.

To assess computational overhead, we run our monitored TinyOS application in a cycle-accurate emulator, MSPSim [9], tailored to the MSP430 microcontroller on the TelosB platform. Such an emulator executes all CPU operations with correct timing up to individual clock ticks, as these operations would also be executed on the given microcontroller architecture; an emulator is generally used for testing hardware or hardware-and-software designs. Cycle-accurate emulator executions of the same sensor software in the same context will always be consistent.

When running the application over the MSP430 emulator, we sample the CPU load and stack use every 20ms. We show the results of CPU overhead (i.e. the *difference* between the load of monitored runs and that of the original unmonitored application run) for a single monitor in Fig. 7. In order to capture the worst-case overhead, for this calculation we considered only those combinations of system events in the LTL formula, and only those intervals of the corresponding emulated executions with a still *active monitor*, i.e., before the monitor detected that the property at hand had been violated, and thus finished its checking procedure.

To note is that the CPU overhead follows the system's duty cycle, as expected, and that it only rises up to about 1% load increase for the 40ms after a new system event. This metric can be easily translated into actual mW of power consumed by the computation overhead, through integration. Also, since at each checking `step()` the monitor will also report whether a violation was

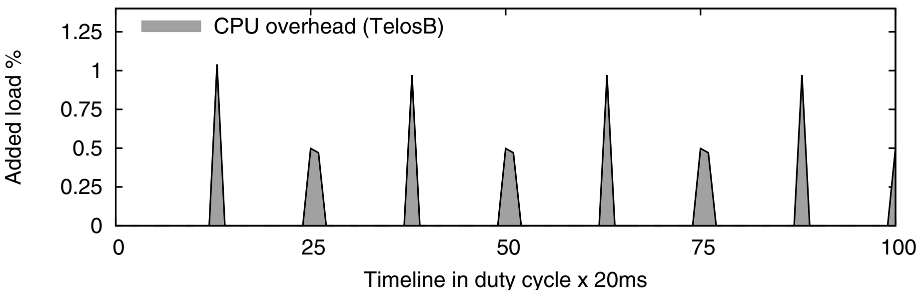


Fig. 7. CPU overhead (the average of 30 emulation runs) with a single running monitor

just encountered, this gives that a *user notification* can be triggered after these 40ms of processing the new event.

Finally, we gather all emulation runs and assess overhead by automaton size; we show results in Fig. 8.

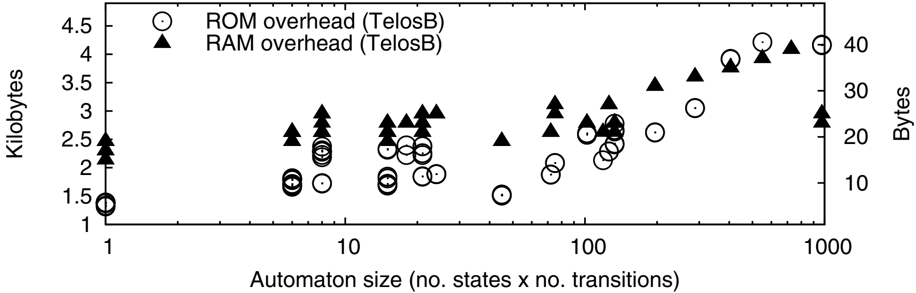


Fig. 8. Collected overhead results by automaton size, for a single monitor. ROM scale on left, RAM scale on right.

5 Related Work and Conclusions

Other than SafeTinyOS [3] and the Interface Contracts for TinyOS [1], NodeMD [16] also contributes a checker for deadlock, livelock (using checkpoints and expiration times), stack overflow, and assertions, based purely on program analysis and code instrumentation. None of these tools applies formal methods for runtime verification, and only support temporal properties expressed as small automata by the programmer, or quantitative temporal properties with heavy code instrumentation to check timing conditions. However, we share with these tools the online, embedded manner of running a checker. The same application area is covered in [20], for probabilistic properties and with the added feature of networking (and thus the added positive of supporting global network properties) and the negative in that this checker, while based on formal methods, runs externally to the WSN on a desktop, with 300k lines of code code added to the simulator. Temporal checking for C using aspect-based code instrumentation and state-machine monitors is covered in [13]. We found valuable insights in [23], an experimental study into the optimization of state-machine monitors for SystemC.

A few closing remarks are in order. Given the resulted feasibility of monitoring TinyOS execution traces on an embedded platform, as shown in this prototype, we may sustain the argument that, for such practical applications with (1) possibility for offline construction of the monitor, and (2) tightly bound online computational resources, alternative solutions could replace our regeneration of the TGBA per each new property with maintained databases of minimal automaton

translations for standard temporal properties, such as the Büchi store [24]. Also, further study is needed in what regards alternative solutions such as nondeterministic automata as checkers, and into quantitative temporal properties.

References

1. Archer, W., Levis, P., Regehr, J.: Interface contracts for TinyOS. In: Proceedings of the International Conference on Information Processing in Sensor Networks (IPSN), pp. 158–165. ACM (2007)
2. Bucur, D., Kwiatkowska, M.: On software verification for sensor nodes. Journal of Systems and Software 84(10), 1693–1707 (2011)
3. Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient memory safety for TinyOS. In: Proceedings of the Conference on Embedded Networked Sensor Systems (SenSys), pp. 205–218. ACM (2007)
4. Couvreur, J.-M.: On-the-Fly Verification of Linear Temporal Logic. In: Wing, J., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–711. Springer, Heidelberg (1999)
5. d'Amorim, M., Rosu, G.: Efficient Monitoring of ω-Languages. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 364–378. Springer, Heidelberg (2005)
6. Duret-Lutz, A.: LTL translation improvements in SPOT. In: Proceedings of the Fifth International Conference on Verification and Evaluation of Computer and Communication Systems, VECoS, pp. 72–83. British Computer Society (2011)
7. Duret-Lutz, A., Poitrenaud, D.: SPOT: An extensible model checking library using transition-based generalized Büchi automata. In: Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, MASCOTS, pp. 76–83. IEEE Computer Society, Washington, DC (2004)
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE, pp. 411–420. ACM, New York (1999)
9. Eriksson, J., Dunkels, A., Finne, N., Österlind, F., Voigt, T.: MSPsim – an Extensible Simulator for MSP430-equipped Sensor Boards. In: European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session, Delft, The Netherlands (2007)
10. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
11. Gay, D., Levis, P., Culler, D.: Software design patterns for TinyOS. In: Proceedings of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pp. 40–49. ACM (2005)
12. Gay, D., Levis, P., von Behren, R.: The nesC language: A holistic approach to networked embedded systems. In: ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 1–11. ACM (2003)
13. Havelund, K.: Runtime Verification of C Programs. In: Suzuki, K., Higashino, T., Ulrich, A., Hasegawa, T. (eds.) TestCom/FATES 2008. LNCS, vol. 5047, pp. 7–22. Springer, Heidelberg (2008)
14. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. SIGPLAN Not. 35(11), 93–104 (2000)

15. Jurdak, R., Wang, X.R., Obst, O., Valencia, P.: Wireless Sensor Network Anomalies: Diagnosis and Detection Strategies. In: Tolk, A., Jain, L.C. (eds.) Intelligence-Based Systems Engineering. ISRL, vol. 10, pp. 309–325. Springer, Heidelberg (2011)
16. Krunic, V., Trumpler, E., Han, R.: NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In: Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys), pp. 43–56. ACM (2007)
17. Li, P., Regehr, J.: T-Check: Bug finding for sensor networks. In: Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN), pp. 174–185. ACM (2010)
18. Mottola, L., Voigt, T., Österlind, F., Eriksson, J., Baresi, L., Ghezzi, C.: Anquiro: Enabling efficient static verification of sensor network software. In: Proceedings of Workshop on Software Engineering for Sensor Network Applications (SESENA) ICSE (2) (2010)
19. Polastre, J., Szewczyk, R., Culler, D.: Telos: Enabling Ultra-Low Power Wireless Research. In: Fourth International Symposium on Information Processing in Sensor Networks (IPSN), pp. 364–369 (April 2005)
20. Sammapun, U., Lee, I., Sokolsky, O., Regehr, J.: Statistical Runtime Checking of Probabilistic Properties. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 164–175. Springer, Heidelberg (2007)
21. Sasnauskas, R., Landsiedel, O., Alizai, M.H., Weise, C., Kowalewski, S., Wehrle, K.: KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In: Proceedings of the 9th International Conference on Information Processing in Sensor Networks (IPSN), pp. 186–196 (2010)
22. Sharma, O., Lewis, J., Miller, A., Dearle, A., Balasubramaniam, D., Morrison, R., Sventek, J.: Towards Verifying Correctness of Wireless Sensor Network Applications Using Insense and Spin. In: Păsăreanu, C.S. (ed.) SPIN 2009. LNCS, vol. 5578, pp. 223–240. Springer, Heidelberg (2009)
23. Tabakov, D., Vardi, M.Y.: Optimized Temporal Monitors for SystemC. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 436–451. Springer, Heidelberg (2010)
24. Tsay, Y.-K., Tsai, M.-H., Chang, J.-S., Chang, Y.-W.: Büchi Store: An Open Repository of Büchi Automata. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 262–266. Springer, Heidelberg (2011)
25. Zheng, M., Sun, J., Liu, Y., Dong, J.S., Gu, Y.: Towards a Model Checker for NesC and Wireless Sensor Networks. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 372–387. Springer, Heidelberg (2011)

Real-Time Runtime Verification on Chip

Thomas Reinbacher[1], Matthias Függer[1], and Jörg Brauer[2,3]

[1] Embedded Computing Systems Group, Vienna University of Technology, Austria
[2] Verified Systems International GmbH, Bremen, Germany
[3] Embedded Software Laboratory, RWTH Aachen University, Germany

Abstract. We present an algorithmic framework that allows on-line monitoring of past-time MTL specifications in a discrete time setting. The algorithms allow to be synthesized into efficient observer hardware blocks, which take advantage of the highly-parallel nature of hardware designs. For the time-bounded Since operator of past-time MTL we obtain a time complexity that is double logarithmic in the time it is executed at and the given time bounds of the Since operator. This result is promising with respect to a non-interfering monitoring approach that evaluates real-time specifications during the execution of the system-under-test. The resulting hardware blocks are reconfigurable and have applications in prototyping and runtime verification of embedded real-time systems.

1 Introduction

In runtime verification monitors are synthesized to automatically evaluate executions of a system-under-test (SUT), typically from a formal specification in a logic that is suitable to cover real-world specifications. In this paper, we present an algorithmic framework to apply runtime verification for safety-critical *embedded real-time systems*.

An approach classically taken in runtime verification is to instrument the code base, a technique that has proven feasible for a number of high-level languages [3] such as C, C++, and Java as well as for hardware description languages such as VHDL and Verilog. However, instrumentation is not directly applicable to embedded real-time systems, as these systems often include non-instrumentable hardware and mechanical parts; events from those might go unnoticed for an instrumenting runtime verification system. Even if instrumentation is applicable, the additional runtime overhead may alter timing behavior [7,11] as well as memory consumption and may make re-certification of the system onerous (e.g., systems certified after DO-178B). Pike et al. [23] recently studied the requirements of runtime verification for ultra-critical systems, identifying four major requirements: functionality (cannot change the target's behavior), certifiability (must avoid re-certification), timing (must not interfere with the target's timing), and swap (must not exhaust size, weight and power tolerances).

Requirements. We aim at runtime verification algorithms that can be directly realized in hardware, e.g., as the Runtime Verification Unit (RVU) in Fig. 1,

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 110–125, 2013.
© Springer-Verlag Berlin Heidelberg 2013

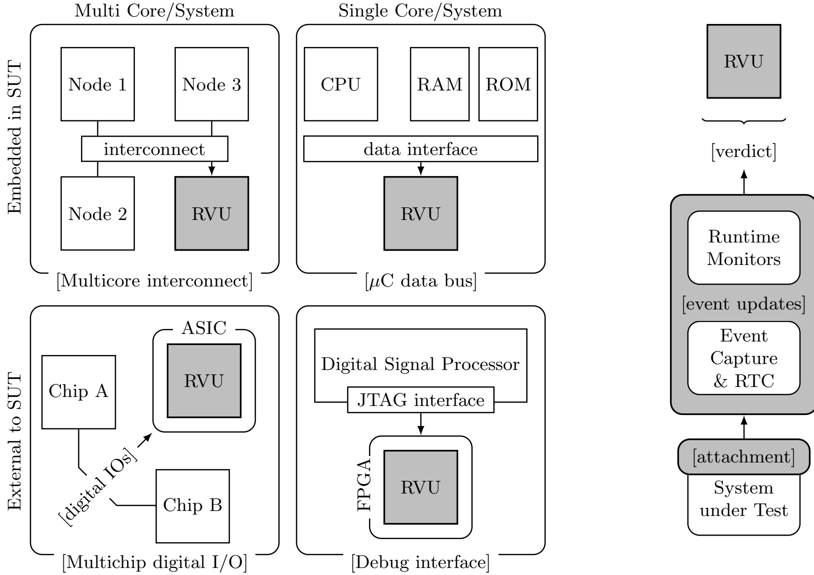


Fig. 1. Sample applications. Top left: RVU embedded into a network-on-chip, monitoring data exchanged among network nodes; Top right: RVU connected to the data interface of a microcontroller IP-core, monitoring microcontroller behavior (software); Bottom left: RVU connected to digital interconnects among chips on a printed circuit board (PCB), monitoring data exchanged through digital I/Os; Bottom right: RVU running on an FPGA attached to a debug interface (e.g. JTAG) of a digital signal processor, monitoring changes of accessible registers and diagnosis indicators.

that can be attached or embedded into various SUTs such as mixed hardware software designs in an FPGA or ASIC environment. For this application domain of runtime verification we arrive at the following requirements:

Stand-Alone. Runtime verification should not only be deployed during the testing phase of the product but also after the product is shipped. Thus, it should not depend on a powerful host computer that executes the monitor.

Non-Intrusive. The resulting monitor should be efficient enough to not alter the timing requirements of the system under test. From the algorithmic viewpoint synthesized monitors with a predictable and low execution time are required to statically determine upper bounds of the execution time of the monitor.

Timed. The behavior of a real-time system is defined by the sequence of data it produces complemented with its temporal behavior. To support correctness claims that involve timed properties, the system should support expressive logics to formalize not only functional but also real-time requirements.

Reconfigurable. For the testing phase, the system should be reconfigurable without requiring to re-synthesize the whole hardware system, which may take dozens of minutes to complete, for example when targeting an FPGA.

Contributions. Our work can be seen as a response to the above requirements: (i) We present on-line observer algorithms that allow to verify whether a past-time MTL (ptMTL) formula holds at (discrete) times $n \in \mathbb{N}_0$. The algorithms make use of basic operations only and are stated in a way that allows for a direct implementation in hardware. (ii) We formally prove the observers' correctness and derive bounds on their time complexity in terms of gate delays and their space complexity in terms of memory bits. For the observer algorithm of the ptMTL Since operator $\phi_1 S_J \phi_2$, where J is a nonempty interval, executed at time n, we obtain a time complexity of $\mathcal{O}(\log_2 \log_2 \max(J \cup \{n\}))$, only. The observer's space complexity is dominated by the size of a list it needs to maintain. We show that the list's space complexity is at most $2\lceil \log_2(n) \rceil \cdot (2\max(J) - \min(J) + 2)/(2 + \max(J) - \min(J))$. (iii) We finally discuss efficient realizations of the proposed observer algorithms in hardware.

2 Logics for Runtime Verification

We briefly summarize the temporal logics ptLTL and ptMTL which are used to specify properties in our framework. Both allow to specify safety, past-time properties over executions. For details, we refer the reader to [1, 10, 12, 17, 19].

2.1 Past Time LTL

A popular logic in runtime verification is the past-time fragment of LTL (ptLTL), mainly due to: (i) observer generation for ptLTL is straightforward [12, Sect. 5], and (ii) ptLTL can easily express typical specifications [18]. Although less expressive than LTL [10, Sect. 2.6], it can be exponentially more succinct [16]. With $\bullet$ in $\{\wedge, \vee, \rightarrow\}$ and Σ in the set AP of atomic propositions, a formula ξ is defined as:

$$\xi ::= true \mid false \mid \Sigma \mid \neg\xi \mid \xi \bullet \xi \mid \odot\xi \mid \diamondsuit\xi \mid \boxdot\xi \mid \xi\ S_s\ \xi \mid \xi\ S_w\ \xi$$

Hereby, $\odot\xi$ is the past-time analogue of next and referred to as *previously* ξ. Likewise, $\diamondsuit\xi$ is referred to as *eventually in the past* ξ and $\boxdot\xi$ as *always in the past*. The duals of the until and the weak-until operators are S_s and S_w, i.e., *strong since* and *weak since*, respectively. Similar as in LTL [13, Thm. 1], ptLTL can be reduced to the propositional operators plus two past-time operators [20], e.g., to $\odot$ and S_s. The satisfaction relation of a ptLTL specification can be defined as follows: Let $e = (s_t)_{t \geq 0}$ be an execution where s_t is a state of the system. An atomic proposition Σ holds on s_t iff $s_t \in \Sigma$. Denote by e^n, for $n \in \mathbb{N}_0$, the *execution prefix* $(s_t)_{0 \leq t \leq n}$. For a ptLTL formula ξ, time $n \in \mathbb{N}_0$ and execution e, we define ξ *holds at time* n *of execution* e, denoted $e^n \models \xi$, inductively as follows:

$e^n \models true$		is true,
$e^n \models false$		is false,
$e^n \models \Sigma$, where $\Sigma \in AP$	iff	Σ holds on s_n,
$e^n \models \neg\xi$	iff	$e^n \not\models \xi$,
$e^n \models \xi_1 \bullet \xi_2$	iff	$e^n \models \xi_1 \bullet e^n \models \xi_2$ with $\bullet \in \{\wedge, \vee, \rightarrow\}$,
$e^n \models \odot\xi$	iff	$e^{n-1} \models \xi$ if $n > 0$, and $e^0 \models \xi$ otherwise,
$e^n \models \xi_1 S_s \xi_2$	iff	$\exists j (0 \leq j \leq n): \big(e^j \models \xi_2 \ \wedge\ \forall k (j < k \leq n):\ e^k \models \xi_1\big)$.

The above syntax is augmented with monitoring operators [12, 17] to add syntactic sugar to ptLTL. Examples are the trigger conditions $\uparrow \xi$ and $\downarrow \xi$, where $\uparrow \xi$ stands for *start* ξ (i.e., ξ was false in the predecessor state s_{n-1} and is true in the current state s_n, equivalent to $\xi \wedge \neg \odot \xi$) and $\downarrow \xi$ for *end* ξ (ξ was true in s_{n-1} and is false in s_n, equivalent to $\neg\xi \wedge \odot\xi$). Checking whether a ptLTL formula holds at time $n \in \mathbb{N}_0$ in some execution $e = (s_t)_{t \geq 0}$ can be determined by evaluating only s_n and the results from s_{n-1} [12].

2.2 Past-Time MTL

Metric Temporal Logic (MTL) [1] extends LTL by replacing the qualitative temporal operators of LTL by quantitative operators that respect time bounds. Since we are interested in on-chip monitoring algorithms, progress of time is provided by the (possibly divided) chip's clock signal, resulting in a discrete time base $\mathbb{N}_0$. Time bounds of quantitative operators are given in form of intervals: For t, t' in $\mathbb{N}_0$, we write $[t, t']$ for the set $\{i \in \mathbb{N}_0 \mid t \leq i \leq t'\}$, and $[t, \infty]$ for the set $\{i \in \mathbb{N}_0 \mid t \leq i\}$. For example, MTL allows to express the property $\Box(\mathit{alarm} \rightarrow \Diamond_{[0,10]} \mathit{shutdown})$, which states that every alarm leads to a shutdown within $[0, 10]$ time units. Similar to ptLTL, a restriction of MTL to its past time fragment (ptMTL) is of interest. Formally, a ptMTL formula η is defined by:

$$\eta ::= \mathit{true} \mid \mathit{false} \mid \Sigma \mid \neg\eta \mid \eta \bullet \eta \mid \eta\ S_J\ \eta$$

where $\Sigma \in AP$, $\bullet \in \{\wedge, \vee, \rightarrow\}$, and $J = [t, t']$ for some $t, t' \in \mathbb{N}_0$. The semantics of true, false, Σ, $\neg\eta$, and $\eta \bullet \eta$ are as before. Recall that in ptLTL $\xi_1\ S\ \xi_2$ expresses ξ_2 *was true in the past and since then* ξ_1 *was true*. By way of contrast, satisfaction of $e^n \models \eta_1\ S_J\ \eta_2$ in ptMTL, does not only depend on the observation that $\eta_1\ S\ \eta_2$ holds in the current state, but also on (i) the time n of the current state and (ii) the times $i \in \mathbb{N}_0$ since when $\eta_1 S\ \eta_2$ was observed to be true: for at least one such i, $e^i \models \eta_2$, and $n - i \in J$ have to hold. Formally, we define:

$$e^n \models \eta_1\ S_J\ \eta_2 \text{ iff } \exists i (0 \leq i \leq n) : (n - i \in J \wedge e^i \models \eta_2 \wedge \forall j (i < j \leq n) : e^j \models \eta_1)$$

3 Observer Design for Real-Time Properties

Next, we discuss the formal design of on-line observer algorithms for ptMTL formulas in a discrete time model. The observer design extends work on observers for ptLTL [12] which have been built in hardware [22, 24].

3.1 Decomposing a Specification

In the following let $e = (s_t)_{t \geq 0}$ be an execution and ϕ a ptMTL formula. Further, let $J = [t, t']$, with $t, t' \in \mathbb{N}_0$, be a non-empty interval. An observer is an algorithm that, given input ϕ and execution e, at each time $n \in \mathbb{N}_0$, returns true if $e^n \models \phi$, and false otherwise. We define the return value of our observer algorithm with input ϕ at time n by structural induction on ptMTL formula ϕ:

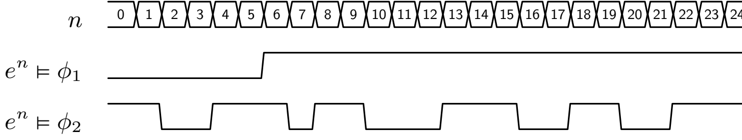


Fig. 2. Validity of $e^n \models \phi_1$ and $e^n \models \phi_2$ for prefix of execution e

(i) $\phi = true$ returns true and $\phi = false$ returns false.
(ii) $\phi = \Sigma$, where $\Sigma \in AP$ returns true if Σ holds on s_n, and false otherwise.
(iii) $\phi = \phi_1 \bullet \phi_2$ is true if $e^n \models \phi_1 \bullet e^n \models \phi_2$, where $\bullet \in \{\wedge, \vee, \rightarrow\}$, and false otherwise.
(iv) If ϕ is a ptLTL formula, we apply the algorithm in [12].
(v) For $\phi = \phi_1 S_J \phi_2$, we collect all times where ϕ_2 was true in the past and since then ϕ_1 remained true and store them in a list. At time n we check if there exists a time τ in the list such that $n - \tau \in J$. If such a τ exists we return true, and false otherwise.

Algorithms for cases (i) – (iii) are straightforward. For case (iv), we use the algorithm of Havelund and Roşu [12], for which a translation into hardware building blocks (specified in terms of VHDL) is known [24]. Finding an efficient algorithm to detect satisfaction of $e^n \models \phi_1 \; S_J \; \phi_2$ requires more sophisticated reasoning, and is the topic of the next sections.

Modularity. Once observer hardware implementations for all cases of subformulas of ϕ are known, one immediately obtains an observer algorithm for ϕ by connecting the (sub)modules' inputs and outputs according to the parse tree of ϕ.

Running Example. In the following, we frequently refer to the execution given in Fig. 2, which describes satisfaction of the two sub-formulas ϕ_1 and ϕ_2 over times $n \in [0, 24]$. We say *transition* ⌝⌞ *(resp.* ⌟⌜*) of* ϕ *occurs at time* n iff $e^n \models\uparrow \phi$ in case $n > 0$ and $e^0 \models \phi$ otherwise (resp. $e^n \models\downarrow \phi$ in case $n > 0$ and $e^0 \models \neg\phi$ otherwise). In the running example, transition ⌟⌜ of ϕ_1 occurs at time 6.

3.2 Observer for Since Operator Based on Rewriting

In a discrete time setting, there is an equivalent ptLTL formula for every ptMTL formula [19], directly leading to an observer algorithm for $\phi_1 S_{[a,b]} \phi_2$. With $\odot^i \phi$ being $\odot$ applied i times to ϕ, a straightforward generic translation is given by:

$$\begin{aligned} e^n \models \phi_1 \; S_{[a,b]} \; \phi_2 \;&\Leftrightarrow\; \exists i (a \leq i \leq b) : (\odot^i \phi_2 \wedge \odot^{i-1} \phi_1 \wedge \odot^{i-2} \phi_1 \wedge \cdots \wedge \phi_1) \\ &\Leftrightarrow\; \textstyle\bigvee_{i=a}^{b} \quad (\odot^i \phi_2 \wedge \bigwedge_{j=0}^{i-1} \odot^j \phi_1) \end{aligned}$$

Note that the translation can be optimized in a way that we group all pairs $\langle \odot^i \phi_2, \odot^j \phi_1 \rangle$ where the indices i and j coincide, and evaluate $\odot^i (\phi_1 \wedge \phi_2)$ in a single computation step. In a hardware implementation, one can make use of shift-registers to store the relevant part of the execution path wrt. the truth

values of ϕ_1 and ϕ_2. This results in an observer algorithm with time complexity $\Theta(b)$ and space complexity $2b-1$, rendering this solution infeasible for large b.

In the following we will present alternative observer algorithms with potentially less time and space complexity. We first consider the two variants $\boxdot_J\phi$ and $\diamonddot_J\phi$ of $\boxdot\phi$ and $\diamonddot\phi$, which are special cases of $\phi_1\, S_J\, \phi_2$.

3.3 The Invariant and Exists Previously Operators

We first discuss specializations of the common operators $\diamonddot_J$ (exists within interval J) and $\boxdot_J$ (invariant within interval J). Observe that with two equivalences [2]

$$\diamonddot_J\ \phi \equiv \mathsf{true}\ S_J\ \phi \qquad\qquad \boxdot_J\ \phi \equiv \neg \diamonddot_J \neg\phi \tag{1}$$

both operators can be expressed in terms of the S_J operator. From a practical point of view, two instances of the *exists within interval* and the *invariant within interval* operators, namely *invariant previously* ($\boxbox_\tau$) and *exists previously* ($\diamonddiamond_\tau$), where $\tau \in \mathbb{N}_0$, are valuable. They have the intended meaning *at least once in the past τ time units* ($\diamonddiamond_\tau$) respectively *invariant for the past τ time units* ($\boxbox_\tau$), and are defined by $\diamonddiamond_\tau \equiv \diamonddot_{[0,\tau]}$ respectively $\boxbox_\tau \equiv \boxdot_{[0,\tau]}$.

For example, $\uparrow(r_a \geq 10) \to \boxbox_{10}\,(r_b = 22)$ expresses that whenever $(r_a \geq 10)$ becomes true, $(r_b = 22)$ holds at all 10 previous time units. For both $\diamonddiamond_\tau$ and $\boxbox_\tau$ we present simplifications that yield space- and time-efficient observers.

Invariant Previously ($\boxbox_\tau\ \phi$) is transformed into $\neg(\mathsf{true}\ S_{[0,\tau]}\ \neg\phi)$ by (1). An observer for $\boxbox_\tau\ \phi$ requires a single register $m_{\boxbox_\tau\phi}$ with domain $\mathbb{N}_0 \cup \{\infty\}$. Initially $m_{\boxbox_\tau\phi} = \infty$. For the observer in Algorithm 1, define predicate $\mathsf{valid}^{\boxbox}(m,\tau,n)$ as:

$$\mathsf{valid}^{\boxbox}(m,\tau,n) \equiv (\max(n-\tau,0) \geq m)\ \ .$$

Intuitively, the predicate holds iff the latest $\lrcorner\!\ulcorner$ transition of ϕ occurred before time $n-\tau$ and no $\urcorner\!\llcorner$ transition of ϕ occurred since then until time n.

Theorem 1. *For all $n \in \mathbb{N}_0$, the observer in Algorithm 1 implements $e^n \models \boxbox_\tau\ \phi$.*

Proof. We first observe the equivalences

$$\begin{aligned} e^n \models \boxbox_\tau\ \phi &\Leftrightarrow e^n \models \neg(\mathsf{true}\ S_{[0,\tau]}\ \neg\phi) \Leftrightarrow \forall i(0 \leq i \leq n) : (n-i \in [0,\tau] \to e^i \models \phi) \\ &\Leftrightarrow \forall i(0 \leq i \leq n) : (i \in n-[0,\tau] \to e^i \models \phi) \Leftrightarrow \forall i : i \in [0,n] \cap [n-\tau,n] \to e^i \models \phi \\ &\Leftrightarrow \forall i : i \in [\max(0,n-\tau),n] \to e^i \models \phi\ \ . \end{aligned} \tag{2}$$

Algorithm 1. Observer for $\boxbox_\tau\phi$

1: At each time $n \in \mathbb{N}_0$:
2: **if** $\lrcorner\!\ulcorner$ transition of ϕ occurs at time n **then**
3: $\quad m_{\boxbox_\tau\phi} \leftarrow n$
4: **end if**
5: **if** $\urcorner\!\llcorner$ transition of ϕ occurs at time n **then**
6: $\quad m_{\boxbox_\tau\phi} \leftarrow \infty$
7: **end if**
8: return $\mathsf{valid}^{\boxbox}(m_{\boxbox_\tau\phi},\tau,n)$

Note that interval $[\max(0, n-\tau), n]$ is never empty. Thus Equation (2) holds iff a $_\!\ulcorner$ transition of ϕ occurred at a time at most $\max(0, n-\tau)$ and no $\urcorner\!_$ transition of ϕ occurred since then until time n. The theorem follows. □

Running Example. Consider $\psi \equiv (\uparrow \phi_1) \rightarrow (\boxbox_2 \phi_2)$ on the execution in Fig. 2. Initially, $m_{\boxbox_2\phi_2} = \infty$. At time 0, ϕ_2 holds and thus $m_{\boxbox_2\phi_2} = 0$. At time 2, we have $m_{\boxbox_2\phi_2} = \infty$ again. Since a $_\!\ulcorner$ transition of ϕ_2 occurs at time 4, $m_{\boxbox_2\phi_2} = 4$. At time 6, $(\uparrow \phi_1)$ becomes true and since $\mathsf{valid}^{\boxbox}(m_{\boxbox_2\phi_2}, 2, 6)$ is true, $e^6 \vDash \psi$ holds.

Exists Previously ($\Diamonddiamond_\tau\, \phi$). From the equivalence $\Diamonddiamond_\tau\, \phi \equiv \neg \boxbox_\tau \neg\phi$, we can immediately derive an observer for $\Diamonddiamond_\tau\, \phi$ from the observer for $\boxbox_\tau\, \phi$. The resulting algorithm can straightforwardly be implemented by checking for a $\urcorner\!_$ (resp. $_\!\ulcorner$) transition of ϕ instead of a $_\!\ulcorner$ (resp. $\urcorner\!_$) transition of $\neg\phi$ in line 2 (resp. line 5).

3.4 The Invariant and Exists within Interval Operators

We will next present observers for the more general operators *invariant within interval* ($\boxdot_J$) and *exists within interval* ($\Diamonddot_J$). Instead of a register, both observers require a list of time-stamp pairs. Clearly, an efficient implementation of this list is vital for an efficient observer. In the following, we present several techniques so as to keep this list succinct, whilst preserving validity of the observer. For a list l, we denote by $|l|$ its length, and by $l[k]$, where $k \in \mathbb{N}$, its kth element. We assume that elements are always appended to the tail of a list.

Invariant within Interval ($\boxdot_J\ \phi$) is transformed into $\neg(\mathsf{true}\ S_J\ \neg\phi)$ by (1). An observer for $\boxdot_J\ \phi$ requires a list $l_{\boxdot_J\phi}$ of elements from $(\mathbb{N}_0 \cup \{\infty\})^2$. For a pair of time-stamps $T \in (\mathbb{N}_0 \cup \{\infty\})^2$, we shortly write $T.\tau_s$ for its first component and $T.\tau_e$ for its second component. Initially, $l_{\boxdot_J\phi}$ is empty. For the observer in Algorithm 2, we define predicates $\mathsf{valid}^{\boxdot}(T, n, J)$ and $\mathsf{feasible}(T, n, J)$, with $T \in (\mathbb{N}_0 \cup \{\infty\})^2$, by:

$$\mathsf{valid}^{\boxdot}(T,n,J) \equiv (T.\tau_s \le \max(0, n-\max(J))) \wedge (T.\tau_e \ge n - \min(J))\ ,$$
$$\mathsf{feasible}(T,n,J) \equiv (T.\tau_e - T.\tau_s \ge \max(J) - \min(J)) \vee (T.\tau_s = 0 \wedge T.\tau_e \ge n - \min(J)).$$

Intuitively, tuples that satisfy $\mathsf{feasible}$ are those that characterize an interval where ϕ holds long enough to possibly satisfy $\boxdot_J\ \phi$.

Theorem 2. *For all $n \in \mathbb{N}_0$, the observer in Algorithm 2 implements $e^n \vDash \boxdot_J\ \phi$.*

Proof. First consider Algorithm 2 without the feasibility check, i.e., line 7 is replaced by "if true then". By analogous arguments as in the proof of Theorem 1, we obtain

$$\begin{aligned} e^n \vDash \boxdot_J\ \phi &\Leftrightarrow \forall i : i \in [0,n] \cap [n-\max(J), n-\min(J)] \rightarrow e^i \vDash \phi \\ &\Leftrightarrow \forall i : i \in [\max(0, n-\max(J)), n-\min(J)] \rightarrow e^i \vDash \phi\ . \end{aligned} \quad (3)$$

We next distinguish two cases for n: (i) $n < \min(J)$, and (ii) $n \ge \min(J)$:

Algorithm 2. Observer for $\boxdot_J \phi$

1: At each time $n \in \mathbb{N}_0$:
2: **if** $_\!\!\ulcorner$ transition of ϕ occurs at time n **then**
3: add (n, ∞) to $l_{\boxdot_J \phi}$
4: **end if**
5: **if** $\urcorner\!\!_$ transition of ϕ occurs at time n **and** $l_{\boxdot_J \phi}$ is non-empty **then**
6: remove tail element (τ_s, ∞) from $l_{\boxdot_J \phi}$
7: **if** $\mathsf{feasible}((\tau_s, n-1), n, J)$ **then**
8: add $(\tau_s, n-1)$ to $l_{\boxdot_J \phi}$
9: **end if**
10: **end if**
11: return $\bigvee_{k=1}^{|l_{\boxdot_J \phi}|} \mathsf{valid}^{\boxdot}(l_{\boxdot_J \phi}[k], n, J)$ in case $n \geq \min(J)$ and true otherwise

(i) In case $n < \min(J)$, interval $[\max(0, n - \max(J)), n - \min(J)]$ is empty, and thus (3) is true. Since Algorithm 2 returns true in this case, the theorem follows for Algorithm 2 without the feasibility check for case (i).

(ii) In case $n \geq \min(J)$, interval $[\max(0, n - \max(J)), n - \min(J)]$ is non-empty. Thus (3) holds iff a $_\!\!\ulcorner$ transition of ϕ occurred at a time at most $\max(0, n - \max(J))$ and no $\urcorner\!\!_$ transition of ϕ occurred since then until time $n - \min(J)$. Since this is the case iff there exists an element (τ_s, τ_e) in list $l_{\boxdot_J \phi}$ with $\tau_S \leq \max(0, n - \mathsf{max}(J))$ and $\tau_e \geq n - \mathsf{min}(J)$, i.e., $[\tau_s, \tau_e] \supseteq [\max(0, n - \max(J)), n - \min(J)]$, the theorem follows for Algorithm 2 without the feasibility check for case (ii).

It remains to show that the theorem holds for Algorithm 2 with original line 7. If we can show that from $\neg\mathsf{feasible}((\tau_s, \tau_e), n, J)$ follows $\neg\mathsf{valid}^{\boxdot}((\tau_s, \tau_e), n', J)$, for all times $n' \geq n$, we may safely remove tuple (τ_s, τ_e) from the algorithm's list without changing the algorithm's return value.

Assume that $\mathsf{valid}^{\boxdot}((\tau_s, \tau_e), n', J)$ holds, with $n' \geq n$. We distinguish two cases for n': (a) $n' < \max(J)$, and (b) $n' \geq \max(J)$:

(a) In case $n' < \max(J)$, it follows from $\mathsf{valid}^{\boxdot}((\tau_s, \tau_e), n', J)$ that $T.\tau_s = 0$ and $T.\tau_e \geq n' - \min(J) \geq n - \min(J)$. Thus $\mathsf{feasible}((\tau_s, \tau_e), n, J)$ holds.

(b) Otherwise $n' \geq \max(J)$, and it follows from $\mathsf{valid}^{\boxdot}((\tau_s, \tau_e), n', J)$ that $T.\tau_s \leq n' - \max(J)$ and $T.\tau_e \geq n' - \min(J)$. Thus $T.\tau_e - T.\tau_s \leq \max(J) - \min(J)$ and thereby $\mathsf{feasible}((\tau_s, \tau_e), n, J)$.

The theorem follows. $\square$

Running Example. Consider $\psi \equiv (\uparrow \phi_1) \rightarrow (\boxdot_{[3,4]} \phi_2)$ and execution e of Fig. 2. At time 0, the element $(0, \infty)$ is inserted into $l_{\boxdot_{[3,4]} \phi_2}$. The $\urcorner\!\!_$ transition of ϕ_2 at time 2 leads to $l_{\boxdot_{[3,4]} \phi_2} = ((0, 1))$, since $\mathsf{feasible}((0, 1), 2, [3, 4])$ holds. At time 4, another pair is added, resulting in $l_{\boxdot_{[3,4]} \phi_2} = ((0, 1), (4, \infty))$. Since at time 6 both $\mathsf{valid}^{\boxdot}(l_{\boxdot_{[3,4]} \phi_2}[1], 6, [3, 4])$ and $\mathsf{valid}^{\boxdot}(l_{\boxdot_{[3,4]} \phi_2}[2], 6, [3, 4])$ are false, $e^6 \not\models \psi$.

Exists within Interval ($\diamondsuit_J\ \phi$). From the equivalence $\diamondsuit_J \phi \equiv \neg \boxdot_J \neg\phi$, we can easily derive an observer for $\diamondsuit_J \phi$ from the observer for $\boxdot_J \phi$. As before, we obtain the observer by swapping $\urcorner\!\!_$ and $_\!\!\ulcorner$ transitions and negating the output.

Algorithm 3. Observer for $\phi_1 \, S_J \, \phi_2$

1: At each time $n \in \mathbb{N}_0$:
2: **if** ϕ_1 holds at time n **then**
3: **if** ⌝⌞ transition of ϕ_2 occurs at time n **then**
4: add (n, ∞) to l_S
5: **end if**
6: **if** ⌟⌜ transition of ϕ_2 occurs at time n **and** l_S is non-empty **then**
7: remove tail element (τ_s, ∞) from l_S
8: **if** $\mathsf{feasible}((\tau_s, n-1), n, J)$ **then**
9: add $(\tau_s, n-1)$ to l_S
10: **end if**
11: **end if**
12: **else**
13: **if** ϕ_2 holds at time n **then**
14: set $l_S = ((0, n-1))$ in case $n \neq 0$ and $l_S = ()$ otherwise
15: **else**
16: set $l_S = ((0, \infty))$
17: **end if**
18: **end if**
19: return $\neg \left(\bigvee_{k=1}^{|l_S|} \mathsf{valid}^{\boxdot}(l_S[k], n, J) \right)$ in case $n \geq \min(J)$ and false otherwise

3.5 The Since within Interval Operator

An observer for $\phi_1 S_J \, \phi_2$ is obtained from a $\diamondminus_J$ observer and additional logic to reset the observer's list. Let l_S be an initially empty list. The $\phi_1 S_J \, \phi_2$ observer is stated in Algorithm 3. In case ϕ_1 holds at time n, the observer executes the same code as a $\diamondminus_J \phi_2$ observer. In case ϕ_1 does not hold at time n, the list l_S is reset to contain only a single entry whose content depends on the validity of ϕ_2. We obtain:

Theorem 3. *For all $n \in \mathbb{N}_0$, Algorithm 3 implements $e^n \models \phi_1 \, S_J \, \phi_2$.*

3.6 Garbage Collection

Thus far, we did not consider housekeeping of either list so as to prevent unlimited growth. It is important to appreciate that each timed operator has a bounded time-horizon on which it depends. This horizon can be exploited to eliminate pairs T from Algorithm 2 and Algorithm 3's lists that can neither validate nor invalidate the specification. Our garbage collector works as specified: at any time $n \in \mathbb{N}_0$, we remove a tuple T from the list if the proposition

$$\mathsf{garbage}(T, n, J) \equiv T.\tau_e < n - \min(J)$$

holds. The main purpose of the garbage collector is to reduce the algorithms' space and time complexity: We will show that by removing tuples, garbage collection considerably reduces the algorithms' space complexity. Further, observe that direct implementations of line 11 of Algorithm 2 and line 19 of Algorithm 3

require searches through a list. We will show that with our garbage collector running in parallel, these lines reduce to checking the list's first element only.

In the following, we show the correctness of our garbage collection strategy for any of the proposed algorithms: We first show that if a tuple T is allowed to be removed by the garbage collector at time n, it cannot fulfill $\mathsf{valid}^{\boxdot}$ at that time or at any later time. It is thus safe to remove it from the list.

Lemma 1. *If* $\mathsf{garbage}(T, n, J)$*, then* $\neg\mathsf{valid}^{\boxdot}(T, n', J)$ *for all* $n \geq n'$.

Proof. Assume that $\mathsf{garbage}(T, n, J)$ holds. Then $T.\tau_e < n - \min(J) \leq n' - \min(J)$. Since $T.\tau_e \geq n' - \min(J)$ is necessary for $\mathsf{valid}^{\boxdot}(T, n', J)$, the lemma follows. □

We next show that always a prefix of a list is removed. This allows the garbage collector to evaluate $\mathsf{garbage}$ iteratively, starting from the head of the list.

Lemma 2. *Let* $l = (\dots, T, T', \dots)$ *be the list of any of the proposed observer algorithms at time* $n \in \mathbb{N}_0$*. If* $\mathsf{garbage}(T', n, J)$*, then* $\mathsf{garbage}(T, n, J)$*.*

Proof. Assume that $\mathsf{garbage}(T', n, J)$ holds. Then $T'.\tau_e < n - \min(J)$. By observing that all of the proposed algorithms ensure that $T.\tau_e \leq T'.\tau_e$ for successive list elements T and T', we obtain $T.\tau_e < n - \min(J)$, i.e., $\mathsf{garbage}(T, n, J)$ holds. The lemma follows. □

We next prove an upper bound on the length of Algorithm 2 and Algorithm 3's lists. We start by showing that there is a minimum distance between successive events in the algorithms' lists.

Lemma 3. *Let* $l = (\dots, T, T', \dots)$ *be the list of any of the proposed observer algorithms at time* $n \in \mathbb{N}_0$*. Then* $T.\tau_e + 2 \leq T'.\tau_s$*.*

Proof. Consider Algorithm 2. By the algorithm, tuple T must have been added by line 8. For line 8 to add $T = (T.\tau_s, n - 1)$, transition ⌉⎿ of ϕ must have occurred at time n. Thus the next tuple added to the list at a time $n' > n$ must have been of the form (n', ∞). Since, by the algorithm, then $T'.\tau_s \geq n'$ must hold, we further obtain $T'.\tau_s \geq (n - 1) + 2 = T.\tau_e + 2$. The lemma follows for Algorithm 2.

For Algorithm 3 the lemma follows by analogous arguments. □

Further the first element in the list that was not removed by the garbage collector cannot be of arbitrary age:

Lemma 4. *Consider a time-bounded formula* $\boxdot_J \phi$*,* $\diamondsuit_J \phi$*, or* $\phi_1 \ S_J \ \phi_2$*. Let* $l = (T, \dots)$ *be the list of the proposed respective observer algorithm at time* $n \in \mathbb{N}_0$*, after garbage collection has run at time* n*. Then* $T.\tau_e \geq n - \min(J)$*.*

Proof. It must hold that $\mathsf{garbage}(T, n, J)$ is false, since otherwise T would have been removed by the garbage collector. Thus $T.\tau_e \geq n - \min(J)$. □

Lemma 5. *Let* $l = (T^1, T^2, \dots, T^k, \dots)$ *be the list of any of the proposed observer algorithms at time* $n \in \mathbb{N}_0$*, after garbage collection has run at time* n*. Then* $T^k.\tau_e \geq n - \min(J) + (k - 1)(2 + \max(J) - \min(J))$*.*

We may now derive an upper bound on the list elements of any of our observer algorithms. Its proof is by Lemmas 4 and 5.

Lemma 6. *Consider a time-bounded formula $\boxdot_J \phi$, $\diamonddot_J \phi$, or $\phi_1 \; S_J \; \phi_2$. Let l be the list of the proposed respective observer algorithm at time $n \in \mathbb{N}_0$, after garbage collection has run at time n. Then l is of length at most* $(2\max(J) - \min(J) + 2)/(2 + \max(J) - \min(J))$.

3.7 Space and Time Complexity

We first give a bound on space complexity in terms of single-bit registers that are required by a hardware implementation of our observer algorithms. Clearly, the space complexity for an observer of ptMTL formula ϕ is the sum of the space complexity of its observers for all subformulas of ϕ, and its time complexity scales with the depth of the parse tree of ϕ. It is thus sufficient to state bounds for $\boxdot_J \phi$, $\diamonddot_J \phi$ and $\phi_1 \; S_J \; \phi_2$. In all these cases the respective observer algorithm's space complexity is dominated by the space complexity of the algorithm's list. Clearly the bit complexity of the τ_s or τ_e component of a tuple added by one of the proposed algorithms to its list before time $n \in \mathbb{N}_0$ is bounded by $\lceil \log_2(n) \rceil$. We thus obtain from Lemma 6 that for any of the time-bounded formulas $\boxdot_J \phi$, $\diamonddot_J \phi$, or $\phi_1 \; S_J \; \phi_2$, our proposed observer algorithms, if executed at time $n \in \mathbb{N}_0$, have to maintain a list of space complexity at most:

$$2\lceil \log_2(n) \rceil \cdot \frac{2\max(J) - \min(J) + 2}{2 + \max(J) - \min(J)} \quad . \tag{4}$$

Note that $\log_2(n)$ is small for realistic experimental setups. For example, allowing to store 52 bit per tuple component is sufficient to monitor executions that are sampled with a 1 GHz clock during a period of over 140 years.

An alternative to storing absolute times in the observer's list, is to adapt the observer algorithms in a way such that only relative times are stored. While this potentially reduces the bound of Equation (4) by substituting $\log_2(n)$ with $\log_2(\max(J))$, it requires updating of the list elements (as these then contain relative times) at every time $n \in \mathbb{N}_0$. Since this would require more complex hardware mechanism and result in a slower on-line algorithm, we decided not to follow this path in our hardware implementation.

We next show that garbage collection allows to reduce time complexity of the proposed observers. The time-determining part of Algorithms 2 and 3 is the evaluation of the predicate $\mathsf{valid}^{\boxdot}$ for all list elements in line 11 and line 19 respectively. However, garbage collection allows to only evaluate the predicate for the first element in the list, thus greatly improving time complexity of the proposed algorithms:

Lemma 7. *Let $l = (T, \dots, T', \dots)$ be the list of any of the observer algorithms at time $n \in \mathbb{N}_0$, after garbage collection has run at time n. Then $\neg \mathsf{valid}^{\boxdot}(T', n, J)$.*

Proof. Assume by means of contradiction that $\mathsf{valid}^{\boxdot}(T', n, J)$ holds. Then $T'.\tau_s \leq \max(0, n - \max(J)) \leq \max(0, n - \min(J))$. For both Algorithm 2 and 3

we observe that $T.\tau_e < T'.\tau_s$ has to hold. Thus $T.\tau_e < \max(0, n - \min(J))$. Since neither Algorithm 2 nor 3 adds tuples with a negative τ_s or τ_e component, we obtain that $T.\tau_e < n - \min(J)$ has to hold; and by that $\mathsf{garbage}(T, n, J)$ holds. A contradiction to the fact that garbage collection has been run at time n: it would have removed tuple T in that case. The lemma follows. □

Since further there exist circuits that perform an addition of two integers of bit complexity $w \in \mathbb{N}$ within time $\mathcal{O}(\log_2(w))$ [15], since evaluating the $\mathsf{valid}^{\boxdot}(T, n, J)$ and $\mathsf{garbage}^{\boxdot}(T, n, J)$ predicates at time $n \in \mathbb{N}_0$ requires addition of integers of bit complexity at most $\max(\log_2(n), \log_2(J))$, and since garbage collection removes at most one tuple at each time, we arrive at an asymptotic time complexity of

$$\mathcal{O}\Big(\log_2 \log_2 \max(J \cup \{n\})\Big) \ ,$$

for any of the observers $\boxdot_J \phi$, $\diamondsuit_J \phi$, and $\phi_1 \ S_J \ \phi_2$ executed at time n.

3.8 Hardware Realization

Fig. 3 shows a hardware realization of an observer for ptMTL formula η. The control logic manages a pool of $\boxdot_\tau \phi$, $\boxdot_J \phi$ and $\phi_1 S_J \phi_2$ hardware observers connected according to η's parse tree. This allows to change η (within resource limitations) without re-synthesizing the hardware observer, which could take tens of minutes for FPGA designs, allowing applications in prototyping and testing.

Time-stamps are internally stored in registers of width $w = \lceil log_2(n_{max}) \rceil + 2$, to allow overflow when performing arithmetical operations and to indicate ∞. For a list $l_{\boxdot_J \phi}$ we turn to block RAMs (abundant on FPGAs) which are organized as ring buffers. Each ring buffer has a garbage collector (GC). To insert a time-stamp pair that satisfies $\mathsf{feasible}((\tau_s, n-1), n, J))$, the write pointer is incremented to point to the next free element in the ring buffer. The GC then adjusts the read pointer to indicate the latest element wrt. n and J that is *recent enough.* In a fresh cycle (indicated by a changed time-stamp n), the GC loads (τ_s, τ_e) using the read pointer, which is incremented iff $\mathsf{garbage}((\tau_s, \tau_e), n, J)$ holds.

Subtraction and relational operators as required by the predicates feasible, garbage and valid can be built around adders. For example (left part of Fig. 3), $\mathsf{valid}^{\boxdot}((\tau_e, \tau_s), n, J)$ is implemented using five w-bit adders: one for $q := n - \min(J)$, one for $r := T.\tau_e \geq q$, one to calculate $p := n - \max(J)$ and two to calculate $t := T.\tau_s \leq \max(p, 0)$. Finally, the unit outputs the verdict $t \wedge r$, where t and r are calculated in parallel.

We successfully implemented a first prototype in an FPGA design, demonstrating the feasibility of our approach.

4 Related Work

Thati and Roşu [25] presented an on-line observer for MTL formulas ψ. Their idea is to reduce the problem of deciding whether $e^n \models \psi$ to deciding several instances

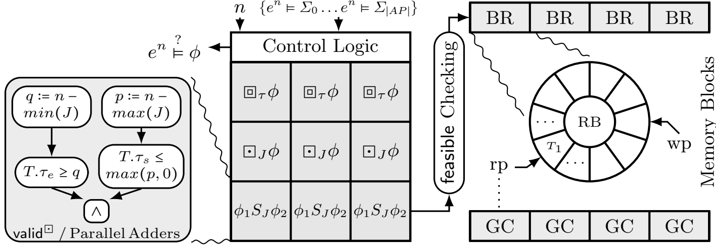


Fig. 3. Hardware overview (GC = Garbage Collector, BR = Block Ram)

of $e^{n'} \models \psi'$, where ψ' is a subformula of ψ and $n' \leq n$. Thereby for each subformula $\phi_1 S_{[a,b]} \phi_2$ of ψ, the formulas $\phi_1 S_{[a-1,b-1]} \phi_2$, $\phi_1 S_{[a-2,b-2]} \phi_2, \ldots, \phi_1 S_{[0,b-a]} \phi_2, \ldots,$ $\phi_1 S_{[0,0]} \phi_2$ are defined to be subformulas of ψ. For example, in case $\psi \equiv \phi_1 S_{[1,3]} \phi_2$, where ϕ_1 and ϕ_2 are atomic propositions, the reduced formulas of ψ are ϕ_1, ϕ_2 as well as $\phi_1 S_{[0,2]} \phi_2$, $\phi_1 S_{[0,1]} \phi_2$, and $\phi_1 S_{[0,0]} \phi_2$. Denoting by m the number of subformulas an MTL formula ψ is reduced to, the space complexity of their observer is within $\mathcal{O}(m2^m)$ and its time complexity is within $\mathcal{O}(m^3 2^{3m})$ for each time n in $\mathbb{N}_0$, the observer is executed at. For the special cases of $\psi \equiv \phi_1 S_J \phi_2$, the observer still requires a memory of at least $2m \geq 2\max(J)$ bit. While this bound is incomparable in general to our bound, for large values of $\max(J)$ we immediately obtain that our solution has less memory complexity. For example for $\phi_1 S_{[5,1500]} \phi_2$ the solution in [25] requires at least 3000 bit of memory, whereas our observer requires 208 bit, assuming time-stamps of 52 bit.

Maler et al. [19] presented an on-line observer algorithm for $\phi_1\ S_J\ \phi_2$ that is based on having active counters for each event of ϕ_2. Divakaran et al. [9] improved the number of counters of bit width $\log \max(J)$ to $2\lceil \min(J)/(\max(J) - \min(J)) \rceil + 1$ and showed its optimality for an observer realized as a timed transition system. While their space complexity is incomparable to ours in general, their solution is very resource intensive for a hardware realization: While we may store list values in cheap RAM blocks, their solution requires to store the current counter values in registers, since their values are incremented at every time step.

The (discrete time) point-based observer algorithm of Basin et al. [4] for formula $\phi_1\ S_J\ \phi_2$ runs in time $\mathcal{O}(\log \max(J \cup \{n\}))$ if executed at time $n \in \mathbb{N}_0$. Their algorithm, however, requires memory in the order of $\max(J)$. They further presented an interval-based observer algorithm for $\phi_1\ S_J\ \phi_2$ with space complexity comparable to our solution. However, the algorithm is clearly motivated with a software implementation in mind, whereas we aim at efficient (highly parallel) circuit implementations. For example, for an arbitrary ptMTL formula ϕ, our time-complexity bounds scale with the depth of the parse tree of ϕ, whereas the bounds in [4] scale with the fourth power of the number of nodes in the parse tree of ϕ. Further, a direct implementation of their algorithm would require considerable hardware overhead, as it makes use of doubly-linked lists to store and manipulate timestamps. In comparison, our ring buffer design can easily be mapped to block RAM elements that are abundant on modern day FPGAs.

The Property Specification Language (PSL) gained momentum in industrial-strength hardware verification. PSL is based on LTL, augmented with regular expressions, thus, we will not compare our work to PSL monitoring algorithms but rather to the hardware architecture of the resulting checkers. Translations from PSL into hardware either follow the modular or the automata based synthesis.

In the modular approach (for example as in [5, 8, 21]), sub-circuits for each operator are built and inter-connected according to the parse tree of the PSL expression being monitored.

Borrione et al. [5] describe a method of translating properties of the PSL foundation layer into predefined primitive components. A component is a hardware unit, consisting of a checking window and an evaluation block. Shift register chains are used to trigger the execution of the evaluation block. Blocks representing a timed operator need to individually count the elapsed time-stamps, while we tailored our algorithms to work with cheap RAM blocks.

In the automata based approach (for example as in [6]), (in general non-deterministic) state machines are synthesized that act as monitor for a PSL property. To avoid a blowup of the automaton size, additional counters are used. However, this is only feasible if the output language natively supports non-deterministic finite automata (NFA); unfortunately, major hardware descriptions languages (e.g., Verilog and VHDL) do not. Consequently, monitors need to be converted to a deterministic finite automaton (DFA) first, which, in the worst case, yields an exponential blowup of the resulting DFA in the size of the NFA [14].

5 Conclusion

We presented an on-line algorithm to check a ptMTL formula ϕ on executions with discrete time domain. At the algorithm's heart is an observer algorithm for the time-bounded Since operator and the special cases *exists/invariant previously* and *within interval*. The presented algorithms have been proven correct and bounds on their time and space complexity have been proven. The promising complexity results are mainly due to the concept of garbage collection and feasibility testing that automatically drop events that can neither validate nor invalidate formula ϕ: The proposed garbage collector does not only keep the algorithm's list size bounded, but also allows to evaluate the list's first element only to determine validity of ϕ. We further discussed a reconfigurable hardware realization of our observer algorithm that provides sufficient flexibility to allow for changes of ϕ without necessarily re-synthesizing the hardware observer.

The predictable and low resource requirements of the presented hardware solution together with its reconfigurability allow for diagnosis of real-time systems during mission time. We plan to work on an extensive experimental evaluation of our approach and to extend our work to (bounded) future time MTL.

Acknowledgement. The work has been supported by the Austrian Research Agency FFG under grant 825891 (CevTes) and (partially) supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF). The authors want to thank Dejan Nickovic for fruitful discussions.

References

1. Alur, R., Henzinger, T.A.: Real-time Logics: Complexity and Expressiveness. In: LICS, pp. 390–401. IEEE (1990)
2. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press (2008)
3. Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.): RV 2010. LNCS, vol. 6418. Springer, Heidelberg (2010)
4. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for Monitoring Real-Time Properties. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 260–275. Springer, Heidelberg (2012)
5. Borrione, D., Liu, M., Morin-Allory, K., Ostier, P., Fesquet, L.: On-line assertion-based verification with proven correct monitors. In: ICICT, pp. 125–143 (2005)
6. Boulé, M., Zilic, Z.: Automata-based assertion-checker synthesis of PSL properties. ACM Transactions on Design Automation of Electronic Systems 13(1) (2008)
7. Colombo, C., Pace, G.J., Schneider, G.: Safe Runtime Verification of Real-Time Properties. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 103–117. Springer, Heidelberg (2009)
8. Das, S., Mohanty, R., Dasgupta, P., Chakrabarti, P.: Synthesis of system verilog assertions. In: DATE, vol. 2, pp. 1–6 (2006)
9. Divakaran, S., D'Souza, D., Mohan, M.R.: Conflict-tolerant real-time specifications in metric temporal logic. In: TIME, pp. 35–42 (2010)
10. Emerson, E.A.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, vol. B, pp. 995–1072. MIT Press (1990)
11. Fischmeister, S., Lam, P.: Time-aware instrumentation of embedded software. IEEE Transactions on Industrial Informatics 6(4), 652–663 (2010)
12. Havelund, K., Roşu, G.: An overview of the runtime verification tool Java PathExplorer. Formal Methods in System Design 24(2), 189–215 (2004)
13. Havelund, K., Roşu, G.: Efficient monitoring of safety properties. International Journal on Software Tools for Technology Transfer 6, 158–173 (2004)
14. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley Longman Publishing Co., Inc. (2006)
15. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. IEEE Trans. Comput. 22(8), 786–793 (1973)
16. Latvala, T., Biere, A., Heljanko, K., Junttila, T.A.: Simple Is Better: Efficient Bounded Model Checking for Past LTL. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 380–395. Springer, Heidelberg (2005)
17. Lee, I., Kannan, S., Kim, M., Sokolsky, O., Viswanathan, M.: Runtime assurance based on formal specifications. In: PDPTA, pp. 279–287 (1999)
18. Lichtenstein, O., Pnueli, A., Zuck, L.: The Glory of the Past. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985)
19. Maler, O., Nickovic, D., Pnueli, A.: Real Time Temporal Logic: Past, Present, Future. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 2–16. Springer, Heidelberg (2005)

20. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent systems. Springer (1992)
21. Morin-Allory, K., Borrione, D.: Proven correct monitors from PSL specifications. In: DATE, pp. 1–6 (2006)
22. Pellizzoni, R., Meredith, P., Caccamo, M., Rosu, G.: Hardware runtime monitoring for dependable COTS-based real-time embedded systems. In: RTSS, pp. 481–491 (2008)
23. Pike, L., Niller, S., Wegmann, N.: Runtime Verification for Ultra-Critical Systems. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 310–324. Springer, Heidelberg (2012)
24. Reinbacher, T., Brauer, J., Horauer, M., Steininger, A., Kowalewski, S.: Past Time LTL Runtime Verification for Microcontroller Binary Code. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 37–51. Springer, Heidelberg (2011)
25. Thati, P., Roşu, G.: Monitoring Algorithms for Metric Temporal Logic specifications. ENTCS 113, 145–162 (2005)

BabelTrace: A Collection of Transducers for Trace Validation

Aouatef Mrad[1], Samatar Ahmed[1], Sylvain Hallé[1,⋆], and Éric Beaudet[2]

[1] Université du Québec à Chicoutimi, Canada
shalle@acm.org, aouatef.mrad@uqac.ca
[2] Novum Solutions, Canada
eric.beaudet@novumsolutions.ca

Abstract. Trace validation is the process of evaluating a formal specification over a log of recorded events produced by a system. In addition to the numerous techniques developed specifically for that purpose over the years, a range of peripheral tools such as model checkers and database engines can also be used as *bona fide* trace validators. We present an evolvable software environment that provides a large set of transducers which, when piped into an appropriate chain, can transform a trace and a formal specification into a suitable input problem for a variety of available tools.

1 Introduction

The analysis of event traces with respect to formal specifications has found an increasing number of uses in the recent past. Various kinds of data collected at runtime during the execution of a software system can be stored into a form of log for *a posteriori* processing. Example scenarios range from business process contract compliance [11] to test trace analysis [4] and intrusion detection [9].

Unfortunately, the tools developed for a particular use case can seldom be re-used in a different context. The traces are generally represented using a slightly different structure across tools, and the specification languages they use are not equivalently expressive, creating a barrier to the use of trace analysis software across application domains. Moreover, there exist tools and techniques designed to solve other problems, such as model checkers and database engines, that could also be used as trace validators, provided that traces and specifications be converted into corresponding objects of their application domain.

The present paper builds upon that observation and presents an evolvable software environment called BabelTrace[1] that provides a large set of converters between trace validation problems expressed in various languages. BabelTrace does not merely operate at the syntactical level: it provides transducers that can reduce the problem of trace validation to particular cases of model checking, XML and SQL query evaluation.

⋆ This work was funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under an Engage Grant.

[1] http://www.github.com/sylvainhalle/TraceAdapter

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 126–130, 2013.
© Springer-Verlag Berlin Heidelberg 2013

2 A Taxonomy of Trace Validation Problems

BabelTrace is organized into a library of available software and scripts, each of which is formally classified according to two characteristics: the trace format read by the tool and the specification language it uses.

Input Trace Type. The first aspect of the classification is the format used to represent event traces, and in particular the restrictions on the way data parameters can be used. BabelTrace currently supports four categories of events.

The first category is called *multi-valued*, and corresponds to the most generic form of trace, where each event is taken as a function $\nu : P \to 2^V$ that associates to parameters $p \in P$ a *set* of values taken from some domain V. Events of such kind are generally represented using XML or an equivalent notation, and can be found, e.g. in web services, for example, in the Amazon E-Commerce Service studied in [7].

A first restriction is to consider *single-valued* events, where each parameter can occur at most once in each event; formally, this imposes that $|\nu(p)| \leq 1$ for every $p \in P$. The Extensible Event Stream (XES) format [5] is one possible notation for events of such kind. Examples of single-valued messages in the literature include system events produced by spacecraft hardware [4]. A further restriction produces events with a *fixed schema*, where all parameters occur exactly once and differ only in the values they carry. Examples of such events include web server logs [9] and data extracted from a (single) relational database table.

Finally, the last category of traces considered is made of *atomic* symbols taken from a known alphabet. Examples include sequences of method calls on program objects (discarding their arguments) [2] and some business process logs where events correspond to enabled activities [11].

Specification Language. The second aspect of the classification is the specification language used to formalize the properties to verify on traces. To this end, a number of formal languages have been suggested over the years, and used as the input specification for various tools. BabelTrace currently supports MFOTL, a monadic first-order temporal logic where each temporal operator is given a "time window" over which it operates [1]. It also includes LTL-FO$^+$, a first-order extension of Linear Temporal Logic with quantification over event parameters [7] where ground terms are equalities between bound variables. It obviously supports classical LTL without quantifiers, where ground terms are event predicates over the current message. Finally, to accomodate atomic traces, a simpler variant of LTL with alphabet symbols as ground terms is also handled.

Each tool is thus associated to a trace/language pair $\langle \mathscr{T}, \mathscr{L} \rangle$ according to the categories described above. Equiped with this classification, a trace validation problem instance is a pair (τ, φ), where τ is a trace expressed in format $\mathscr{T}$ and φ is a specification expressed in language $\mathscr{L}$. The process of validating a trace consists in computing whether φ is satisfied on τ, which we write $\tau \models_{\langle \mathscr{T}, \mathscr{L} \rangle} \varphi$.

3 The Many Ways to Compute $\tau \models \varphi$

BabelTrace does not perform trace validation by itself; it rather calls external tools that can execute that computation.

Basic Workflow. To this end, BabelTrace offers a collection of classes that can process an input file and retrieve a tool's result.

1. *Readers* take a concrete input trace file and convert it into an internal data structure in either of the four trace types $\mathscr{T}$ described above. Input formats currently supported are XML, XES and database tables exported as comma-separated values (CSV) files.
2. *Translators* take a problem instance (φ, τ) and automatically generate the input file(s) in the syntax required to perform trace validation on some target tool.
3. *Launchers* then build the command line syntax to start the execution of some tool on a given input, parse the tool's command-line output and convert it to decide whether $\tau \models_{\langle \mathscr{L}, \mathscr{T} \rangle} \varphi$.

This architecture allows the processing of event traces to be done in a uniform manner, independent of the particular syntax of a given tool or the file format used to represent the traces one wishes to validate.

Figure 1 shows the trace-language pairs (rectangles) and tools (ovals) currently supported by BabelTrace. A curved dashed line denotes an implemented translator from a pair $\langle \mathscr{L}, \mathscr{T} \rangle$ to the syntax of a specific tool. One can see that among software specifically designed for trace validation, BabelTrace supports BeepBeep [7], MonPoly [1], and a rewriting-based algorithm for atomic LTL developed for Maude [10].

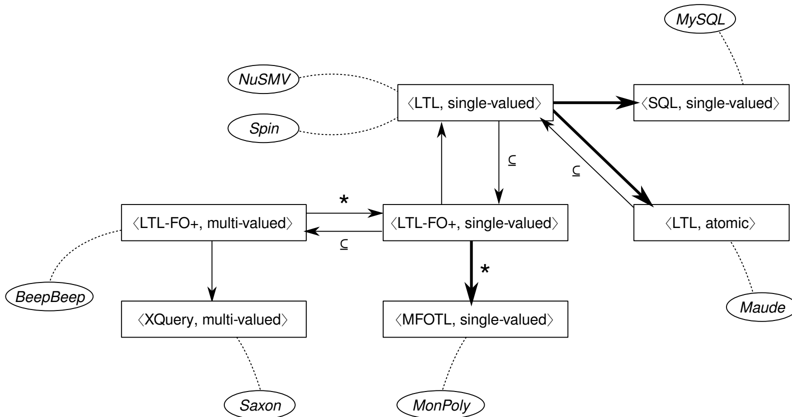


Fig. 1. A map of tools, input formats and transducers. An arrow labelled with $A \overset{\subseteq}{\rightarrow} B$ indicates that the transduction from A to B is trivially implemented since problem A is a particular case of problem B. A starred arrow indicates that the transduction is equivalent for a fragment of the input language. Variable- and fixed-schema are assimilated in this graph.

Transducers. This basic workflow, however, works only if the input trace and specification are expressed directly using the trace type and specification language of the tool one wishes to use. Yet, we have argued that the problem of validating a specification on a trace may also be converted into an equivalent problem of some other domain; this problem need not even be expressed in terms of traces and specifications, as long as its solution can be used to infer the answer to the original trace validation question. Therefore, in addition to translators, BabelTrace adds a set of *transducers* whose task is to transform a problem instance pair $(\varphi, \tau) \in \langle \mathscr{L}, \mathscr{T} \rangle$ into a new problem $(\varphi', \tau') \in \langle \mathscr{L}', \mathscr{T}' \rangle$, in such a way that $\tau \models_{\langle \mathscr{L}, \mathscr{T} \rangle} \varphi$ if and only if $\tau' \models_{\langle \mathscr{L}', \mathscr{T}' \rangle} \varphi'$.

In Figure 1, straight arrows between boxes indicate transducers currently implemented in BabelTrace. The detailed presentation of each transducer is beyond the scope of this paper. However, one shall keep in mind that the translation generally involves a transformation of both the trace and the specification in order to preserve equivalence, and that such translation is not always polynomial. Nevertheless, a few transducers are worthy of mention:

– The transduction from LTL to SQL transforms a trace into a database table, and an LTL formula into an equivalent SQL expression. Trace validation is hence rephrased as a special case of SQL query evaluation.
– The translation to NuSMV [3] and Spin [8] is done by building a completely deterministic transition system producing only the event trace to validate. Trace validation is hence rephrased as a special case of LTL model checking.
– The transduction from LTL-FO$^+$ to XQuery is an implementation of a result shown in [6]. Trace validation is rephrased as a special case of XML query processing.
– The transduction from propositional LTL to atoms fetches all values queried by the LTL formula and creates one atomic symbol per combination of those values.

4 Applications and Future Work

One can see how, by following arrows in the graph, it is possible to pipe transducers and reformulate a given trace validation in many different but ultimately equivalent problems. Hence BabelTrace offers great flexibility in the choice of the tool used to solve a problem instance. A first obvious consequence is the possibility to easily devise a thorough benchmark of the various methods available. Figure 2 shows a foretaste of the results that can be obtained, through the validation of a set of 500 randomly generated traces over the LTL-FO$^+$ formula $\mathbf{F}\,(\exists x \in p : x = 0)$ for a subset of 6 different tools. It shows —to our surprise— that the best performance is achieved by converting the problem as an equivalent SQL query.

BabelTrace's Java interface is currently being adapted for online use through a web portal, aiming at easy access of a large set of known trace validation tools and convenient conversion of a problem into various input languages. We shall finally mention that this list of formats, specification languages and tools is growing. For example, we omitted the language Eagle due to the unavailability of the tools using them (notably Logscope [4] and Monid [9]). ProM and SEQ.OPEN are planned to be added to the graph in the near future, as well as support for regular expressions and MapReduce.

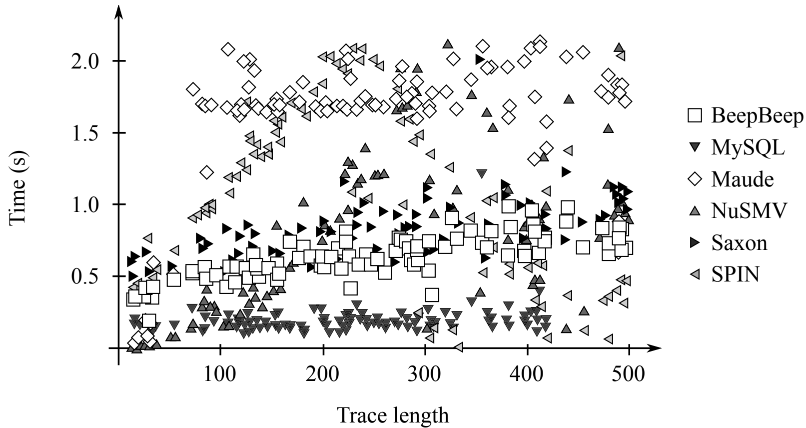


Fig. 2. Validation time for six different tools, on a Lenovo ThinkStation E20 with Ubuntu 11.04

References

1. Basin, D., Klaedtke, F., Müller, S.: Policy Monitoring in First-Order Temporal Logic. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 1–18. Springer, Heidelberg (2010)
2. Chen, F., d'Amorim, M., Rosu, G.: Checking and correcting behaviors of Java programs at runtime with Java-MOP. Electr. Notes Theor. Comput. Sci. 144(4), 3–20 (2006)
3. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
4. Groce, A., Havelund, K., Smith, M.H.: From scripts to specifications: the evolution of a flight software testing effort. In: Kramer, J., Bishop, J., Devanbu, P.T., Uchitel, S. (eds.) ICSE (2), pp. 129–138. ACM (2010)
5. Günther, C.W.: Extensible event stream standard definition 1.0. Technical report (2009)
6. Hallé, S., Villemaire, R.: XML Methods for Validation of Temporal Properties on Message Traces with Data. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part I. LNCS, vol. 5331, pp. 337–353. Springer, Heidelberg (2008)
7. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. on Services Computing (2011)
8. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional (2003)
9. Naldurg, P., Sen, K., Thati, P.: A Temporal Logic Based Framework for Intrusion Detection. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 359–376. Springer, Heidelberg (2004)
10. Rosu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. 12(2), 151–197 (2005)
11. van der Aalst, W.M.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)

Quantitative Trace Analysis Using Extended Timing Diagrams

Andreas Richter and Klaus Kabitzsch

Dresden University of Technology, Institute of Applied Computer Science,
Chair of Technical Information Systems, D-01062 Dresden, Germany
{andreas.richter1,klaus.kabitzsch}@tu-dresden.de

Abstract. Putting runtime verification into everyday industrial practice, shows the demand for formalisms that are easily understandable and usable, also for non-experts. To gain deeper insight into a system's behaviour, methods must be sufficiently expressive and support the evaluation of quantitative properties. We present a graphical specification language for quantitative trace analysis based on timing diagrams to meet these requirements. The proposed formalism allows the verification of functional requirements against traces in combination with the determination of quantitative parameters that characterize the system and render original requirements more precisely. We successfully included the timing diagram specification formalism and the corresponding evaluation methods into commercial trace analysis and test tools which are used to examine measurement data from the industrial automation and automotive domains.

Keywords: quantitative analysis, timing diagrams, trace analysis, embedded systems, automotive.

1 Introduction

The application of interconnected embedded controllers in vehicles and manufacturing facilities has permanently increased over the last decades. Verification and quality assurance for these systems have become main topics in the respective domains. Due to the complexity and reactivity of the systems, not all problems are detectable or avoidable by means of traditional test and diagnosis. Runtime monitoring in combination with subsequent trace analysis often can help to increase the degree of validation. The application of runtime verification into everyday industrial practice shows great need for formalisms that are easily understandable and also usable for practitioners without detailed knowledge in theoretical computer science. To gain deeper insight into concrete system behaviour, methods must be sufficiently expressive and support the evaluation of quantitative properties. In this paper we show extensions of timing diagrams (TD) and their application for intuitive quantitative trace analysis.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 131–135, 2013.
© Springer-Verlag Berlin Heidelberg 2013

2 Related Work

Traditionally, many verification techniques operate on qualitative properties and provide qualitative (i. e. boolean) verdicts. For industrial application it is often also desirable to calculate numerical or statistical analysis results, e. g.: How often did a pattern occur? When and to which extent was a timing constraint violated? Is the time of occurrence of an event more and more drifting towards one endpoint of a given timing interval? Quantitative trace analysis can answer those questions and extend a trace by the computation of new values as it operates on concrete system executions and is able to access concrete system state values and timings. Quantitative extensions to linear temporal logic (LTL) were for example shown in [1].

There are several visual specification formalisms (e. g. state charts, message sequence charts, live sequence charts) with varying suitability for different domains and system types. Timing diagrams are a common and established graphical specification language in the engineering domains. The diagrams focus on the states of signals, the changes of these states over time and the corresponding timing relations. By that they are well suited for the specification of the input/output behaviour of networked devices. Because timing diagrams resemble the signal display of oscilloscopes, they are intuitively comprehensible to many engineers. Timing diagrams are also part of the UML 2 specification and thereby can contribute in closing the gap between computer science and other engineering disciplines.

Timing diagrams have been used for interaction modelling during software design (UML), as specification language for model checking [2], for the verification of hardware designs [3] or for VHDL code synthesis [4]. Many publications describe specific enhancements of timing diagrams and give different definitions for their syntax and semantics, like Symbolic timing diagrams [5], real-time enhancements [6] or composition of multiple timing diagrams [7].

3 Quantitative Timing Diagrams

A timing diagram consists of one or more *signals* along the vertical axis with the *timeline* running in positive horizontal direction. Each signal is associated a *waveform*, formed by a sequence of *edges*. Edges hold *state expressions* that constrain the expected signal values. Thus a waveform determines the allowed sequence of a signal's state values over time. During analysis, measured signal values are matched against the state expressions of the current diagram states so that the diagram is traversed from left to right. *Relationships* (arrows) with interval annotations [min,max] describe timing constraints between edges of (different) waveforms. The interval [0,0] defines synchronisation. The pattern depicted by the diagram is successfully identified if all waveforms are traversed until their final states are reached and no timing constraints were violated.

To enable quantitative trace analysis, we made the following extensions to the original syntax and semantics of timing diagrams and implemented a graphical diagram editor (Fig. 1) along with evaluation functionality:

Diagram activation. We implemented two different activation modes for diagrams. One can either use one signal with an activating edge that triggers the diagram evaluation or a set of multiple precondition signals to model more complex activation conditions.

Event and conditional edges. We distinguish between event and conditional edges. Event edges expect exactly one sample value that satisfies the expression of the edge and immediately proceed to the next edge of the waveform. In contrast, conditional edges accept multiple consecutive samples that match the edge expression without advancing to the next edge.

State expressions. As state expressions, we allow all evaluable relational expressions for the considered diagram and trace. We defined the state expression grammar in EBNF. The grammar utilizes several keywords for enhanced convenience and expressiveness (e. g. VALUE / TIME accesses the current signal value / trace timestamp, SIG(signalname) accesses current values of other signals, AS(variablename) accesses preceding internal variable assignments, PARA(parametername) accesses parameters that are passed to the diagram from outside the evaluation engine).

Assignments. Following the approach of Finkbeiner et al. [1], we allow the definition of value assignment for all diagram elements (edges, relationships and the whole diagram itself) denoted as *element : assignment*. An assignment is evaluated when the defining element is evaluated to TRUE. Similar to state expression, assignment expressions have an underlying grammar that allows complex value calculations. An evaluated assignment is characterized by its unique name, the assigned value and the trace time stamp of its evaluation.

Evaluation modes. Evaluation can be run either in validation mode, where every activated TD instance must complete as specified or in pattern matching mode that succeeds if the TD is identified at least once in a given trace.

All assignments, together with other concrete values and timings gathered during evaluation (e. g. diagram activation and completion times, time stamps for individual edge activations, relationship durations), are recorded into a detailed XML-based result file. Result files can be passed to subsequent components of the quantitative analysis tool chain like visualization interfaces (see Fig. 2), statistical tools or result databases.

Figure 1 shows a simple TD example with two signals, each having a waveform consisting of two edges, and one relationship. The diagram checks that every change in *Signal_0* resulting in *Signal_0* being *Signal_0_HIGH* is answered by a similar change in *Signal_1* such that *Signal_1* equals *Signal_1_HIGH* within 0.8 seconds. *Signal_0* is a precondition signal which creates a new diagram evaluation instance every time the signal changes from 0 to the value of the parameter *Signal_0_HIGH*. The ability to parameterize expressions allows specification of generic diagrams that are reusable in different analysis runs under variable conditions. The diagram also shows the use of assignments within different diagram elements. The assignment *dur* calculates the concrete duration of the relationship via the keyword DURATION, *ts* is assigned the trace timestamp when *Signal_1* changes from 0 to the value of *Signal_1_HIGH* using the keyword TIME.

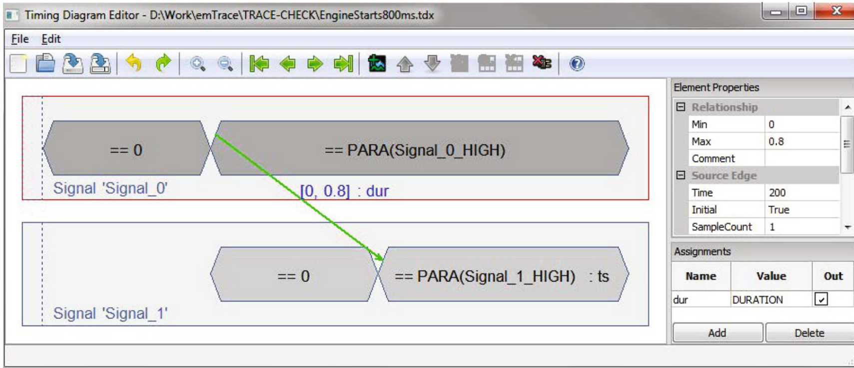


Fig. 1. Editor for quantitative timing diagrams

4 Application

We integrated our verification approach into the commercial trace analysis software TRACE-CHECK [8] which is widely used in the automotive industry and factory automation. Besides timing diagrams, the tool also supports verification of properties formulated in a real-time logic similar to Metric Temporal Logic (MTL) and via Python-implemented scripts. TRACE-CHECK and the TD evaluation module can operate on traces with non-equidistant time stamps.

We successfully applied timing diagram specification on several use cases within the automotive domain. We found that test engineers appreciate timing diagram specifications. They often struggle with translating functional requirements into temporal logic formulas. There the specification of complex timing relationships between signal curves unavoidably leads to complicated, strongly nested expressions. Experience has also shown that timing diagram specification nicely complements temporal logic as it focuses on the "good-cases" or "positive patterns" of functional requirements specification. Whereas in logic it is easier

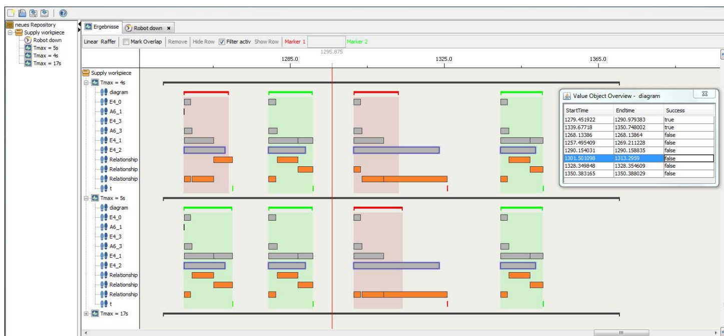


Fig. 2. Visualization of timing diagram analysis results

to formulate that something must not happen, quantitative timing diagrams testify whether a finite trace segment behaves like expected and give detailed information about the execution conditions.

To aggregate and overview analysis results we developed a visualisation component (Fig. 2) that processes the detailed report files. For an analyzed trace, consecutive diagram activations are plotted along a horizontal timeline. The colours green and red are used to distinguish between successfully identified and failed diagram entities. Users can zoom, filter and access concrete timings and values of all diagram elements. It is also possible to stack and overlay multiple analyses for comparison.

5 Conclusion and Future Work

In this paper, we showed the adaption of timing diagrams as a specification language for quantitative trace analysis. We gave an overview about the diagram editor and the analysis functionality provided by its integration into verification tools from the automotive and factory automation domains. We presented first application results and suggested a prototype for result visualisation. In future we will concentrate our research on the use of continuous signal descriptions inside state expressions and methods for hierarchically combining multiple diagrams. We will also give formal syntax and semantics for timing diagram evaluation over finite traces.

Acknowledgements. This work was funded by the German Federal Ministry of Education and Research (BMBF) within the research project *emTrace : Enhanced model-based trace analysis* under the reference number 01IS11004B. The authors are responsible for the content of this publication.

References

1. Finkbeiner, B., Sankaranarayanan, S., Sipma, H.B.: Collecting statistics over runtime executions. In: Proc. of Runtime Verification (RV 2002), pp. 36–55. Elsevier (2002)
2. Fisler, K.: Timing diagrams: Formalization and algorithmic verification. Journal of Logic, Language and Information 8, 323–361 (1999), doi:10.1023/A:1008345113376
3. Damm, W., Josko, B., Schlör, R.: Specification and validation methods, pp. 331–409. Oxford University Press, Inc., New York (1995)
4. Grass, W., Grobe, C., Lenk, S., Tiedemann, W.D., Kloos, C., Marin, A., Robles, T.: Transformation of timing diagram specifications into VHDL code. In: Design Automation Conference, Proc. ASP-DAC 1995/CHDL 1995/VLSI 1995 (1995)
5. Schlör, R.: Symbolic timing diagrams: a visual formalism for model verification. PhD thesis, Universität Oldenburg (2002)
6. Feyerabend, K.: Real time symbolic timing diagram. Technical report, Carl von Ossietzky Universität Oldenburg (1996)
7. Lenk, S.: Extended timing diagrams as a specification language. In: EURO-DAC 1994: Proceedings of the Conference on European Design Automation, pp. 28–33. IEEE Computer Society Press, Los Alamitos (1994)
8. Deutschmann, R., Fruth, M., Zabelt, M.: Neue Absicherungsstrategien für Steuergerätesoftware. In: Moderne Elektronik im Kfz V, Haus der Technik (2010)

Maximal Causal Models for Sequentially Consistent Systems*

Traian Florin Șerbănuță[1,2], Feng Chen[1], and Grigore Roșu[1,2]

[1] University of Illinois at Urbana-Champaign
[2] University "Alexandru Ioan Cuza" Iași

Abstract. This paper shows that it is possible to build a *maximal and sound* causal model for concurrent computations from a given execution trace. It is sound, in the sense that any program which can generate a trace can also generate all traces in its causal model. It is maximal (among sound models), in the sense that by extending the causal model of an observed trace with a new trace, the model becomes unsound: there exists a program generating the original trace which cannot generate the newly introduced trace. Thus, the maximal sound model has the property that it comprises *all* traces which *all* programs that can generate the original trace can also generate. The existence of such a model is of great theoretical value as it can be used to prove the soundness of non-maximal, and thus smaller, causal models.

1 Introduction

Traces of events describing concurrent computations have been employed in a plethora of methods for testing and analyzing concurrent systems. The common pattern for all these methods (e.g., [2,3,5,7,12–16,18–21]) is: (1) the program is instrumented to trace the execution of programs; then (2) *one* execution trace is recorded; then (3) an abstraction of that trace, i.e., a *model*, is derived; and finally, (4) the obtained model is used to "predict" (problematic) event patterns occurring in other possible executions abstracted by it.

Consider, for example, the conventional happens-before causality: if two conflicting accesses to an object are not causally ordered, then a data-race is reported [15]. But is this the best one can do? Of course, not. A series of papers propose more relaxed happens-before causal models where one can also permute blocks protected by the same lock, provided that they access disjoint variables [12], thus discovering new concurrency bugs not observable with plain happens-before. But is this the best one can do? Of course, not. Other papers propose models where one can also permute semantic blocks provided that each read access continues to correspond to the same write [16,18,21]. Others go even further. Section 5 discusses a series of existing causal models; we only study *sound* models here, i.e., ones which only report real problems in the analyzed

* This work was supported in part by Contract 161/15.06.2010, SMISCSNR 602-12516 (DAK), by NSA contract H98230-10-C-0294 and by NSF grant CCF-0916893.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 136–150, 2013.
© Springer-Verlag Berlin Heidelberg 2013

systems, allowing developers to focus on fixing those real problems and not on additionally sorting them out from false positives. We would naturally like to know whether there is an end to the question “Is this the best we can do?”, that is, whether there is any causal model that can be associated to a given execution trace which comprises the maximum number of causally equivalent traces.

Although most runtime analysis techniques are built upon some underlying sound causal model, possibly relaxed for efficiency reasons, each effort seems to focus more on how to capture it efficiently rather than proving its soundness (often implicitly assumed) or studying its relationship to existing models (other than empirically comparing the number of found bugs). Moreover, since such approaches attempt to extract information from *one* observed trace and to find property violations, they actually deal with *causal properties* (e.g., causal datarace, causal atomicity), which are instances of desired system-wise properties that can be detected using only the causal information gathered from the observed trace. Since what can be inferred from a trace intrinsically depends on the chosen causal model, definitions of causal properties differ from technique to technique, with the undesirable effect that a causal property (e.g., a datarace) in one model might not be recognized as such by another model.

1.1 Motivating Examples

Each example in Figure 1 shows a two-threaded program, together with one of its possible executions, in which Thread 1 is executed completely before Thread 2 starts. In this representation of executions, synchronized blocks are boxed, while write and read operations on shared locations are denoted by $\leftarrow$ (receiving a value), and $\rightarrow$ (yielding a value), respectively. Both *programs* exhibit a race condition between the two write operations on y. However, are the observed executions also exhibiting a causal datarace?

When analyzing the observed execution in Figure 1(a), a simple happens-before approach ordering all accesses to concurrent objects [15] cannot observe

(a)

Thread 1	*Thread 2*	*Execution*
sync(l) {		1:
$y = 1;$		[$y \leftarrow 1$
$x = 1;$		$x \leftarrow 1$
if $(x == 2)$		$x \rightarrow 1$]
$z = 1;$		
}		
	sync(l){	2:
	$x = 2;$	[$x \leftarrow 2$]
	}	
	$y = 2;$	$y \leftarrow 2$

(b)

Thread 1	*Thread 2*	*Execution*
sync(l) {		1:
$x = 1;$		[$x \leftarrow 1$]
}		
$y = 1;$		$y \leftarrow 1$
sync(l) {		
$x = 1;$		[$x \leftarrow 1$]
}		2:
	sync(l) {	
	if $(x > 0)$	[$x \rightarrow 1$
	$y = 2;$	$y \leftarrow 2$]
	}	

Fig. 1. Motivating examples

a causal datarace: the release operation of the lock in Thread 1 is required to happen-before the acquire of the lock in Thread 2. Happens-before with lock atomicity [12] is not able to infer a causal datarace either: although the lock atomicity would allow for the two lock-blocks to be permuted, the read of x in Thread 1 is still required to happen-before the write of x in Thread 2. Yet, the race condition can be captured as a causal datarace of the observed execution by weaker happens-before models [16, 18, 21], since in those models, one can additionally permute a write before a read of same location, as long as it is permuted before the write corresponding to that read. Thus, the trace generated by the program in Figure 1(a) *has or does not have* a causal datarace, depending upon the particular causal model employed.

However, none of the approaches mentioned above can detect the race condition in Figure 1(b) as a causal datarace for the observed execution. The reason for this is that all models enforce at least the read-after-write dependency (i.e., a read should always follow the latest write event of the same variable), and therefore would not allow the permutation of the last two lock-blocks of the execution, since the read of x in Thread 2 must follow the last write of x in Thread 1. Nevertheless, there is enough information in the observed execution to be able to detect the race: since both writes of x in Thread 1 write the same value, it is actually possible to permute the last two lock blocks, and thus detect the race. Moreover, since one could conceive a technique specialized for finding such cases, it can be rightfully claimed that the observed execution has in fact a causal datarace, although not captured by any existing definition.

Given this ever increasing (regarding coverage) sequence of causal models and definitions for causal properties, it is only natural to ask the following question:

> *Is there any causal model that generalizes all existing models, and which cannot be surpassed?*

We answer this question positively in the context of sequential consistency [9]. While we believe the presented approach can be applied to other memory models, we chose sequential consistency here for two reasons: (1) it is broadly accepted, popular and intuitive; (2) it is subsumed by other memory models: errors detected under sequential consistency are also errors for other memory models.

Contributions. The main result of this paper is a semantic framework that allows to prove maximality of causal models, and a proof that our proposed model is indeed the maximal causal model for the observed execution. This means that it comprises precisely *all* traces which can be generated by all programs which can generate the observed trace. Concretely, we show that: (1) all programs which can produce the observed execution can generate all traces in the model; and (2) for any trace not in the model there exists a program generating the observed trace which cannot generate it. To our knowledge, this is the first such result for causal models. We then prove (the implicitly assumed) soundness for a series of existing causal models by showing they are submodels of the proposed model.

Paper structure. Section 2 introduces some notation and discusses sequential consistency. Section 3 axiomatizes consistent concurrent systems and defines our proposed causal models. Section 4 formally defines the maximality claim and proves our model maximal among sound models. Section 5 shows how existing models are included in ours, thus proving their soundness. Section 6 reviews related research and discusses several research ideas connected with the presented work. Section 7 concludes.

2 Execution Model

Assume a machine that can execute arbitrarily many threads in parallel. The execution environment contains a set of *concurrent objects* (shared memory locations, locks, ...), which are accessed by threads to share data and synchronize. *Threads*, which can only interact through the execution environment, are abstracted as *sequences of operations on concurrent objects*. The only source of thread non-determinism is the execution environment, that is, if the interaction between a thread and the environment is the same across executions, the thread will execute the same operations, in the same order. To simplify the presentation, we assume no dynamic creation of threads (this presents no technical difficulty).

2.1 Concurrent Objects, Serial Specification

We adopt the definition of concurrent objects and serial specifications proposed by Herlihy and Wing [8]. A concurrent object is behaviorally defined through a set of atomic operations, which any thread can perform on it, and a serial specification of its legal behavior in isolation. The serial specification describes the valid sequences of operations which can be performed on the object. We next describe two common types of concurrent objects.

Shared memory locations. Each shared memory location can be regarded as a shared object with read and write operations, whose serial specification states that each read yields the same value as the one of the previous write. Moreover, to avoid non-determinism due to the initial state of the memory, we will further require that all memory locations are initialized, that is, the first operation for each location is a write.

Mutexes. Each mutex can be regarded as a concurrent object providing *acquire* and *release* operations. Their mutual exclusion property is achieved through the serial specification which accepts only those sequences in which the difference between the number of *acquire* and *release* operations is either 0 or 1 for each prefix, and all consecutive pairs of *acquire*-*release* share the same thread.

To keep the proofs simple and the concepts clear, we refrain here from adding more concurrency constructs (such as spawn/join, wait/notify, or semaphores). Note, however, that this would not introduce additional complexity, but just further constrain the notion of consistency.

2.2 Events and Traces

Operations performed by threads on concurrent objects are recorded as *events*. We consider events to be abstract entities from an infinite "collection" *Events*, and describe them as tuples of *attribute-value* pairs. The only attributes considered here are: *thread*—the unique id of the thread generating the event, *op*—the operation performed (e.g., *write*, *read*, *acquire*, or *release*), *target*—the concurrent object accessed by the event, and *data*—the value sent/received by the current event, if such exists (e.g., for the *write*/*read* operations). For example, ($thread{=}t_1$, $op{=}write$, $target{=}x$, $data{=}1$) describes an event recording a write operation by thread t_1 to memory location x with value 1. When there is no confusion, we only list the attribute values in an event, e.g., $(t_1, write, x, 1)$. Our choice for deciding what attributes to record in an event considers a monitor which can observe memory and synchronization operations and the identity of the thread performing them, but has no access to the actual code. Section 6 includes a discussion on possible variations on the set of attributes recorded for an event.

For any event e and attribute *attr*, $attr(e)$ denotes the value corresponding to the attribute *attr* in e, and $e[v/attr]$ to denote the event obtained from e by replacing the value of attribute *attr* by v. An *execution trace* is abstracted as a sequence of events. Given a trace τ, a concurrent object o and a thread t, let $\tau\restriction_o$ and $\tau\restriction_t$ denote the restriction of τ to events involving only o, and only t, respectively. Let $latest_o(\tau)$ be the latest event of τ having the *op* attribute o. If o is omitted, it simply means the latest event in τ.

Sequential consistency can be now elegantly defined:

Definition 1 ([1]). *Let τ be any trace.*
*(1) τ is **legal** if and only if $\tau\restriction_o$ satisfies o's serial specification for any object o;*
*(2) An **interleaving** of τ is a trace τ' such that $\tau'\restriction_t = \tau\restriction_t$ for each thread t.*
*(3) A trace τ is **(sequentially) consistent** if it admits a legal interleaving.*

Since we restrict ourselves to sequential consistency, from here on when we say that a trace is sequentially consistent we automatically mean that it is also legal.

3 Feasibility Model

This section introduces an axiomatization for a machine producing consistent traces, and uses it to associate a sound-by-definition causal model to any observed execution, comprising all executions which can potentially be inferred from that execution alone, without additional knowledge of the system generating it.

The two properties/axioms (presented below) we base our approach on are *trace consistency* and *feasible executions*. A consistent trace (Definition 1) disallows "wrong" behaviors, such as reading a value different from the one which was written, or proceeding when a lock cannot be acquired. Feasible executions, defined below, refer to *sets* of execution traces and aim at capturing *all* the behaviors that a given system or program can manifest. No matter what task a concurrent system or program accomplishes, its set of traces must obey some

basic properties. First, feasible traces are generate-able, meaning that any prefix of any feasible trace is also feasible; this is captured by our first axiom of feasible traces, *prefix closedness*. Second, we assume that thread interleaving is the only source of non-determinism in producing traces; this is captured by our second axiom of feasible traces, *local determinism*.

Each particular multithreaded system or programming environment, say $\mathcal{S}$, has its own notion of feasible execution, given by its specific intended semantics. Let us call all (possibly incomplete) traces that $\mathcal{S}$ can yield $\mathcal{S}$*-feasible*, and let $\mathit{feasible}(\mathcal{S})$ be their set. Instead of defining $\mathit{feasible}(\mathcal{S})$, which requires a formal definition of $\mathcal{S}$ and is therefore $\mathcal{S}$-specific (and tedious), we here *axiomatize* it:

Prefix Closedness. *Events are indivisible and generated in execution order; hence,* $\mathit{feasible}(\mathcal{S})$ *must be prefix closed*: if $\tau_1\tau_2$ is $\mathcal{S}$-feasible, then τ_1 is $\mathcal{S}$-feasible. Prefix closedness ensures that each event is generated individually, with the possibility of interleaving happening in-between any of them. For example, although the `++ x` instruction generates two events, a read, follow by a write on `x` these are not necessarily consecutive: if the instruction is not properly synchronized, another thread could write `x` after the first event, yielding an atomicity violation.

Local Determinism. *The execution of a concurrent operation is determined by the previous events in the same thread, and can happen at any consistent moment after them.* Formally, if $\tau e, \tau' \in \mathit{feasible}(\mathcal{S})$ and $\tau\restriction_{\mathit{thread}(e)} = \tau'\restriction_{\mathit{thread}(e)}$ then: if $\tau' e$ is consistent then $\tau' e \in \mathit{feasible}(\mathcal{S})$; moreover, if $\mathit{op}(e) = \mathit{read}$ and there exists an event e' such that $e = e'[\mathit{data}(e)/\mathit{data}]$ and $\tau' e'$ is consistent, then $\tau' e \in \mathit{feasible}(\mathcal{S})$. The second part says that if a *read* operation is enabled, i.e., all previous events have been generated, then it can be executed at any consistent time (despite the fact that the value it receives might be different from that observed in the original trace). Allowing traces where read events observe a different value than in the original trace might seem like a source of unsoundness. Note, however, that the same local determinism property prohibits the thread on which such a read event occurred to continue after producing this event, by stating that an additional event for a thread is generated *only* if the current trace for that thread is *exactly* the same (including the value) as in the original trace. Suppose, for example, that two threads, identified by t_1 and t_2, assign 1, then execute an increment operation on the same location l. One potential observed trace could be: $(t_1, \mathit{write}, l, 1)(t_2, \mathit{write}, l, 1)(t_1, \mathit{read}, l, 1)(t_1, \mathit{write}, l, 2)(t_2, \mathit{read}, l, 2)(t_2, \mathit{write}, l, 3)$. Local determinism ensures that we can also obtain the (partial) trace

$$(t_1, \mathit{write}, l, 1)(t_2, \mathit{write}, l, 1)(t_1, \mathit{read}, l, 1)(t_2, \mathit{read}, l, 2)(t_1, \mathit{write}, l, 2).$$

This shows that we can use local determinism to interleave threads differently than their original scheduling, as long as consistency is respected and threads produce the same events. Note that, (1) event $e = (t_2, \mathit{read}, l, 2)$ can be generated although it reads a different value than it originally did; and (2) thread t_1 can continue after e was generated (since it concerns a different thread), but thread t_2 cannot (because, e.g., e could be guarding a control statement).

Definition 2. *$\mathcal{S}$ is* ***consistent*** *iff feasible$(\mathcal{S})$ satisfies the axioms above.*

A major goal of trace-based analysis is to infer/analyze as many traces as possible using a recorded trace. When one does not know (or does not want to use) the source code of the multithreaded program being executed, one can only infer potential traces of the system resembling the observed trace. Let us now define the proposed causal model, termed *feasibility closure*, as the set of executions which can be inferred from an observed execution—they correspond to the traces obtainable from τ using the feasibility axioms.

Definition 3. *The* ***feasibility closure*** *of a consistent trace τ, written feasible(τ), is the smallest set of traces containing τ which is prefix-closed and satisfies the local determinism property. A trace in feasible(τ) is called* ***τ-feasible****.*

Without dwelling into details here, as this is proved elsewhere [17], intuitively the feasibility closure of a trace contains all interleavings of the observed trace, where each thread is stopped once it read a value from the memory different from the one observed originally, as well as all prefixes of these traces.

The following result formalizes the soundness of the proposed model. Assuming the base axioms are sound, the closure properties guarantee that all traces in our causal model are feasible. In addition, Proposition 1 shows that *any* system/program which can generate one trace, can also generate *all* traces comprised by its causal model.

Proposition 1. *If $\mathcal{S}$ consistent and $\tau \in$ feasible$(\mathcal{S})$ then feasible$(\tau) \subseteq$ feasible$(\mathcal{S})$. Moreover, if τ' is consistent and $\tau \in$ feasible(τ'), then feasible$(\tau) \subseteq$ feasible(τ').*

The intuition for $\tau \in$ *feasible*(τ') is that if a run of any program executed on $\mathcal{S}$ can produce τ', then there is also some run of the same program executed also on $\mathcal{S}$ that can produce τ.

4 Maximality

In this section we show that the proposed causal model is *maximal* among sound models, in the sense that any extension to it is done at the expense of soundness. We will prove therefore that given a trace τ' which is not in the feasibility closure of a trace τ, there exists a program p which can generate τ but not τ'; therefore, if the model were extended to include τ' and used τ' as a witness that a property is satisfied/invalidated by a program generating τ, this would be a false witness if the program which generated τ was p.

To prove our claim, we propose CONC, a very simple (not even Turing complete) concurrent language. The benefit of such a simple language is that it can conceivably be simulated in any real language; therefore, proving the maximality result for CONC proves the model is maximal for all languages. Figure 2 presents the grammar and SOS semantics of CONC. The grammar specifies a parallel composition of named threads. Each thread is a succession of statements

CONC SYNTAX: $Proc ::= Proc \mid\mid Proc \mid Int : Stmt$

$Stmt ::= Stmt \; ; \; Stmt \mid \mathsf{nop} \mid \mathsf{if}\; Int\; \mathsf{then}\; Stmt$

$\mid \mathsf{load}\; Loc \mid Loc := Int \mid \mathsf{acquire}\; Loc \mid \mathsf{release}\; Loc$

CONC SEMANTICS:

$$\frac{\langle p_1, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_1', \sigma', \delta', \rho'\rangle}{\langle p_1 \mid\mid p_2, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_1' \mid\mid p_2, \sigma', \delta', \rho'\rangle} \quad (Par_1)$$

$$\frac{\langle p_2, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_2', \sigma', \delta', \rho'\rangle}{\langle p_1 \mid\mid p_2, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle p_1 \mid\mid p_2', \sigma', \delta', \rho'\rangle} \quad (Par_2)$$

$$\frac{\langle s, \sigma, \delta, \rho, t\rangle \xrightarrow{\tau} \langle s', \sigma', \delta', \rho', t\rangle}{\langle t : s, \sigma, \delta, \rho\rangle \xrightarrow{\tau} \langle t : s', \sigma', \delta', \rho'\rangle} \quad (Thread)$$

$$\frac{\langle s_1, \sigma', \delta', \rho', t\rangle \xrightarrow{\tau} \langle s_1', \sigma', \delta', \rho', t\rangle}{\langle s_1 \; ; \; s_2, \sigma, \delta, \rho, t\rangle \xrightarrow{\tau} \langle s_1' \; ; \; s_2, \sigma', \delta', \rho', t\rangle} \quad (Seq)$$

$$\frac{\cdot}{\langle \mathsf{nop} \; ; \; s, \sigma, \delta, \rho, t\rangle \xrightarrow{\epsilon} \langle s, \sigma, \delta, \rho, t\rangle} \quad (Nop)$$

$$\frac{\cdot}{\langle \mathsf{if}\; i\; \mathsf{then}\; s, \sigma, \delta, \rho, t\rangle \xrightarrow{\epsilon} \langle s, \sigma, \delta, \rho, t\rangle} \quad \textbf{if } \rho(t) = i \quad (If_{true})$$

$$\frac{\cdot}{\langle \mathsf{if}\; i\; \mathsf{then}\; s, \sigma, \delta, \rho, t\rangle \xrightarrow{\epsilon} \langle \mathsf{nop}, \sigma, \delta, \rho, t\rangle} \quad \textbf{if } \rho(t) \neq i \quad (If_{false})$$

$$\frac{\cdot}{\langle \mathsf{load}\; x, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, read, x, i)} \langle \mathsf{nop}, \sigma, \delta, \rho[t \leftarrow i], t\rangle} \quad \textbf{where } i = \sigma(x) \quad (Read)$$

$$\frac{\cdot}{\langle x := i, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, write, x, i)} \langle \mathsf{nop}, \sigma[x \leftarrow i], \delta, \rho, t\rangle} \quad (Write)$$

$$\frac{\cdot}{\langle \mathsf{acquire}\; x, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, acquire, x)} \langle \mathsf{nop}, \sigma, \delta[x \leftarrow t], \rho, t\rangle} \quad \textbf{if } \delta(x) = \perp \quad (Acq)$$

$$\frac{\cdot}{\langle \mathsf{release}\; x, \sigma, \delta, \rho, t\rangle \xrightarrow{(t, release, x)} \langle \mathsf{nop}, \sigma, \delta[x \leftarrow \perp], \rho, t\rangle} \quad \textbf{if } \delta(x) = t \quad (Rel)$$

Fig. 2. Syntax and SOS semantics for the CONC language

and uses one internal register to load data from the shared memory. $\mathsf{load}\; x$ loads the value at location x into the internal register of the thread, $x := i$ stores integer i at location x, $\mathsf{acquire}$ and $\mathsf{release}$ have the straight-forward semantics, and $\mathsf{if}\; i\; \mathsf{then}\; s$ executes s only if the internal register has value i. A running configuration of CONC is a tuple $\langle p, \sigma, \delta, \rho\rangle$ where p is the remainder of the program being executed, σ maps variables to values, δ maps each lock to the id of the thread holding it, and ρ gives for each thread the value of its internal register. In the SOS derivation rules we additionally use configurations of the

form $\langle p, \sigma, \delta, \rho, t\rangle$, where t is the thread id obtained in the (*Thread*) rule, which is propagated by all following rules. Assuming p has n threads, the initial configuration of the system is $START(p) = \langle p, \sigma_\epsilon, \delta_\epsilon, \rho^n_\epsilon\rangle$ where σ_ϵ, δ_ϵ, and ρ^n_ϵ, initialize all locations, locks, and registers for the n threads with $\bot$, respectively.

We have chosen this minimal language both because it is sufficiently expressive to generate all (finite) legal traces, and because it is quite easy to mimic in any other language. In Java, for example, each thread would be modeled by a thread object, and all threads could be started in a loop by the main thread. Since beginnings of threads do not generate events, this is as-if all threads start together in parallel. The running method of each Java thread object would declare a local variable r to stand for the register, and then the two CONC instructions dealing with the register translate as follows: load l becomes $r = l$, and if i then s becomes if $(r == i)\ \ s$.

It is straightforward to associate to each event an instruction producing it. Let *code* be the mapping defined on events as follows:

$$code(e) = \begin{cases} \textsf{load } x & \textbf{if } e = (t, read, x, i) \\ x := i & \textbf{if } e = (t, write, x, i) \\ \textsf{acquire } x & \textbf{if } e = (t, acquire, x) \\ \textsf{release } x & \textbf{if } e = (t, release, x) \end{cases}$$

Given a program p, let $p\restriction_t$ be its projection on thread t, that is, the statement labeled by t in the parallel composition.

The following result shows that, except for the code, the running configuration is completely determined by the trace generated up to that point:

Proposition 2. *If* $\mathrm{CONC} \vdash START(p) \xrightarrow{\tau}^* \langle p', \sigma_\tau, \delta_\tau, \rho^n_\tau\rangle$, *where n is the number of threads of p, then:*

(1) $\sigma_\tau(x) = data(latest_{write}(\tau\restriction_x))$;
(2) $\delta_\tau(x) = \begin{cases} thread(latest(\tau\restriction_x)), \text{ if} op(latest(\tau\restriction_x)) = acquire \\ \bot, \text{ otherwise} \end{cases}$;
(3) $\rho^n_\tau(t) = data(latest_{read}(\tau\restriction_t))$.

Therefore, in the sequel we will use $\mathrm{CONC} \vdash p \xrightarrow{\tau}^* p'$ instead of $\mathrm{CONC} \vdash START(p) \xrightarrow{\tau}^* \langle p', \sigma_\tau, \delta_\tau, \rho^n_\tau\rangle$

Now, let us prove that the semantics of CONC does indeed satisfy the sequential consistency axioms. Let p be a CONC program and let *feasible*(p) be the set of all p-feasible traces; that is τ is p-feasible if there exists a program p' such that $\mathrm{CONC} \vdash p \xrightarrow{\tau}^* p'$. We sill show that *feasible*(p) satisfies the *strong local determinism* property, namely, not only that an enabled event can be generated at any point by its thread, but that it also must be the (unique) next event generated by that thread (ignoring the *data* attribute for *read* events). Formally,

Definition 4. *feasible*(p) *satisfies the* ***strong local determinism*** *property if it satisfies local determinism and if $\tau_1 e_1$ and $\tau_2 e_2$ are p-feasible,* $thread(e_1) = thread(e_2) = t$, *and* $\tau_1\restriction_t = \tau_2\restriction_t$, *then* $op(e_1) = op(e_2)$, *and* $target(e_1) = target(e_2)$; *if additionally* $op(e_i) = write$, *then also* $data(e_1) = data(e_2)$.

The following result shows that every CONC program p is a consistent system in the sense of Definition 2.

Proposition 3 (CONC consistency). *$feasible(p)$ satisfies prefix closedness and strong local determinism.*

Proof (Sketch). Prefix closedness is obvious, since the semantics can emit at most one event for each execution step. The second property follows by case analysis on the rule SOS rule applied to produce the relevant event for local determinism.

Now, given a trace τ, let us build the canonical CONC program generating it. *code* can be naturally extended on traces by $code(e\tau) = code(e) \; ; \; code(\tau)$. Let $\{t_1, t_2, \ldots, t_n\}$ be the set of thread ids appearing in τ. Then the program associated to a trace τ is defined by $program(\tau) = t_1 : code(\tau\restriction_{t_1}) \mid\mid \cdots \mid\mid t_n : code(\tau\restriction_{t_n})$.

Let us also define the empty program with n threads as $program^n(\epsilon) = t_1 : \mathsf{nop} \mid\mid \cdots \mid\mid t_n : \mathsf{nop}$. The following result shows that the program corresponding to a consistent trace can indeed generate that trace.

Proposition 4. *If τ is a consistent trace with n threads, then* $\mathrm{CONC} \vdash program(\tau) \xrightarrow{\tau}^{*} program^n(\epsilon)$.

The following theorem justifies the maximality claims for the proposed model.

Theorem 1 (Maximality). *For any consistent trace τ' which is not τ-feasible there exists a program generating τ but not τ'.*

Proof (Sketch). Because of prefix closeness and thread determinism, the only interesting case to analyze is when τ' continues the execution on a thread after reading a value distinct from the one recorded in an event e of τ. in that case, we create a new program p from $program(\tau)$ by inserting a conditional write instruction right after that generating event e. We then show that program p can still generate τ, but cannot generate τ'.

5 Proving Soundness for Existing Causal Models

Focusing on identifying concurrency anomalies and measuring success based on the number of bugs found, almost no causal model in the literature is actually proved sound. The authors of a causal model usually give some common-sense arguments for their choice and informally rely on the soundness of Happens-Before [9]. However, intuition can sometimes be misleading: in Section 5.4 we reveal a soundness problem with the model of Sen et al. [16]. Moreover, even when proved sound, the proofs are quite laborious, each having to repeat the formalization of an execution model. Proving soundness of other causal models by embedding them in our already proven sound model eliminates the need for an execution model and reduces proofs to checking closure properties.

We start with the following result, which can be regarded as a sufficient criterion for feasibility:

Theorem 2. *Any consistent prefix of an interleaving of τ is τ-feasible.*

The remainder of this section shows that existing sound causal models are captured by the feasibility closure as simple instances of Theorem 2. Another important consequence of Theorem 2 is that is basically shows there is a unique feasibility closure associated to a concurrent computation, regardless of the representative trace [17].

5.1 Happens Before Relation on Mazurkiewicz Traces

One elegant way to capture the happens-before trace equivalence is the Mazurkiewicz trace [10] associated to the dependence given by the happens-before relation.

The happens-before dependence is a set $T \cup D$, where $T = \bigcup_t \{(e_1, e_2) : \tau\restriction_t = \tau_1 e_1 e_2 \tau_2\}$ is the intra-thread sequential dependence relation and $D = \bigcup_x \{(e_1, e_2) : \tau\restriction_x = \tau_1 e_1 e_2 \tau_2$ *such that* e_1 *or* e_2 *is a write of* $x\}$ is the sequential memory dependence relation. Given this happens-before dependence, the Mazurkiewicz trace associated with τ is defined as the least set $[\tau]$ of traces containing τ and being closed under permutation of consecutive independent events [10]: if $\tau_1 e_1 e_2 \tau_2 \in [\tau]$ and $(e_1, e_2) \notin T \cup D$, then $\tau_1 e_2 e_1 \tau_2 \in [\tau]$.

The following result shows that the feasibility closure is closed under the equivalence relation generated by happens-before, that is, happens-before is captured by our model, and thus re-shown sound for consistent executions:

Proposition 5. *If $\tau_1 e_1 e_2 \tau_2$ is τ-feasible and $(e_1, e_2) \notin T \cup D$, then $\tau_1 e_2 e_1 \tau_2$ is τ-feasible. Given any τ-feasible trace τ', $[\tau'] \subseteq feasible(\tau)$. Hence, $[\tau] \subseteq feasible(\tau)$.*

5.2 Weak Happens Before

Several more recent trace analysis techniques [16,18,21] argue that the happens-before model can be further relaxed, noticing that the only purpose of the write-after-read happens-before order is to guarantee that a read event always reads the same write event as before in any feasible interleaving of the original trace. Therefore, one only needs to preserve the *read-after-write dependence*:

Definition 5. *Suppose $\tau = \tau_1 e_1 \tau_2 e_2 \tau_3$. Then e_2* ***write-read depends on*** *e_1 in τ, written $e_1 <_\tau^{wr} e_2$, if $target(e_1) = target(e_2)$, $op(e_1) = write$, $op(e_2) = read$, and for all $e \in \mathcal{E}_{\tau_2}$, either $target(e) \neq target(e_1)$, or $op(e) \neq write$.*

That is, $e_1 <_\tau^{wr} e_2$ iff the value read by e_2 is the value written by e_1.

Sen et al. [16] introduce the notion of *atomic* sets associated to each *write* event, containing itself and all read events which write-read depend on it, accepting as feasible executions all linearizations of the transitive closure of the combined $<_\tau^{wr}$ and thread ordering, satisfying the additional requirement that the atomic sets are preserved. However this can be simply restated as follows [21]:

Definition 6. *$\tau \sim \tau'$ if τ is an interleaving of τ' and $<_\tau^{wr} = <_{\tau'}^{wr}$.*

That is, the $\sim$-equivalence class of τ contains all interleavings of τ which have exactly the same write-read dependence relation. Next result shows that this model is also captured by our model.

Proposition 6. *If τ_1 is τ-feasible, and $\tau_1 \sim \tau_2$, then τ_2 is also τ-feasible.*

5.3 Happens-Before with Synchronization

A conservative and sound approach, requiring no implementation changes, to handle locks in happens-before-based trace analysis techniques is to assume that *acquire* and *release* operations on the same lock yield the same happens-before dependence as if they were particular *write* and *read* operations (on the lock variable) [15]. However, this prevents synchronized blocks from being permuted, and thus imposes coverage limitations. The lock-set approaches, also called hybrid happens-before [12], propose to handle locks separately, associating with each event the set of locks [14] protecting them, hereby not enforcing any particular order between synchronized blocks.

We here group the events protected by locks in *atomic blocks*. Events e_1 and e_2 from a consistent trace τ, both generated by thread t, are *l-atomic* in τ, written $e_1 \Updownarrow_l^\tau e_2$, if and only if there is some *acquire* event e on lock l generated by t before both e_1 and e_2, and there is no *release* event e' on l generated by t between e and either of e_1, e_2. For each lock l, let $[e]_l$ denote the *l-atomic equivalence class of e*. Assuming a trace in which all acquired locks are eventually released, l-atomic equivalence classes consist of all events belonging the the same acquire-release block of l. A trace τ' is *consistent with the lock atomicity of* τ if there exists no lock l and decomposition $\tau_1 e_1 \tau_2 e_2 \tau_3 e_3 \tau_4 e_4 \tau_5$ such that $e_1 \Updownarrow_l^\tau e_3$ and $e_2 \Updownarrow_l^\tau e_4$ and $[e_1]_l \neq [e_2]_l$. Let $\prec_{hb}^\tau$ be the transitive closure of the union between happens-before and thread orderings of τ. The following holds:

Proposition 7. *Let τ' be a τ-feasible trace. Any linearization of $\prec_{hb}^{\tau'}$ consistent with the lock atomicity of τ' is τ-feasible.*

5.4 Weak-Happens-Before with Synchronization

We next present two approaches to handling synchronization in weak-happens-before models and show they are both embeddable in our model.

Lock Atomicity via Write-Read Atomicity [16]. Since the notion of write-read atomicity already allows atomic sets to be permuted, it seems reasonable to use the conservative idea from standard happens-before methods, and treat *acquire* as a *write* event and *release* as a read event. Formally, given the consistent trace τ, one could additionally introduce an atomic dependence relation $<_\tau^a$ given by $e_1 <_\tau^a e_2$ if $\tau = \tau_1 e_2 \tau_2 e_2 \tau_3$, $target(e_1) = target(e_2)$, $op(e_1) = acquire$, $op(e_2) = release$, and there is no event e in τ_2 such that $target(e) = target(e_1)$, and $op(e) = acquire$. With this definition, equivalent traces to an observed trace τ are those interleavings of τ having the same write-read and atomic dependencies.

However, this definition needs a careful approach. Consider the example in Figure 1(b), and suppose that we observe a similar execution, but that the program is stopped after the *read* of x in Thread 2. Since no *release* event has been generated, the *acquire* in Thread 2 has no event depending on it, and thus it can be permuted (without the *read* event on x it was supposed to protect) before the last lock-block of Thread 1. Then, the final *read* of x itself can be permuted past the final *release* of l in Thread 1, exhibiting a spurious causal datarace.

Nevertheless, these models are sound for *synchronization complete* traces, that is, traces in which each acquired lock is eventually released.

Proposition 8. *Let τ_1 be a synchronization complete τ-feasible trace. Any interleaving τ_2 of τ_1 satisfying that $<^{wr}_{\tau_2}=<^{wr}_{\tau_1}$ and $<^{a}_{\tau_2}=<^{a}_{\tau_1}$ is τ-feasible.*

Lock Atomicity via Locksets. Wang and Stoller [21] propose a weak-happens-before model based on write-read dependence, while using locksets to handle locks as individual objects. In this model, a trace τ' is equivalent with a consistent trace τ if τ' is an interleaving of τ having the same write-read dependence relation and being consistent with the lock atomicity of τ.

Proposition 9. *Let τ_1 be a τ-feasible trace. Any interleaving τ_2 of τ_1, consistent with the lock atomicity of τ_1 and satisfying that $<^{wr}_{\tau_2}=<^{wr}_{\tau_1}$ is τ-feasible.*

6 Related Work and Discussion

Beginning with the introduction of the Happens-Before ordering by Lamport [9], there has been a considerable amount of research on models and techniques to abstract executions for the purpose of inferring causally equivalent executions satisfying/violating particular but important properties, such as dataraces or atomicity/serializability [2,5,7,12,14–16,19–21]. Section 5 shows that the sound causal models upon which the above mentioned techniques were based [10,12,15,16,21] are subsumed by the maximal causal model; their soundness follows as a corollaries of Theorem 1.

Ganai and Gupta [6] apply a similar technique for software model checking, attempting to reduce the state space to be explored using sequential consistency constraints. Similarly, building on a previous draft of this paper, Said et al. [13] encode the axioms of our proposed model (extended with constructs for thread creation and wait/notify) into an SMT solver and use that to effectively search the model for potential dataraces in Java programs.

Another interesting and productive line of research attempts to use information about the actual program code to either statically detect potential bad behaviors [11], or to use information about the program and about the property to be checked to further relax the models of executions [3,20].

Adding or removing attributes from events. Our choice of which attributes to be included in an event was based on the idea of observing the execution of any

multithreaded program executed on any machine offering no guarantees other than sequential consistency. Therefore, no semantical information is assumed about the program other than the identity of the thread performing an operation. There are other possible choices, each with their benefits. For example, Sinha et al. [18] choose not to record the value read/written in the memory. Thus, their model must preserve the read-after-write dependence, and the set of comprised traces is thus comparable with that of Wand and Stoller [21], and Sen et al. [16]. We believe a similar maximality result could be proved for traces of this kind, but that has not been attempted yet. In contrast, Wang et al. [20] enrich the events with symbolic information extracted from the program executing them. This allows them to obtain more comprehensive models at the expense of having to analyze the code. Since analyzing the code statically leads quickly to undecidability issues, and thus static analyzers need to be conservative, we believe there might indeed be no similar maximality result for these types of models, their coverage increasing with the power of the analysis.

Causal properties of traces. Since our model associates for a trace all traces which can be obtained by all programs which can obtain that trace, this allows for program-independent definitions of causal properties. For example,Wang and Stoller [21] propose serializability of a trace τ as the property that there exists an alternative execution of the program producing an interleaving of τ in which each transaction is a sequential block. Farzan and Madhusudan [4] relax this constraint by requiring that for each transaction there exists an alternative execution of the program producing an interleaving of τ containing that transaction as a sequential block. Sen et al. [16] say that a trace exhibits a datarace if there exists an alternative execution of the program producing an (partial) interleaving of τ in which the conflicting events are consecutive.

The program-independent properties associated to any of the above (program-dependent) definitions can be obtained by simply replacing the (rather informal) "alternative execution of the program producing an interleaving of τ" with "a τ-feasible trace", as defined by Definition 3. Formal definitions of these causal properties can be found in the companion technical report [17].

7 Conclusion

We have shown that, by axiomatizing basic properties of (sequentially consistent) concurrent systems, one can obtain *maximally sound causal models* for concurrent executions, which can be naturally associated to each observed trace, capturing all feasible traces which could be inferred from it. The maximality result has two important theoretical implications. First, verifying the soundness claims for any causal model is reduced to proving that it is a submodel of the maximal one. Second, since the maximal model captures all causally equivalent traces, it allows for universal, program-independent definitions for causal properties. Although this paper focuses on proving the maximality claim of our model, the companion technical report [17] additionally provides a constructive characterization of the proposed model, as well as a model checking algorithm.

References

1. Attiya, H., Welch, J.L.: Sequential consistency versus linearizability. TOCS 12, 91–122 (1994)
2. Banerjee, U., Bliss, B., Ma, Z., Petersen, P.: A theory of data race detection. In: PADTAD 2006, pp. 69–78. ACM, New York (2006)
3. Chen, F., Roşu, G.: Parametric and Sliced Causality. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 240–253. Springer, Heidelberg (2007)
4. Farzan, A., Madhusudan, P.: Causal Atomicity. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 315–328. Springer, Heidelberg (2006)
5. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: POPL 2004, pp. 256–267 (2004)
6. Ganai, M.K., Gupta, A.: Efficient Modeling of Concurrent Systems in BMC. In: Havelund, K., Majumdar, R. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 114–133. Springer, Heidelberg (2008)
7. Helmbold, D.P., McDowell, C.E., Wang, J.Z.: Determining possible event orders by analyzing sequential traces. TPDS 4(7), 827–840 (1993)
8. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. TOPLAS 12, 463–492 (1990)
9. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess progranm. IEEE Trans. Comput. 28(9), 690–691 (1979)
10. Mazurkiewicz, A.: Trace Theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 279–324. Springer, Heidelberg (1987)
11. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for Java. In: PLDI 2006, pp. 308–319 (2006)
12. O'Callahan, R., Choi, J.-D.: Hybrid dynamic data race detection. SIGPLAN Not. 38(10), 167–178 (2003)
13. Said, M., Wang, C., Yang, Z., Sakallah, K.: Generating Data Race Witnesses by an SMT-Based Analysis. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 313–327. Springer, Heidelberg (2011)
14. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: a dynamic data race detector for multithreaded programs. TOCS 15(4), 391–411 (1997)
15. Schonberg, E.: On-the-fly detection of access anomalies. Best of PLDI 1979-1999 39, 313–327 (2004)
16. Sen, K., Roşu, G., Agha, G.: Detecting Errors in Multithreaded Programs by Generalized Predictive Analysis of Executions. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 211–226. Springer, Heidelberg (2005)
17. Şerbănuţă, T.F., Chen, F., Roşu, G.: Maximal causal models for sequentially consistent systems. Technical Report, University of Illinois at Urbana-Champaign (October 2011), http://hdl.handle.net/2142/27708
18. Sinha, A., Malik, S., Wang, C., Gupta, A.: Predictive analysis for detecting serializability violations through trace segmentation. In: MEMOCODE 2011 (2011)
19. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: POPL 2006, pp. 334–345 (2006)
20. Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic Predictive Analysis for Concurrent Programs. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 256–272. Springer, Heidelberg (2009)
21. Wang, L., Stoller, S.D.: Accurate and efficient runtime detection of atomicity errors in concurrent programs. In: PPoPP 2006, pp. 137–146 (2006)

Monitoring Compliance Policies over Incomplete and Disagreeing Logs*

David Basin, Felix Klaedtke, Srdjan Marinovic, and Eugen Zălinescu

Institute of Information Security, ETH Zurich

Abstract. When monitoring system behavior to check compliance against a given policy, one is sometimes confronted with incomplete knowledge about system events. In IT systems, such incompleteness may arise from logging infrastructure failures and corrupted log files, or when the logs produced by different system components disagree on whether actions took place. In this paper, we present a policy language with a three-valued semantics that allows one to explicitly reason about incomplete knowledge and handle disagreements. Furthermore, we present a monitoring algorithm for an expressive fragment of our policy language. We illustrate through examples how our approach extends compliance monitoring to systems with logging failures and disagreements.

1 Introduction

Laws, inter-business contracts, security policies, and similar normative regulations define compliance requirements that IT systems need to enforce. For example, IT systems in US hospitals must enforce HIPAA [1], which regulates the dissemination of medical records and the subsequent obligations that medical staff are expected to fulfill. For banks, separation-of-duty constraints should reduce the risk of fraud [2]. Data-usage contracts between different businesses regulate how sensitive documents are exchanged and subsequently disposed. Checking whether implemented IT systems comply with a body of regulations or policies is a problem of growing importance, since non-compliant behavior can lead to serious security breaches, monetary penalties, and the erosion of stakeholder's internal standards and commitments.

Runtime-verification techniques [4,5,19,22–24] offer a promising approach for automated compliance checking of IT systems. These techniques require logging mechanisms for recording policy-relevant system actions (represented as events), a suitable language for expressing policies and unambiguously defining permissible and prohibited system behavior, and a monitoring algorithm for determining and reporting policy violations.

In complex IT systems, which are usually composed of numerous interacting subsystems, the problem of incomplete knowledge about performed actions arises. In particular, logs may contain gaps due to corrupted files, logging-mechanism crashes, network failures, and so forth. Furthermore, when multiple

* This work was partly supported by Google Inc.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 151–167, 2013.
© Springer-Verlag Berlin Heidelberg 2013

logs are required to verify compliant behavior, they may disagree whether certain actions took place. For example, sharing a sensitive document between two parties may require the recipient to fulfill certain obligations. Thus, when analyzing the recipient's and the sender's logs against this policy, we need to treat all disagreements over the transfer of the document as incomplete knowledge, since favoring one log over the other may result in missed violations or false positives. Most runtime monitors, however, do not distinguish between a gap and a non-occurrence of an event. Thus applying them to incomplete logs can yield *wrong* results. For example, consider a policy like *a subject can access a document if the subject is not blacklisted*. If it is unknown whether a subject is blacklisted, then the subject is incorrectly reported as compliant.

In this paper, we present a policy language and an accompanying monitoring algorithm that accounts for possibly incomplete and disagreeing logs. At the core of our approach is a three-valued truth space [25]. In addition to the classical Boolean values t (true) and f (false), which respectively represent the occurrence and non-occurrence of an event, we represent a knowledge gap about an event's occurrence by the third truth value $\perp$. Furthermore, when evaluating policies, their interpretation is as follows: the Boolean values t and f correspond to policy compliance and policy violation and $\perp$ represents an inconclusive answer, which can be due to knowledge gaps of event occurrences or disagreeing events.

Our policy language is a variant of a first-order temporal logic [7,17]. First-order temporal logics have been a good fit in various case studies for formally expressing and monitoring compliance policies, see, e.g., [5,23]. Special care must be taken when defining the semantics of a logic with additional truth values besides the classical Boolean values. In particular, a vital requirement for monitoring incomplete and disagreeing logs is to ensure that reported violations cannot be retracted if or when the log is eventually completed, for example, by recovering lost files. Otherwise, these results are of no value. More precisely, formalized policies must be monotonic with respect to the underlying partial ordering on knowledge, i.e., $\perp$ is less than f and t, and f and t are incomparable [9,10,20]. Our policy language guarantees this monotonicity requirement. Furthermore, the third truth value $\perp$ is a first-class citizen at the object-level of our policy language: the classical logical connectives are extended to the three-valued truth space and there are specific connectives that guarantee expressive-completeness with respect to the set of knowledge-monotonic operators. Such monotonic operators are needed in our application context to express at the logic's object-level how disagreements between logged events should be resolved.

The monitoring algorithm presented in this paper for this three-valued setting is inspired by the one from [6,7] for the standard Boolean setting. It iteratively scans the logged actions and soundly reports violations, i.e., whenever a violation is reported, it indeed is a policy violation. It also soundly reports potential violations, i.e., depending on how the knowledge gaps are filled, these might turn out to be real policy violations. However, our monitoring algorithm is not complete in the sense that some policy violations might not be reported. This limitation stems from the expressivity of our policy language over infinite

domains. Importantly, however, for an expressive fragment, which retains all the language's connectives but limits the usage of free variables within a formula, we show that our monitoring algorithm guarantees completeness.

In summary, our main contribution is a solution to the problem of checking policy compliance in the presence of logging failures and disagreements between logged events. Our solution comprises a policy language and a monitoring algorithm. The policy language supports reasoning with incomplete knowledge. The monitoring algorithm may be used either off-line (for audit) or on-line (at runtime), and reports all policy violations and potential policy violations for an expressive fragment of our language. Although several features of our solution are present in related work—see Section 6 for a comparison—combining them to solve the stated problem is novel. In particular, our language is the first compliance language to consider three truth values at the object level, and our monitoring algorithm is the first algorithm to guarantee both soundness and completeness in a three-valued first-order setting.

The remainder of the paper is structured as follows. In Section 2, we describe our abstract logging setting. In Section 3, we introduce our policy language. In Section 4, we analyze our policy language with respect to monotonicity and expressiveness. In Section 5, we present our monitoring algorithm. Finally, in Sections 6 and 7, we discuss related work and draw conclusions. Technical details are omitted due to space limitations. These are given in the full version of the paper, which can be found on the authors' web pages.

2 Logging Knowledge Base

We abstract from a particular *physical* log file structure, and view a logging infrastructure as producing a single logging knowledge base, which is evaluated against a compliance policy. A logging knowledge base uses the three-valued truth space $\mathbf{3} := \{\mathsf{t}, \mathsf{f}, \bot\}$ to explicitly distinguish between what is known and unknown regarding event occurrences.

To formally define a logging knowledge base over $\mathbf{3}$, we introduce a *logging signature* S, which is a tuple (C, R, ι), where C is a finite set of constant symbols, R is a finite set of predicates disjoint from C, and the function $\iota : R \to \mathbb{N}$ assigns each predicate $r \in R$ an arity $\iota(r)$. Each predicate r denotes an action, and its arguments $\bar{a}$ denote the action's parameters, $r(\bar{a})$ denoting an event. A *logging structure* $\mathcal{D}$ over the signature S consists of a domain $|\mathcal{D}| \neq \emptyset$ and interpretations $c^{\mathcal{D}} \in |\mathcal{D}|$, and $r_{\mathsf{t}}^{\mathcal{D}} \subseteq |\mathcal{D}|^{\iota(r)}$ and $r_{\mathsf{f}}^{\mathcal{D}} \subseteq |\mathcal{D}|^{\iota(r)}$, for each $c \in C$ and $r \in R$, such that $r_{\mathsf{t}}^{\mathcal{D}}$ and $r_{\mathsf{f}}^{\mathcal{D}}$ are disjoint. We let $r_{\bot}^{\mathcal{D}} := |\mathcal{D}|^{\iota(r)} \setminus (r_{\mathsf{t}}^{\mathcal{D}} \cup r_{\mathsf{f}}^{\mathcal{D}})$. We define a *logging knowledge base* over the signature S as a sequence $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \dots)$ of logging structures over S, with the following properties:

1. $\bar{\mathcal{D}}$ has constant domains, that is, $|\mathcal{D}_i| = |\mathcal{D}_{i+1}|$, for all $i \geq 0$. We denote the domain by $|\bar{\mathcal{D}}|$.
2. Each constant symbol $c \in C$ has a rigid interpretation, that is, $c^{\mathcal{D}_i} = c^{\mathcal{D}_{i+1}}$, for all $i \geq 0$. We denote c's interpretation by $c^{\bar{\mathcal{D}}}$.

We call the indices of the elements in the sequence $\bar{\mathcal{D}}$ *time points* and denote them with the Greek letter τ. We interpret a logging knowledge base $\bar{\mathcal{D}}$ as follows:

– If $\bar{a} \in r_{\mathsf{t}}^{\mathcal{D}_\tau}$, then the event $r(\bar{a})$ happened at the time point τ.
– If $\bar{a} \in r_{\mathsf{f}}^{\mathcal{D}_\tau}$, then the event $r(\bar{a})$ did not happen at the time point τ.
– If $\bar{a} \in r_{\perp}^{\mathcal{D}_\tau}$, then $\bar{\mathcal{D}}$ contains a *knowledge gap* at the time point τ with regard to whether the event $r(\bar{a})$ happened at τ. In practice, a gap is determined by additional information about logging failures.

Thus a logging knowledge base states explicitly whether logging information is complete at a time point τ. In case of incomplete knowledge, we have $r_{\perp}^{\mathcal{D}_\tau} \neq \emptyset$.

We extend the classical logging assumption, whereby there are only finitely many events happening at each time point, to a three-valued setting.

Assumption 1. *Let $\bar{\mathcal{D}}$ be a logging knowledge base over the signature (C, R, ι). For each $r \in R$ and $\tau \in \mathbb{N}$, either $r_{\mathsf{t}}^{\mathcal{D}_\tau}$ is finite and $r_{\perp}^{\mathcal{D}_\tau} = \emptyset$, or $r_{\perp}^{\mathcal{D}_\tau} = |\bar{\mathcal{D}}|^{\iota(r)}$.*

This assumption formalizes that as long as a particular logging process is running, it correctly records all events. If the process crashes, then nothing is recorded until the process is restarted. In line with our model of a logging knowledge base, this means that at each time point τ and for each relation r either $r_{\perp}^{\mathcal{D}_\tau} = \emptyset$ or $r_{\perp}^{\mathcal{D}_\tau} = |\bar{\mathcal{D}}|^{\iota(r)}$.

Note that a logging knowledge base does not differentiate between multiple instances of the same event happening at the same time point. To do so, one would have to ensure that either the time points' granularity is sufficient to render this scenario impossible, or to add unique artificial parameters (such as counters) for each such event instance.

3 Compliance Policy Language

In this section, we define our policy language $\mathcal{L}_3$ and illustrate with examples how policies are formalized and evaluated in the presence of incomplete knowledge. We also show how disagreements can be handled with $\mathcal{L}_3$'s operators.

Syntax and Semantics. In the following, let $S = (C, R, \iota)$ be a signature and let V be a countably infinite set of variables, where $V \cap (C \cup R) = \emptyset$. Also, let $\mathbb{I}$ be the set of nonempty intervals over $\mathbb{N}$. We often write an interval in $\mathbb{I}$ as $[b, b') := \{a \in \mathbb{N} \mid b \leq a < b'\}$, where $b \in \mathbb{N}$, $b' \in \mathbb{N} \cup \{\infty\}$, and $b < b'$.

Definition 2. *The $\mathcal{L}_3$ formulas over the signature S are given by the grammar*

$$\varphi ::= \mathsf{f} \mid r(t_1, \ldots, t_{\iota(r)}) \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \otimes \varphi \mid \forall x.\, \varphi \mid \varphi \,\mathsf{S}_I\, \varphi \mid \varphi \,\mathsf{U}_I\, \varphi\,,$$

where r ranges over the elements in R, the t_is over the elements in $C \cup V$, x over the elements in V, and I over the elements in $\mathbb{I}$.

Before formally defining the evaluation semantics, Figure 1(a) shows $\mathcal{L}_3$'s interpretation of the logical connectives over $\mathbf{3}$. We mildly abuse notation and

$\neg$	
t	f
f	t
$\bot$	$\bot$

$\wedge$	t	f	$\bot$
t	t	f	$\bot$
f	f	f	f
$\bot$	$\bot$	f	$\bot$

$\otimes$	t	f	$\bot$
t	t	$\bot$	$\bot$
f	$\bot$	f	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$

(a) primitive operators

$\vee$	t	f	$\bot$
t	t	t	t
f	t	f	$\bot$
$\bot$	t	$\bot$	$\bot$

$\rightarrow$	t	f	$\bot$
t	t	f	$\bot$
f	t	t	t
$\bot$	t	$\bot$	$\bot$

(b) derived operators

Fig. 1. Truth tables for three-valued operators (strong Kleene logic [25])

use same symbols to denote logical connectives and their corresponding three-valued operators. The classical connectives $\neg$ and $\wedge$ retain their interpretation when restricted to the Boolean values t and f. The $\otimes$ connective does not have a classical counterpart. Intuitively, it represents a *consensus* on how much truth can be agreed upon and is useful for combining different sources of knowledge when neither t nor f should be preferred over the other.

In the following, a *valuation* is a mapping $\theta : V \to |\bar{\mathcal{D}}|$. For a valuation θ, the variable vector $\bar{x} = (x_1, \ldots, x_n)$, and $\bar{d} = (d_1, \ldots, d_n) \in |\bar{\mathcal{D}}|^n$, $\theta[\bar{x} \mapsto \bar{d}]$ is the valuation mapping x_i to d_i, for $i \in \{1, \ldots, n\}$, and the other variables' valuation is unaltered. We abuse notation by applying a valuation θ also to constant symbols $c \in C$, with $\theta(c) := c^{\bar{\mathcal{D}}}$.

Definition 3. *Let $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \ldots)$ be a temporal structure over the signature S, θ a valuation, and $\tau \in \mathbb{N}$ a time stamp. We inductively define the mapping $[\![\cdot]\!]^{\bar{\mathcal{D}},\theta,\tau}$ from formulas over S to values in $\mathbf{3}$ as follows:*

$$\begin{aligned}
[\![\mathsf{f}]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \mathsf{f} \\
[\![r(t_1, \ldots, t_{\iota(r)})]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= v \text{ if } \big(\theta(t_1), \ldots, \theta(t_{\iota(r)})\big) \in r_v^{\mathcal{D}_\tau}, \text{ where } v \in \mathbf{3} \\
[\![\neg\varphi]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \neg[\![\varphi]\!]^{\bar{\mathcal{D}},\theta,\tau} \\
[\![\varphi_1 \wedge \varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= [\![\varphi_1]\!]^{\bar{\mathcal{D}},\theta,\tau} \wedge [\![\varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} \\
[\![\varphi_1 \otimes \varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= [\![\varphi_1]\!]^{\bar{\mathcal{D}},\theta,\tau} \otimes [\![\varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} \\
[\![\forall x.\, \varphi]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \textstyle\bigwedge_{d \in |\bar{\mathcal{D}}|} [\![\varphi]\!]^{\bar{\mathcal{D}},\theta[x \mapsto d],\tau} \\
[\![\varphi_1 \,\mathsf{S}_I\, \varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \textstyle\bigvee_{\tau - \tau' \in I} \big([\![\varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau'} \wedge \bigwedge_{\tau'' \in (\tau', \tau]} [\![\varphi_1]\!]^{\bar{\mathcal{D}},\theta,\tau''}\big) \\
[\![\varphi_1 \,\mathsf{U}_I\, \varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau} &:= \textstyle\bigvee_{\tau' - \tau \in I} \big([\![\varphi_2]\!]^{\bar{\mathcal{D}},\theta,\tau'} \wedge \bigwedge_{\tau'' \in [\tau, \tau')} [\![\varphi_1]\!]^{\bar{\mathcal{D}},\theta,\tau''}\big)
\end{aligned}$$

In this definition, $\bigwedge$ and $\bigvee$ are respectively the (possibly infinitary) meet and join over the ordering $\mathsf{f} \leq \bot \leq \mathsf{t}$. Note that they match the corresponding operators in Figure 1. The temporal connectives are accompanied by intervals and a formula of the form $\varphi \,\mathsf{S}_I\, \psi$ or $\varphi \,\mathsf{U}_I\, \psi$ is only satisfied in $\bar{\mathcal{D}}$ at the time point τ if it is satisfied within the bounds given by the interval I of the respective temporal operator. We may omit the interval I if it is $[0, \infty)$.

We introduce the following additional syntactic sugar. We write t for $\neg\mathsf{f}$, $\varphi \vee \psi$ for $\neg(\neg\varphi \wedge \neg\psi)$, $\varphi \to \psi$ for $\neg\varphi \vee \psi$, and $\exists x.\, \varphi$ for $\neg\forall x.\, \neg\varphi$. For a vector of variables $\bar{x} = (x_1, x_2, \ldots, x_n)$, with $n \geq 0$, we write $\forall\bar{x}.\, \varphi$ for $\forall x_1.\, \forall x_2 \ldots \forall x_n.\, \varphi$. Moreover, we define the temporal connectives $\blacklozenge_I\, \psi$ and $\lozenge_I\, \psi$ as $\mathsf{t}\,\mathsf{S}_I\, \psi$ and $\mathsf{t}\,\mathsf{U}_I\, \psi$, respectively. Intuitively, $\blacklozenge_{[b,b')}\, \psi$ is t at τ, if ψ is t at least at one *past* time point in the time interval $[\max(0, \tau - b' - 1), \tau - b]$. If ψ is f at all these

time points, then $\blacklozenge_{[b,b')}\,\psi$ is f at τ. The presence of at least one $\bot$ and no t results in the truth value $\bot$ for $\blacklozenge_{[b,b')}\,\psi$ at τ, since depending on how the incompleteness is resolved either outcome (t or f) is possible. The interpretation of $\Diamond_{[b,b')}\,\psi$ is similar for *future* time points. The dual temporal connectives are $\Box_I\,\psi := \neg\Diamond_I\,\neg\psi$ and $\blacksquare_I\,\psi := \neg\blacklozenge_I\,\neg\psi$. We use standard conventions concerning the binding strength of connectives to omit parentheses. For instance, temporal connectives bind weaker than the other connectives. Furthermore, $\rightarrow$ binds weaker than $\vee$, which in turn binds weaker than $\wedge$ and $\otimes$.

Finally, we introduce some additional notation. Given a formula φ, we denote by $fv(\varphi)$ and $\bar{fv}(\varphi)$ the set and respectively the vector of free variables of φ. We call a formula φ *closed* if $fv(\varphi) = \emptyset$. For a formula φ with $\bar{fv}(\varphi) = \bar{x} = (x_1, \ldots, x_n)$, we define the set of elements of $|\bar{\mathcal{D}}|^n$ for which φ evaluates to $v \in \mathbf{3}$ at a time point $\tau \in \mathbb{N}$ as

$$[\![\varphi]\!]_v^{\bar{\mathcal{D}},\tau} := \left\{\bar{d} \in |\bar{\mathcal{D}}|^n \;\middle|\; [\![\varphi]\!]^{\bar{\mathcal{D}},\theta[\bar{x}\mapsto\bar{d}],\tau} = v, \text{ for some valuation } \theta\right\}.$$

Compliance Policies. Regardless of the policy language, compliance policies are usually given as a set of regulative *normative* statements (norms), which expess what an agent is obliged to do given some actions it has performed, or which conditions need to hold (or to have held) for an agent to be permitted to execute some actions. Norms are meant to be applied at all times within a system, and it has also been argued [11, 12] that deadlines are of essential importance in regulating temporal norms. Following these notions, compliance policies in $\mathcal{L}_3$ are formalized as follows:

Definition 4. *A* compliance policy *represented in $\mathcal{L}_3$ is a closed formula of the form $\Box\,\forall\bar{x}.\,\psi$, where each future temporal connective in ψ is bounded.*

The outermost unbounded $\Box$ connective specifies that a policy must be fulfilled at each time point. Bounded inner future temporal connectives guarantee that each obligation has a deadline.

We map the truth values onto policy evaluations as follows: t/f denotes that a policy is satisfied/violated, and $\bot$ denotes that it is unknown whether a policy is satisfied or violated. Furthermore, for a compliance policy $\Box\,\forall\bar{x}.\,\psi$, it is often useful to report additional information regarding its violations, which is given by the aforementioned sets $[\![\psi]\!]_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$, $[\![\psi]\!]_{\bot}^{\bar{\mathcal{D}},\tau}$, and $[\![\psi]\!]_{\mathsf{t}}^{\bar{\mathcal{D}},\tau}$, for a time point τ. Their interpretation is as follows:

- The elements in $[\![\psi]\!]_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$ witness a policy violation at time point τ.
- For elements in $[\![\psi]\!]_{\bot}^{\bar{\mathcal{D}},\tau}$, it is unknown whether they violate the policy at time point τ. They are potential violations.
- The elements in $[\![\psi]\!]_{\mathsf{t}}^{\bar{\mathcal{D}},\tau}$ satisfy the policy at time point τ.

In Section 4, we show that all reported violations and satisfactions at τ persist regardless of how incompleteness is resolved.

Examples. We begin with the following security policy requiring that *if a request is serviced at a web-server then it must not have been denied by a firewall.*

In practice, this policy would be a part of a larger specification. However, this excerpt is enough to illustrate how $\mathcal{L}_3$'s semantics deal with logging failures. We formalize this policy as $\Box \forall r.\, \psi_1$, where

$$\psi_1 := \mathit{service}(r) \to \neg \blacklozenge_{[0,4)}\, \mathit{deny}(r)\,.$$

When there are no failures, then any serviced request that has previously been denied violates the policy, and is contained in $[\![\psi_1]\!]_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$. If the web-server's logger crashes at a time point τ, i.e. $\mathit{service}_{\perp}^{\mathcal{D}_\tau} = |\bar{\mathcal{D}}|$, then all requests that had been denied at the previous four time points by the firewall potentially violate the policy, i.e. $[\![\psi_1]\!]_{\perp}^{\bar{\mathcal{D}},\tau} = \bigcup_{\tau'} \mathit{deny}_{\mathsf{t}}^{\mathcal{D}_{\tau'}}$, where $\tau - 4 < \tau' \leq \tau$. If, however, there are no denied requests in the designated interval, the set $[\![\psi_1]\!]_{\perp}^{\bar{\mathcal{D}},\tau}$ is empty and the policy is therefore satisfied. This shows that not all logging failures must result in potential violations. We note that if all unknown events are treated as not to have happened, then the policy would be *wrongly* reported as satisfied.

For our second example, we focus on formalizing inter-business contracts. These contracts often specify obligations that the signing parties must enforce regarding the treatment of sensitive documents used during the collaborations. To ensure that each party complies with its obligations, a policy must specify how events are combined from different logs belonging to different stakeholders. For example, when two companies exchange sensitive information, the contract might say that *all received documents must be paid for within 5 days.* A straightforward, but naive, formalization of this policy is $\Box \forall d.\, \psi_2$, where

$$\psi_2 := \mathit{receive}(d) \to \Diamond_{[0,6)}\, \mathit{pay}(d)\,.$$

The *receive* event is taken from the receiving stakeholder's log. This specification assumes that the receiving stakeholder is honest, since if its IT system does not log a received document, the stakeholder's behavior is trivially compliant according to the given specification. We can attempt to expand the formalization to include the sender's *send* event (from the sender's log) as follows

$$\psi_2' := \mathit{send}(d) \vee \mathit{receive}(d) \to \Diamond_{[0,6)}\, \mathit{pay}(d)\,.$$

In this case, the receiver is obliged to pay if either it receives a document, or the sender says that it has sent the document. However, this is also unsatisfactory, as the sender can cheat and insert fictitious *send* events causing policy violations. In $\mathcal{L}_3$ we can combine the logs with the $\otimes$ operator and obtain[1]

$$\psi_2'' := \mathit{send}(d) \otimes \mathit{receive}(d) \to \Diamond_{[0,6)}\, \mathit{pay}(d)\,.$$

[1] We assume that the time granularity is coarse enough to allow *receive* and *send* happen at the same time point. If a *receive* can happen with a delay of, e.g., at most one time unit after a *send*, a more elaborate formalization is required:

$$\begin{aligned}\Box \forall d.\, \big(&\mathit{send}(d) \wedge (\mathit{send}(d) \otimes \Diamond_{[0,2)}\, \mathit{receive}(d)) \to \Diamond_{[0,6)}\, \mathit{pay}(d)\big) \wedge \\ &\big(\mathit{receive}(d) \wedge (\mathit{receive}(d) \otimes \blacklozenge_{[0,2)}\, \mathit{send}(d)) \to \Diamond_{[0,5)}\, \mathit{pay}(d)\big)\,.\end{aligned}$$

In this case, all disagreements at some τ about payments are in $[\![\psi_2'']\!]_{\perp}^{\bar{\mathcal{D}},\tau}$, since $\perp \to \mathsf{f}$ is $\perp$. The specification no longer favors one stakeholder over the other. This has the benefit of not requiring additional pre-processing of logs, which would need its own language and semantics. We remark that the given specification cannot be directly expressed in existing compliance policy languages because $\perp$ does not exist at the object level in those languages.

For our third example, we consider a form of separation-of-duty constraint [2]: *a subject s may access an object o if it has not previously accessed some object o', where o''s dataset conflicts with o's.* One possible formalization of this requirement is

$$\square\, \forall s.\, \forall o.\, \forall d.\, \forall o'.\, \forall d'.\, \mathit{access}(s,o,d) \land (\blacklozenge\, \mathit{access}(s,o',d')) \to \neg \mathit{conflict}(d,d')\,.$$

In this example, $\mathit{access}(s,o,d)$ records that s accessed o in a dataset d. The predicate *conflict* does not correspond to an event; it describes a property of a system state. When having the events $\mathit{conflict}_s$ and $\mathit{conflict}_f$ at hand, which mark the start point and the end point of two datasets being conflicting, the formula $\neg \mathit{conflict}_f(d,d')\,\mathsf{S}\,\mathit{conflict}_s(d,d')$ can be used to describe this state property. For the sake of brevity, we assume that an object belongs to at most one dataset. In case s accessed an o, and it is unknown whether s had any other accesses, then if there exists d' in conflict with d, such an access is a potential violation.

Notice that the above formalization only considers whether the data items are in conflict at the time point when o is accessed. This means that even if the datasets are in conflict just before the access, the policy is not violated. With respect to the separation-of-duty requirement, one may say that this behavior is in a compliance *gray* area. In $\mathcal{L}_3$, we define the following temporal connective C_I that treats such *gray* areas as $\perp$, signaling that it is unclear whether the policy is satisfied or violated:

$$\mathsf{C}_I\psi := (\blacklozenge_I\, \psi) \otimes (\blacksquare_I\, \psi)\,.$$

Intuitively, $\mathsf{C}_I\psi$ insists that the truth value of ψ does not change in the given past interval I. Any change results in $\perp$, and otherwise the truth value is not changed. We can define a similar temporal connective using $\square$ and $\lozenge$ to mark a future gray zone. We make use of C_I by changing the original formalization to

$$\square\, \forall s.\, \forall o.\, \forall d.\, \forall o'.\, \forall d'.\, \mathit{access}(s,o,d) \land (\blacklozenge\, \mathit{access}(s,o',d')) \to \mathsf{C}_{[0,2)} \neg \mathit{conflict}(d,d')\,,$$

where $[0,2)$ is a two-day *gray* zone interval.

4 Monotonicity and Compositional Expressiveness

A logging knowledge base may grow in knowledge by resolving missing information about the occurrences and non-occurrences of events, i.e., moving elements from $r_{\perp}^{\mathcal{D}_\tau}$ to the relations $r_{\mathsf{t}}^{\mathcal{D}_\tau}$ or $r_{\mathsf{f}}^{\mathcal{D}_\tau}$.

Definition 5. *An* extension *of a logging knowledge base $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \dots)$ over $S = (C, R, \iota)$ is a logging knowledge base $\bar{\mathcal{D}}^\star = (\mathcal{D}_0^\star, \mathcal{D}_1^\star, \dots)$ over S with $|\bar{\mathcal{D}}^\star| = |\bar{\mathcal{D}}|$, $c^{\bar{\mathcal{D}}} = c^{\bar{\mathcal{D}}^\star}$ for all $c \in C$, and $r_b^{\mathcal{D}_\tau} \subseteq r_b^{\mathcal{D}_\tau^\star}$ for all $b \in \{\mathsf{t}, \mathsf{f}\}$, $\tau \in \mathbb{N}$, and $r \in R$.*

Under Assumption 1, an extension either does not alter a relation $r_{\perp}^{\mathcal{D}_\tau}$ or empties $r_{\perp}^{\mathcal{D}_\tau}$ by moving finitely many elements to $r_{\mathsf{t}}^{\mathcal{D}_\tau}$ and the remaining elements to $r_{\mathsf{f}}^{\mathcal{D}_\tau}$.

We say that a policy specification is *monotonic* if the t and f evaluations, over a given logging knowledge base, can never be retracted for any of its extensions. In other words, regardless of how the logging base's incompleteness is resolved, the policy violations and satisfactions persist. Monotonicity is a vital requirement for a compliance policy, because monotonic specifications prevent a non-compliant behavior from being turned into a compliant behavior by holding back information. In the following, we establish that for $\mathcal{L}_3$ all policy specifications are monotonic by construction. To formalize monotonicity, we first order the truth values with a partial ordering $\leq_k$ as follows: $\perp \leq_k \mathsf{f}$, $\perp \leq_k \mathsf{t}$, and f and t are incomparable. In short, f and t contain more knowledge than $\perp$. The following theorem states that the evaluations of $\mathcal{L}_3$'s formulas do not reduce the amount of knowledge, when incompleteness is resolved in a logging knowledge base's extension.

Theorem 6. *Given an $\mathcal{L}_3$ formula ψ, a valuation θ, and a logging knowledge base $\bar{\mathcal{D}}$, then $[\![\psi]\!]^{\bar{\mathcal{D}},\theta,\tau} \leq_k [\![\psi]\!]^{\bar{\mathcal{D}}^\star,\theta,\tau}$, for all extensions $\bar{\mathcal{D}}^\star$ of $\bar{\mathcal{D}}$ and all $\tau \in \mathbb{N}$.*

Proof. From the definition of a logging knowledge base's extension, and by structural induction using the fact that all of $\mathcal{L}_3$ connectives' corresponding operators are $\leq_k$-monotonic, including the infinitary operators for temporal connectives.

As a corollary, given a compliance policy $\Box\forall\bar{x}.\,\psi$, a logging knowledge base $\bar{\mathcal{D}}$, a valuation θ, and a time point τ, if $[\![\psi]\!]^{\bar{\mathcal{D}},\theta,\tau}$ is t or f, then this evaluation persists at τ, for all extensions $\bar{\mathcal{D}}^\star$. Moreover, we have $[\![\psi]\!]_{\mathsf{f}}^{\bar{\mathcal{D}}^\star,\tau} \supseteq [\![\psi]\!]_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$ and $[\![\psi]\!]_{\mathsf{t}}^{\bar{\mathcal{D}}^\star,\tau} \supseteq [\![\psi]\!]_{\mathsf{t}}^{\bar{\mathcal{D}},\tau}$, for all extensions $\bar{\mathcal{D}}^\star$ and $\tau \in \mathbb{N}$. Therefore, even with incomplete knowledge it is sound to report the elements in $[\![\psi]\!]_{\mathsf{f}}^{\bar{\mathcal{D}},\tau}$ as policy violations when monitoring $\bar{\mathcal{D}}$.

Given that all $\mathcal{L}_3$ policies are monotonic, an important question is: *Can all monotonic compositional operators for combining events from different logs be defined as syntactic sugar in $\mathcal{L}_3$?* If the answer is positive, then $\mathcal{L}_3$ does not need to be further extended. An n-ary three-valued operator $O : \mathbf{3}^n \to \mathbf{3}$ is *representable* using a set $\mathcal{C}$ of operators if O can be written as the functional composition of operators in $\mathcal{C}$. We utilize the following theorem to show that any monotonic operator can be expressed in $\mathcal{L}_3$.

Theorem 7 (Blamey [10]). *For any $n \in \mathbb{N}$, every $\leq_k$-monotonic n-ary operator over the $\mathbf{3}$ truth space is representable using the set $\{\mathsf{f}, \neg, \wedge, \otimes\}$ of operators.*

Blamey's proof is constructive and yields a function that given a monotonic operator produces an expression showing how to compose the operators f, $\neg$, $\wedge$, and $\otimes$. As $\mathcal{L}_3$ has all the corresponding connectives, such an expression can be seen as a formula in $\mathcal{L}_3$. Hence $\mathcal{L}_3$ can express any n-ary three-valued $\leq_k$-monotonic operator, including those for combining different logs.

5 Monitoring Algorithm

The input of our algorithm consists of a compliance policy $\Box\,\forall\bar{x}.\,\varphi$ and a logging knowledge base $\bar{\mathcal{D}}$ over a signature $S=(C,R,\iota)$. The algorithm iteratively processes the logging structures $\mathcal{D}_\tau$, for each $\tau\in\mathbb{N}$. To process a structure $\mathcal{D}_\tau$ for formulas with bounded future operators, the algorithm might need to process structures $\mathcal{D}_{\tau'}$ with $\tau'>\tau$ as well. When run in the *on-line* mode, the algorithm waits until such structures become available. For the rest of this section, we fix the signature S, the logging knowledge base $\bar{\mathcal{D}}$, and the policy $\Box\,\forall\bar{x}.\varphi$. Furthermore, we assume that the domain $|\bar{\mathcal{D}}|$ is infinite.

At each iteration τ, the algorithm outputs a triple $(S^\tau_{\mathsf{t}}, S^\tau_{\mathsf{f}}, S^\tau_{\perp})$, where for each $v\in\mathbf{3}$, the element S^τ_v is either $\mathsf{Fin}\ V$, CoFin, or None, where Fin, CoFin, and None are labels standing respectively for "finite set", "cofinite set", and "inconclusive", and V is a finite set.

Our algorithm is sound, i.e. if $S^\tau_v=\mathsf{Fin}\ V$ then $V=[\![\varphi]\!]^{\bar{\mathcal{D}},\tau}_v$, for all $v\in\mathbf{3}$ and $\tau\in\mathbb{N}$. However, our algorithm is not *complete*, where completeness means that the algorithm always returns a value from which one can deduce all compliant tuples ($[\![\varphi]\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{t}}$), all violations ($[\![\varphi]\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{f}}$), and all potential violations ($[\![\varphi]\!]^{\bar{\mathcal{D}},\tau}_{\perp}$). Note that when φ has free variables, all these sets cannot be explicitly output, as at least one is infinite. However, if two sets are finite, then the third one is cofinite, and it is thus implicitly determined. Therefore our algorithm is complete when at least two of the elements of the returned triples are of the form $\mathsf{Fin}\ V$. When φ is closed, completeness means that at each iteration a truth value is returned, as the triples $(\mathsf{Fin}\ \{()\}, \mathsf{Fin}\ \emptyset, \mathsf{Fin}\ \emptyset)$, $(\mathsf{Fin}\ \emptyset, \mathsf{Fin}\ \{()\}, \mathsf{Fin}\ \emptyset)$, and $(\mathsf{Fin}\ \emptyset, \mathsf{Fin}\ \emptyset, \mathsf{Fin}\ \{()\})$ correspond respectively with the truth values t, f, and $\perp$.

Incompleteness of our algorithm is rooted in the standard issues that arise when dealing with infinite domains [3], which $\mathcal{L}_3$ inherits from first-order queries in the Boolean setting. Consider for instance the formula $\psi=p(x)\vee q(y)$ with $x\neq y$ and assume that $p^{\mathcal{D}_\tau}_{\mathsf{t}}$ and $q^{\mathcal{D}_\tau}_{\mathsf{t}}$ are finite and non-empty, and $p^{\mathcal{D}_\tau}_{\perp}=q^{\mathcal{D}_\tau}_{\perp}=\emptyset$, for some $\tau\in\mathbb{N}$. Then $[\![\psi]\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{t}}$ and $[\![\psi]\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{f}}$ are neither finite nor cofinite, hence our algorithm cannot deal with it: at τ, it returns $(\mathsf{None}, \mathsf{None}, \mathsf{Fin}\ \emptyset)$. Formulas such as ψ are problematic in the Boolean setting, since their evaluation results are domain-dependent [3]. In the three-valued setting, there are similar issues, even for formulas that are non-problematic in the Boolean setting. Consider the formula $\psi'=p(x)\wedge q(y)$ with $p^{\mathcal{D}_\tau}_{\mathsf{t}}$ finite and non-empty and $q^{\mathcal{D}_\tau}_{\perp}=|\bar{\mathcal{D}}|$, for some $\tau\in\mathbb{N}$. Then both $[\![\psi']\!]^{\bar{\mathcal{D}},\tau}_{\mathsf{t}}$ and $[\![\psi']\!]^{\bar{\mathcal{D}},\tau}_{\perp}$ are infinite and domain-dependent.

Even though the algorithm is incomplete on $\mathcal{L}_3$, we obtain completeness for a fragment of $\mathcal{L}_3$, presented at the end of this section.

Algorithmic Overview. We briefly describe the main ideas underlying the algorithm. Due to space constraints, a detailed presentation is deferred to the full paper.

The algorithm's core is the procedure eval, whose arguments are a formula ψ, a finite set $\Gamma=\{(r,E_r)\mid r\in R\}$ representing the relations of the logging structure $\mathcal{D}_\tau$, and a time point τ. The values E_r, i.e., the second component of

```
proc init(φ)
  for each ψ ∈ sf(φ) with ψ = ψ S_I ψ' do
    L_ψ ← ⟨⟩

proc eval(φ, Γ, τ)
  case φ = f
    return (Fin ∅, Fin {()}, Fin ∅)

  case φ = r(t̄)
    E_r ← get_value(r, Γ)
    return eval_predicate(φ, E_r)

  case φ = ¬ψ
    return eval_neg(eval(ψ, Γ, τ))

  case φ = ψ ∧ ψ'
    E_ψ ← (ψ, eval(ψ, Γ, τ))
    E_ψ' ← (ψ', eval(ψ', Γ, τ))
    return eval_and(E_ψ, E_ψ')

  case φ = ψ ⊗ ψ'
    E_ψ ← (ψ, eval(ψ, Γ, τ))
    E_ψ' ← (ψ', eval(ψ', Γ, τ))
    return eval_times(E_ψ, E_ψ')

  case φ = ∀x̄.ψ
    E_ψ ← eval(ψ, Γ, τ)
    return eval_forall(x̄, ψ, E_ψ)

  case φ = ψ S_I ψ'
    E_ψ ← eval(ψ, Γ, τ)
    E_ψ' ← eval(ψ', Γ, τ)
    return eval_since(φ, τ, E_ψ, E_ψ')
```

Fig. 2. The init and eval procedures

elements in Γ, as well as the return value of the eval procedure, are triples of the form $(S_{\mathsf{t}}, S_{\mathsf{f}}, S_{\perp})$, where each S_v with $v \in \mathbf{3}$ is either Fin V, CoFin, or None. Such values satisfy (either by Assumption 1 or by construction) the following invariant with regard to some formula γ and time point τ: if $S_v = \mathsf{Fin}\ V$, then $[\![\gamma]\!]_v^{\bar{\mathcal{D}},\tau}$ is a finite subset of $|\bar{\mathcal{D}}|^{|fv(\gamma)|}$ and $V = [\![\gamma]\!]_v^{\bar{\mathcal{D}},\tau}$; if $S_v = \mathsf{CoFin}$, then $[\![\gamma]\!]_v^{\bar{\mathcal{D}},\tau}$ is a cofinite subset of $|\bar{\mathcal{D}}|^{|fv(\gamma)|}$ and the other two elements of the triple are of the form $S_{v'} = \mathsf{Fin}\ V'$, for $v' \in \mathbf{3} \setminus \{v\}$. This invariant is denoted as $Inv(\gamma, \tau, E)$, where $E = (S_{\mathsf{t}}, S_{\mathsf{f}}, S_{\perp})$. By Assumption 1, the values E_r from the set Γ satisfy the invariant $Inv(r(\bar{x}), \tau, E_r)$, where $\bar{x}$ is a sequence of distinct variables of length $\iota(r)$. We prove in Theorem 9 that the return value E of eval(φ, Γ, τ) satisfies the invariant $Inv(\varphi, \tau, E)$, thus establishing the correctness of our algorithm.

The eval procedure, given in Figure 2, is called recursively over ψ's subformulas. The procedure performs a case distinction on all possible top-level connectives. Some of the sub-procedures used by eval are in Figure 3, while the remaining the pseudo-code is given in the full paper.

Next, we sketch each case of the eval procedure. The simplest case is when ψ is the truth value f. In this case we simply return the triple (Fin $\emptyset$, Fin $\{()\}$, Fin $\emptyset$). When ψ is of the form $r(\bar{t})$ for some predicate r, we first retrieve the value E_r associated with r from the set Γ of pairs. We then retrieve the sets $[\![r(\bar{t})]\!]_v^{\bar{\mathcal{D}},\tau}$ from $r_v^{\mathcal{D}_\tau}$, for each $v \in \mathbf{3}$, by filtering the relations $r_v^{\mathcal{D}_\tau}$ according to the implicit constraints present in the sequence $\bar{t}$ of constants and variables.

To evaluate formulas ψ whose top-most connective is a non-temporal connective, we first evaluate the direct sub-formulas of ψ and then compute, whenever possible, the sets $[\![\psi]\!]_v^{\bar{\mathcal{D}},\tau}$ for $v \in \mathbf{3}$, using the equalities given in Lemma 8 below. These equalities extend the standard equalities that express the relationship between first-order logic and relational algebra, from the Boolean to the three-valued setting. They use the relational algebra operators *projection* and *join* [3]. We refer to the full paper for their formal definitions, and here we proceed with their intuitive description. As the temporal aspect is not relevant in this case of

eval, we also fix the time point τ and drop the superscript in $[\![\psi]\!]_v^{\bar{\mathcal{D}},\tau}$, i.e., we just write $[\![\psi]\!]_v$, for $v \in \mathbf{3}$ and a formula ψ.

Given a formula ψ and a truth value $v \in \mathbf{3}$, we can see the set $[\![\psi]\!]_v$ as a *named relation*, where columns in $[\![\psi]\!]_v$ are named by the free variables in $\bar{fv}(\psi)$. Given a free variable x of ψ, the *projection* of the tuples in the relation $[\![\psi]\!]_v$ on the columns corresponding to other free variables is denoted $\pi_x([\![\psi]\!]_v)$. For instance, if $[\![p(x,y)]\!]_{\mathsf{t}} = \{(0,2),(1,2),(1,3)\}$, then $\pi_x([\![p(x,y)]\!]_{\mathsf{t}}) = \{(2),(3)\}$. For $v, v' \in \mathbf{3}$, the *natural join* of the sets $[\![\psi]\!]_v$ and $[\![\psi']\!]_{v'}$, denoted $[\![\psi]\!]_v \bowtie [\![\psi']\!]_{v'}$, is the set of tuples for which the projections on the columns, corresponding to ψ's and ψ''s free variables, are in $[\![\psi]\!]_v$ and respectively in $[\![\psi']\!]_{v'}$, and the fields of which match on the common free variables. For instance, if $[\![q(y,z)]\!]_{\mathsf{t}} = \{(2,4)\}$, then $[\![p(x,y)]\!]_{\mathsf{t}} \bowtie [\![q(y,z)]\!]_{\mathsf{t}} = \{(0,2,4),(1,2,4)\}$. We adopt the convention that $\bowtie$ binds stronger than $\cup$.

Lemma 8. *Let $\bar{\mathcal{D}}$ be a logging knowledge base, τ be a time point, and ψ and ψ' be $\mathcal{L}_3$ formulas. The following equalities hold:*

$$\begin{aligned}
[\![\neg\psi]\!]_v &= [\![\psi]\!]_{\neg v}, \textit{ if } v \in \mathbf{3}\\
[\![\psi \wedge \psi']\!]_{\mathsf{t}} &= [\![\psi]\!]_{\mathsf{t}} \bowtie [\![\psi']\!]_{\mathsf{t}}\\
[\![\psi \wedge \psi']\!]_{\mathsf{f}} &= [\![\psi]\!]_{\mathsf{f}} \cup [\![\psi']\!]_{\mathsf{f}}, \textit{ if } fv(\psi) = fv(\psi')\\
[\![\psi \wedge \psi']\!]_{\perp} &= [\![\psi]\!]_{\mathsf{t}} \bowtie [\![\psi']\!]_{\perp} \cup [\![\psi]\!]_{\perp} \bowtie [\![\psi']\!]_{\mathsf{t}} \cup [\![\psi]\!]_{\perp} \bowtie [\![\psi']\!]_{\perp}\\
[\![\psi \otimes \psi']\!]_b &= [\![\psi]\!]_b \bowtie [\![\psi']\!]_b, \textit{ if } b \in \{\mathsf{t},\mathsf{f}\}\\
[\![\psi \otimes \psi']\!]_{\perp} &= [\![\psi]\!]_{\perp} \bowtie [\![\psi']\!]_{\perp} \cup [\![\psi]\!]_{\mathsf{t}} \bowtie [\![\psi']\!]_{\mathsf{f}} \cup [\![\psi]\!]_{\mathsf{f}} \bowtie [\![\psi']\!]_{\mathsf{t}}\\
[\![\forall x.\, \psi]\!]_{\mathsf{t}} &= \emptyset, \textit{ if } [\![\psi]\!]_{\mathsf{t}} \textit{ is finite and } x \in fv(\psi)\\
[\![\forall x.\, \psi]\!]_{\mathsf{f}} &= \pi_x([\![\psi]\!]_{\mathsf{f}}), \textit{ if } x \in fv(\psi)\\
[\![\forall x.\, \psi]\!]_{\perp} &= \pi_x([\![\psi]\!]_{\perp}) \setminus \pi_x([\![\psi]\!]_{\mathsf{f}}), \textit{ if } x \in fv(\psi)
\end{aligned}$$

These equalities provide a method to compute, under the stated conditions, the relations $[\![\psi]\!]_v$ from the corresponding relations for ψ's direct sub-formulas. For instance, if $\psi = \psi_1 \wedge \psi_2$ and $[\![\psi_1]\!]_{\mathsf{t}}$, $[\![\psi_2]\!]_{\mathsf{t}}$ are finite relations, then $[\![\psi]\!]_{\mathsf{t}}$ is a finite relation given by the join of the other two relations. Furthermore, when $[\![\psi_1]\!]_{\mathsf{t}}$ is finite, $[\![\psi_2]\!]_{\mathsf{t}}$ is cofinite, and $fv(\psi_2) \subseteq fv(\psi_1)$, then $[\![\psi]\!]_{\mathsf{t}}$ is a finite relation that we can compute as $[\![\psi_1]\!]_{\mathsf{t}} \bowtie [\![\psi_2]\!]_{\mathsf{t}} = [\![\psi_1]\!]_{\mathsf{t}} \bowtie \big(|\bar{\mathcal{D}}|^{|fv(\psi_2)|} \setminus ([\![\psi_2]\!]_{\mathsf{f}} \cup [\![\psi_2]\!]_{\perp})\big)$. Note that the condition $fv(\psi_2) \subseteq fv(\psi_1)$ is essential, as otherwise $[\![\psi_1]\!]_{\mathsf{t}} \bowtie [\![\psi_2]\!]_{\mathsf{t}}$ may be infinite. For example, if $\bar{fv}(\psi_1) = (x)$ and $\bar{fv}(\psi_2) = (x,y)$ with $[\![\psi_1]\!]_{\mathsf{t}} = \{(1)\}$, $[\![\psi_2]\!]_{\mathsf{f}} = \{(1,2)\}$, and $[\![\psi_2]\!]_{\perp} = \{(3,4)\}$, then $[\![\psi]\!]_{\mathsf{t}} = \{1\} \times (|\bar{\mathcal{D}}| \setminus \{2,4\})$. The same method is applied to each of the other sub-cases of the binary connectives.

The described approach is implemented through the procedures $\mathsf{eval_neg}$, $\mathsf{eval_and}$, $\mathsf{eval_times}$, and $\mathsf{eval_forall}$, given in Figure 3. Each procedure returns a triple $(R_{\mathsf{t}}, R_{\mathsf{f}}, R_{\perp})$, where R_v is a value computed based on the identities in Lemma 8 using the procedures join and union, which are given in the full paper. The join procedure takes as arguments tuples (ψ, E) and (ψ', E'), and truth values v and v'. Provided that the invariants $Inv(\psi,\tau,E)$ and $Inv(\psi',\tau,E')$ are satisfied, the return value is either Fin $([\![\psi]\!]_v \bowtie [\![\psi']\!]_{v'})$ or None, depending on whether a finite relation can be computed. The union procedure has similar arguments and return values. The auxiliary procedures $\mathsf{update_cofin}$ and update from Figure 3 handle the following corner case: If two elements of the newly

```
proc eval_and(H_ψ, H_ψ')
  R_t ← join(H_ψ, H_ψ', t, t)
  R_f ← union(H_ψ, H_ψ', f, f)
  R_⊥ ← eval_and_⊥(H_ψ, H_ψ')
  return update_cofin(ψ ∧ ψ', R_t, R_f, R_⊥)

proc eval_and_⊥(H_ψ, H_ψ')
  R_1 ← join(H_ψ, H_ψ', t, ⊥)
  R_2 ← join(H_ψ, H_ψ', ⊥, t)
  R_3 ← join(H_ψ, H_ψ', ⊥, ⊥)
  case R_1, R_2, R_3 = Fin V_1, Fin V_2, Fin V_3
    return Fin (V_1 ∪ V_2 ∪ V_3)
  otherwise
    return None

proc eval_neg(S_t, S_f, S_⊥)
  return (S_f, S_t, S_⊥)

proc eval_forall(x̄, ψ, (S_t, S_f, S_⊥))
  (R_t, R_f, R_⊥) ← (None, None, None)
  case S_t = Fin T
    R_t ← Fin ∅
    case S_⊥ = Fin U
      R_⊥ ← Fin ∅
  case S_f = Fin F
    s̄ ← get_positions(x̄, ψ)
    R_f ← Fin (π_s̄(F))
    case S_⊥ = Fin U
      R_⊥ ← Fin (π_s̄(U) \ π_s̄(F))
  return update_cofin(∀x̄.ψ, R_t, R_f, R_⊥)

proc eval_times(H_ψ, H_ψ')
  R_t ← join(H_ψ, H_ψ', t, t)
  R_f ← join(H_ψ, H_ψ', f, f)
  R_⊥ ← eval_times_⊥(H_ψ, H_ψ')
  return update_cofin(ψ ⊗ ψ', R_t, R_f, R_⊥)

proc eval_times_⊥(H_ψ, H_ψ')
  R_1 ← union(H_ψ, H_ψ', ⊥, ⊥)
  R_2 ← join(H_ψ, H_ψ', t, f)
  R_3 ← join(H_ψ, H_ψ', f, t)
  case R_1, R_2, R_3 = Fin V_1, Fin V_2, Fin V_3
    return Fin (V_1 ∪ V_2 ∪ V_3)
  otherwise
    return None

proc update_cofin(ψ, R_t, R_f, R_⊥)
  R_t ← update(ψ, R_t, R_f, R_⊥)
  R_f ← update(ψ, R_f, R_t, R_⊥)
  R_⊥ ← update(ψ, R_⊥, R_t, R_f)
  return (R_t, R_f, R_⊥)

proc update(ψ, R_1, R_2, R_3)
  case R_2 = Fin _ and R_3 = Fin _
    if fv(ψ) ≠ ∅ then return CoFin
    else if R_2 = Fin ∅ and R_3 = Fin ∅ then
      return Fin {()}
    else
      return Fin ∅
  otherwise
    return R_1
```

Fig. 3. The eval_neg, eval_and, eval_times, and eval_forall procedures

formed triple $(R_{\mathsf{t}}, R_{\mathsf{f}}, R_{\perp})$ are of the form Fin V and the remaining element is None, then update_cofin$(\psi, R_{\mathsf{t}}, R_{\mathsf{f}}, R_{\perp})$ changes None to either CoFin if $fv(\varphi) \neq \emptyset$, or otherwise (when $fv(\varphi) = \emptyset$) to Fin $\{()\}$ or Fin $\emptyset$ depending on the truth value that should be returned. This ensures that the invariant *Inv* is preserved by the return value of the eval_and, eval_times, and eval_forall procedures.

Finally, we consider the temporal operators. Let $\psi = \alpha \mathsf{S}_I \beta$. For efficiency, eval maintains between iterations a sequence L_ψ, which is initialized by the init procedure with the empty sequence. The sequence L_ψ contains values $E_{\tau'}$ that satisfy the invariant $Inv(\alpha \mathsf{S}_{[\delta,\delta]} \beta, \tau, E_{\tau'})$, where $\delta = \tau - \tau'$ and τ' is such that $0 \leq \tau - \tau' < b$, with $I = [a, b)$. In this way, the sub-formulas α and β are not re-evaluated at previous time points τ'. Instead, the result of their evaluation is stored in L_ψ. The return value is computed by iteratively calling eval_or on the elements $E_{\tau'}$ of L_ψ for which $(\tau - \tau') \in I$. This last step reflects the equivalence between $\alpha \mathsf{S}_I \beta$ and $\bigvee_{\delta \in I} \alpha \mathsf{S}_{[\delta,\delta]} \beta$. Given two formulas ψ_1 and ψ_2 and two values E_1 and E_2 satisfying respectively the invariants $Inv(\psi_1, \tau, E_1)$ and $Inv(\psi_2, \tau, E_2)$, the procedure eval_or returns a value E that satisfies $Inv(\psi_1 \vee \psi_2, \tau, E)$.

The case for Until is analogous to Since. The only significant difference is that the procedure must delay its answer until all relevant events have occurred. Various optimizations, which we mention in the full paper, can further improve the efficiency of handling temporal operators.

The following theorem establishes termination and soundness of our algorithm. To state it formally, we first explicitly define the relationship between the arguments Γ_τ of the eval procedure, and the logging structures $\mathcal{D}_\tau$ of $\bar{\mathcal{D}}$. We let

$$triples(\mathcal{D}_\tau) := \left\{ \left(r, (val(r_{\mathsf{t}}^{\mathcal{D}_\tau}), val(r_{\mathsf{f}}^{\mathcal{D}_\tau}), val(r_{\perp}^{\mathcal{D}_\tau}))\right) \mid r \in R \right\},$$

where $val(V)$ is $\mathsf{Fin}\ V$ if V is finite, and is CoFin otherwise. Thus $\Gamma_\tau = triples(\mathcal{D}_\tau)$.

Theorem 9. *Let $\bar{\mathcal{D}}$ be a logging knowledge base, φ a formula in $\mathcal{L}_3$, and $\tau \in \mathbb{N}$ a time point. The procedure $\mathsf{eval}(\varphi, \Gamma_\tau, \tau)$ returns a value E that satisfies the invariant $Inv(\varphi, \tau, E)$, whenever $\mathsf{init}(\varphi)$, $\mathsf{eval}(\varphi, \Gamma_0, 0)$, ..., $\mathsf{eval}(\varphi, \Gamma_{\tau-1}, \tau-1)$ were called previously in this order, where $\Gamma_{\tau'} = triples(\mathcal{D}_{\tau'})$, for $\tau' \leq \tau$.*

A Complete Fragment. In general, our algorithm is incomplete. However, by limiting the usage of free variables, we obtain the fragment $\mathcal{L}_3^c$ for which we guarantee completeness.

Definition 10. *The set $\mathcal{L}_3^c$ of formulas is inductively defined:*

- $\mathsf{f} \in \mathcal{L}_3^c$ *and* $r(t_1, \dots, t_{\iota(r)}) \in \mathcal{L}_3^c$,
- *if* $\varphi \in \mathcal{L}_3^c$, *then* $\neg\varphi \in \mathcal{L}_3^c$ *and* $\forall x.\, \varphi \in \mathcal{L}_3^c$,
- *if* $\varphi, \psi \in \mathcal{L}_3^c$ *and either* $fv(\varphi) = fv(\psi)$, $fv(\varphi) = \emptyset$, *or* $fv(\psi) = \emptyset$, *then* $\varphi \wedge \psi \in \mathcal{L}_3^c$, $\varphi \otimes \psi \in \mathcal{L}_3^c$, $\varphi\ \mathsf{S}_I\ \psi \in \mathcal{L}_3^c$, *and* $\varphi\ \mathsf{U}_I\ \psi \in \mathcal{L}_3^c$.

Note that $\mathcal{L}_3^c$ allows universal quantification and, by using $\neg$, also existential quantification of free variables, and both quantifiers can be nested freely. But if an $\mathcal{L}_3^c$ formula contains a sub-formula with no quantifiers and two or more predicates, they must have the same free variables. As all of $\mathcal{L}_3$'s connectives are retained and their application is not restricted, $\mathcal{L}_3^c$ can still express all monotonic finitary operators. However, they cannot be used as liberally as in $\mathcal{L}_3$.

The first and second policy examples in Section 3 fall within $\mathcal{L}_3^c$. However, due to the free-variable restriction, the following formula is not in $\mathcal{L}_3^c$:

$$\Box\, \forall s.\, \forall r.\, \forall m.\, send(s, r, m) \rightarrow \Diamond_I\, authorize(m)\,.$$

It says that all messages m, sent by s to r must be subsequently authorized. This is a typical compliance policy from the HIPAA Privacy Rule [1]. By pushing the quantification of s and r inside the antecedent, we obtain a formula in $\mathcal{L}_3^c$:

$$\Box\, \forall m.\, \big(\exists s.\, \exists r.\, send(s, r, m)\big) \rightarrow \Diamond_I\, authorize(m)\,.$$

One can check that evaluating $\forall x.\, \varphi \rightarrow \psi$ and $(\exists x.\, \varphi) \rightarrow \psi$, as well as $\exists x.\, \varphi \wedge \psi$ and $(\exists x.\, \varphi) \wedge \psi$, where $x \notin fv(\psi)$, over an arbitrary logging knowledge base and an arbitrary time point yields the same truth value.

It is not always possible to rewrite a formula such that the result falls into $\mathcal{L}_3^c$. Recall the third example (the separation-of-duty requirement) from Section 3. Clearly, it does not fall within $\mathcal{L}_3^c$. However, if there are finitely many datasets, we can partially ground the formula, obtaining a family of formulas $\varphi_{\mathrm{d},\mathrm{d}'}$, where d and d′ range over the datasets. Each is in $\mathcal{L}_3^c$ after similar rewriting as above:

$$\varphi_{\mathrm{d},\mathrm{d}'} := \Box\, \big(\exists s.\, (\exists o.\, access(s, o, \mathrm{d})) \wedge \exists o'.\, \blacklozenge\, access(s, o', \mathrm{d}')\big) \rightarrow \mathsf{C}_{[0,2)} \neg conflict(\mathrm{d}, \mathrm{d}')\,.$$

Syntactic rewriting and partial grounding cannot always be applied. Still, $\mathcal{L}_3^c$ is an expressive fragment that captures a wide-range of compliance policies.

Finally, we state our result on the algorithm's completeness on $\mathcal{L}_3^c$ formulas. To do so, we define the stronger invariant $Inv_c(\varphi, \tau, E)$ which, in addition to $Inv(\varphi, \tau, E)$, requires that there are $v', v'' \in \mathbf{3}$ with $v' \neq v''$ such that $S_{v'} = \mathsf{Fin}\ V'$ and $S_{v''} = \mathsf{Fin}\ V''$ for some sets V', V'', where $E = (S_\mathsf{t}, S_\mathsf{f}, S_\perp)$.

Theorem 11. *Let $\bar{\mathcal{D}}$ be a logging knowledge base, φ a formula in $\mathcal{L}_3^c$, and $\tau \in \mathbb{N}$ a time point. The procedure $\mathsf{eval}(\varphi, \Gamma_\tau, \tau)$ returns a value E that satisfies the invariant $Inv_c(\varphi, \tau, E)$, whenever $\mathsf{init}(\varphi)$, $\mathsf{eval}(\varphi, \Gamma_0, 0), \ldots, \mathsf{eval}(\varphi, \Gamma_{\tau-1}, \tau-1)$ were called previously in this order, where $\Gamma_{\tau'} = triples(\mathcal{D}_{\tau'})$, for $\tau' \leq \tau$.*

6 Related Work

The only work we are aware of that addresses the problem of compliance checking with incomplete knowledge is Garg et al. [21]. Their policy language is a restricted first-order logic. It has a more liberal usage of free variables compared to $\mathcal{L}_3^c$, but it does not consider $\perp$ at the object-level and cannot express the $\otimes$ operator. They adopt a weaker logging assumption, whereby a finite or an infinite number of event occurrences can be unknown. However, their compliance algorithm is not suitable for on-line monitoring and, more importantly, it is incomplete, even with our logging assumption. Recall our first policy example in Section 3. If the web-server's logger crashes and there are no denials, their algorithm does not report that there are no violations. Instead, it wrongly reports that there may be potential violations, where in fact there are none. Similarly, it may also fail to report violations. For example, given a specification of the form

$$\square\, \forall \bar{x}.\, c(\bar{x}) \rightarrow \exists \bar{y}.\, c'(\bar{x}, \bar{y}) \wedge \forall \bar{z}.\, \varphi(\bar{x}, \bar{y}, \bar{z})\,,$$

then all $\bar{x}$ that violate the policy by making c true and φ false, but for which all c' events are missing, are not reported. This is because their algorithm evaluates formulas in a top-down fashion: it first finds all $\bar{x}$ that satisfy c, then it partially grounds[2] the consequent, then it finds all $\bar{y}$ that satisfy c', and then partially grounds φ, and so forth. However, if there are no partial groundings, the algorithm stops further evaluations. In contrast, since our algorithm works in a bottom-up fashion, it does not have this problem.

The problem of incompleteness and disagreements is also present in other fields, and some approaches there are also based on many-valued logics. Some access-control policy languages [15, 18] use multiple truth values to represent different access-control decisions. These languages are propositional and do not support temporal reasoning. Several model-checking approaches [13, 14, 16] also consider a many-valued truth space. However, their many-valued semantics do not guarantee policy-compliance monotonicity. Furthermore, their specification languages only have the classical Boolean and temporal connectives.

[2] Their logging assumption and language restrictions guarantee that there are always only finitely many satisfying ground instances.

Bauer et al. [8] extend the classical LTL semantics by also assigning non-Boolean truth values to finite and complete prefixes of infinite traces. Their semantics differentiate whether all or some extensions of a finite trace satisfy a property. However, the Boolean and temporal operators are not extended over the additional truth values. Furthermore, they do not consider the ordering $\leq_k$ of the truth values in knowledge.

Another approach to dealing with incompleteness is to make quantitative statements, e.g., how certain it is whether a property is violated. Stoller et al. [26] present such an approach for monitoring traces with gaps. Their solution first assigns probabilities to whether events happened during gaps, and then computes the overall probability that a temporal property is violated. This solution is orthogonal to ours. It requires a reliable training set to derive appropriate probability assignments for different event occurrences.

7 Conclusions

In complex IT systems, logging failures happen and knowledge about the occurrence of system actions is incomplete when monitoring the system. Furthermore, system components can disagree on whether actions took place. Approaches for checking system compliance based on the classical Boolean setting are insufficient since they may incorrectly report policy violations. A three-valued truth space allows us to correctly distinguish between violations and potential violations. The solution presented in this paper carefully adopts a three-value truth space so that policy evaluations are correct regardless of how knowledge gaps are resolved. The presented monitoring algorithm shows that policy violations and potential violations can be soundly and completely determined.

As future work we will investigate how to efficiently resolve potential violations as prior knowledge gaps are incrementally resolved. We also plan case studies to evaluate our monitoring algorithm in real-world settings. Finally, we would like to explore different truth spaces to distinguish between different kinds of knowledge gaps and disagreements.

Acknowledgments. We thank Germano Caronni and Matúš Harvan for fruitful discussions on this topic.

References

1. The Health Insurance Portability and Accountability Act of 1996 (HIPAA), Public Law 104-191 (1996)
2. Gramm-Leach-Bliley Act of 1999 (GLBA), Public Law 106-102 (1999)
3. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
4. Barringer, H., Groce, A., Havelund, K., Smith, M.: Formal analysis of log files. J. Aero. Comput. Inform. Comm. 7, 365–390 (2010)
5. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: Monitoring usage-control policies in distributed systems. In: TIME 2011, pp. 88–95 (2011)

6. Basin, D., Harvan, M., Klaedtke, F., Zălinescu, E.: MONPOLY: Monitoring Usage-Control Policies. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 360–364. Springer, Heidelberg (2012)
7. Basin, D., Klaedtke, F., Müller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: FSTTCS 2008. Leibniz International Proceedings in Informatics (LIPIcs), vol. 2, pp. 49–60 (2008)
8. Bauer, A., Leucker, M., Schallhart, C.: Comparing LTL semantics for runtime verification. J. Logic Comput. 20(3), 651–674 (2010)
9. Belnap Jr., N.D.: A useful four-valued logic. In: Dunn, J.M., Epstein, G. (eds.) Modern Uses of Multiple-Valued Logic. Episteme, vol. 2, pp. 7–37. D. Reidel Publishing Company (1977)
10. Blamey, S.: Partial logic. In: Gabbay, D.M., Guenthner, F. (eds.) Handbook of Philosophical Logic, vol. 5, pp. 261–353. Kluwer Academic Publishers (2002)
11. Boella, G., Broersen, J., van der Torre, L.: Reasoning about Constitutive Norms, Counts-As Conditionals, Institutions, Deadlines and Violations. In: Bui, T.D., Ho, T.V., Ha, Q.T. (eds.) PRIMA 2008. LNCS (LNAI), vol. 5357, pp. 86–97. Springer, Heidelberg (2008)
12. Broersen, J.: On the Logic of 'Being Motivated to Achieve ϱ, Before δ'. In: Alferes, J.J., Leite, J. (eds.) JELIA 2004. LNCS (LNAI), vol. 3229, pp. 334–346. Springer, Heidelberg (2004)
13. Bruns, G., Godefroid, P.: Model Checking Partial State Spaces with 3-Valued Temporal Logics. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 274–287. Springer, Heidelberg (1999)
14. Bruns, G., Godefroid, P.: Model Checking with Multi-valued Logics. In: Díaz, J., Karhumäki, J., Lepistö, A., Sannella, D. (eds.) ICALP 2004. LNCS, vol. 3142, pp. 281–293. Springer, Heidelberg (2004)
15. Bruns, G., Huth, M.: Access control via Belnap logic: Intuitive, expressive, and analyzable policy composition. ACM Trans. Inform. Syst. Secur. 14(1) (2011)
16. Chechik, M., Devereux, B., Easterbrook, S., Gurfinkel, A.: Multi-valued symbolic model-checking. ACM Trans. Softw. Eng. Meth. 12(4), 371–408 (2003)
17. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. 20(2), 149–186 (1995)
18. Crampton, J., Morisset, C.: PTaCL: A Language for Attribute-Based Access Control in Open Systems. In: Degano, P., Guttman, J.D. (eds.) POST 2012. LNCS, vol. 7215, pp. 390–409. Springer, Heidelberg (2012)
19. Dinesh, N., Joshi, A., Lee, I., Sokolsky, O.: Checking Traces for Regulatory Conformance. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 86–103. Springer, Heidelberg (2008)
20. Fitting, M.: Kleene's logic, generalized. J. Log. Comput. 1(6), 797–810 (1991)
21. Garg, D., Jia, L., Datta, A.: Policy auditing over incomplete logs: Theory, implementation and applications. In: CCS 2011, pp. 151–162 (2011)
22. Groce, A., Havelund, K., Smith, M.: From scripts to specification: The evaluation of a flight testing effort. In: ICSE 2010, vol. 2, pp. 129–138 (2010)
23. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Trans. Serv. Comput. 5(2), 192–206 (2012)
24. Hvitved, T., Klaedtke, F., Zălinescu, E.: A trace-based model for multiparty contracts. J. Log. Algebr. Program. 81(2), 72–98 (2012)
25. Kleene, S.C.: Introduction to Metamathematics. D. Van Nostrand, Princeton (1950)
26. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime Verification with State Estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012)

Adaptive Runtime Verification

Ezio Bartocci[2], Radu Grosu[2], Atul Karmarkar[1], Scott A. Smolka[1], Scott D. Stoller[1], Erez Zadok[1], and Justin Seyster[1]

[1] Department of Computer Science, Stony Brook University, USA
[2] Department of Computer Engineering, Vienna University of Technology

Abstract. We present *Adaptive Runtime Verification* (ARV), a new approach to runtime verification in which overhead control, runtime verification with state estimation, and predictive analysis are synergistically combined. Overhead control maintains the overhead of runtime verification at a specified target level, by enabling and disabling monitoring of events for each monitor instance as needed. In ARV, predictive analysis based on a probabilistic model of the monitored system is used to estimate how likely each monitor instance is to violate a given temporal property in the near future, and these *criticality levels* are fed to the overhead controllers, which allocate a larger fraction of the target overhead to monitor instances with higher criticality, thereby increasing the probability of violation detection. Since overhead control causes the monitor to miss events, we use Runtime Verification with State Estimation (RVSE) to estimate the probability that a property is satisfied by an incompletely monitored run. A key aspect of the ARV framework is a new algorithm for RVSE that performs the calculations in advance, dramatically reducing the runtime overhead of RVSE, at the cost of introducing some approximation error. We demonstrate the utility of ARV on a significant case study involving runtime monitoring of concurrency errors in the Linux kernel.

1 Introduction

In [11], we introduced the concept of *runtime verification with state estimation* (RVSE), and showed how it can be used to estimate the probability that a temporal property is satisfied by a partially or incompletely monitored program run. In such situations, there may be *gaps* in observed program executions, making accurate estimation challenging.

Incomplete monitoring can arise from a variety of sources. For example, in real-time embedded systems, the sensors might have intrinsically limited fidelity, or the scheduler might skip monitoring of internal or external events due to an impending deadline for a higher-priority task. Incomplete monitoring also arises from overhead control frameworks, such as [5], which repeatedly disable and enable monitoring of selected events, to maintain the overall overhead of runtime monitoring at a specified target level. Regardless of the cause, simply ignoring the fact that unmonitored events might have occurred gives poor results.

The main idea behind RVSE is to use a statistical model of the monitored system, in the form of a Hidden Markov Model (HMM), to “fill in” gaps in event sequences. We then use an extended version of the forward algorithm of [7] to calculate the probability

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 168–182, 2013.
© Springer-Verlag Berlin Heidelberg 2013

that the property is satisfied. The HMM can be learned automatically from training runs, using standard algorithms [7].

When the cause of incomplete monitoring is overhead control, a delicate interplay exists between RVSE and overhead control, due to the runtime overhead of RVSE itself: the matrix-vector calculations performed by the RVSE algorithm to process an observation symbol—which can be a program event or a gap symbol paired with a discrete probability distribution describing the length of the gap—are expensive. Note that we did not consider this interplay in [11], because the RVSE calculations were performed post-mortem in the experiments described there.

The relationship between RVSE and overhead control can be viewed as an accuracy-overhead tradeoff: the more overhead RVSE consumes processing an observation symbol, with the goal of performing more accurate state estimation, the more events are missed (because less overhead is available). Paradoxically, these extra missed events result in more gap symbols, making accurate state estimation all the more challenging.

This tension between accurate state estimation and overhead control can be understood in terms of Heisenberg's uncertainty principle, which essentially states that the more accurately one measures the position of an electron, the more its velocity is perturbed, and vice versa. In the case of RVSE, we are estimating the position (state) and velocity (execution time) of a "computation particle" (program counter) flowing through an instrumented program.

With these concerns in mind, this paper presents *Adaptive Runtime Verification* (ARV), a new approach to runtime verification in which overhead control, runtime verification with state estimation, and predictive analysis are synergistically combined. In ARV, as depicted in Figure 1, each monitor instance[1] has an associated *criticality level*, which is a measure of how "close" the instance is to violating the property under investigation. As criticality levels of monitor instances rise, so will the fraction of monitoring resources allocated to these instances, thereby increasing the probability of violation detection and concomitant adaptive responses to property violations.

The main contributions of this paper are:

- In ARV, the overhead-control subsystem and the RVSE-enabled monitoring subsystem are coupled together in a feedback control loop: overhead control introduces gaps in event sequences, whose resolution requires HMM-based state estimation (RVSE); state estimation informs overhead control, closing the loop. Up-to-date state estimates enable the overhead-control subsystem to make intelligent, criticality-based decisions about how to allocate the available overhead among monitor instances.
- A key aspect of the ARV framework is a new algorithm for RVSE that performs the calculations offline (in advance), dramatically reducing the runtime overhead of RVSE, at the cost of introducing some approximation error. We analyze the cumulative approximation error incurred by this algorithm.
- To compute the criticality levels of monitor instances, the ARV framework performs reward-based reachability queries over the Discrete Time Markov Chain

[1] A *monitor instance* is a runtime instance of a parameterized monitor. For example, our monitor for concurrency errors in the Linux kernel is parameterized by the id (address) of the structure being monitored.

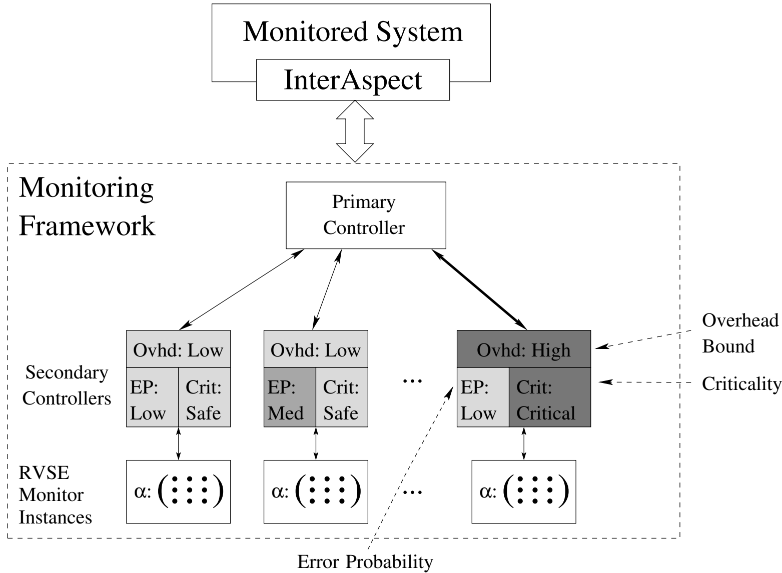


Fig. 1. The Adaptive Runtime Verification Framework

(DTMC) derived from the composition of the HMM model of the monitored program and the monitor, represented as a Deterministic Finite State Machine (DFSM). These queries determine the expected distance to the monitor's error state. These queries are also computed in advance, and the results are stored in a data structure.
- We demonstrate the utility of the ARV approach on a significant case study involving runtime monitoring of concurrency errors in the Linux kernel.

2 Background

Hidden Markov Models (HMMs). An HMM [7] is a tuple $H = \langle S, A, V, B, \pi \rangle$ containing a set S of states, a transition probability distribution A, a set V of observation symbols (also called "outputs"), an observation probability distribution B, and an initial state distribution π. The states and observations are indexed (i.e., numbered), so S and V can be written as $S = \{s_1, s_2, \ldots, s_{N_s}\}$ and $V = \{v_1, \ldots, v_{N_o}\}$, where N_s is the number of states, and N_o is the number of observation symbols. Let $\Pr(c_1 \mid c_2)$ denote the probability that c_1 holds, given that c_2 holds. The transition probability distribution A is an $N_s \times N_s$ matrix indexed by states in both dimensions, such that $A_{i,j} = \Pr(\text{state is } s_j \text{ at time } t+1 \mid \text{ state is } s_i \text{ at time } t)$. The observation probability distribution B is an $N_s \times N_o$ matrix indexed by states and observations, such that $B_{i,j} = \Pr(v_j \text{ is observed at time } t \mid \text{ state is } s_i \text{ at time } t)$. Following tradition, we define $b_i(v_k) = B_{i,k}$. Prior distribution π_i is the probability that the initial state is s_i.

An example of an HMM is depicted in Figure 3 a). Each state is labeled with observation probabilities in that state; for example, `P(LOCK)`=0.99 in state s_1 means

$B_{1,LOCK} = 0.99$. Edges are labeled with transition probabilities; for example, 0.20 on the edge from s_2 to s_3 means $A_{2,3} = 0.20$.

Learning HMMs. Given a set of traces of a system and a desired number of states of the HMM, it is possible to learn an HMM model of the system using standard algorithms [7]. The main idea behind these algorithms is to maximize the probability that the HMM generates the given traces. In our experiments, we chose an HMM model with three states, used the Baum-Welch learning algorithm [1], and provided the learning algorithm with 1,000 traces as input. Figure 3 a) depicts the transition and observation probability distributions of the resulting HMM model. The related case study (Section 6) provides further details.

Deterministic Finite State Machines (DFSMs). We assume that the temporal property ϕ to be monitored is expressed as a parametrized deterministic finite state machine. A DFSM is a tuple $M = \langle S_M, m_{init}, V, \delta, F\rangle$, where S_M is the set of states, m_{init} in S_M is the initial state, V is the alphabet (also called the set of input symbols), $\delta : S_M \times V \to S_M$ is the transition function, and F is the set of accepting states (also called "final states"). Note that δ is a total function. A trace O satisfies the property iff it leaves M in an accepting state.

RVSE Algorithm. In [11], we extended the forward algorithm to estimate the probability of having encountered an error (equivalent to be in an accepting state) in the case where the observation sequence O contains the symbol $gap(L)$ denoting a possible gap with an unknown length. The length distribution L is a probability distribution on the natural numbers: $L(\ell)$ is the probability that the gap has length ℓ.

The Hidden Markov Model $H = \langle S, A, V, B, \pi\rangle$ models the monitored system, where $S = \{s_1, \ldots, s_{N_s}\}$ and $V = \{v_1, \ldots, v_{N_o}\}$. Observation symbols of H are observable actions of the monitored system. H need not be an exact model of the system.

The property ϕ is represented by a DFSM $M = \langle S_M, m_{init}, V, \delta, F\rangle$. For simplicity, we take the alphabet of M to be the same as the set of observation symbols of H. It is easy to allow the alphabet of M to be a subset of the observation symbols of H, by modifying the algorithm so that observations of symbols outside the alphabet of M leave M in the same state.

The goal is to compute $\Pr(\phi \mid O, H)$, i.e., the probability that the system's behavior satisfies ϕ, given observation sequence O and model H. Let $Q = \langle q_1, q_2, \ldots, q_T\rangle$ denote the (unknown) state sequence that the system passed through, i.e., q_t denotes the state of the system when observation O_t is made. We extend the forward algorithm [7] to compute $\alpha_t(i, m) = \Pr(O_1, O_2, \ldots, O_t, q_t = s_i, m_t = m \mid H)$, i.e., the joint probability that the first t observations yield $O_1, O_2, \ldots, O_t$ and that q_t is s_i and that m_t is m, given the model H. We refer to a pair (j, n) of an HMM state and a DFSM state as a *compound state*, and we sometimes refer to α_t as a probability distribution over compound states. The extended algorithm appears in Figure 2. The desired probability $\Pr(\phi \mid O, H)$ is the probability that the DFSM is in an accepting state after observation sequence O, which is $p_{\mathrm{sat}}(\alpha_{|O|+1})$, where $p_{\mathrm{sat}}(\alpha) = \sum_{j\in 1..N_s, n\in F} \alpha(j, n) \;/\; \sum_{j\in 1..N_s, n\in S_M} \alpha(j, n)$. The probability of an error (i.e., a violation of the property) is $p_{\mathrm{err}}(\alpha) = 1 - p_{\mathrm{sat}}(\alpha)$.

$$p_i(m,n) = \sum_{v \in V \text{ s.t. } \delta(m,v)=n} b_i(v) \tag{1}$$

$$g_0(i,m,j,n) = (i = j \wedge m = n)\,?\,1:0 \tag{2}$$

$$g_{\ell+1}(i,m,j,n) = \sum_{i' \in [1..N_s], m' \in S_M} g_\ell(i,m,i',m') A_{i',j} p_j(m',n) \tag{3}$$

$$\alpha_1(j,n) = \tag{4}$$
$$\begin{cases} (n = \delta(m_{init}, O_1))\,?\,\pi_j b_j(O_1) : 0 & \text{if } O_1 \neq gap(L) \\ L(0)(n = m_{init}\,?\,\pi_j : 0) + \sum_{\ell>0, i \in [1..N_s]} L(\ell)\pi_i g_\ell(i, m_{init}, j, n) & \text{if } O_1 = gap(L) \end{cases}$$
$$\text{for } 1 \leq j \leq N_s \text{ and } n \in S_M$$

$$\alpha_{t+1}(j,n) = \begin{cases} \left(\sum_{\substack{i \in [1..N_s] \\ m \in \text{pred}(n, O_{t+1})}} \alpha_t(i,m) A_{i,j} \right) b_j(O_{t+1}) & \text{if } O_{t+1} \neq gap(L) \\ L(0)\alpha_t(j,n) + \sum_{\ell>0} L(\ell) \sum_{\substack{i \in [1..N_s] \\ m \in S_M}} \alpha_t(i,m) g_\ell(i,m,j,n) & \text{if } O_{t+1} = gap(L) \end{cases} \tag{5}$$
$$\text{for } 1 \leq t \leq T-1 \text{ and } 1 \leq j \leq N_s \text{ and } n \in S_M$$

Fig. 2. Forward algorithm for Runtime Verification with State Estimation. $\text{pred}(n, v)$ is the set of predecessors of n with respect to v in the DFSM, i.e., the set of states m such that M transitions from m to n on input v.

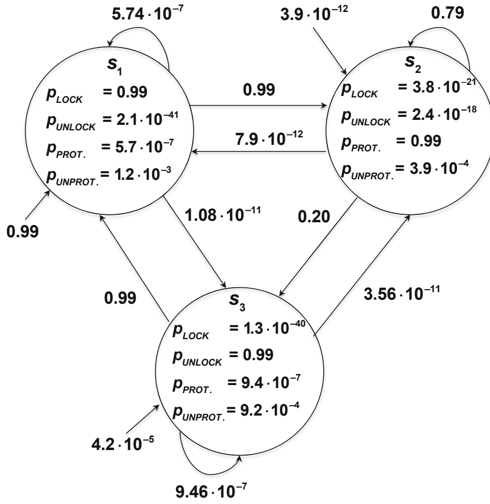
3 The ARV Framework: Architecture and Principles

Figure 1 depicts the architecture of the ARV framework. ARV uses InterAspect [8], an aspect-oriented program-instrumentation framework that we developed for the GCC compiler collection, to insert code that intercepts monitored events and sends them to the *monitoring framework*. The monitoring framework maintains a separate RVSE-enabled monitor instance for each monitored object.

Each monitor instance uses the RVSE algorithm in Section 4 to compute its estimate of the composite HMM-DFSM state; specifically, it keeps track of which pre-computed probability distribution over compound states characterizes the current system state.

Each instance uses this probability distribution over compound states to compute the its *error probability* (EP), i.e., the probability that a property violation has occurred, as described in Section 4. Each instance also uses this probability distribution over compound states to compute its *criticality level*, based on the expected number of transitions before a violation occurs, using the predictive analysis of Section 5.

The overhead-control subsystem is structured, as in SMCO [5], as a *cascade controller* comprising one primary controller and a number of secondary controllers, one per monitor instance. The primary controller allocates monitoring resources (overhead), and the secondary controllers enforce the overhead allocation by disabling monitoring



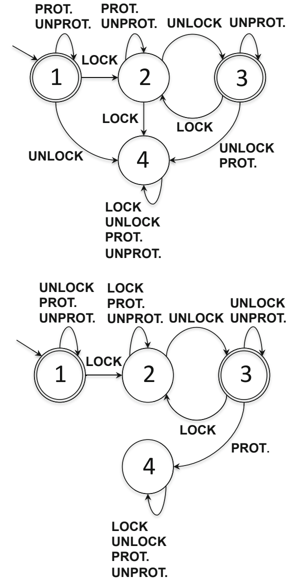


Fig. 3. Left (a): An example of an HMM. Right (b): Two examples of DFSM. States with a double border are accepting states.

when necessary. A key feature of ARV's design is the ability to redistribute overhead so that more critical monitor instances are allowed more monitoring overhead.

4 Pre-computation of RVSE Distributions

Performing the matrix calculations in the RVSE algorithm during monitoring incurs very high overhead. This section describes how to dramatically reduce the overhead by pre-computing compound-state probability distributions α and storing them in a rooted graph. Each edge of the graph is labeled with an observation symbol. At run-time, the algorithm maintains (for each monitor instance) a pointer $curNode$, indicating the node associated with the current state. The probability distribution in the current state is given by the matrix associated with $curNode$. Initially, $curNode$ points to the root node. Upon observing an observation symbol O, the algorithm finds the node n' reachable from $curNode$ by an edge labeled with O, and then assigns n' to $curNode$. Note that this takes constant time, independent of the sizes of the HMM and the monitor.

In general, an unbounded number of probability distributions may be reachable, in which case the graph would be infinite. We introduce an approximation in order to ensure termination. Specifically, we introduce a binary relation $\mathrm{closeEnough}$ on compound-state probability distributions, and during the graph construction, we identify nodes that are close enough.

Pseudo-code for the graph construction appears in Figure 4. $\mathrm{successor}(\alpha, O)$ is the probability distribution obtained using the forward algorithm—specifically, equation

```
α₀ = the probability distribution with α₀(j, m_init) = π₀(j), and α₀(j, n) = 0 for n ≠ m_init
workset = {α₀}
nodes = {α₀}
while workset ≠ ∅
 α = workset.removeOne();
 for each observation symbol O in V
  α′ = normalize(successor(α, O))
  if dead(α′)
   continue
  endif
  if α′ ∈ nodes
   add an exact edge labeled with O from α to α′
  else if there exists α″ in nodes such that closeEnough(α′, α″)
   add an approximate edge labeled with O from α to α′
  else
   add α′ to nodes and workset
   add an exact edge labeled with O from α to α′
  endif
 endfor
endwhile
```

Fig. 4. Pseudo-code for graph construction

(5])—to update compound-state probability distribution α based on observation of observation symbol O. Note that each edge is marked as *exact* or *approximate*, indicating whether it introduces any inaccuracy. $\mathrm{normalize}(\alpha)$ is the probability distribution obtained by computing $\sum_{j,n} \alpha(j,n)$ and then dividing every entry in α by this sum; the resulting matrix α' satisfies $\sum_{j,n} \alpha'(j,n) = 1$. Normalization has two benefits. First, it helps reduce the number of nodes, because normalized matrices are more likely to be equal or close-enough than un-normalized matrices. Second, normalization helps reduce the inaccuracy caused by the use of limited-precision numerical calculations in the implementation (*cf.* [7, Section V.A]), which uses the term "scaling" instead of "normalization"). Normalization is compatible with our original RVSE algorithm—in particular, it does not affect the value calculated for $\Pr(\phi \mid O, H)$—and it provides the second benefit described above in that algorithm, too, so we assume hereafter that the original RVSE algorithm is extended to normalize each matrix α_t.

A state s of a DFSM is *dead* if it is non-accepting and all of its outgoing transitions lead to s. A probability distribution is *dead* if the probabilities of compound states containing dead states of the DFSM sum to 1. The algorithm does not bother to compute successors of dead probability distributions, which always have error probability 1.

We define the close-enough relation by: $\mathrm{closeEnough}(\alpha, \alpha')$ iff $||\alpha - \alpha'||_{\mathrm{sum}} \leq \epsilon$, where ϵ is an implicit parameter of the construction, and $||\alpha||_{\mathrm{sum}} = \sum_{i,j} |\alpha(i,j)|$. Note that, if we regard α as a vector, as is traditional in HMM theory, then this norm is the vector 1-norm.

Termination. We prove termination of the graph construction using the pigeonhole principle. Consider the space of $N_s \times N_m$ matrices with entries in the range $[0..1]$, where

$N_m = |S_m|$. Partition this space into cells (hypercubes) with edge length ϵ/N_sN_m. If two matrices α and α' are in the same cell, then the absolute value of the largest element in $\alpha - \alpha'$ is at most ϵ/N_sN_m, and $||\alpha - \alpha'||_{\mathrm{sum}}$ is at most the number of elements times the largest element, so $||\alpha - \alpha'||_{\mathrm{sum}} \leq N_sN_m\epsilon/N_sN_m$, hence $||\alpha - \alpha'||_{\mathrm{sum}} \leq \epsilon$. The contrapositive of this conclusion is: if two matrices satisfy $||\alpha - \alpha'||_{\mathrm{sum}} > \epsilon$, then they are in different cells. Therefore, the number of nodes in the graph is bounded by the number of cells in this grid, which is $(N_sN_m/\epsilon)^{N_sN_m}$. Note that this termination proof applies even if normalization is omitted from the algorithm.

Cumulative Inaccuracy. Use of the closeEnough relation during graph construction introduces inaccuracy. We characterize the inaccuracy by bounding the difference between the probability distribution matrix associated with *curNode* and the probability distribution matrix that would be computed by the original RVSE algorithm. Let $\alpha'_1, \alpha'_2, \ldots, \alpha'_t$ be the sequence of matrices labeling the nodes visited in the graph, for a given observation sequence O. Let $\alpha_1, \alpha_2, \ldots, \alpha_t$ be sequence of matrices calculated by the RVSE algorithm for the same observation sequence O. The cumulative inaccuracy is expressed as a bound err_t on $||\alpha_t - \alpha'_t||_{\mathrm{sum}}$. First, we consider inaccuracy assuming that the original and new RVSE algorithms do *not* normalize the probability distributions (recall that normalization is not needed to ensure soundness or termination), and we show that the cumulative inaccuracy does not increase along an exact edge and increases by at most ϵ along an approximate edge.

We define err_t inductively. The base case is $err_0 = 0$. For the induction case, we suppose $||\alpha'_t - \alpha_t||_{\mathrm{sum}} \leq err_t$ and define err_{t+1} so that $||\alpha'_{t+1} - \alpha_{t+1}||_{\mathrm{sum}} \leq err_{t+1}$.

If the transition from α'_t to α'_{t+1} traverses an exact edge, then the inaccuracy remains unchanged: $err_{t+1} = err_t$. To prove this, we show that the following inequality holds: $||\mathrm{successor}(\alpha'_t, O_{t+1}) - \mathrm{successor}(\alpha_t, O_{t+1})||_{\mathrm{sum}} \leq err_t$. To prove this, we expand the definition of successor and simplify. There are two cases, depending on whether O_{t+1} is a gap. If O_{t+1} is not a gap,

$$
\begin{aligned}
& \textstyle\sum_{j\in[1..N_s],n\in S_M} |\mathrm{successor}(\alpha'_t, O_{t+1}) - \mathrm{successor}(\alpha_t, O_{t+1})| \\
= & \textstyle\sum_{j\in[1..N_s],n\in S_M} \sum_{i\in[1..N_s],m\in\mathrm{pred}(n,O_{t+1})} |\alpha'_t(i,m) - \alpha_t(i,m)| A_{i,j} b_j(O_{t+1}) \\
& \text{// } M \text{ is deterministic, so each } m \text{ is predecessor of at most one } n \text{ for given } O_{t+1}, \\
& \text{// so for any } f, \textstyle\sum_{n\in S_M, m\in\mathrm{pred}(n,O_{t+1})} f(m) \leq \sum_{m\in S_M} f(m). \\
\leq & \textstyle\sum_{j\in[1..N_s],i\in[1..N_s],m\in S_M} |\alpha'_t(i,m) - \alpha_t(i,m)| A_{i,j} b_j(O_{t+1}) \\
& A \text{ is stochastic, i.e., } \textstyle\sum_{j\in S_M} A_{i,j} = 1, \text{ and } b_j(O_{t+1}) \leq 1 \\
\leq & \textstyle\sum_{i\in[1..N_s],m\in S_M} |\alpha'_t(i,m) - \alpha_t(i,m)| \\
\leq & \; err_t
\end{aligned}
$$

If O_{t+1} is a gap,

$$
\begin{aligned}
& \textstyle\sum_{j\in[1..N_s],n\in S_M} |\mathrm{successor}(\alpha'_t, O_{t+1}) - \mathrm{successor}(\alpha_t, O_{t+1})| \\
= & \textstyle\sum_{j\in[1..N_s],n\in S_M} L(0)|\alpha'_t(j,n) - \alpha_t(j,n)| \\
& + \textstyle\sum_{\ell>0} L(\ell) \sum_{i\in[1..N_s],m\in\mathrm{pred}(n,O_{t+1})} |\alpha'_t(i,m) - \alpha_t(i,m)| g_\ell(i,m,j,n) \\
& \text{// definition of } err_t \\
\leq & \; L(0) err_t \\
& + \textstyle\sum_{\ell>0} L(\ell) \sum_{i\in[1..N_s],m\in\mathrm{pred}(n,O_{t+1})} |\alpha'_t(i,m) - \alpha_t(i,m)| \sum_{j\in[1..N_s],n\in S_M} g_\ell(i,m,j,n) \\
& \text{// } g_\ell(i,m,\cdot,\cdot) \text{ is stochastic, i.e., } \textstyle\sum_{j\in[1..N_s],n\in S_M} g_\ell(i,m,j,n) = 1, \text{ and def. of } err_t \\
\leq & \; L(0) err_t + \textstyle\sum_{\ell>0} L(\ell) err_t \\
& \text{// } \textstyle\sum_{\ell\geq 0} L(\ell) = 1 \\
\leq & \; err_t
\end{aligned}
$$

If the transition from α'_t to α'_{t+1} traverses an approximate edge, then, by definition of closeEnough, the traversal may add ϵ to the cumulative inaccuracy, so $err_{t+1} = err_t + \epsilon$. Note that the same argument used in the case of an exact edge implies that the inaccuracy in err_t is not amplified by traversal of an approximate edge.

Now we consider the effect of normalization on cumulative inaccuracy. We show that normalization does not increase the inaccuracy. Let $\hat{\alpha}'_t$ and $\hat{\alpha}_t$ be the probability distributions computed in step t before normalization; thus, $\alpha'_t = \text{normalize}(\hat{\alpha}'_t)$ and $\alpha_t = \text{normalize}(\hat{\alpha}_t)$. Note that $\sum_{j,n} \hat{\alpha}'_t(j,n)$ and $\sum_{j,n} \hat{\alpha}_t(j,n)$ are at most 1; this is a property of the forward algorithm (*cf.* [7, Section V.A]). Also, every element of $\hat{\alpha}'_t$, $\hat{\alpha}_t$ α'_t, and α_t is between 0 and 1. Thus, normalization moves each element of $\hat{\alpha}'_t$ and $\hat{\alpha}_t$ to the right on the number line, closer to 1, or leaves it unchanged. For concreteness, suppose $\sum_{j,n} \hat{\alpha}'_t(j,n) < \sum_{j,n} \hat{\alpha}_t(j,n)$; a completely symmetric argument applies when the inequality points the other way. This inequality implies that, on average, elements of $\hat{\alpha}'_t$ are to the left of elements of $\hat{\alpha}_t$. It also implies that, on average, normalization moves elements of $\hat{\alpha}'_t$ farther (to the right) than it moves elements of $\hat{\alpha}_t$. These observations together imply that, on average, corresponding elements of $\hat{\alpha}'_t$ and $\hat{\alpha}_t$ are closer to each other after normalization than before normalization, and hence that $||\alpha'_t - \alpha_t||_{\text{sum}} \leq ||\hat{\alpha}'_t - \hat{\alpha}_t||_{\text{sum}}$. Note that elements of $\hat{\alpha}'_t$ cannot move so much farther to the right than elements of $\hat{\alpha}_t$ that they end up being farther, on average, from the corresponding elements of $\hat{\alpha}_t$, because both matrices end up with the same average value for the elements (namely, $1/N_s N_m$).

Stricter Close-Enough Relation. To improve the accuracy of the algorithm, a slightly stricter close-enough relation is used in our experiments: $\text{closeEnough}(\alpha, \alpha')$ holds iff $||\alpha - \alpha'||_{\text{sum}} \leq \epsilon \;\wedge\; (p_{\text{dead}}(\alpha) = 0 \Leftrightarrow p_{\text{dead}}(\alpha') = 0)$, where $p_{\text{dead}}(\alpha)$ is the sum of the elements of α corresponding to compound states containing a dead state of M. It is easy to show that the algorithm still terminates, and that the above bound on cumulative inaccuracy still holds.

5 Predictive Analysis of Criticality Levels

Criticality Level. We define the *criticality level* of a monitor instance to be the inverse of the expected distance (number of steps) to a violation of the property of interest. To compute this expected distance for each compound state, we compute a Discrete Time Markov Chain (DTMC) by composing the HMM model H of the monitored program with the DFSM M for the property. We then add a reward state structure to it, assigning a cost of 1 to each compound state. We use PRISM [6] to compute, as a reward-based reachability query, the expected number of steps for each compound state to reach compound states containing dead states of M. Note that these queries are issued in advance of the actual runtime monitoring, with the results stored in a table for efficient access.

Discrete-Time Markov Chain (DTMC). A Discrete-Time Markov Chain (DTMC) [6] is a tuple $\mathcal{D} = (S_D, \tilde{s}_0, \mathbf{P})$, where S_D is a finite set of states, $\tilde{s}_0 \in S_D$ is the initial state, and $\mathbf{P} : S_D \times S_D \to [0,1]$ is the transition probability function. $\mathbf{P}(\tilde{s_1}, \tilde{s_2})$ is the probability of making a transition from $\tilde{s_1}$ to $\tilde{s_2}$.

Reward Structures. DTMCs can be extended with a reward (or cost) structure [6]. A *state reward function* $\underline{\rho}$ is a function from states of the DTMC to non-negative real numbers, specifying the reward (or cost, depending on the interpretation of the value in the application of interest) for each state; specifically, $\underline{\rho}(\tilde{s})$ is the reward acquired if the DTMC is in state $\tilde{s}$ for 1 time-step.

Composition of an HMM with a DFSM. Given an HMM $H = \langle S, A, V, B, \pi \rangle$ and a DFSM $M = \langle S_M, m_{init}, V, \delta, F \rangle$, their composition is a DTMC $\mathcal{D} = (S_D, \tilde{s}_0, \mathbf{P})$, where $S_D = (S \times S_M) \cup \{\tilde{s}_0\}$, $\tilde{s}_0$ is the initial state, and the transition probability function $\mathbf{P}$ is defined by:

- $\mathbf{P}(\tilde{s}_0, (s_i, m_{init})) = \pi$, with $1 \le i \le |S|$,
- $\mathbf{P}((s_{i_1}, m_{j_1}), (s_{i_2}, m_{j_2})) = A_{i_1,i_2} \sum_{\forall v_k \in V : \delta(m_{i_1}, v_k) = m_{i_2}} b_{i_1}(v_k)$.

We extend $\mathcal{D}$ with the state reward function such that $\underline{\rho}(\tilde{s}) = 1$ for all $\tilde{s} \in S_D$. With this reward function, we can calculate the expected number of steps until a particular state of the DTMC occurs.

Computing the Expected Distance. The expected distance $ExpDist(\tilde{s}, T)$ of a state $\tilde{s}$ of the DTMC to reach a set of states $T \subseteq S_D$ is defined as the expected cumulative reward and is computed as follows:

$$\mathrm{ExpDist}(\tilde{s}, T) = \begin{cases} \infty & \text{if } \mathrm{PReach}(\tilde{s}, T) < 1 \\ 0 & \text{if } \tilde{s} \in T \\ \underline{\rho}(\tilde{s}) + \sum_{\tilde{s}' \in S_D} \mathbf{P}(\tilde{s}, \tilde{s}') \cdot \mathrm{ExpDist}(\tilde{s}', T) & \text{otherwise} \end{cases}$$

where $\mathrm{PReach}(\tilde{s}, T)$ is the probability to eventually reach a state in T starting from $\tilde{s}$. For further details on quantitative reachability analysis for DTMCs, see [6]. The expected distance for a monitor instance with compound-state probability distribution α is then defined by $\mathrm{ExpDist}(\alpha, T) = \sum_{i,j} \alpha(i, j) \cdot \mathrm{ExpDist}((s_i, m_j), T)$.

6 Case Study

We evaluate our system by designing a monitor for the lock discipline property and applying it to the Btrfs file system. This property is implicitly parameterized by a `struct` type S that has a lock member, protected fields, and unprotected fields. Informally, the property requires that all accesses to protected fields occur while the lock is held.

The DFSM $M^{LD}(t, o)$ for the lock discipline property is parameterized by a thread t and an object o, where o is a particular `struct` with type S. There are four kinds of events: $\mathrm{LOCK}(t, o)$ (thread t acquires the lock associated with object o), $\mathrm{UNLOCK}(t, o)$ (thread t releases the lock associated with object o), $\mathrm{PROT}(t, o)$ (thread t accesses a protected field of object o), and $\mathrm{UNPROT}(t, o)$ (thread t accesses an unprotected field of object o). The DFSM $M^{LD}(t, o)$ is shown in the lower part of Figure 3(b); the parameters t and o are elided to avoid clutter. It requires that thread t's accesses to protected fields occur while thread t holds the lock associated with object o, except for accesses to protected fields before the first time t acquires that lock (such accesses are assumed to be part of initialization of o).

7 Implementation

Implementing the case study requires a gap-aware monitor and instrumentation that can intercept monitored events. Both these subsystems must integrate with our overhead control mechanism. The monitor must be able to recognize potential gaps caused by overhead control decisions, and the instrumentation must provide a means for the controller to disable monitoring by halting the interception of events. In addition, our implementation adapts to RVSE's criticality estimates by allocating hardware debugging resources to exhaustively monitor a small number of risky objects. This section discusses the implementation of these systems.

7.1 Gaps

On updating a monitor instance, the monitor processes a gap event before processing the current intercepted event if monitoring was disabled since the last time the monitor instance was updated. The gap event indicates that the monitor may have missed one or more events for the given instance during the time that monitoring was disabled.

The monitor determines whether a gap event is necessary by comparing the time of the last update to the monitor instance's state, which is stored along with the state, with the last time that monitoring was disabled for the current thread. For efficiency, we measure time using a counter incremented each time monitoring is disabled—a logical clock—rather than a real-time clock.

7.2 Instrumentation

For our case study, we monitor the lock discipline property for the `btrfs_space_info` struct in the Linux Btrfs file system. Each `btrfs_space_info` object has a spinlock, eight fields protected by the spinlock, and five fields not protected by the spinlock.

Using a custom GCC plug-in, we instrument every function that operates on a `btrfs_space_info` object, either by accessing one of its fields or by acquiring or releasing its spinlock. The instrumented function first has its function body *duplicated* so that there is an *active* path and an *inactive* path. Only the active path is instrumented for full monitoring. This allows monitoring to be efficiently enabled or disabled at the granularity of a function execution. Selecting the inactive path effectively disables monitoring. When a duplicated function executes, it first calls a *distributor* function that calls the overhead control system to decide which path to take. We enable and disable monitoring at the granularity of function executions, because deciding to enable or disable monitoring at the granularity of individual events would incur too much overhead.

Every `btrfs_space_info` operation in the active path is instrumented to call the monitor, which updates the appropriate monitor instance, based on the thread and the `btrfs_space_info` object involved. For fast lookup, all monitor instances associated with a thread are stored in a hash table local to that thread and indexed by object address.

7.3 Hardware Supervision

Our system prioritizes monitoring of objects with high criticality by placing them under hardware supervision. Specifically, we use debug registers to monitor every operation

on these objects even when other monitoring is disabled (i.e., when the inactive path is taken). The debug registers cause the CPU to raise a debug exception whenever an object under hardware supervision is accessed, allowing the monitor to observe the access. Note that this allows monitoring to be enabled and disabled on a per-object basis, for a limited number of objects, in contrast to the per-function-execution basis described above. The overhead remaining after monitoring the hardware supervised objects is distributed to the other objects in the system using the normal overhead control policy.

Our current implementation keeps track of the most critical object in each thread. Each thread can have its own debug register values, making it possible to exhaustively track events for one monitor instance in each thread for any number of threads.

Because an x86 debug register can at most watch one 64-bit memory location, we need a small amount of additional instrumentation to monitor all 13 fields in a supervised `btrfs_space_info` object. Our plug-in instruments every `btrfs_space_info` field access in the *inactive* path with an additional read to a dummy field in the same object. Setting the debug register to watch the dummy field of a supervised object causes the program to raise a debug exception whenever any field of that object is accessed from the inactive path. The debug exception handler calls the monitor to update the monitor instance for the supervised object.

For `btrfs_space_info` spinlock acquire and release operations, we instrument the inactive path with a simple check to determine if the spinlock belongs to one of the few supervised objects that should be updated even though monitoring is disabled. We could use debug registers to remove the need for this check, but we found that overhead from checking directly was very low, because lock operations occur infrequently compared to field accesses.

7.4 Training

We collected data from completely monitored runs to train the HMM and learn the gap length distribution. During training runs for a given overhead level, the distributor makes monitoring decisions as if overhead control were in effect but does not enforce those decisions; instead, it always takes the active path. As a result, the system knows which events would have been missed by taking the inactive path. Based on this information, for each event that would have triggered processing of a gap event, we compute the actual number of events missed for the corresponding monitor instance. The gap length distribution for the given overhead level is the distribution of those numbers.

Our case study uses a simple overhead-control mechanism in which the target "overhead level" is specified by the fraction f of function executions to be monitored. For each function execution, the distributor flips a biased coin, which says "yes" with probability f, to decide whether to monitor the current function execution. We tested three different sampling probabilities: 50%, 75%, 85%, and 95%. For each sampling probability, we precomputed the RVSE distributions with $\epsilon = 0.1$, thereby obtaining four RVSE graphs having 12,177, 33,234, 30,645 and 11,622 nodes, respectively.

7.5 Evaluation

We used two different tests to measure how well our prioritization mechanism improved ARV's effectiveness. The first test runs with an unmodified version of Btrfs, which does not contain any lock discipline violations, in order to test how well prioritization avoids false alarms. The second test runs on a version of Btrfs with an erroneous access that we inserted, to test if prioritization improves our chances of detecting it. For both of these tests, we run Racer [12], a workload designed specifically to stress file system concurrency, on top of a Btrfs-formatted file system, and we report results that are averaged over multiple runs.

We tested three configurations: 1) hardware supervision disabled, 2) randomly assigned hardware supervision, and 3) adaptive hardware supervision that prioritizes critical objects, as described above. Most threads in the Racer workload had two associated monitor instances. At any time, our prioritization chose one of those from each thread to supervise.

The table below shows the results for these tests. Each row in the table is for one of the three sampling probabilities. For our false alarm test, the columns labeled FalseAlarm in the table show how many monitor instances had an error probability higher than 0.8 at the end of the run. Because the run had no errors, lower numbers are better in this test. For our error detection test, we checked the corresponding monitor instance immediately after our synthetic error triggered; the columns labeled ErrDet in the table show the percentage of the times that we found that monitor instance to have an error probability higher than 0.8, indicating it correctly inferred a likely error. For this test, higher numbers are better. All results are averaged over multiple runs.

Sampling	No Supervision		Random Supervision		Adaptive Supervision	
Probability	FalseAlarm	ErrDet	FalseAlarm	ErrDet	FalseAlarm	ErrDet
50%	30.3	23.0%	11.7	57.4%	12	50.1%
75%	47	31.2%	36	69.3%	17	79.4%
85%	5502	34.1%	5606	72.3%	5449	85.1%

In all cases, hardware supervision improved the false alarm rate and the error detection rate. For the 75% and 85% sampling profiles, adaptive prioritization provides greater improvement than simply choosing objects at random for supervision. With 50% sampling, adaptive sampling does worse than random, however. In future work, we intend to improve our criticality metric so that it performs better at lower overheads. The table also shows that ARV takes advantage of increased sampling rates, successfully detecting more errors in the error detection test. We are currently investigating why performance in the false alarm test declines with higher sampling rates.

8 Related Work

In [2], the authors propose a method for the automatic synthesis and adaptation of invariants from the observed behavior of an application. Their overall goal is adaptive application monitoring, with a focus on interacting software components. In contrast to our approach, where we learn HMMs, the invariants learned are captured as finite automata (FA). These FA are necessarily much larger than their corresponding HMMs. Moreover, error uncertainty, due to inherently limited training during learning, must be dealt with at runtime, by modifying the FA as needed. They also do not address the problem of using the synthesized FA for adaptive-control purposes.

A main aspect of our work is our approximation of the RVSE forward algorithm for state estimation, which pre-computes compound-state probability distributions and stores them in a graph. In the context of the runtime monitoring of HMMs, the authors of [10] propose a complementary method for accelerating the estimation of the current (hidden) state: Particle filters [4]. This sequential Monte-Carlo estimation method is particularly useful when the number of states of the HMM is very large, in particular, much larger than the number of particles (i.e., samples) necessary for obtaining a sufficiently accurate approximation. This, however, is typically not the case in our setting, where the HMMs are relatively small. Consequently, the Particle filtering method would have introduced at least as much overhead as the forward algorithm, and would have therefore also required a priori (and therefore approximate) state estimation.

The runtime verification of HMMs is explored in [9,3], where highly accurate deterministic and randomized methods are presented. In contrast, we are considering the runtime verification of actual programs, while using probabilistic models of program behavior in the form of HMMs to fill in gaps in execution sequences.

9 Conclusions

We have presented Adaptive Runtime Verification, a new approach approach to runtime verification that synergistically combines overhead control, runtime verification with state estimation, and predictive analysis of monitor criticality levels. We have demonstrated the utility of the ARV framework through a significant case study involving the monitoring of concurrency errors in the Linux kernel.

Future work will involve extending the ARV framework with a recovery mechanism that will come into play when a property violation is detected or imminent. We will also consider additional case studies, including those that use SMCO [5] for their overhead control. Fully integrating SMCO will require a new method to compute the probability distribution on the length of gaps introduced by SMCO for any given target overhead.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. Research supported in part by AFOSR Grant FA9550-09-1-0481, NSF Grants CCF-1018459, CCF-0926190, and CNS-0831298, and ONR Grant N00014-07-1-0928.

References

1. Baum, L.E., Petrie, T., Soules, G., Weiss, N.: A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. The Annals of Mathematical Statistics 41(1), 164–171 (1970)
2. Denaro, G., Mariani, L., Pezze, M., Tosi, D.: Adaptive runtime verification for autonomic communication infrastructures. In: Proc. of the International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM), vol. 2, pp. 553–557. IEEE Computer Society (2005)
3. Gondi, K., Patel, Y., Sistla, A.P.: Monitoring the Full Range of ω-Regular Properties of Stochastic Systems. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 105–119. Springer, Heidelberg (2009)
4. Gordon, N., Salmond, D., Smith, A.: Novel approach to nonlinear/non-Gaussian Bayesian state estimation. In: IEEE Proceedings on Radar and Signal Processing, vol. 140, pp. 107–127. IEEE (1993)
5. Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S.A., Stoller, S.D., Zadok, E.: Software monitoring with controllable overhead. International Journal on Software Tools for Technology Transfer (STTT) 14(3), 327–347 (2012)
6. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Model Checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
7. Rabiner, L.R.: A tutorial on hidden Markov models and selected applications in speech recognition. Proceedings of the IEEE 77(2), 257–286 (1989)
8. Seyster, J., Dixit, K., Huang, X., Grosu, R., Havelund, K., Smolka, S.A., Stoller, S.D., Zadok, E.: InterAspect: Aspect-oriented instrumentation with GCC. Formal Methods in System Design (2012), accepted on condition of minor revisions
9. Sistla, A.P., Srinivas, A.R.: Monitoring Temporal Properties of Stochastic Systems. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 294–308. Springer, Heidelberg (2008)
10. Sistla, A.P., Žefran, M., Feng, Y.: Monitorability of Stochastic Dynamical Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 720–736. Springer, Heidelberg (2011)
11. Stoller, S.D., Bartocci, E., Seyster, J., Grosu, R., Havelund, K., Smolka, S.A., Zadok, E.: Runtime Verification with State Estimation. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 193–207. Springer, Heidelberg (2012)
12. Modak, S.: Linux Test Project (LTP) (2009), http://ltp.sourceforge.net/

Malware Riding Badware: Challenges in Analyzing (Malicious/Benign) Web Applications

Giovanni Vigna

University of California at Santa Barbara

Abstract. The Web has become a dangerous place, where simple rules of behavior no longer keep a user out of trouble. Sophisticated drive-by download attacks delivered through compromised web sites are reaching users through seemingly innocuous search engine queries. Malicious web pages are turning helpless victims into armies of bots that participate in a historically unprecedented transfer of wealth in the form of intellectual property, trade secrets, and classified information. This talk describes how this problem can be tackled from two different points of view: the identification of the web application vulnerabilities that allow for site compromise and the detection of web-based malware that attacks the users' browsers. These two issues go hand-in-hand, and require automated approaches in order to keep up with the pace at which cybercriminals devise new ways to exploit and hijack web applications and browsers.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, p. 183, 2013.
© Springer-Verlag Berlin Heidelberg 2013

MapReduce for Parallel Trace Validation of LTL Properties

Benjamin Barre, Mathieu Klein, Maxime Soucy-Boivin, Pierre-Antoine Ollivier, and Sylvain Hallé*

Département d'informatique et de mathématique
Université du Québec à Chicoutimi, Canada
shalle@acm.org

Abstract. We present an algorithm for the automated verification of Linear Temporal Logic formulæ on event traces using an increasingly popular cloud computing framework called MapReduce. The algorithm can process multiple, arbitrary fragments of the trace in parallel, and compute its final result through a cycle of runs of MapReduce instances. Compared to classical, single-instance solutions, a proof-of-concept implementation shows through experimental evaluation how the algorithm reduces by as much as 90% the number of operations that must be performed linearly, resulting in a commensurate speed gain.

1 Introduction

Over the recent years, the volume and complexity of interactions between information systems has been steadily increasing. Large amounts of data are gathered about these interactions, forming a trace of events, also called a *log*, that can be stored, mined, and audited. Web servers, operating systems, database engines and business processes of various kinds all produce event logs, crash reports, test traces or dumps in some format or another.

One possible use of such a log is to perform *trace validation*: given a specification of the expected or agreed-upon interaction (or inversely, of invalid behaviour), the trace of actions recorded at runtime can then be searched automatically for patterns satisfying or violating that specification. The specification generally relates events to some sequence of actions, method calls or events: the validity of each event cannot be assessed individually, but must rather be evaluated according to the event's position with respect to surrounding events, both before and after. As we shall see in Section 2, there exists a variety of scenarios where event traces are subject to sequencing constraints, and the use of a language such as Linear Temporal Logic represents a reasonable mean of expressing these constraints formally.

Various solutions have been proposed in the past to automate the task of trace validation [3–5, 7, 12, 22], either based on temporal logic or other kinds of formal specifications. While these solutions allow the expression of intricate relationships between events in a log, the scalability of many of them is jeopardized by the growing amount

* With financial support from the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Fonds de recherche du Québec – Nature et technologies (FRQNT).

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 184–198, 2013.
© Springer-Verlag Berlin Heidelberg 2013

of data generated by today's systems. Recently, the advent of *cloud computing* has been put forward as a potential remedy to this problem, in particular for the tasks of process discovery and conformance checking [20]. By allowing the distributed processing of data spread across a network of commodity hardware, cloud computing opens the way to dramatic improvements in the performance of many applications.

Given the growing amount of collected trace data and the observed move towards distributed computing infrastructures, it is crucial that existing trace validation methodologies be ported to the cloud paradigm. However, the prospect of parallel processing of temporal constraints in general, and LTL formulæ in particular, is held back precisely because of the sequential nature of the properties to verify: since the validity of an event may depend on past and future events, the handling of parts of the trace in parallel and independent processes seems to be disqualified at the onset. A review of available solutions in Section 3 observes, perhaps unsurprisingly, that most existing trace validation tools are based on algorithms that do not take advantage of parallelism, while those that do offer very limited specification languages where sequential relationships between events are excluded.

The present paper addresses this issue by presenting a *parallelizable* algorithm for the automated validation of LTL properties in event traces. The algorithm uses a recent and popular execution framework, called MapReduce [9], which is described in Section 4. MapReduce provides an environment particularly suitable to the breaking up of a task into small, independent processes that can be distributed across multiple nodes in a network, and is currently being used in large-scale applications such as the Google search engine for the computation of the PageRank index [17]. The algorithm, detailed in Section 5, exploits this framework by splitting the original property into subformulæ that can be evaluated separately through cycles of MapReduce jobs.

The algorithm has been implemented in a proof-of-concept application that was then experimentally evaluated on traces of up to 100,000 events. Results from these experiments, described in Section 6, show that for some classes of constraints, a very large portion of the work can be dispatched to independent processes running in parallel, indicating a high potential for speedup. To the best of our knowledge, the present work is the first application of MapReduce for the verification of temporal logic properties on event traces.

2 Trace Validation Use Cases

We shall first recall basic concepts related to the validation of event traces in various contexts. For the needs of this paper, an *event trace* $m_0 m_1 \ldots$, noted $\overline{m}$, represents a sequence of events over a period of time. Each event is an individual entity, made of one or more parameter-value pairs of arbitrary names and types. The schema (that is, the number and names of each parameter in each event) is not assumed to be known in advance, or even to be consistent across all events.

2.1 Constraints on Event Sequences: Linear Temporal Logic

Given an event trace, one is then interested in expressing properties or *constraints* that must be fulfilled either by individual events or sequences thereof. A variety of formal

languages are available to describe constraints of different kinds; one of them is a logical formalism called Linear Temporal Logic (LTL). The basic building blocks of LTL formulæ are *propositional variables* p, q, ..., expressing Boolean conditions on particular messages of the trace. In the present context, each propositional variable is an assertion of the form parameter = value, which evaluates to true if the equality holds for the current message, and to false otherwise.

One can evaluate when a trace $\overline{m}$ satisfies a given formula φ, written as $\overline{m} \models \varphi$, by giving conditions to be evaluated recursively on the structure of the formula. On top of propositional variables, LTL allows *Boolean connectives* $\vee$ (or), $\wedge$ (and), $\neg$ (not), bearing their usual meaning and *temporal operators* to express constraints on the sequence of events. The temporal operator **G** means "globally"; the formula $\mathbf{G}\,\varphi$ means that formula φ is true in every event of the trace, starting from the current event. The operator **F** means "eventually"; the formula $\mathbf{F}\,\varphi$ is true if φ holds for some future event of the trace. The operator **X** means "next"; it is true whenever φ holds in the next event of the trace. Finally, the **U** operator means "until"; the formula $\varphi\,\mathbf{U}\,\psi$ is true if φ holds for all events until some event satisfies ψ. [1] The formal semantics for LTL is left out due to lack of space, but the reader is referred to [8] for a classical coverage of LTL and other temporal logics.

Two concepts bear particular importance in this paper. Given some operator $\star$ and a formula φ of the form $\star\varphi'$ or $\varphi' \star \psi$, expressions φ' and ψ are called the *direct subformulæ* of φ. Subformulæ form a partial ordering; we will denote as $\varphi' \prec \varphi$ the fact that φ' is a direct subformula of φ. The *depth* of a formula φ, noted $\delta(\varphi)$, is then defined as the maximum number of nested subformulæ it contains. For example, the expression $\mathbf{G}\,(p \wedge \mathbf{F}\,q)$ is of depth 3, and its set of subformulæ is $\{p \wedge \mathbf{F}\,q, p, \mathbf{F}\,q, q\}$. For a set of subformulæ S, we will say that φ is a (direct) *superformula* of ψ if $\varphi, \psi \in S$ and $\psi \prec \varphi$.

There exists a variety of scenarios where constraints on event traces can be modelled as LTL properties. As an example, we recall an earlier work where an Amazon bookstore business process was modelled as a set of constraints [21, p. 34]. We also mention that the same techniques used for LTL business process compliance can be reused for the verification of web service interface contracts [13], the detection of network intrusions in web server logs [16], and the analysis of system events produced by spacecraft hardware during testing [3]. This issue has also gained considerable importance in the past decade with the advent of anti-fraud regulation such as the Sarbanes-Oxley Act (SOX) [1] or the Payment Card Industry Data Security Standard (PCI) [2], which require some form of storage and analysis of log files, such as database transaction history.

3 Related Work

Existing solutions for the validation of event traces can be split into two categories. On one side are formal trace validation tools, mostly experimental or academic, offering a rich input language but for which no parallel processing algorithms are available; on the other side lie distributed log analysis products whose input language and validation capabilities are relatively limited.

[1] We implicitly assume a finite-trace semantics where $\varepsilon \not\models \mathbf{X}\,\varphi$, $\varepsilon \not\models \mathbf{F}\,\varphi$, and $\varepsilon \models \mathbf{G}\,\varphi$.

3.1 Formal Trace Analysis

A first category of tools is made of so-called "formal" trace analyzers. Complex sequential patterns of events are expressed using a rich, mathematically-based notation such as finite-state machines, temporal logic or Petri nets. Algorithms are then developed to process these specifications and automatically check that some trace satisfies the given pattern.

In this realm, a wide variety of techniques have been developed for different purposes. When the specifications are written as temporal logic formulæ, algorithms can manipulate the expressions symbolically, and progressively rewrite the original specification as the trace is being read; the pattern is violated when this rewriting process transforms the specification into a contradiction. This idea has been implemented in two independent tools, respectively based on the Maude engine [19] and the Java programming language [13].

An alternate approach consists of storing the events into a database, and to transform the sequential patterns into an equivalent database query. This has been experimented with traditional relational databases and SQL [7], and more recently using XML databases and the XQuery language [12]. The database approach has also been followed, to some degree, by the Monpoly tool [5], which associates to each event in the trace a set of conditions on its values.

ProM [22] is an open-source environment aimed at the mining of patterns in large sets of log data. Among the many plugins developed for ProM, one can find a tool for the automated verification of LTL formulæ on process logs. Also worthy of mention are Logscope [3] and RuleR [4], which use their own input language loosely based on logic and finite-state machines. However, none of the aforementioned tools is reported to offer parallel processing capabilities, and in particular the leveraging of cloud-based infrastructures, such as MapReduce, to that end.

3.2 Distributed Trace Analysis

The second category of related work comprises so-called "log analysis" solutions. Most products in that category are commercial software aimed at the filtering of event data (such as database or server logs) to search for the presence of specific patterns. Notable examples include Snare[2], ManageEngine[3] or Splunk[4]; even operating systems such as Windows provide viewing and filtering capabilities for internal events. These tools can be seen as refined variants of the well-known "Grep" function, which performs pattern matching over an input file and returns lines corresponding to some regular expression. Indeed, such mechanism has also been proposed as the basis of trace validation tools in the past [11].

A problem arises, however, when one wants to query an event trace using a more articulate query language than single-line regular expressions. Linear Temporal Logic is a prime illustration of this problem: if p and q define single event patterns, a temporal expression like $\mathbf{G}\,(p \rightarrow \mathbf{X}\,q)$ validates whether an event that satisfies p is always

[2] http://www.intersectalliance.com/projects/index.html
[3] http://www.manageengine.com
[4] http://www.splunk.com

immediately followed by an event that satisfies q. Events (or lines) are no longer compared individually, but rather with respect to their sequential relationship. The main hypothesis of the aforementioned techniques, namely that event processing can be done individually, no longer holds. If the two lines of some temporal pattern are stored on different chunks of the trace, and processed by independent parallel threads, the sequential relationship will be missed.

Therefore, in all the aforementioned solutions, the filtering process is generally limited to single events taken in isolation. For example, it is possible to obtain the list of all events satisfying some criterion on the event's attributes or to compute aggregate numerical statistics on events collected (such as total throughput, average delay, etc.), but not to fetch events in relation with other events, or satisfying some sequence or temporal pattern.

A close cousin to the approach presented in this paper has been exposed by Bauer and Falcone [6]. In this setting, multiple components in a system each observe a subset of some global event trace. Given an LTL property φ, their goal is to create sound formulæ derived from φ that can be monitored on each local trace, while minimizing inter-component communication. However, this work assumes that the projection of the global trace upon each component is well-defined and known in advance. Moreover, all components consume events from the trace synchronously, such that the distribution of monitoring does not result in a speed-up of the whole process.

4 An Overview of MapReduce

Since the emergence of the concept of cloud computing a few years ago, a variety of distributed computing environments have been released. One notable proponent is MapReduce, a framework introduced by Google in 2004 for the processing of large amounts of data [9]. It is one of the forerunners of the so-called "NoSQL" trend, which has seen the development and rising popularity of alternative data processing schemes steering away from mainstream relational databases.

4.1 Processing Steps

Figure 1 summarizes the schematics of MapReduce. Data processing starts by the reading of some piece of data (typically an input file) by an Input Reader, whose task is to convert the input stream into a set of tuples. Each tuple is a key-value pair, denoted $\langle q_i, v\rangle$, where both keys and values can be of arbitrary types.

As Figure 1 shows, multiple instances of the Input Reader can run in parallel, and typically process separate fragments of the input data simultaneously. The tuples produced by the Input Reader are then sent one by one to a Mapper, whose task is to convert each input tuple $\langle q_i, v\rangle$ into some output tuple $\langle k_i, v'\rangle$. The processing is stateless —that is, each tuple must be transformed independently of any previously-seen tuple, and regardless of the order in which tuples are received. For an input tuple, the Mapper may as well decide not to produce any output tuple.

The pool of tuples from all Mapper instances then goes through a shuffling step; all tuples with the same key are grouped and dispatched to the same instance of Reducer.

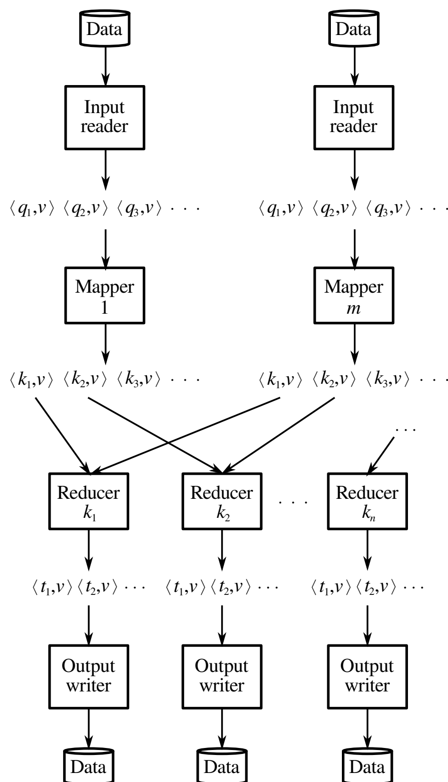


Fig. 1. The different steps of MapReduce data processing

Therefore, a Reducer that receives a tuple $\langle k_i, v\rangle$ is guaranteed to receive all other tuples $\langle k_i, v'\rangle$ for that same key k_i. For the sake of clarity, we can safely assume that each Reducer instance receives the tuples for exactly one key; we can hence parameterize each such instance with the key it has been assigned.

Contrarily to the Mapper, the Reducer receives its input tuples at once, and is hence allowed to iterate through and retain information about previously seen tuples. Again, the Reducer's task is to read the input tuples, and produce as output one final set of tuples of the form $\langle t_i, v\rangle$. This set of tuples can then be read, and formatted back to some output format by an Output Writer.

In some cases, the Input Reader and Mapper may be fusioned into a single processing step, as is the case for the Reducer and Output Writer. Moreover, some definitions of MapReduce also imply that tuples are sorted according to their value before being fed to the Reducer, although we do not assume such sorting in the present paper.

Popular frameworks such as Google's or Apache's Hadoop[5] provide an environment and code libraries allowing one to write data processing tasks as MapReduce jobs. It

[5] http://hadoop.apache.org

generally suffices to write the (Java or Python) code for the Map and Reduce phases of the processing, compile it and send it to the nodes of the cloud infrastructure.

One can see from this simple description that the keys and values produced by a processing step need not be (and generally are not) the same for input and output. In the same way, there is no fixed relationship between the number of tuples read and the number of tuples sent out; a Mapper or Reducer processing some tuple may return zero, one, or even more than one tuple as output.

Moreover, it is possible to chain multiple MapReduce phases. It suffices to take the output of the Reducers as input for a subsequent cycle of Mappers. Google's PageRank algorithm is computed through three MapReduce phases, the second of which is repeated until convergence of some numerical value is reached [17]. The algorithm for Mappers and Reducers differs from phase to phase.

Although the MapReduce scheme is arguably less natural than a classical, linear program to an inexperienced developer, its architecture presents one key advantage: once a problem has been correctly split into Map and Reduce jobs, scaling up the processing to multiple nodes in the cloud becomes straightforward. Indeed, multiple Input Readers can simultaneously take care of a separate chunk of the input data. Then, since the Map step processes each tuple regardless of any past or future tuple, an arbitrary number of Mappers can process the tuples generated by the Input Readers in parallel. Similarly, the processing done by each Reducer only requires access to tuples of the same key, which entails that up to one Reducer per key can run in parallel. All in all, the whole processing chain greatly decreases the number of steps that require to be done in sequence. A good review of MapReduce's pros and cons can be found in Lee et al. [15].

5 LTL Trace Validation with MapReduce

Despite the potential parallelism brought about by the use of the MapReduce paradigm, the fundamental question of whether LTL trace validation is parallelizable remained open until very recently. We have already shown that, if one is to leverage distributed cloud frameworks for LTL querying of event traces, simple mechanisms such as Distributed Grep and their derivatives cannot be used directly.

Kuhtz and Finkbeiner showed in 2009 that LTL path checking belongs to the complexity class $\mathsf{AC}^1(\mathsf{logDCFL})$ [14]; this result entails that the process can be efficiently split by evaluating entire blocks of events in parallel. Rather than sequentially traversing the trace, their work considers the circuit that results from "unrolling" the formula over the trace. However, while the evaluation of this unrolling can be done in parallel, a specific type of Boolean circuit requires to be built in advance, which depends on the length of the trace to evaluate. Moreover, the formal demonstration of the result shows that, while a fixed number of gates of this circuit can be contracted in parallel at each step of the process, the algorithm itself requires a shared and global access to the trace from every parallel process. As such, it does not lend itself directly to distributed computing frameworks.

We take an alternate approach, and describe in this section an algorithm that performs LTL trace validation on event traces directly using the MapReduce computing paradigm. The algorithm evaluates an LTL formula in an iterative fashion. At the first iteration, all

the states where ground terms are true are evaluated. In the next iteration, these results are used to evaluate all subformulæ directly using one of those ground terms. More generally, at the end of iteration i of the process, the events where all subformulæ of depth i hold are computed. It follows that, in order to evaluate an LTL formula of depth n, the algorithm will require exactly n MapReduce cycles. Each map-reduce cycle effectively acts as a form of temporal tester [18] processing a trace made of the evaluation of lower-level testers.

This does not mean, however, that the event trace must be read as many times. In fact, the input trace is entirely read only once, at the first iteration of the procedure. Afterwards, only sequential numbers referring to those events need to be passed between mappers and reducers. The contents of the original trace never need to be consulted ever again.

The system is described by providing details on each component of the MapReduce algorithm described in Figure 1. We suppose that every instance of the process (Input Reader, Mapper, Reducer, Output Writer) are parameterized by the formula to verify φ, and the length of the trace, ℓ.

5.1 Trace Format and Input Reader

The Input Reader is responsible for the processing of an event trace chunk and the generation of a first set of key-value tuples from that chunk. We assume that each event is sequentially numbered, or that its position in the whole trace can be easily computed otherwise. For some event e, we will refer by $\#(e)$ this event's sequential number.

The Input Reader, whose algorithm is given in Figure 2(a), iterates through each event of the trace chunk, and evaluates on each event the ground terms present in φ. For each propositional variable a and each event e, it outputs a tuple $\langle a, (i, 0)\rangle$ where i is the event's sequential number in the trace. The ground terms of a formula φ are computed using the function atom(φ).

One should remark that this initial processing step does not require that the trace be located on a single node, or even that each node's fragment consist of blocks of successive events. As long as each event can be placed in some total order (such as the

Procedure Input Reader$_{\varphi,\ell}$(*chunk*)
  $A[]$:= atoms(φ)
  For each e in *chunk* **do**
    $i := \#(e)$
    For each a in A **do**
      If $e \models a$ **then**
        output $\langle a, (i, 0)\rangle$
      End if
    End
  End

(a) Pseudo-code for the LTL Input Reader

Procedure Mapper$_{\varphi,\ell}$($\langle \psi, (n, i)\rangle$)
  If $i \leq \delta(\psi)$
    $S[]$:= superformulæ(φ, ψ)
    For each ξ in S **do**
      output $\langle \xi, (\psi, n, i+1)\rangle$
    End If
  End
  End If

(b) Pseudo-code for the LTL Mapper.

Fig. 2. InputReader and Mapper

value of a global, shared clock), any number of nodes can host any subset of the trace. This is particularly useful if event collection and storage is performed in a distributed fashion.

5.2 Mapper

The Mapper takes as input tuples of the form $\langle \psi, (n,i) \rangle$, either from the Input Reader or from the output of a previous MapReduce cycle. Each such tuple reads as “the process is at iteration i, and subformula ψ is true on event n”. One can see, in particular, how the tuples returned by the Input Reader express this fact for ground terms of the formula to verify.

The Mapper, shown in Figure 2(b), is responsible for lifting these results, computed for some ψ, up into every formulæ ψ' of which ψ is a *direct* subformula (these are obtained using the function superformulæ(φ, ψ)). For example, if the states where p is true have been computed, then these results can be used to determine the states where $\mathbf{F}\, p$ is true. To this end, the Mapper takes every tuple $\langle \psi, (n,i) \rangle$, and will output a tuple $\langle \psi', (\psi, n, i+1) \rangle$, where ψ is a subformula of ψ'. This tuple reads “the process is at iteration $i+1$, subformula ψ is true on event n, and this must be used to evaluate ψ'”. In the definition of the reducer, ξ stands for whatever subformula the input tuple is build from.

5.3 Reducer

The mappers are mostly used to prepare results from the last iteration to be used for the current iteration. In contrast, each instance of the reducer performs the actual evaluation of one more layer of the temporal formula to verify. After the shuffling step, each individual instance of the reducer receives all generated tuples of the form $\langle \psi', (\psi, n, i) \rangle$ for some formula ψ', and where ψ is a direct subformula of ψ'. Hence, the reducer is given information on all the event numbers for which ψ' holds, and is asked to compute the states where ψ holds based on this information. This task can then be decomposed depending on the top-level connective in ψ'. The algorithm for each reducer is shown in Figure 3.

When the top-level formula to evaluate is $\mathbf{X}\, \psi$, the events that satisfy the formula are exactly those immediately preceding an event where ψ holds. Consequently, the reducer iterates through its input tuples of $\langle \mathbf{X}\, \psi, (\psi, n, i) \rangle$ and produces for each one an output tuple $\langle \mathbf{X}\, \psi, (n-1, i) \rangle$.

When the top-level formula to evaluate is $\mathbf{F}\, \psi$, the events that satisfy the formula are exactly those for which some event in the future is such that ψ holds. The corresponding reducer iterates through the input tuples and computes the highest event number c for which ψ holds. All events preceding c satisfy $\mathbf{F}\, \psi$. Consequently, the reducer generates as output all tuples of the form $\langle \mathbf{F}\, \psi, (k, i) \rangle$, for each $k \in [0, c]$.

The reducer for $\neg\psi$ iterates through all tuples and stores in a Boolean array whether $e_i \models \psi$ for each event i in the trace. It then outputs a tuple $\langle \neg\psi, (k, i) \rangle$ for all event numbers k that were not seen in the input. The reducer for $\mathbf{G}\, \psi$ proceeds in reverse. It first iterates through all tuples in the same way. If we let c be the index of the last

```
Procedure Reducer_{φ,ℓ}(F ψ, tuples[])
  m := -1
  For each ⟨F ψ, (ξ, n, i)⟩ in tuples do
    If n > m then m := n
  End
  For k from 0 to m do
    output ⟨F ψ, (k, i)⟩
  End

Procedure Reducer_{φ,ℓ}(¬ψ, tuples[])
  For each ⟨¬ψ, (ξ, n, i)⟩ in tuples do
    s[n] := ⊤
  End
  For k from 0 to ℓ do
    If s[k] ≠ ⊤ then
      output ⟨¬ψ, (k, i)⟩
    End If
  End

Procedure Reducer_{φ,ℓ}(G ψ, tuples[])
  For each ⟨G ψ, (ξ, n, i)⟩ in tuples do
    s[n] := ⊤
  End
  For k from ℓ to 0 do
    If s[k] ≠ ⊤ break
    output ⟨G ψ, (k, i)⟩
  End

Procedure Reducer_{φ,ℓ}(ψ ∨ ψ', tuples[])
  For each ⟨ψ ∨ ψ', (ξ, n, i)⟩ in tuples do
    If δ(ψ ∨ ψ') ≠ i then
      output ⟨ξ, (n, i)⟩
    Else
      output ⟨ψ ∨ ψ', (n, i)⟩
    End If
  End
```

```
Procedure Reducer_{φ,ℓ}(X ψ, tuples[])
  For each ⟨X ψ, (ξ, n, i)⟩ in tuples do
    output ⟨X ψ, (n − 1, i)⟩
  End

Procedure Reducer_{φ,ℓ}(ψ ∧ ψ', tuples[])
  For each ⟨ψ ∧ ψ', (ξ, n, i)⟩ in tuples do
    If δ(ψ ∧ ψ') ≠ i then
      output ⟨ξ, (n, i)⟩
      s_ξ[n] := ⊤
    End If
    If s_ψ[n] := ⊤ and s_ψ'[n] := ⊤ then
      output ⟨ψ ∧ ψ', (n, i)⟩
    End If
  End

Procedure Reducer_{φ,ℓ}(ψ U ψ', tuples[])
  For each ⟨ψ U ψ', (ξ, n, i)⟩ in tuples do
    If δ(ψ U ψ') ≠ i then
      output ⟨ξ, (n, i)⟩
    End If
    s_ξ[n] := ⊤
  End
  b := ⊥
  For k from ℓ to 0 do
    If s_ψ'[n] = ⊤ then
      output ⟨ψ U ψ', (k, i)⟩
      b := ⊤
    Else If s_ψ[n] := ⊤ and b = ⊤ then
      output ⟨ψ U ψ', (k, i)⟩
    Else
      b := ⊥
    End If
  End
```

Fig. 3. Pseudo-code for the LTL Reducers

event for which ψ does not hold, the reducer will then output all tuples $\langle \mathbf{G}\,\psi, (k,i)\rangle$ for $k \in [c+1, \ell]$. This indeed corresponds to all events for which $\mathbf{G}\,\psi$ holds.

The case of binary connectives $\vee$ and $\wedge$ is slightly more delicate. Special care must be taken to persist tuples whose result will be used in a later iteration. Consider the case of formula $(\mathbf{F}\,p) \wedge q$. The states where ground terms p and q hold will be computed by the Input Reader at iteration 0. However, although q is a direct subformula of $(\mathbf{F}\,p) \wedge q$, one has to wait until iteration 2 to combine it to $\mathbf{F}\,p$, evaluated at iteration 1. More precisely, a tuple $\langle \psi \star \psi', (\psi, n, i)\rangle$ can only be evaluated at iteration $\delta(\psi \star \psi')$; in all previous iterations, tuples $\langle \psi, (n,i)\rangle$ must be put back in circulation. The first condition in both reducers' algorithm takes care of this situation.

Otherwise, when the top-level formula to evaluate is $\psi \vee \psi'$, the reducer outputs a tuple $\langle \psi \vee \psi', (n,i)\rangle$ whenever it reads input tuples $\langle \psi \vee \psi', (\psi, n, i)\rangle$ or $\langle \psi \vee \psi', (\psi', n, i)\rangle$.

When the top-level formula is $\psi \wedge \psi'$, the reducer must memorize event numbers n for which it has read tuples $\langle \psi \wedge \psi', (\psi, n, i) \rangle$ and $\langle \psi \wedge \psi', (\psi', n, i) \rangle$, and outputs $\langle \psi \wedge \psi', (n, i) \rangle$ as soon as it has seen both.

The last case to consider is that of a formula of the form $\psi \, \mathbf{U} \, \psi'$. The reducer first iterates through all its input tuples and memorizes the event numbers for which ψ holds, and those for which ψ' holds. It then proceeds backwards from the last event of the trace, and outputs $\langle \psi \, \mathbf{U} \, \psi', (n, i) \rangle$ for some state n if ψ' holds for n, or if ψ holds for n and there exists an uninterrupted sequence of states leading to a state n' for which ψ' holds. This last information is handled through the Boolean variable b.

As one can see, the tuples produced by each reducer is of the form $\langle \psi, (n, i) \rangle$, carrying the exact same meaning as those originally produced by the Input Reader, albeit for formulæ of greater depth. Therefore, the result of one MapReduce cycle can be fed back as input of a new cycle; as we have seen, it takes exactly $\delta(\varphi)$ such cycles to completely evaluate some LTL formula φ.

5.4 Output Writer

At the end of the last MapReduce cycle, one is left with tuples $\langle \varphi, (n, \delta(\varphi)) \rangle$. These represent all event numbers n such that $\overline{m}^n \models \varphi$. The output writer, shown in Figure 4, translates the last set of tuples into the truth value of the formula to evaluate. By the semantics of LTL, an event trace satisfies the formula φ if $\overline{m}^0 \models \varphi$. Hence the output writer simply writes "true" if $\langle \varphi, (0, \delta(\varphi)) \rangle$ is found, and false otherwise.

```
Procedure Output Writer_{φ,ℓ}(tuples[])
  For each ⟨φ,(n,i)⟩ in tuples do
    If n = 0 then
      output "Formula is true"
      Break
    End if
  End
  output "Formula is false"
```

Fig. 4. Pseudo-code for the LTL Output Writer.

6 Experimental Results

To illustrate the concept and evaluate its feasibility, a proof-of-concept implementation of the algorithm was developed. The algorithm consists of the implementation of two Java classes providing the Map and Reduce algorithms described earlier. For the needs of the experiments, the actual coordination of Map and Reduce jobs is done locally on a single machine using a single-thread implementation of MapReduce: the data source is fed tuple by tuple to the mapper, the output tuples are collected, split according to their keys, and each list is sent to the reducer, again in a sequential fashion. As such, this sequential workflow reproduces exactly the processing done by MapReduce environments, without the distribution of computation. This was done on purpose, so that

the running time of each mapper and reducer instance could be easily measured. The potential speedup incurred by parallelizing will be computed from those measurements in Section 6.2.

6.1 Experimental Setup

At the onset, a first observation that can be made is that the validator is very simple: excluding the code for coordinating Mappers and Reducers, the total implementation of the validator amounts to 1,000 lines of Java code. This should be put in contrast with another simple trace validator from the same author, BeepBeep, which is also implemented in Java and rather uses the classical, on-the-fly algorithm for the evaluation of LTL formulæ on traces [13]; BeepBeep is is made up of twice as many lines of Java code.

To assess the running time of the MapReduce validation algorithm, we built a dataset consisting of traces of randomly-generated events, with each event being made of up to ten random parameters, labelled $p_0, \ldots p_9$, each carrying five possible values. Each trace has a length between 1 and 100,000 events, and 500 such traces were produced. In total, this dataset amounts to more than one gigabyte of randomly-generated event data.

Four properties, with increasing complexity, were verified on these traces. Property #1 is $\mathbf{G}\, p_0 \neq 0$, and simply asserts that in every event, parameter p_0, when present, is never equal to 0. Property #2 is $\mathbf{G}\,(p_0 = 0 \rightarrow \mathbf{X}\, p_1 = 0)$: it expresses the fact that whenever $p_0 = 0$ in some event, the next event is such that $p_1 = 0$. Property #3 is a generalization of Property #2:

$$\forall x \in [0,9] : \mathbf{G}\,(p_0 = x \rightarrow \mathbf{X}\, p_1 = x)$$

This property asserts that whatever value taken by p_0 will be taken by p_1 in the next event. The universal and existential quantifiers are meant as a shorthand notation; the actual LTL formula to be validated is the logical conjunction of the previous template for all possible values of x between 0 and 9, and reads

$$(\mathbf{G}\,(p_0 = 0 \rightarrow \mathbf{X}\, p_1 = 0)) \wedge (\mathbf{G}\,(p_0 = 1 \rightarrow \mathbf{X}\, p_1 = 1)) \ldots$$

Finally, Property #4 checks that *some* parameter p_m alternates between two possible values; this is true when the value of p_m in the current event is the same as the value two events from the current one, and is written:

$$\exists m \in [0,9] : \forall x \in [0,9] : \mathbf{G}\,(p_m = x \rightarrow \mathbf{X}\,\mathbf{X}\, p_m = x)$$

Again, the quantifiers are meant as a shorthand.

6.2 Results

Each formula was validated on each trace, and various statistics on the process were computed and are shown in Table 1.

The first measurement is the number of tuples produced by the algorithm. This value is taken as the sum of T_i, the total number of tuples processed at the Map phase of each

Table 1. Verification statistics for each of the four properties. All values are averaged over the 500 traces used in the experiment.

	Property #1	Property #2	Property #3	Property #4
Number of tuples	55,009	119,871	599,425	4,987,124
Time per event (μs)	19	23	75	985
Sequential ratio	100 %	92 %	19 %	3 %
Inferred time per event (μs)	19	21	14	30

map-reduce cycle number i, for all cycles $i \in [1, \delta(\varphi)]$. One can see that the number of tuples increases with the complexity of the formula: while Property #1 produces 55,000 tuples, the validation of Property #4 on a trace generates on average almost 5 million such units.

Using MapReduce inherently implies a tradeoff between processing speed and bandwidth consumed; the proposed algorithm is no different in that respect. While the number of tuples can seems large at first sight, we shall mention that generally, cloud providers such as Amazon EC2 charge users for CPU time, not for bandwidth —hence decreasing processing time is the key factor. Also note that in closed environments (server farms, clusters) bandwidth consumption is not really an issue.

The second measurement is the running time per event. For each trace, the running time per event is the total processing time divided by the number of events in the trace; the value shown in Table 1 is the average of these values over all traces. One can see that the MapReduce algorithm takes between 20 and approximately 1,000 microseconds per event, and that this value grows with the complexity of the formula to verify.

We also computed the "sequential ratio" of the validation process. At each map-reduce cycle, we keep the largest number of tuples processed by a single instance of a reducer. This value, noted t_i, represents the minimum number of tuples that must be processed sequentially in that particular cycle. If all reducers for that cycle were allowed to run in parallel, and assuming similar processing time for each tuple, the ratio t_i/T_i is an indicator of the time the "parallel" cycle requires with respect to the "sequential" version. The global sequential ratio shown in Table 1 is taken as

$$s = \frac{\sum_{i=1}^{\delta(\varphi)} t_i}{\sum_{i=1}^{\delta(\varphi)} T_i}$$

This sequential ratio shows one of the limits of the validation algorithm in its present incarnation: the potential for parallelism is bounded by the structure of the formula to validate, as there can be at most one instance of reducer for every possible subformula of the property to verify. Therefore, for simple formulæ such as Property #1 and #2, which have very few different subformulæ at each map-reduce cycle, almost all the work must be done sequentially (100% in the case of Property #1, and 92% in the case of Property #2). However, as soon as the property becomes more complex, as is the case for Properties #3 and #4, the situation is reversed, and each reducer handles a small fraction of the total number of tuples. Property #4 is most dramatic in that

respect, since 97% of all tuples involved can be processed in parallel. The presence of quantifiers accounts for a large part of this phenomenon, as it rapidly blows up the size of the actual LTL formula passed to the trace validator: 50 copies of the same template are validated, with various combinations of values for m and x.

From the sequential ratio s and the average sequential running time per event r obtained for each property, we can then infer the average validation time in the maximally-parallel case by computing $r \times s$; this inferred running time is shown in the last line of Table 1. While these figures should be interpreted with caution at this point in the study, it is relatively safe to assume that the distributed processing of a trace should not exceed 50 μs per event, regardless of which of the four properties is validated.

7 Conclusion

In this paper, we have presented an algorithm for the automated validation of Linear Temporal Logic properties on large traces of events using the MapReduce development framework. We have shown experimentally on a sample dataset how this algorithm presents reasonable running times even when the MapReduce environment is restricted to a single thread. In addition, the breaking up of the algorithm into several phases of independent mappers and reducers presents the potential of reducing the number of operations that must be performed linearly by executing these processes in parallel, yielding a potential speedup of 90% in some cases. As far as we know, this work is the first published algorithm that leverages the MapReduce framework for the validation of temporal logic properties on large event traces. It opens the way to the use of cloud computing services for the efficient compliance checking of program traces and event logs of various kinds.

The promising results obtained on the proof-of-concept implementation discussed in this paper lead to a number of extensions and improvements over the current method. First, the algorithm presents an interest in that it can be reused as a basis for other temporal languages that intersect with LTL. This is the case, for example, of specifications written as finite-state machines, PSL [10] or DecSerFlow [21]. Second, the technique itself could be expanded to take into account data parameters and quantification; the formulæ described in Section 6.1 gave a foretaste of such quantification and initial results indicate that quantification is a fertile ground for parallelism. The proposed implementation is currently being ported as a free software suite for Apache Hadoop.

Finally, we have seen that the potential for parallelism is bounded by the structure of the formula to validate, as there can be at most one instance of reducer for every possible subformula of the property to verify. This entails that one cannot freely distribute the processing of the trace to an arbitrary number of parallel processes: for a simple formula, or one that contains few nested expressions, few reducers can be started in parallel. Therefore, a sought after refinement of the current method is currently being worked on, which will allow multiple Reducer instances for the same key to be merged in a later step.

References

1. An act to protect investors by improving the accuracy and reliability of corporate disclosures made pursuant to the securities laws, and for other purposes, U.S. Pub.L. 107-204, 116 Stat. 745 (July 30, 2002)
2. Payment card industry data security standard, version 2.0 (2010), https://www.pcisecuritystandards.org/security_standards/pci_dss.shtml
3. Barringer, H., Groce, A., Havelund, K., Smith, M.: Formal analysis of log files. Journal of Aerospace Computing, Information, and Communication 7(11), 365–390 (2010)
4. Barringer, H., Rydeheard, D., Havelund, K.: Rule systems for run-time monitoring: From Eagle to RuleR. Journal of Logic and Computation 20(3), 675–706 (2010)
5. Basin, D., Klaedtke, F., Müller, S.: Policy Monitoring in First-Order Temporal Logic. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 1–18. Springer, Heidelberg (2010)
6. Bauer, A., Falcone, Y.: Decentralized LTL monitoring. Technical Report arXiv:1111.5133v3 (2011)
7. Böhlen, M.H., Chomicki, J., Snodgrass, R.T., Toman, D.: Querying TSQL2 Databases with Temporal Logic. In: Apers, P.M.G., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 325–341. Springer, Heidelberg (1996)
8. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (2000)
9. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
10. Eisner, C., Fisman, D.: A Practical Introduction to PSL. Springer (2006)
11. Garavel, H., Mateescu, R.: SEQ.OPEN: A Tool for Efficient Trace-Based Verification. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 151–157. Springer, Heidelberg (2004)
12. Hallé, S., Villemaire, R.: XML Methods for Validation of Temporal Properties on Message Traces with Data. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part I. LNCS, vol. 5331, pp. 337–353. Springer, Heidelberg (2008)
13. Hallé, S., Villemaire, R.: Runtime enforcement of web service message contracts with data. IEEE Transactions on Services Computing (2011), doi:10.1109/TSC.2011.10
14. Kuhtz, L., Finkbeiner, B.: **LTL** Path Checking Is Efficiently Parallelizable. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009, Part II. LNCS, vol. 5556, pp. 235–246. Springer, Heidelberg (2009)
15. Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with MapReduce: a survey. SIGMOD Record 40(4), 11–20 (2011)
16. Naldurg, P., Sen, K., Thati, P.: A Temporal Logic Based Framework for Intrusion Detection. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 359–376. Springer, Heidelberg (2004)
17. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab (November 1999)
18. Pnueli, A., Zaks, A.: On the Merits of Temporal Testers. In: Grumberg, O., Veith, H. (eds.) 25 Years of Model Checking. LNCS, vol. 5000, pp. 172–195. Springer, Heidelberg (2008)
19. Rosu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. 12(2), 151–197 (2005)
20. van der Aalst, W.M.P.: Distributed Process Discovery and Conformance Checking. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 1–25. Springer, Heidelberg (2012)
21. van der Aalst, W.M.P., Pesic, M.: Specifying and monitoring service flows: Making web services process-aware. In: Baresi, L., Nitto, E.D. (eds.) Test and Analysis of Web Services, pp. 11–55. Springer (2007)
22. Verbeek, H.M.W., Buijs, J.C.A.M., van Dongen, B.F., van der Aalst, W.M.P.: XES, XESame, and ProM 6. In: Soffer, P., Proper, E. (eds.) CAiSE Forum 2010. LNBIP, vol. 72, pp. 60–75. Springer, Heidelberg (2011)

Path-Aware Time-Triggered Runtime Verification

Samaneh Navabpour[1], Borzoo Bonakdarpour[2], and Sebastian Fischmeister[1]

[1] Department of Electrical and Computer Engineering, University of Waterloo
{snavabpo,sfischme}@uwaterloo.ca
[2] School of Computer Science, University of Waterloo
borzoo@cs.uwaterloo.ca

Abstract. A time-triggered monitor runs in parallel with the program under inspection and periodically samples the program state to evaluate a set of properties. However, a time-triggered monitor working with a fixed sampling frequency often suffers from *redundant* sampling, which results in excessive overhead. In this paper, we propose an effective approach to reduce redundant sampling. Our approach calculates the sampling frequency with respect to the program behavior at run time. We further advance this approach to dynamically adjust the sampling frequency at run time by predicting the program behavior using symbolic execution. Experiments show that our approach reduces the sampling frequency, runtime overhead, and the number of redundant samples by up to 3.5 times, 69%, and 86%, respectively.

1 Introduction

Achieving system correctness is a major problem for today's large software systems. A recent NIST report estimates that 59.6 billion dollars are lost every year because of software errors [11]. Verification and testing are arguably the two most common approaches to ensure program correctness. However, verification may suffer from the state explosion problem, and testing may not be able to cover all possible execution scenarios of the system. These limitations argue for *runtime verification* [3,9,12], where a *monitor* inspects a program to evaluate a set of properties at run time.

Most monitoring approaches in runtime verification are *event-triggered*. In these approaches, the occurrence of an event of interest triggers the monitor for property evaluation. This technique leads to defects such as *unpredictable monitoring overhead* and potentially *bursts of monitoring invocations* at run time. Such defects can cause unpredictable behavior at run time; especially in real-time embedded safety/mission-critical systems, where it can result in catastrophic consequences. To tackle these drawbacks, in [6], we proposed *time-triggered* runtime verification, where the monitor runs in parallel with the program and samples the program state at *fixed* time intervals (i.e., the sampling period) to evaluate a set of properties. A challenge in implementing such a monitor is to compute the *longest sampling period* (LSP), such that all events of interest are observed. The approach

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 199–213, 2013.
© Springer-Verlag Berlin Heidelberg 2013

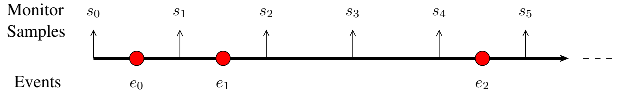


Fig. 1. Redundant monitoring intervention with fixed sampling period

in [6] computes the *fixed* longest sampling period with respect to all events of interest in a program by using the control-flow graph (CFG) of the program.

Although employing a time-triggered monitor results in observing bounded monitoring overhead and predictable monitoring invocation at run time [6], applying the fixed longest sampling period may result in unnecessary monitoring invocations. For example, consider the execution time line in Figure 1. Events e_0, e_1, and e_2 occur on the execution path and the monitor samples the program execution with the fixed longest sampling period at points $s_0, s_1, \ldots, s_5$. It is straightforward to observe that samples s_3 and s_4 are redundant, as no events of interest occur from sample s_2 until sample s_4. A large number of redundant samples cause the monitor to impose excessive overhead at run time. Thus, it is highly desirable to design a time-triggered monitor that can adjust the sampling period at run time based upon the characteristics of the program execution path.

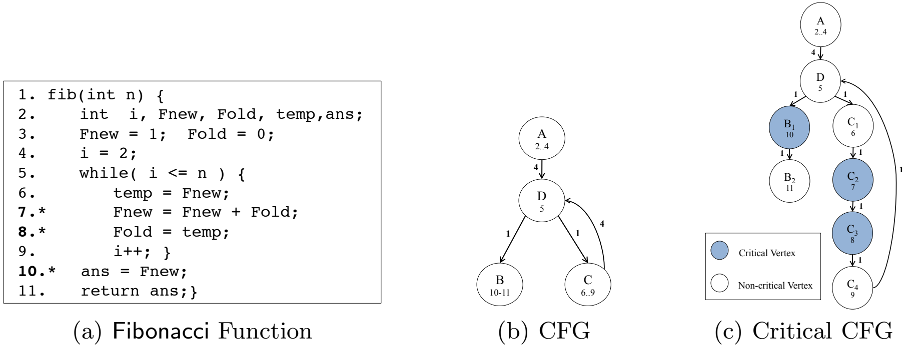
With this motivation, in this paper, we propose the notion of *path-aware time-triggered monitoring* for sequential programs. We present an approach which leverages symbolic execution [16] to predict the execution path of the program with respect to its input values. Hence, to compute the *path-aware* longest sampling period, our approach only considers the events of interest executed within the predicted execution path.

Moreover, we introduce a method that allows a time-triggered monitor to *adapt* its sampling period at run time, based on the events of interest to be executed in the near future. In particular, our method partitions the predicted execution path of the program into *LSP regions* and computes the path-aware longest sampling period of each region. Hence, when the program execution enters a new region at run time, the monitor adapts the path-aware longest sampling period of that region, thus avoiding redundant samples. Our method also ensures that the overhead of adapting the sampling period at run time imposes low overhead.

Finally, we present a fully automated tool chain that implements both path-aware and adaptive path-aware monitoring approaches. We study the effect of both approaches with the SNU benchmark suite [1]. The experiments show highly promising results; i.e., our approach reduces the sampling frequency, runtime overhead, and the number of redundant samples by up to 3.5 times, 69%, and 86%, respectively.

2 Background

In time-triggered runtime verification (TTRV) a monitor samples the *state* of a program at regular time intervals to evaluate a set of properties. The state of the



(a) Fibonacci Function (b) CFG (c) Critical CFG

Fig. 2. Fibonacci and its CFG and critical CFG

program is defined by a set of *variables of interest* that affect the evaluation of the set of properties. A time-triggered monitor exhibits the following characteristics which make it suitable for monitoring time-sensitive systems: (1) bounded overhead (i.e., the monitor imposes approximately the same amount of overhead at each sample), and (2) predictable invocation (i.e., the monitor samples the program execution at constant intervals, called the *sampling period*).

Let P be a program and Π be a logical property, where P is expected to satisfy Π. Let $\mathcal{V}_\Pi$ be the set of variables of interest. We leverage control-flow analysis to estimate the time intervals between consecutive state changes with respect to variables in $\mathcal{V}_\Pi$.

Definition 1. *The* control-flow graph *of a program P is a weighted directed simple graph $CFG_P = \langle V, v^0, A, w\rangle$, where:*

- *V: is a set of* vertices*, each representing a basic block of P. Each basic block consists of a sequence of instructions in P.*
- *v^0: is the* initial vertex *with indegree 0, which represents the initial basic block of P.*
- *A: is a set of* arcs *of the form (u, v), where $u, v \in V$. An arc (u, v) exists in A, if and only if the execution of basic block u can immediately lead to the execution of basic block v.*
- *w: is a function $w : A \to \mathbb{N}$, which defines a* weight *for each arc in A. The weight of an arc is the* best-case execution time *(BCET) of the source basic block.* □

For example, consider the Fibonacci function in Figure 2(a) from the fibcall program of the SNU benchmark [1]. Assuming that the BCET of each instruction is one time unit in this example, the resulting CFG_P is shown in Figure 2(b), where each vertex is annotated with the corresponding line numbers of the program. In order to calculate the longest sampling period (LSP), we modify CFG_P in two steps:

Step 1 (Extracting the Critical Vertices)
In this step, we modify CFG_P, such that each *critical instruction* (i.e., an instruction that updates the value of a variable in $\mathcal{V}_\Pi$) resides in a vertex by itself. We refer to such a vertex as a *critical vertex*. For example, if $\mathcal{V}_\Pi = \{\mathsf{Fnew}, \mathsf{Fold}, \mathsf{ans}\}$, then instructions 7, 8 and, 10 are critical instructions in Fibonacci and Figure 2(c) shows the evolved CFG_P. We call this graph a *critical control-flow graph (critical CFG)*.

Step 2 (Calculating the Longest Sampling Period)
As mentioned earlier, the main challenge in using TTRV is accurate state reconstruction. To preserve all state changes, the monitor must sample at a sampling period such that it does not overlook any state changes that could occur in P at run time. This sampling period is called the *longest sampling period* (LSP).

Definition 2. *Let* $CFG = \langle V, v^0, A, w\rangle$ *be a critical control-flow graph and* $V_c \subseteq V$ *be the set of critical vertices of* CFG*. The* longest sampling period (LSP) *for* CFG *is the minimum shortest path between two vertices in* V_c*.* □

For example, LSP of Fibonacci is 1 time unit. In this paper, we refer to such a sampling period as *fixed* LSP. Since there needs to be an initial sample in the beginning of execution, we assume that the initial vertex v^0 is also a critical vertex. We also assume that precompiled libraries are correct and, hence, need not be monitored. Thus, these libraries do not take part in calculating LSP.

In [6], we observed that a time-triggered monitor with fixed LSP imposes 170% overhead on average. This is due to *redundant sampling*; i.e., the monitor may take samples even when the program has not executed a critical instruction since the last sample. To reduce the number of redundant samples, one can employ *auxiliary memory* to build a *history* of state changes between consecutive samples. In other words, let v be a critical vertex in a critical CFG, where the critical instruction $Inst$ updates the value of a variable x. The following graph transformation [6] results in a new critical CFG with a greater fixed LSP: it (1) removes v, (2) merges the incoming and outgoing arcs of v, and (3) adds an instruction $Inst' : x' \leftarrow x$ after instruction $Inst$ in the program source code, where x' is an auxiliary memory location. For example, applying this transformation to vertex C_2 in Figure 2(c) results in a graph where C_2 and all its incoming and outgoing arcs are removed, and a new arc from C_1 to C_3 with weight 2 is added. In the new graph, we have fixed $LSP = 2$.

3 Path-Aware Time-Triggered Monitoring

Although the experimental results from [6] show that by using auxiliary memory, the monitoring overhead can be reduced on average by 60%, the overhead still remains larger than the overhead of event-triggered runtime verification frameworks [8, 13, 15]. This is because the method in [6] uses the CFG of the program to compute the fixed LSP and not the realized execution paths at run time. Thus, the fixed LSP tends to be conservative; i.e., the monitor may take redundant samples.

To clarify, consider the critical CFG in Figure 2(c) with fixed $LSP = 1$. This value is optimal when the program executes the instruction sequence $\langle v_A, v_D, v_{C_1}, v_{C_2}, v_{C_3}, v_{C_4}, v_D, v_{B_1}, v_{B_2}\rangle$ at run time (e.g., n = 2 as input). On the contrary, when the program executes the instruction sequence $\langle v_A, v_D, v_{B_1}, v_{B_2}\rangle$ (e.g., n = 0 as input), the fixed LSP is too small. In the latter case, the monitor can observe all values of variables of interest with $LSP = 5$. In this case, the monitor takes 85% less redundant samples compared to using the fixed $LSP = 1$.

The above example motivates the idea to compute LSP with respect to the program's execution path. This is achieved before the program runs by the following two steps:

1. Predict the program's execution path with respect to the program's input values.
2. Using the predicted path from Step 1, compute LSP by only considering the sequence of critical instructions within the execution path.

Step 1 (Path Prediction)
Let P be a program, $CFG_P = \langle V, v^0, A, w\rangle$ be its control-flow graph, and $\mathcal{I}_P$ be the *input domain* of P. The input domain is the set of all values that can be provided as input to P.

Definition 3. *An* execution path *is a sequence of the form* $\gamma = \langle (v_0, \omega_0, v_1), (v_1, \omega_1, v_2), ...\rangle$, *where:*
- $v_0 = v^0$.
- *For all* $i \geq 0$, $v_i \in V$.
- *For all* (v_i, ω_i, v_{i+1}), *where* $i \geq 0$, *there exists an arc* (v_i, v_{i+1}) *in* A.
- *For all* $i \geq 0$, $\omega_i = w(v_i)$.
- *If* P *is a terminating program, then* $\gamma = \langle (v_0, \omega_0, v_1), \ldots, (v_{n-1}, \omega_{n-1}, v_n)\rangle$ *is finite and* v_n *is a vertex in* V *with outdegree of zero.* □

For instance, in Fibonacci, the input value n=0 leads to the execution of path $\gamma_1 = \langle (v_A, 4, v_D), (v_D, 1, v_{B_1}), (v_{B_1}, 1, v_{B_2})\rangle$. In this paper, we only focus on *possible* execution paths; i.e., an execution path for which there exists some input in $\mathcal{I}_P$ that enables P to take the path at run time. We denote the set of all (possible) execution paths of program P as $\mathcal{P}_P$.

To *predict* the execution path(s) of P, we require a mechanism that takes the value-set of the inputs of P and returns the set of execution paths of P. We refer to this mechanism as the *path prediction* function.

Definition 4. *Let* P *be a program. The* path prediction *function* $\psi_P : \mathcal{I}_P \rightarrow 2^{\mathcal{P}_P}$, *maps an input from the input domain of* P *to a subset of execution paths of* P. □

Note that in a deterministic program, ψ_P maps an input to one and only one execution path. In practice, ψ_P can be implemented using symbolic execution [16] before the actual program run. In particular, symbolic execution creates a bijection from each execution path γ of a program to a *path constraint.* A path constraint projects the conditions (e.g., in if-then-else and loop structures) that need to be satisfied in order for the program to execute γ at run time.

Notation: For each path γ, $PC(\gamma)$ denotes the path constraint of γ. For instance, for execution path $\gamma_1 = \langle (v_A, 4, v_D), (v_D, 1, v_{B_1}), (v_{B_1}, 1, v_{B_2}) \rangle$ of Fibonacci, $PC(\gamma_1) = (n < 2)$. Thus, ψ_P in fact, uses the input values (e.g., n = 0) and $PC(\gamma)$ to find the path constraint(s) satisfied by the input values, and extracts the set of associated execution path(s).

Step 2 (Computing the Sampling Period)
Given a predicted execution path γ from Step 1, the *LSP* of γ is computed as follows. We refer to the following sampling period as *path-aware longest sampling period (paLSP).*

Definition 5. *Let $x \in \mathcal{I}_P$ be an input and $\psi_P(x) = \{\gamma\}$. The* path-aware longest sampling period *paLSP for γ is the minimum subpath length between two critical vertices of γ.* □

Now, consider a program that includes a loop structure. It is likely that an execution path γ of the program has multiple occurrences of a subpath of the form $\langle (v_i, \omega_i, v_{i+1}), (v_{i+1}, \omega_{i+1}, v_{i+2}), ..., (v_n, \omega_n, v_i) \rangle$. We refer to such a subpath as a *loop sequence*. Observe that multiple occurrences of a loop sequence in γ does not affect the value of *paLSP*. Hence, before computing *paLSP*, our approach transforms γ, such that each of its loop sequences consecutively occur at most *twice*. We refer to the resulting execution path as the *unique* version of γ.

Definition 6. *Let γ be an execution path. The* unique execution path *of γ, denoted γ^{unq}, is a path, where each consecutive occurrence of at least 2 for a loop sequence L in γ is represented by 2 consecutive occurrences of L in γ^{unq}.* □

Thus, our approach computes *paLSP* of γ using γ^{unq}. For a program P, an algorithm for computing *paLSP* of a unique path γ^{unq} takes the following steps:

1. Extract the subgraph CFG'_P of CFG_P that only includes path γ^{unq}.
2. Create the critical control-flow graph of CFG'_P.
3. Compute *paLSP* according to Definition 5.

We refer to TTRV that uses *paLSP* as *path-aware TTRV* (pa-TTRV). In Section 6, we will show that pa-TTRV is quite effective in practice to reduce the runtime overhead.

4 Adaptive Path-Aware Time-Triggered Monitoring

Although pa-TTRV can effectively reduce the number of redundant samples, it still imposes excessive redundant samples when only a small fraction of the execution paths need to be sampled using the computed *paLSP*. For instance, if Fibonacci (Figure 2(c)) takes the hypothetical execution path $\gamma_2 = \langle v_A, v_D, v_{B_1}, v_{B_2}, v_A, v_D, v_{C_1}, v_{C_2}, v_{C_3}, v_{C_4}, v_D, v_{B_1}, v_{B_2} \rangle$, then $paLSP = 1$. However, if we apply $paLSP = 5$ up until vertex v_{C_2} (which can sample all critical instructions) and adjust is to $paLSP = 1$ afterwards, then the number of samples drops by 62%.

4.1 *LSP* Regions

Intuitively, our idea to reduce redundant samples in an execution path is to dynamically change *paLSP* according to the *LSP regions* of the execution path.

Definition 7. *Let γ be a unique execution path. An LSP* region *is a set of subpaths of γ with the same paLSP, where each subpath is* maximal*. That is, for each LSP region, if a subpath is extended, it no longer belongs to that region.* □

Since each subpath has a unique *paLSP*, each *LSP* region is an equivalence class.

To sample an execution path with an *adaptive paLSP*, our approach needs to somehow *regionalize* the path based on Definition 7. In this case, when the program starts executing a subpath of an *LSP* region, the monitor *adapts* to *paLSP* of that *LSP* region at run time. Clearly, the regionalization can partition an execution path in various ways. Our general objective is to regionalize an execution path such that adapting *paLSP* at run time does not add excessive overhead. We break down our objective as follows:

1. Reducing the number of *LSP* regions; i.e., since change of *LSP* region and, hence, sampling period at run time incurs some overhead.
2. Reducing the number of samples taken on the execution path.
3. Maintaining the absolute jitter of *paLSP* (i.e., the difference between the minimum and maximum *paLSP* of *LSP* regions) below a predefined threshold Δ_{LSP} provided by the designer. Note that this objective ensures *predictable monitor invocation.*

We refer to a TTRV framework that uses this method as *adaptive pa-TTRV.*

4.2 A Regionalization Algorithm

The algorithm Regionalize addresses the above objectives (see Algorithm 1). It takes as input (1) the bound Δ_{LSP} on the absolute jitter, (2) the overhead of changing the sampling period O_{LSP}, and (3) a unique execution path γ. Its output is a regionalization. The intuitive idea is that the algorithm creates all possible regionalizations and chooses the one with the least monitoring overhead.

The algorithm creates all possible regionalizations using three nested loops: (1) the for-loop (Lines 4-38), (2) the while-loop (Lines 8-37), and (3) the for-loop (Lines 15-35). Each iteration of each loop creates a new regionalization, where each regionalization is different from the other (created in the same loop) by one vertex. Notice that the first for-loop adds/removes vertices from subpath $\langle(v_0, w_0, v_1), \ldots, (v_i, w_i, v_{i+1})\rangle$, the while-loop adds/removes vertices from subpath $\langle(v_{i+1}, w_{i+1}, v_{i+2}), \ldots, (v_{base-1}, w_{base-1}, v_{base})\rangle$, and the second for-loop adds/removes vertices from subpath $\langle(v_{base}, w_{base}, v_{base+1}), \ldots, (v_{n-1}, w_{n-1}, v_n)\rangle$.

When a regionalization $temp_{reg}$ is created (Line 23), the algorithm computes the monitoring overhead of $temp_{reg}$. To this end, it computes *paLSP* (line 26), and the BCET of each *LSP* region of the regionalization (line 27). Respectively, it computes the monitoring overhead by considering the number of samples taken

Algorithm 1. Regionalize

Input: Δ_{LSP}:bound on absolute jitter, O_{LSP}: overhead of changing LSP regions, γ: a unique execution path
Output: A regionalization
1: $regionalization \leftarrow \emptyset$
2: $Overhead_{reg} \leftarrow \infty$
3: $n \leftarrow Length(\gamma)$
// *iterate over the vertices of* γ
4: **for** $i = 0$ to $n-1$ **do**
5: $temp_{reg} \leftarrow \emptyset$ /* *new regionalization* */
6: $reg \leftarrow \langle (v_0, \gamma_0, v_1), \ldots, (v_i, \gamma_i, v_{i+1}) \rangle$
// *regionalization of remainder of* γ
7: $base \leftarrow i+1$
8: **while** $(base \le n-1)$ **do**
9: $temp_{reg} \leftarrow temp_{reg} \cup reg$
// *put vertex in* γ *up to base in separate regions*
10: **for** $m = i+1$ to $base-1$ **do**
11: $reg' \leftarrow \langle (v_m, \gamma_m, v_{m+1}) \rangle$
12: $temp_{reg} \leftarrow temp_{reg} \cup reg'$
13: **end for**
14: $reg'' \leftarrow \langle (v_{base}, \gamma_{base}, v_{base+1}) \rangle$
15: **for** $m = base$ to $n-1$ **do**
16: **for** $j = base+1$ to m **do**
17: $reg'' \leftarrow append(reg'', (v_j, \gamma_j, v_{j+1}))$ /* iterates only when $m > base$ */
18: **end for**
19: $temp_{reg} \leftarrow temp_{reg} \cup reg''$
// *put the remainder of* γ *in separate regions*
20: **for** $q = m+1$ to $n-1$ **do**
21: $reg''' \leftarrow \langle (v_q, \gamma_q, v_{q+1}) \rangle$
22: $temp_{reg} \leftarrow temp_{reg} \cup reg'''$
23: **end for**
// *calculate overhead for new regionalization*
24: $Overhead \leftarrow 0$
25: **for all** $reg \in temp_{reg}$ **do**
26: Compute $paLSP_{reg}$ based on Definition 5
27: $total \leftarrow$ The sum of weights of arcs in reg
28: $Overhead \leftarrow Overhead + \frac{paLSP_{reg}}{total} + O_{LSP}$
29: **end for**
30: $\Delta_{reg} \leftarrow$ Absolute jitter of $paLSP$s of $temp_{reg}$
// *Update the best regionalization*
31: **if** $Overhead < Overhead_{reg}$ **and** $\Delta_{reg} \le \Delta_{LSP}$ **then**
32: $regionalization \leftarrow temp_{reg}$
33: $Overhead_{reg} \leftarrow Overhead$
34: **end if**
35: **end for**
36: $base \leftarrow base + 1$
37: **end while**
38: **end for**
39: **return** $regionalization$

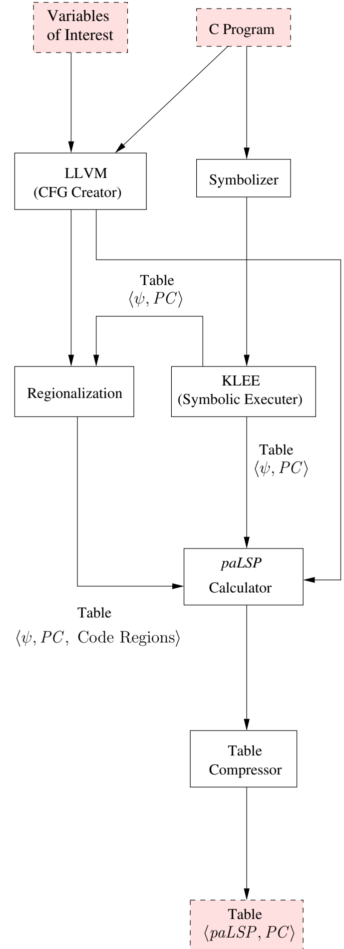


Fig. 3. The tool chain

in each LSP region and the cost of changing LSP regions (line 28). If $temp_{reg}$ has a lower monitoring overhead compared to the previously chosen regionalization, and the absolute jitter of $paLSP$s is bounded by Δ_{LSP}, the algorithm chooses $temp_{reg}$ as the solution (lines 31- 34).

4.3 General Code Regionalization

Observe that Definition 7 and Algorithm 1 identify a regionalization for an execution path. Hence, if two execution paths in a program share a common subpath, the subpath does not necessarily reside in the same *LSP* region. For instance, consider the following execution paths: $\gamma_1 = \langle(v_0, 5, v_1), (v_1, 10, v_2), (v_2, 15, v_3)\rangle$ and $\gamma_2 = \langle(v_0, 5, v_1), (v_1, 1, v_5), (v_5, 2, v_6)\rangle$, where $\Delta_{LSP} = O_{LSP} = 5$. Algorithm 1 computes the following two *LSP* regions for γ_1: (1) $reg_1 = \{\langle(v_0, 5, v_1)\rangle\}$, where $paLSP_{reg_1} = 5$, and (2) $reg_2 = \{\langle(v_1, 10, v_2), (v_2, 15, v_3)\rangle\}$, where $paLSP_{reg_2} = 10$. On the contrary, for γ_2, the algorithm computes a single *LSP* region $reg_{\gamma_2} = \{\gamma_2\}$, where $paLSP_{reg_{\gamma_2}} = 1$. Hence, subpath $\langle(v_0, 5, v_1)\rangle$ resides in different regions with different *paLSP*s for execution paths γ_1 and γ_2. Thus, in environments where a unique regionalization among all execution paths of the program is desirable, we generalize the regionalization process as follows.

Definition 8. *Let $CFG = \langle V, v^0, A, w\rangle$ be a control-flow graph. In* general regionalization*, each arc $(u, v) \in A$ appears in one and only one LSP region.* □

In this case, the monitor *adapts* the *paLSP* of an *LSP* region *reg* at run time when (1) the program initiates the execution of a subpath in *reg*, and (2) *reg* differs from the *LSP* region of the previously executed subpath. Obtaining a general regionalization that optimally satisfies the three objectives mentioned in Subsection 4.1 has exponential complexity. In Section 5, we present an efficient approach to implement general regionalization.

5 Implementation

We have implemented a tool chain for computing *paLSP* and adaptive *paLSP* (see Figure 3). The tool's input is a C program and a set of variables of interest. First, it extracts the CFG and subsequently the critical CFG of the program. In order to implement the path prediction function ψ_P (Definition 4), the tool chain first symbolizes the input variables of the program. Then, it feeds the symbolized program to the symbolic execution tool KLEE [7]. As a result, KLEE creates a mapping table from each unique execution path of the program to its path constraint. We modified KLEE using a patch, such that it converts an execution path to its unique version (Definition 6). In case there exist duplicate unique paths, our KLEE patch only keeps the path with the weakest path constraint.

To compute the adaptive *paLSP*, the tool chain regionalizes the program using general regionalization (see Definition 8). To this end, the tool chain considers all the arcs in between two consecutive conditional statements in the CFG of the program as one *LSP* region. Consequently, the tool chain maps each execution path to the set of its *LSP* regions. The *paLSP* calculator uses the critical CFG to compute *paLSP* and adaptive *paLSP* of each execution path in the mapping table. For adaptive *paLSP*, the *paLSP* calculator computes *paLSP* of each region in each execution path.

In general, the size of the (execution path to path constraint) mapping table may grow exponentially with respect to the number of execution paths. This

implies that looking up the mapping table at run time imposes a large overhead. Thus, it is desirable to construct a smaller version of the table to be used at run time. To this end, the tool chain applies two techniques to eliminate entries:

1. *Implication Reduction*: This technique groups the execution paths whose *paLSP* is defined by the same arc (u, v) in the critical CFG. For each group, it extracts the path constraint whose satisfaction leads to the execution of (u, v). Then, it represents the execution paths in the group with a table entry, that maps the extracted path constraint to *paLSP* of the execution paths in the group. This table entry also incorporates the *union* of the set of *LSP* regions of the execution paths in the group.
2. *paLSP Reduction*: This technique removes all entries from the mapping table, where *paLSP* of the execution path is similar to the fixed *LSP*.

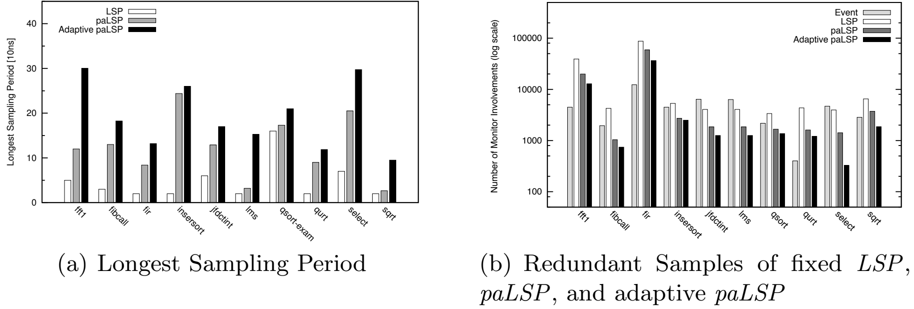
The final mapping table maps a path constraint to a *paLSP* and a set of *LSP* regions along with their *paLSP*. These techniques, on average, reduce the mapping table of SNU programs by 78% without loss of precision. In other words, *paLSP* and adaptive *paLSP* of an execution path do not change. When the input values do not satisfy a path constraint, the satisfiable path constraint is removed by *paLSP* reduction, hence, the monitor sets its sampling period to the fixed *LSP*.

In cases where Klee can not process all execution paths because of its limitations, the tool chain takes two conservative approaches: (1) when there is unanalyzed code, it assumes that this code is executed at all times and, hence, appends it to all execution paths, and (2) when the analysis of an execution path γ is incomplete, the tool finds all possible unique subpaths that can be executed after γ and, hence, creates a new path by appending these subpaths to γ.

6 Experimental Results

We use a selected set of programs from the SNU benchmark [1] to evaluate our approaches. Programs not discussed in this section exhibit similar behavior. The experimental setting is as follows. In each program, the `main` function runs 100 times, where at each iteration the `main` function receives new input values from the environment. The input values are such that each unique execution path of the program executes at least once. The program and the time-triggered monitor run on an MCB1700 board with RTX real-time operating system. The time-triggered monitor runs in four modes: (1) fixed *LSP*, (2) path-aware *LSP*, (3) adaptive *paLSP*, where Δ_{LSP} and O_{LSP} are 50ns, and (4) program augmented with history (cf. Section 2) with the sampling periods of $50 \times$ fixed *LSP*, $50 \times$ *paLSP*, and $50 \times$ adaptive *paLSP*. We measure the following metrics to evaluate our approaches:

1. The values of the fixed *LSP*, *paLSP*, and adaptive *paLSP*.
2. The number of redundant samples taken at run time by the monitor.
3. The execution time of the monitored program. This value projects the amount of monitoring overhead.



(a) Longest Sampling Period

(b) Redundant Samples of fixed *LSP*, *paLSP*, and adaptive *paLSP*

Fig. 4. Sampling period and redundant samples

6.1 Sampling Period of pa-TTRV and Adaptive pa-TTRV

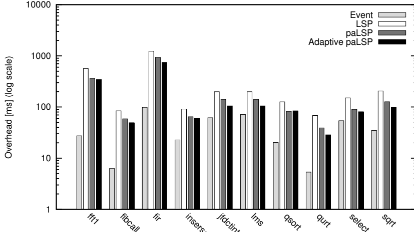
Figure 4(a) shows the fixed *LSP*, *paLSP*, and adaptive *paLSP* of each program. The *paLSP* of each program is the average *paLSP* over all unique execution paths of the program. As for the adaptive *paLSP* for each unique execution path, we consider the average *paLSP* over all the *LSP* regions of the execution path. Respectively, the adaptive *paLSP* of each program is the average adaptive *paLSP* over all unique execution paths of the program. The results show that the *paLSP* and adaptive *paLSP* of all programs are on average 2.4 and 3.34 times greater than their fixed *LSP*.

Observe that in some programs, *paLSP* is considerably greater than the fixed *LSP* (e.g., in insertsort this is 12.2 times). Our studies show that such programs have at least one of the following characteristics:

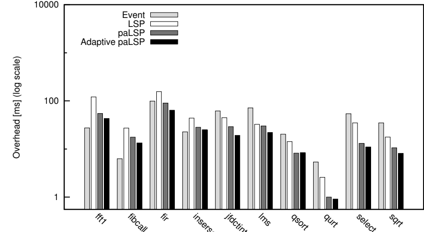
- The majority of the execution paths do not incorporate critical instructions and, hence, do not require monitoring. For instance, 66.66% of the execution paths of insertsort do not require monitoring.
- In the majority of the execution paths, the critical instructions are sparsely distributed and, hence, the required sampling period is greater than the fixed *LSP*. For instance, for 50% of the execution paths of select, *paLSP* is 130ns while the fixed *LSP* of select is 70ns.

On the contrary, in programs such as sqrt and lms, *paLSP* is moderately larger than the fixed *LSP*. Our studies show that such programs have at least one of the following characteristics:

- The majority of the execution paths execute the two consecutive critical instructions that define the fixed *LSP* of the program and, hence, their *paLSP* is equal to the fixed *LSP*. For instance, for 75% of sqrt's execution paths, *paLSP* is 20ns which is the same as sqrt's fixed *LSP*.
- In the majority of the execution paths, the critical instructions are densely distributed. For instance, 54% of the execution paths of lms have *paLSP* of 40ns while lms's fixed *LSP* is 20ns.



(a) Overhead of fixed *LSP*, *paLSP*, and Adaptive *paLSP*



(b) Overhead of $50 \times LSP$, $50 \times paLSP$, and $50 \times$ adaptive*paLSP*

Fig. 5. Monitoring overhead

In addition, in programs such as select, adaptive *paLSP* is considerably greater than *paLSP*. In the execution path of such programs, the critical instructions are densely concentrated in a small fraction of the execution path, while in the remaining of the path the critical instructions are sparse. For instance, in select, the critical instructions *only* reside in the function SWAP, where *paLSP* of an execution path executing SWAP is as low as 70ns. Hence, the adaptive *paLSP* of such execution paths is 48.75% larger than their *paLSP*.

6.2 Redundant Samples of pa-TTRV and Adaptive pa-TTRV

Figure 4(b) shows the number of redundant samples of a time-triggered monitor when using fixed *LSP*, *paLSP*, and adaptive *paLSP*. Note that the y-axis is in log scale. The *Event* bar shows the number of critical instructions executed throughout the program run. Bars *LSP*, *paLSP*, and adaptive *paLSP* show the number of redundant samples in these monitoring modes. The number of redundant samples is the difference between the total number of taken samples and the number of executed critical instructions. On average, by using *paLSP*, redundant samples decrease by 44.87%, and by using adaptive *paLSP*, redundant samples decrease by 64.04%. Our analysis shows that in programs such as qurt and select, if the execution of paths with *paLSP* greater than the fixed *LSP* dominate the execution scenarios, then using *paLSP* results in larger reductions in the number of redundant samples. On the contrary, for programs such as sqrt and fir, we see small reduction in the number of redundant samples, since the majority of the executed paths at run time have *paLSP* equal to the fixed *LSP*. Note that a large percentage of paths with *paLSP* equal to the fixed *LSP* does not imply that the program's execution scenario is dominated by these paths.

6.3 Monitoring Overhead of pa-TTRV and Adaptive pa-TTRV

Figure 5(a) shows the monitoring overhead of a time-triggered monitor when using fixed *LSP*, *paLSP*, and adaptive *paLSP*. Each *Event* bar shows the execution time of the monitored program when using an event-triggered monitor.

Bars *LSP*, *paLSP*, and adaptive *paLSP* show the execution time of the monitored program in these monitoring modes. On average, monitoring overhead decreases by 39.34% when using *paLSP*, and by 51.28% when using adaptive *paLSP*. In programs such as qurt and select, when using *paLSP*, the monitoring overhead does not decrease in the same proportion as the redundant samples. For instance, in select, the number of redundant samples deceases by 72.04%, while the monitoring overhead decreases 51.83%. This is because, the overhead caused by the monitor to find the satisfied path constraint (using the lookup table discussed in Section 5) and adjust its sampling period, is large. Hence, we see less reduction in the monitoring overhead.

The same side effect is seen when using adaptive *paLSP* in programs such as select and fibcall. The overall overhead of looking up the mapping table in the adaptive path-aware monitor is larger compared to the path-aware monitor, since the monitor looks up the table more frequently (i.e., at each entry to a new *LSP* region). In some cases, the overall look up overhead is such that the monitoring overhead of the adaptive path-aware monitor exceeds the monitoring overhead of the path-aware monitor, although the adaptive path-aware monitor reduces more redundant samples. For instance, in qsort, the monitoring overhead of the adaptive path-aware monitor is 84.388ms, and the monitoring overhead of the path-aware monitor is 82.477ms, while the adaptive path-aware monitor removes 18.22% more redundant samples.

Figure 5(a) shows that event-based monitoring imposes less overhead than pa-TTRV and adaptive pa-TTRV when the history mechanism is not employed. This is because a time-triggered monitor still introduces redundant samples with *paLSP* and adaptive *paLSP*. Since event-based monitoring is impractical for real-time systems, we need to further reduce the redundant samples to achieve a cost-worthy time-triggered monitor. To this end, we augment each program with history (see Section 2) to increase the fixed *LSP*, *paLSP*, and adaptive *paLSP* by a factor of 50. Experimental results show that for the SNU programs, the sampling periods of $50 \times$ fixed *LSP*, $50 \times$ *paLSP*, and $50 \times$ adaptive *paLSP* cause zero redundant samples. Furthermore, Figure 5(b) shows that in 66% of the programs, the overhead of the path-aware monitor is less than the overhead of the event-triggered monitor, and in 75% of the programs, the overhead of the adaptive path-aware monitor is less than the overhead of the event-triggered monitor. Note that a sampling period of at least the *maximum* time interval between two consecutive critical instructions, eliminates all redundant samples. In addition, by using history, the maximum increase in the memory usage of the programs is 646 bytes, which is an inconsiderable amount with respect to available resources.

7 Related Work

Regardless of the type of monitor, runtime verification frameworks must impose low monitoring overhead. [3] reduces the overhead by rewriting safety properties

such that the evaluation of properties requires the least information regarding the program execution. [5] reduces the number of instrumentations, by determining locations in the program that do not affect property evaluation. [4] distributes the instrumentation cost among multiple users. [14] controls the overhead by temporarily disabling monitoring of selected data, by using supervisory control theory of discrete event systems and PID-control theory of discrete-time systems. [10] extracts only a subset of the data required to evaluate program properties, by removing/adding instrumentation relevant to the program state at run time.

[2] is the closest work to our path-aware and adaptive path-aware approaches. This approach discards instrumentation with respect to the execution path of the program. Our methods surpass [2] in the following aspects. To our knowledge, the approach in [2] manually extracts the path constraints for each path and, hence, it can only handle very small programs. On the contrary, our techniques are fully automated and can handle medium size programs. Moreover, unlike our rigorous analysis of multiple case studies, [2] only presents the manual analyses of two small case studies and lacks strong evidence on the effectiveness of their method. In addition, [2] is only applicable when all input values are known a priori. On the other hand, our approaches can handle inputs provided dynamically throughout the program run. Finally, [2] does not intelligently discard instrumentation at run time. On the contrary, the adaptive path-aware technique dynamically adjusts the sampling period at run time to further reduce monitoring overhead.

8 Conclusion

In this paper, we presented an effective method for reducing the overhead of time-triggered runtime verification (TTRV). The main drawback of TTRV is its excessive runtime overhead due to redundant sampling, where the monitor may take samples from the program even if no new event for monitoring has occurred. Our proposed method in this paper leverages symbolic execution [16] in order to predict the program's execution path and intelligently choose the sampling period. In particular, we proposed *path-aware* TTRV, where the monitor adjusts its sampling period based on the given input values to the program under inspection. We also introduced *adaptive* path-aware TTRV, where the monitor adjusts its sampling period at run time based on the density of occurrence of events that need to be monitored in an *LSP* region. Our techniques are implemented in a tool chain and the result of experiments show that adaptive path-aware TTRV reduces the runtime overhead and the number of redundant samples by up to 69% and 86%, respectively.

One open problem is merging rigorous execution time analysis with our method to accurately measure and incorporate the cost of switching the sampling period in the algorithm presented in Section 4.

References

1. SNU Real-Time Benchmarks, http://www.cprover.org/goto-cc/examples/snu.html
2. Artho, C., Drusinksy, D., Goldberg, A., Lowry, K.H.M., Pasareanu, C., Roşu, G., Visser, W.: Experiments with Test Case Generation and Runtime Analysis. In: Börger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 87–108. Springer, Heidelberg (2003)
3. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-Based Runtime Verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)
4. Bodden, E., Hendren, L., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative Runtime Verification with Tracematches. In: Sokolsky, O., Taşıran, S. (eds.) RV 2007. LNCS, vol. 4839, pp. 22–37. Springer, Heidelberg (2007)
5. Bodden, E., Hendren, L., Lhoták, O.: A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 525–549. Springer, Heidelberg (2007)
6. Bonakdarpour, B., Navabpour, S., Fischmeister, S.: Sampling-Based Runtime Verification. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 88–102. Springer, Heidelberg (2011)
7. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI 2008, pp. 209–224 (2008)
8. Chen, F., Roşu, G.: Java-MOP: A Monitoring Oriented Programming Environment for Java. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 546–550. Springer, Heidelberg (2005)
9. Colin, S., Mariani, L.: Run-Time Verification. In: Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems. LNCS, vol. 3472, pp. 525–555. Springer, Heidelberg (2005)
10. Dwyer, M.B., Kinneer, A., Elbaum, S.: Adaptive online program analysis. In: Proceedings of the 29th International Conference on Software Engineering, ICSE 2007, pp. 220–229 (2007)
11. Gallaher, M., Kropp, B.: The economic impacts of inadequate infrastructure for software testing. National Institute of Standards & Technology Planning Report 02–03 (2002)
12. Havelund, K., Goldberg, A.: Verify Your Runs. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 374–383. Springer, Heidelberg (2008)
13. Havelund, K., Roşu, G.: An overview of the runtime verification tool java pathexplorer. Form. Methods Syst. Des. 24(2), 189–215 (2004)
14. Huang, X., Seyster, J., Callanan, S., Dixit, K., Grosu, R., Smolka, S.A., Stoller, S.D., Zadok, E.: Software monitoring with controllable overhead. Software Tools for Technology Transfer (STTT) 14(3), 327–347 (2012)
15. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Form. Methods Syst. Des. 24(2), 129–155 (2004)
16. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (1976)

Fast-Forward Runtime Monitoring — An Industrial Case Study

Christian Colombo and Gordon J. Pace

Department of Computer Science, University of Malta
{christian.colombo,gordon.pace}@um.edu.mt

Abstract. Amongst the challenges of statefully monitoring large-scale industrial systems is the ability to efficiently advance the monitors to the current state of the system. This problem presents itself under various guises such as when a monitoring system is being deployed for the first time, when monitors are changed and redeployed, and when asynchronous monitors fall too much behind the system.

We propose fast-forward monitoring — a means of reaching the monitoring state at a particular point in time in an efficient manner, without actually traversing all the transitions leading to that state, and which we applied to a financial transaction system with millions of transactions already affected. In this paper we discuss our experience and present a generic theory of monitor fast-forwarding instantiating it for efficient monitor deployment in real-life systems.

1 Introduction

Our experience with the financial transactions industry has shown us that runtime verification is generally perceived as an intrusive addition which modifies the system code even if not its functionality — slowing down the system at best and introducing extra bugs at worst. The reason is mainly that, with possibly few exceptions, the existing industrial systems have not been designed with runtime verification in mind. To address the industry's concern, we have advocated asynchronous runtime verification which is completely non-intrusive given that all system events were already being logged in a database. Our architecture, embodied in the tool asyncLarva[1] and depicted in Fig. 1, consists of a system recording events (represented by a circle as in a tape recorder) in a database and subsequently, the monitor plays back (represented by a triangle) the events to check correctness. Note that, as in standard asynchronous monitoring, the system head can move forward without waiting for the monitor head to keep up. Any detected problems are communicated to an administrator since by the time the problem is detected it would usually be too late to take corrective action.

Asynchronous runtime monitoring has been successfully applied to an industrial case study [1] and proved effective in discovering issues which would have

[1] http://www.cs.um.edu.mt/svrg/Tools/asynclarva

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 214–228, 2013.
© Springer-Verlag Berlin Heidelberg 2013

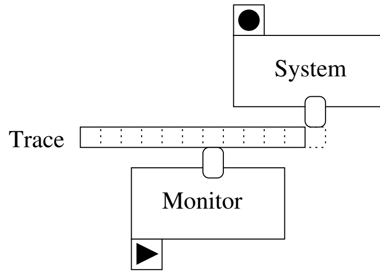


Fig. 1. Asynchronous monitoring

otherwise gone unnoticed. While this approach gives administrators a monitoring tool, it does not provide any possibility of triggering corrective action automatically to mitigate discovered problems.

As a means of enabling asynchronous monitors to take corrective action, we introduced a compensation mechanism [2] — although violations may be discovered late, the system is stopped and its state may be reversed to the point of violation. This could be relatively easily done in a financial transaction context where compensations are inherently part of the system. Also, since frequently financial systems have short bursts of peak load but a lower load on average, the monitor does not fall behind the system indefinitely. Using a box to represent stopping the system and a backward triangle to represent compensations, Fig. 2 depicts the modified architecture.

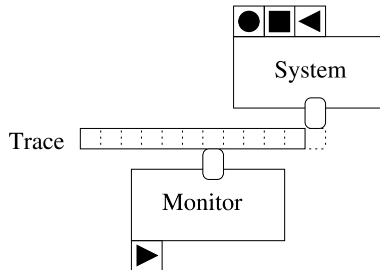


Fig. 2. Asynchronous monitoring with the possibility of compensation

As more confidence was gained in the runtime monitoring system, the next phase was to introduce the possibility of synchrony in which some monitors can run synchronously while the rest of the less crucial monitors are asynchronous. This setup [3] was particularly useful for monitoring untrusted system users who should not be allowed to proceed unless each step is approved by their monitor, *i.e.,* synchronous monitoring. In this way trusted users can still be monitored asynchronously without experiencing any service deterioration due to

monitoring. To implement synchronous and asynchronous monitoring, the system should now be capable of pausing and unpausing to wait for the monitor verdict — shown in Fig. 3 as the rightmost button with a diagonal signifying the two actions. Furthermore, it was noted that sometimes fine-grained compensations have to be discarded and replaced by coarser-grained ones. For example, if a fraud has been detected when it is too late to reverse the related purchases, it might be enough to block the offending user's account. This is shown in the diagram as a double backward triangle.

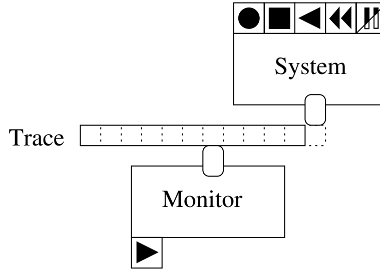


Fig. 3. Synchronous and asynchronous monitoring with the possibility of coarse-grained compensations

In the same way as coarse-grained compensations may be useful because of out-dated compensations, there are cases when 'coarse-grained' monitoring is required. For example, given a system which has already been running for a number of years, this approach can be particularly useful to bring the monitor up to scratch, taking into account the past transactions of the users. If a violation occurred over a year ago, we might not be interested in discovering it, but we still require the monitoring state to be as though the monitors have been running along with the system throughout the years it has been in operation. Another application for coarse-grained monitoring is to restart monitoring from a particular point in time if for some reason monitoring data is lost, or the monitor has been significantly modified. Coarse-grained monitoring can be thought of as a way of fast-forwarding monitoring through a trace by allowing it to skip parts of it. The full architecture is depicted in Fig. 4 showing fast-forward monitoring as two forward triangles at the monitor side.

While the full architecture without fast-forwarding has been presented in [3], in this paper we focus solely on fast-forward monitoring which proved to be necessary to monitor an industrial case study where properties are changed and augmented regularly. This dynamicity requires the monitors to fast-forward through years of data so that the property monitors can update their state with that of the system as fast as possible. To this extent, we propose a methodology whereby apart from the normal (slow) monitor processing the system trace of events, a (fast) abstract monitor is available which can process an abstraction of the system trace. By having translation functions which enable going back and forth

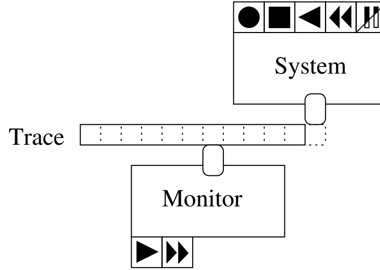


Fig. 4. The proposed architecture with coarse-grained monitoring

between the slow and the fast version, monitoring can be fast-forwarded to ignore irrelevant intermediate monitoring states. To the best of our knowledge, monitor fast-forwarding has not been treated in the literature and thus our contributions are as follows:

– We formalise the notion of monitor fast-forwarding in general and explain what it means for fast-forwarding to be correct. (Section 3.1)
– Next, we show how the theory is applicable to monitoring with a particular monitoring tool, LARVA. (Section 3.2)
– Through an industrial case study we show the usefulness of our approach, particularly in the context of an industrial case study (Section 4).

2 Background

To instantiate fast-forward monitoring, we have used the monitoring tool, LARVA [4], which consists of dynamic automata with timers and events (DATEs). These automata trigger on particular system events after checking that a corresponding condition holds. Conditions can be specified both on the system state and on the monitor state, the latter possibly including stopwatches and Java objects. Following a transition trigger, an action can be executed to modify stopwatches, the system or monitor state, or synchronise with other automata. Furthermore, DATEs enable monitoring on a *foreach* object basis such that a monitor is replicated for each object of a particular type. Thus, a LARVA monitor is in fact a vector of automata accompanied by a function which may dynamically launch new automata and add them to the vector.

Example 1. As an example of a LARVA monitor, below is a property which manages inactive users of a financial system. To ensure inactive accounts are safe, users who are inactive for more than six month are suspended, *i.e.*, put in *dormancy mode*, and an administration fee is charged. If a user asks for his account to be reactivated, then the request is granted but the account is switched to dormant once more if the user still remains inactive for another three months. The corresponding LARVA property checks a number of sub-properties:

1. The account is switched to dormant after six/three months of inactivity (otherwise *expiredDorm* bad state is reached).
2. The account is not put to dormant before six/three months of inactivity (otherwise *unexpectedDorm* bad state is reached).
3. The applicable fee is paid correctly (otherwise *failedPay* bad state is reached).
4. No transaction is carried out if the account is dormant (otherwise *unexpectedTx* bad state is reached).

This property is monitored for each system user and thus a replica of each monitor is instantiated for each user. Excerpts of the LARVA script which specifies the dormancy property is given in Fig. 5 (top) while the depiction of the DATE automaton is given in Fig. 5 (bottom). Note that each transition is an *event**condition**action* triple and octagons represent bad states. For brevity we use *dorm* for *dormancy*, *Tx* for *transaction*, and *T* for *Timer*.

System events in the context of DATEs can be system methods calls, timer events (*e.g.*, a monitor stopwatch triggers an event after 30 minutes since it was reset), synchronisation events, or a disjunction of events. Since basic events contribute to disjunctions, then at any time instant several events may fire simultaneously.

Definition 1. *A system trace $s \in \mathcal{S}$ is a sequence of time instants such that each instant, s_i, is composed of a set of events, $E \in 2^{event}$ and a timestamp $t \in \mathbb{R}_0^+$: $s_i = (E_i, t_i)$. The alphabet of possible instants will be written as Σ: $\Sigma \stackrel{df}{=} 2^{event} \times \mathbb{R}_0^+$.*

A DATE automaton has a set of states Q connected with transitions triggering on system or timer events and guarded by conditions on the system and monitor symbolic state (ranging over Θ) and stopwatches (ranging over $\mathcal{CT}$). A monitor consists of a set of initial automata and directives to instantiate new automata dynamically upon receiving certain events. Full details of the the formal semantics of DATEs can be found in [4]. For the needs of this paper, it suffices to identify the configuration of a vector of DATE automata and explain how they form a run depending on the system events observed.

Definition 2. *A configuration $c \in \mathcal{C}$ of a vector of DATE automata, $\overline{M} \in \overline{\mathcal{M}}$, consists of the current system and monitor state, $\theta \in \Theta$, the current state of the stopwatches, $ct \in \mathcal{CT}$, a vector of locations representing the location each automaton is in, $\overline{q} \in \overline{Q}$, and the vector of automata itself: $c = (\theta, ct, \overline{q}, \overline{M})$.*

We can now outline the semantics of a vector of DATEs.

Definition 3. *The semantics of a vector of DATEs, $\overline{M} \in \overline{\mathcal{M}}$, can be given by transforming $\overline{M}$ into a labelled transition system over the configurations $\langle \mathcal{C}, c_0, \rightarrow_c \rangle$ — with states $\mathcal{C}$ (the configurations of $\overline{M}$), initial state $c_0 \in \mathcal{C}$, and transition function labelled by events and timestamps, $\rightarrow_c \in (\mathcal{C} \times \Sigma) \rightarrow \mathcal{C}$. We write $c \xrightarrow{a}_c c'$ to refer to particular $(c, a, c') \in \rightarrow_c$ and $c \stackrel{w}{\Rightarrow}_c c'$ (with $w \in \Sigma^*$) for the transitive closure of $\rightarrow_c$.*

```
Foreach (String user) {
   Variables {
      Clock T6, T3; }
   Events {
      expired6 = {T6 @ 183 days}
      anyTx = {event(currency, amount, type) where type = "generic"}
      ...   }
   Property dormancy {
      States {
         Bad { expiredDorm, unexpectedDorm, failedPay, unexpectedTx }
         Normal { dorm, thawed, feepaid }
         Starting { nonDorm } }
      Transitions {
         nondorm -> nondorm [anyTx\\resetT6]
         nondorm -> dorm [dorm\T6>180 days]
         ... } } }
```

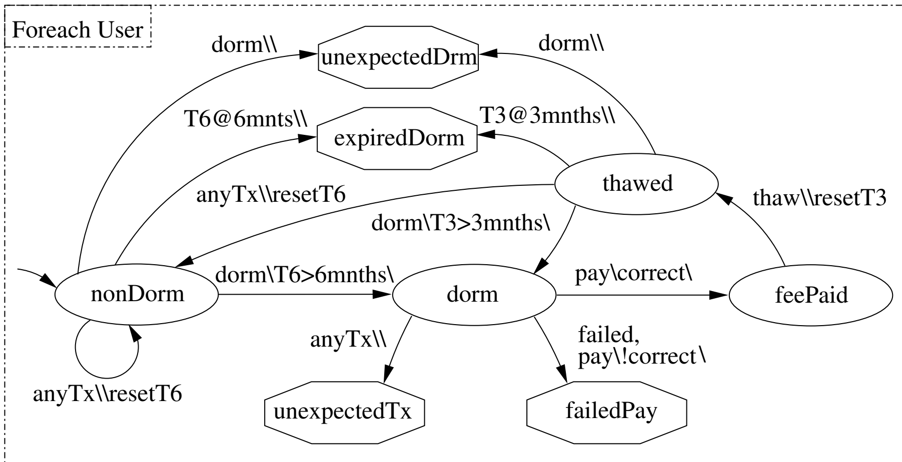


Fig. 5. The dormancy property expressed as a LARVA script (top) and as a DATE (bottom)

The set of bad configurations, $\mathcal{C}_B \subseteq \mathcal{C}$, correspond to the configurations arising from the states which are tagged as undesirable in the original DATEs. We will assume that the transition system guarantees that recovery from a bad state is not possible — if $c \in \mathcal{C}_B$ and $c \xrightarrow{a}_c c'$, then $c' \in \mathcal{C}_B$.

The set of bad traces starting from a configuration c, written $\mathcal{B}(c)$, are the strings leading to a bad configuration: $\mathcal{B}(c) = \{w \mid \exists c' \in \mathcal{C}_B \cdot c \stackrel{w}{\Rightarrow} c'\}$. A configuration c_2 is said to be as strict or stricter than c_1 (written $c_1 \sqsubseteq_c c_2$) if c_2 rejects all traces rejected by c_1 (and possibly more): $\mathcal{B}(c_1) \subseteq \mathcal{B}(c_2)$. We say that they are equivalent if they reject the same traces: $c_1 =_c c_2 \stackrel{df}{=} c_1 \sqsubseteq_c c_2 \wedge c_2 \sqsubseteq_c c_1$.

3 Monitor Fast-Forwarding

Whenever new or modified stateful properties are to be monitored on a system, monitoring has to start processing the system trace from its beginning (which might be years worth of logs) since ignoring any part of the trace might yield wrong monitoring results — the monitor state would not have correctly evolved over the whole trace. Similar problems occur when monitor state is lost due to a system crash or when asynchronous monitors fall too much behind the system. Fast-forward monitoring provides a means of going through the trace (or an abstraction of it) to infer the state (or a somewhat similar state) the monitor would have reached if the trace was monitored normally. By ignoring the intermediate monitoring states, fast-forward monitoring promises to be significantly faster than normal monitoring.

In this section we define a generic theory of monitor fast-forwarding and then we instantiate the approach on the monitoring tool LARVA and explain how we support users in specifying an application of monitor fast-forwarding.

3.1 A Theory of Monitor Fast-Forwarding

The idea behind monitor fast-forwarding is to allow the monitor to skip parts of the trace but still reach the same configuration that would have been reached had the monitor progressed normally. At its simplest level, this would just involve a function to abstract traces into shorter ones. However, in practice, this is usually not sufficient, and one would require abstracting also the transition system induced by the DATE against which the trace is verified. Given a monitoring system, its fast-forward version is another monitoring system, with three additional components: (i) a mapping from the original monitoring system to the fast one; (ii) a trace abstraction which transforms an actual trace into a shorter or a more efficiently processable one; and (iii) a mapping from the fast monitoring system to the original one.

Definition 4. *Given a monitoring transition system $M = \langle \mathcal{C}, c_0, \rightarrow_c \rangle$ with bad states $\mathcal{C}_B$, the transition system $\langle \mathcal{A}, a_0, \rightarrow_A \rangle$ with total domain translation functions $\blacktriangleright\!\blacktriangleright \in \mathcal{C} \rightarrow \mathcal{A}$, $\blacktriangleleft\!\blacktriangleleft \in \mathcal{A} \rightarrow \mathcal{C}$, and a trace abstraction function $\alpha \in \Sigma^* \rightarrow \Sigma^*$, is said to be a fast-forward version of M.*

Typically, for actual fast-forwarding, the length of an abstracted trace is shorter than the original trace:[2] $\mathit{length}(\alpha(w)) \leq \mathit{length}(w)$. The trace abstraction function is assumed to map the empty string to itself: $\alpha(\varepsilon) = \varepsilon$ (which follows directly if the abstraction always shortens traces).

Definition 5. *A transition system A is an* exact *fast-forward version of a monitoring transition system M, with functions $\blacktriangleright\!\blacktriangleright$, $\blacktriangleleft\!\blacktriangleleft$ and α if whenever $c \stackrel{w}{\Rightarrow}_c c'$:*

$$\forall a' \cdot \blacktriangleright\!\blacktriangleright(c) \xRightarrow{\alpha(w)}_A a' \implies c' =_c \blacktriangleleft\!\blacktriangleleft(a')$$

[2] We do not enforce this in the definition to allow for abstractions to an extended alphabet which although lengthens the trace, would still be processed faster.

It is said to be an over-approximated *fast-forward version if whenever* $c \overset{w}{\Rightarrow}_c c'$*:*

$$\forall a' \cdot \blacktriangleright\!\blacktriangleright(c) \overset{\alpha(w)}{\Longrightarrow}_A a' \implies c' \sqsubseteq_c \blacktriangleleft\!\blacktriangleleft(a')$$

It is an under-approximation *if* $c \overset{w}{\Rightarrow}_c c'$ *implies:*

$$\forall a' \cdot \blacktriangleright\!\blacktriangleright(c) \overset{\alpha(w)}{\Longrightarrow}_A a' \implies \blacktriangleleft\!\blacktriangleleft(a') \sqsubseteq_c c'$$

We will write $c \overset{w}{\Rightarrow}_{\blacktriangleright\blacktriangleright} c'$ *if there exist* a *and* a' *such that* (i) $a = \blacktriangleright\!\blacktriangleright(c)$*;* (ii) $c' = \blacktriangleleft\!\blacktriangleleft(a')$*; and* (iii) $a \overset{\alpha(w)}{\Longrightarrow}_A a'$*.*

Proposition 1. *Abstracting to an exact fast-forwarded system and transforming back does not change the observable behaviour of the resulting monitor:* $\blacktriangleleft\!\blacktriangleleft \circ \blacktriangleright\!\blacktriangleright \subseteq =_c$*.*

Proof. The proof follows directly from the definition of exact fast-forwarded systems and the fact that $\alpha(\varepsilon) = \varepsilon$*.*

Example 2. Referring back to the dormancy example, we illustrate how the above theory can be instantiated for initialising dormancy monitors for the existing users of a system. For the sake of this example we will focus on deciding whether a user is in the *nonDorm* state, or the *dorm* state and set the clocks accordingly. To facilitate this decision, we need two pieces of information: (i) a list of system users, and for each user (ii) the timestamp of the latest transaction or the latest switch to dormancy, whichever is most recent.

Thus, assuming a monitor $\langle \mathcal{C}, c_0, \rightarrow_c \rangle$ and a system trace s, the corresponding translation function $\blacktriangleright\!\blacktriangleright$ transforms c_0 into a configuration of type $\mathcal{C} \times \Psi$ (using information from the system state) where Ψ includes a list of users currently active in the system. The reverse translation function $\blacktriangleleft\!\blacktriangleleft$ simply drops Ψ from the resulting configuration. The trace abstraction function α drops all events except for the most recent activity of each user (with respect to the point of initialisation). As for the fast-forward transition system, for each user there are conceptually two options: either his or her last activity was a normal transaction, in which case the user is in the *nonDorm* state and the stopwatch should be set to trigger six months from the latest transaction, or the last activity turned the user into dormant and hence the user is in the *dorm* state.

Given a monitoring transition system M and an exact fast-forwarded version M_A, then monitoring a trace partially in M and partially in the fast-forwarded version M_A is equivalent to monitoring it completely with the original monitor M.

Theorem 1. *Given a monitoring transition system* $M = \langle \mathcal{C}, c_0, \rightarrow_c \rangle$ *with bad states* $\mathcal{C}_B$*, and an exact fast-forward transition system* $M_A = \langle \mathcal{A}, a_0, \rightarrow_A \rangle$ *with functions* $\blacktriangleright\!\blacktriangleright$*,* $\blacktriangleleft\!\blacktriangleleft$ *and* α*, given* $w = w_1 w_2 \ldots w_n$*, then* $w \in \mathcal{B}(c_0)$ *if and only if there exists* $c_n \in \mathcal{C}_B$ *and states* $c_i \in \mathcal{C}$ *such that:*

$$c_0 \overset{w_1}{\Longrightarrow}_C c_1 \overset{w_2}{\Longrightarrow}_{\blacktriangleright\blacktriangleright} c_2 \overset{w_3}{\Longrightarrow}_C c_3 \overset{w_4}{\Longrightarrow}_{\blacktriangleright\blacktriangleright} \ldots \overset{w_{n-1}}{\Longrightarrow}_{\blacktriangleright\blacktriangleright} c_{n-1} \overset{w_n}{\Longrightarrow}_C c_n$$

If the fast-forward system is an over-approximation, then the above is a forward implication. Similarly, for an under-approximation, only the backward implication is guaranteed to hold.

This result follows by induction on the number of parts string w is split into.

Interestingly, while it is generally undesirable to use over- or under-approximations, in certain scenarios one might be ready to compromise having false negatives in the case of an over-approximation and false positives in the case of an under-approximation. In the following list we suggest a number of applications of fast-forwarding monitoring, highlighting where it is preferable to use exact or approximate fast-forward monitoring:

Fast monitor bootstrapping: Whenever monitors have to be instantiated on a system with a long recorded history, running the standard monitor on the long traces may take prohibitively long. An alternative is to process the traces using an exact fast-forwarded version of the monitor. Approximate fast-forwarding can also be useful if, either we are assured that there are no errors to be caught on the stored history (in which case we can use an under-approximation) or if we prefer to process the history quickly ensuring that any errors are caught (in which case, an over-approximation would be applicable).

Burst monitoring: In systems where resources are committed only at particular points in time, it can be beneficial to accumulate and process the system trace only at these moments in time. For instance, in a transaction processing system where all database modifications are committed at the end of a transaction one may, for example, collect the full trace of a transaction and process it using a fast-forwarded monitor. If an exact fast-forward may still be too expensive to check and performance is an issue, one may choose to apply over-approximations for transactions by blacklisted users and under-approximations for whitelisted ones.

Synchronous/asynchronous monitoring: In the case of asynchronous monitoring, fast-forward monitoring can be used whenever the monitor is lagging too much behind the system. Moreover, in monitoring systems such as [3] where asynchronous monitoring can be synchronised at runtime, fast-forward monitoring can be used for a quick synchronisation.

In the rest of the paper, we will focus on the first of the above applications of fast-forwarded monitoring, showing the applicability of the approach and its gains.

3.2 Instantiating Fast-Forwarding to Larva

In Larva we use monitor fast-forwarding to start monitors from a particular point in a system's history, *i.e.*, fast monitor bootstrapping. This is crucial for industrial systems so that when properties are modified, the monitor comes up to scratch with the system as soon as possible — monitoring years' worth of data would waste monitoring time which could be used to start monitoring more recent (and thus more relevant) data.

Instantiating the theory of fast-forwarding to monitor bootstrapping in Larva would include deciding the two translation functions ►► and ◄◄, the trace abstraction function α, and the fast-forward transition relation $\rightarrow_A$. The following list expands each of these aspects, generalising the approach taken in Example 2:

- In the case of Larva it is assumed that the translation function $\blacktriangleright\!\blacktriangleright$ obtains the list of objects which should be monitored during fast-forwarding while the reverse translation function $\blacktriangleleft\!\blacktriangleleft$ drops the additional information.
- The trace which originally includes all the system events, is collapsed to the trace elements which are required for deciding the state of each monitor.
- The fast-forward monitor which handles fast bootstrapping, first obtains $\overline{M}$ by instantiating a monitor for each entity indicated by $\blacktriangleright\!\blacktriangleright$ and subsequently allows each entity to perform one step updating its monitor state based on the abstracted trace. Such a configuration step should include three aspects corresponding to the configuration components $\overline{q}$, ct, and θ respectively:
  1. The state of the respective monitor (an element of $\overline{q}$) is updated (*e.g.*, monitors of users whose account has been put into dormant state in the past should be in the state *dorm*).
  2. The clocks of each monitor are set (*e.g.*, when the last financial transaction took place so that it can be ensured that inactive users are actually put into the dormant state).
  3. The values of the variables of each monitor are set (*e.g.*, counting the number of transactions which the user has carried out so that monitors can check that the allowed quota has not been exceeded).

To facilitate the specification of the abstracted monitor for Larva users, we have introduced some minor additions to Larva scripts as explained in the following.

3.3 Adapting Larva Scripts

To enable users to easily program fast monitor bootstrapping, we have augmented the Larva script structure. Recall that Larva provides the *foreach* construct which enables monitors to be replicated for distinct objects of a particular type. Furthermore, *foreach* components can be nested (*e.g.*, for monitoring each credit card of each user) and the outermost *foreach* components are enclosed in a *global* component which can be used to monitor properties which are not replicated, *i.e.*, not related to particular objects. To initialise a *global* component what is required is to give a value to variables, clocks and update the state of any global property automata. For this reason we have added the *initializeIf* component which triggers on a particular condition (indicating that the monitor is in fast-forward mode) where the user can specify a Java method which returns a hashmap with variable/clock/automata names as keys and the corresponding intended values as the hashmap values. Note that no setting needs to be done if the variables/clocks/states have not progressed from their default initialisation. In Fig. 6 we show how using two SQL queries we deduce when the last successful transaction occurred for a particular user and whether the user has been recently (since the last successful transaction) put into dormant state. Using this information we set the corresponding clocks to trigger when the user should be put to dormant in the future. Moreover, if the user is currently dormant, then the corresponding *dormancy* automaton (shown in Fig. 5) is to be in state *dorm*.

Note that we assume that the current system state does not contain errors and consequently our fast-forwarding is an under-approximation.

Yet it is not enough to be able to initialise a monitor for one particular user — the LARVA script should also allow the script writer to specify a means of deducing the number of users in the system for whom a monitor should be replicated and initialised. Note that this is useless for a *global* component for which no replication takes place anyway. For this reason, each *foreach* may contain an *initially* component (apart from an *initializeIf* component) which can specify a method returning an array with all the objects for which a monitor should be created[3]. In our example the *initially* method returns an array of user ids.

The approach described in this section has been successfully applied to the live data of an industrial case study with promising results as elaborated in the following section.

4 Case Study

We have applied our architecture on Entropay, an online prepaid payment service offered by Ixaris Systems Ltd[4]. Entropay users deposit funds through funding instruments (such as their own personal credit card or through a bank transfer mechanism) and spend such funds through spending instruments (such as a virtual VISA card or a Plastic Mastercard). The service is used worldwide and thousands of transactions are processed on a daily basis.

In our case study 15 properties written as DATEs (each including several sub-properties as explained in the *dormancy* example) have been monitored on Entropay which can be loosely classified under:

Life cycle properties checking that operations occur at the right stage of a user's life cycle (*e.g.*, a user cannot carry out financial transactions if his account is dormant).

Real-time properties checking that actions which should be system-triggered are carried out on time (*e.g.*, the system should automatically put to dormant accounts which have been inactive for more than six months).

Rights properties checking that the user has the appropriate rights before a transaction is permitted (*e.g.*, for a user to log into the system he or she must have the *login* right).

Limits properties checking that the frequency and value of certain transactions fall within the stipulated limits (*e.g.*, no more than 100 purchases are allowed each month).

The case study was successfully executed on a sanitized[5] database of 300,000 users with around a million virtual cards and a number of issues have been detected through the monitoring system[6].

[3] LARVA supports *foreach* components for tuples of objects. Thus, the *initially* method actually returns an array of arrays where each array supplies an element of the tuple.

[4] `http://www.ixaris.com`

[5] User information was obfuscated for the purpose of this study.

[6] The issues have been extensively reported in [2,3].

```
Foreach (String user) {
  Initializeif (init) {
    static HashMap<String, Object> initializeifUser(String user) {
      HashMap<String, Object> list = new HashMap<String, Object>();

    //obtain last successful user transaction
      rs = st.executeQuery(
           SELECT timestamp FROM transaction_table
           WHERE id=@user AND timestamp < @initializationTime
           ORDER BY timestamp DESC);
      latestTrans = rs.getLong("timestamp");

    //check if user is currently dormant
      rs = st.executeQuery(
           SELECT timestamp FROM log_table WHERE id=@user
           AND event="USER_DORMANT" AND timestamp < @initializationTime
           ORDER BY timestamp DESC);
      latestDorm = rs.getLong("timestamp");

      if (latestDorm > latestTrans) {
        //i.e. user is currently dormant
        //therefore put automaton into "dorm" state
      } else {
        //i.e. user is not dormant
        //set clock to expire 6 months after last transaction occurred
      }
...
     return list; } }

//code given in the previous example starts here
  Variables {...} Events {...} Property ...
//code given in the previous example ends here

  Initially {
    static ArrayList initiallyUsers() {
      ...
      rs = s.executeQuery(SELECT id FROM users_table;);
      while (rs.next())
        list.add(rs.getString("id"));
      return list;
  } } }
```

Fig. 6. The dormancy example augmented with fast bootstrapping code

Since Entropay had been up and running for more than a year at the time of applying monitoring and it was envisaged that monitors would have to be modified or added regularly, fast monitor bootstrapping was crucial in making monitoring feasible. More details are given in the following subsection.

4.1 Results

The monitor was deployed on data representing activities starting from 23 December 2008. Data before this date was considered to be too old and would waste monitoring time which could more beneficially be used to monitor more recent data pertaining to users which are more probably still active at the time of monitoring. To quickly bootstrap the monitors up to 23 December 2008, we used the fast-forward technique on 58 weeks of data starting from 8 November 2007.

Using a Dual Core AMD Opteron Processor at 1.81GHz running Windows XP x64 with 2Gb RAM, the monitors successfully fast-forwarded through 58 weeks in 35 hours. Subsequently, the monitors were run on the available data (at the time when this case study was carried out) dating till 8 September 2009 (including 37 weeks of data) consisting of millions of transactions. This process took 552 hours (approximately 23 days) equating to less than 15 hours of processing per one week's data. Proportionately, monitoring the 58 weeks of data would have roughly taken the monitor 36 days to come at par with the live system as opposed to the day and a half with fast-forwarded initialisation. This time saving is crucial when one would need to receive immediate feedback upon deploying new monitors, particularly if remedy actions can be taken based on monitoring results.

We have not tried out this case study using other runtime verification tools. Thus, although past experiments [4] have shown our tool's performance to be comparable to that of other tools, it is difficult to discuss the performance in itself. However, these experiments do clearly highlight the effectiveness of fast-forward monitoring in significantly reducing the monitoring time.

4.2 Discussion

The downside of the current instantiation of monitor fast-forwarding is that the user has to program the fast-forwarding abstractions manually. From our experience, coming up with fast-forwarding monitors is more challenging than devising normal monitors. The reason is that normal monitoring is usually more similar to the typical specifications accompanying industrial systems, while the logic needed for fast-forward monitoring can only be obtained by having an intimate knowledge of the system at hand. For example, referring back to the dormancy example, the industrial specifications are written in the following imperative style: (*i*) *The cycle starts when a registered user is inactive for six months, at that point the user account must be put to dormant.* (*ii*) *Whilst dormant, the user may not perform any transactions but may ask to be reactivated.* (*iii*) *If the user has been reactivated by does not carry out a financial transaction for another three months, the user account is deactivated again.* This logic can almost be directly translated into normal monitors with states and transitions. On the other hand, programming under-approximating (*i.e.,* assuming the system worked correctly before monitoring started) fast-forward monitoring would requires declarative knowledge such as: (*i*) *If the user has performed financial transactions since the last time he or she has been dormant, then the user must*

be active. (*ii*) *On the other hand, if no transactions have been carried out since, then the user is still dormant.* (*iii*) *Yet another possibility is that if the user has carried out (only) non-financial transactions, then the user has been thawed but not fully activated yet.* Although statements such as the latter can usually be inferred from the former, they are not typically written in technical specifications and engineers are more accustomed to the imperative style of specifications.

One use of more generic fast-forwarding in our case study would be to enable the monitor to keep up with the system in case asynchronous monitoring is consistently slower than the system. However, as yet, we have never encountered monitors which are not able to keep up with the system. In our case study results it is noteworthy that once monitors come at par with the system, it is not a problem for the monitors to keep up; with slightly more than two hours of processing for a day's events. Still, if one had to adapt the above given code for fast-forwarding asynchronous monitoring, this can be done by simply adding a condition in the SQL statements to ignore entries before a particular date (the date where the slow asynchronous monitors have reached).

5 Conclusion

To the best of our knowledge the idea of fast-forward monitoring is novel. A notion which relates to fast-forward monitoring is counterexample shrinking [6,7] from the area of testing. For example QuickCheck [5], a model-based testing tool for Erlang, attempts to find a shorter trace when a bug is found. This simplifies the developers' task of debugging since it is easier to understand what happened in a simpler trace. Note that counterexample shrinking is a special case of our fast-forwarding theory: (*i*) the translation functions are the identity functions, (*ii*) the trace abstraction function returns a shorter or simpler trace, and (*iii*) the same identical oracle that is used during testing is used during shrinking. Exact fast-forwarding ensures that the same bug that was exhibited during the original trace is also exhibited when monitoring the simpler one.

In a monitoring environment where the monitoring impact on system performance should be strictly minimal, fast-forward monitoring can be useful to enable the monitor to keep up with the system so that the monitoring effort is spent on monitoring the most relevant events. This problem is not only encountered the first time monitoring is deployed on the live system but also whenever a new version of monitoring code is used or a new property is added. In all such cases, the monitor has to update its state so that it comes in line with the system state. The theory of fast-forwarding has been instantiated for the monitoring tool Larva by enhancing its script with two new components which enable users to specify fast monitor bootstrapping. We have shown the usefulness of this approach for an industrial case study where monitors have to process millions of records before starting to process relevant events. In the future this technique can be incorporated with heuristics (in a similar fashion to [3]) so that the monitor can be fast-forwarded at the start of a critical section where timely monitoring is crucial. This approach might be more practical than the approach proposed in [3] of simply pausing the system to wait for the monitor to keep up.

References

1. Colombo, C., Pace, G.J., Abela, P.: Offline runtime verification with real-time properties: A case study. Tech. rep., Department of Computer Science, University of Malta, internal report 01-WICT-2009 (2009)
2. Colombo, C., Pace, G.J., Abela, P.: Compensation-Aware Runtime Monitoring. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 214–228. Springer, Heidelberg (2010)
3. Colombo, C., Pace, G.J., Abela, P.: Safer asynchronous runtime monitoring using compensations. Formal Methods in System Design 40, 1–26 (2012)
4. Colombo, C., Pace, G.J., Schneider, G.: Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 135–149. Springer, Heidelberg (2009)
5. Hughes, J.: QuickCheck Testing for Fun and Profit. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 1–32. Springer, Heidelberg (2007)
6. Leitner, A., Oriol, M., Zeller, A., Ciupa, I., Meyer, B.: Efficient unit test case minimization. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 417–420. ACM (2007)
7. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Trans. Software Eng. 28(2), 183–200 (2002)

Runtime Enforcement of Timed Properties

Srinivas Pinisetty[1], Yliès Falcone[2], Thierry Jéron[1], Hervé Marchand[1], Antoine Rollet[3] and Omer Landry Nguena Timo[4]

[1] INRIA Rennes - Bretagne Atlantique, France
First.Last@inria.fr
[2] LIG, Université Grenoble I, France
Ylies.Falcone@ujf-grenoble.fr
[3] LaBRI, Université de Bordeaux - CNRS, France
Antoine.Rollet@labri.fr
[4] IRIT, France
Omerlandry.Nguenatimo@enseeiht.fr

Abstract. Runtime enforcement is a powerful technique to ensure that a running system respects some desired properties. Using an enforcement monitor, an (untrustworthy) input execution (in the form of a sequence of events) is modified into an output sequence that complies to a property. Runtime enforcement has been extensively studied over the last decade in the context of untimed properties.

This paper introduces runtime enforcement of timed properties. We revisit the foundations of runtime enforcement when time between events matters. We show how runtime enforcers can be synthesized for any safety or co-safety timed property. Proposed runtime enforcers are time retardant: to produce an output sequence, additional delays are introduced between the events of the input sequence to correct it. Runtime enforcers have been prototyped and our simulation experiments validate their effectiveness.

1 Introduction

Runtime verification [1–6] (resp. enforcement [7–9]) refers to the theories, techniques, and tools aiming at checking (resp. ensuring) the conformance of the executions of systems under scrutiny w.r.t. some desired property. The first step of those monitoring approaches consists in instrumenting the underlying system so as to partially observe the events or the parts of its global state that may influence the property under scrutiny. A central concept is the verification or enforcement *monitor* that is generally synthesized from the property expressed in a high-level formalism. Then, the monitor can operate either *online* by receiving events in a lock-step manner with the execution of the system or *offline* by reading a log of system events. When the monitor is only dedicated to verification, it is a decision procedure emitting verdicts stating the correctness of the (partial) observed trace generated from the system execution.

Three categories of runtime verification frameworks can be distinguished according to the formalism used to express the input property. In *propositional* approaches, properties refer to events taken from a finite set of propositional names. For instance, a propositional specification may rule the ordering of function calls in a program. Monitoring such kind of specifications has received a lot of attention. *Parametric* approaches

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 229–244, 2013.
© Springer-Verlag Berlin Heidelberg 2013

have received a growing interest in the last five years. Here, events are augmented with formal parameters, instantiated at runtime. In *timed* approaches, the observed time between events may influence the truth-value of the property. It turns out that monitoring of (continuous) time specifications is a much harder problem. Intuitively, when monitoring a timed specification, the problem that arises is that the overhead induced by the monitor (i.e., the time spent executing monitor's code) influences the truth-value of the monitored specification. Consequently, not much information can be gained from the verdicts produced by the monitor. Few attempts have been made on monitoring systems w.r.t. timed properties (see Sec. 8 for related work). Two lines of work can be distinguished: synthesis of automata-based decision procedures for timed formalisms (e.g., [1, 3–5]), and, tools for runtime verification of timed properties [10, 11].

In runtime enforcement, an enforcement monitor (EM) is used to transform some (possibly) incorrect execution sequence into a correct sequence w.r.t. the property of interest. In the propositional case, the transformation performed by an EM should be *sound* and *transparent*. Soundness means that the resulting sequence obeys the property. Transparency means that, if the input sequence already conforms to the property, the monitor has to modify it in a minimal way. According to how a monitor is allowed to modify the input sequence (i.e., the primitives afforded to the monitor), several models of enforcement monitors have been proposed [7–9]. In a nutshell, an EM can definitely block the input sequence (as done by security automata), suppress an event from the input sequence (as done by suppression automata), insert an event to the input sequence (as done by insertion automata), or perform any of these primitives (as is the case with edit-automata). Moreover, according to how transparency is effectively formalized, several definitions of runtime enforcement have been proposed (see [9] for an overview).

In this paper we focus on *online enforcement of timed properties*. To the best of our knowledge, no approach was proposed to enforce timed properties. Motivations for extending runtime enforcement to timed properties abound. First, timed properties are a more precise tool to specify desired behaviors of systems since they allow to explicitly state how time should elapse between two events. Moreover, several applications of runtime enforcement of timed properties can be considered. For instance, in the context of security monitoring, enforcement monitors can be used as firewalls to prevent denial of service attacks by ensuring a minimal delay between input events (carrying some request for a protected server). On a network, enforcement monitors can be used to synchronize streams of events together, or, ensuring that a stream of events conforms to the pre-conditions of some service.

Contributions. We propose a context where, under some reasonable assumptions, runtime enforcement of timed properties is possible. For this purpose, we adapt soundness and transparency to a timed context. Runtime enforcement monitors are built from safety and co-safety properties expressed by timed automata. In contrast with previous runtime enforcement approaches, we afford only the primitives of being able to delay the input events to our enforcer. By possibly increasing delays between events of the input sequence, the output timed sequence conforms to the property. Delays are modified by monitors using an internal memory where (sequence of) events are stored and released after appropriate delays. Experiments have been performed on prototype monitors to show their effectiveness and the feasibility of our approach.

Paper organization. Section 2 introduces preliminaries and notation. Section 3 introduces the notion of enforcement for timed properties. Sections 4 and 5 describe how one can enforce safety and co-safety properties, respectively. Our prototype implementations of monitors and experiments are in Sec. 6 and Sec. 7, respectively. Section 8 discusses related work. Finally, conclusions and open perspectives are drawn in Sec. 9.

2 Preliminaries and Notation

Untimed notions. An alphabet is a finite set of elements. A (finite) word over an alphabet A is a finite sequence of elements of A. The *length* of a word w is noted $|w|$. The empty word over A is denoted by ϵ_A or ϵ when clear from context.The set of all (resp. non-empty) words over A is denoted by A^* (resp. A^+). A *language* over A is a subset $\mathcal{L} \subseteq A^*$. The concatenation of two words w and w' is noted $w \cdot w'$. For an interval $[j,k]$ in $\mathbb{N}$, by $\bigodot_{i\in[j,k]}(a_i)$ we denote the concatenation $a_j \cdot a_{j+1} \cdots a_k$. A word w' is a prefix of a word w, noted $w' \preccurlyeq w$, whenever there exists a word w'' such that $w = w' \cdot w''$. For a word w and $1 \leq i \leq |w|$, the i-th letter (resp. prefix of length i, suffix starting at position i) of w is noted $w(i)$ (resp. $w_{[\cdots i]}$, $w_{[i\cdots]}$) – with the convention $w_{[\cdots 0]} \stackrel{\text{def}}{=} \epsilon$. $\text{pref}(w)$ denotes the set of prefixes of w and by extension, $\text{pref}(\mathcal{L}) \stackrel{\text{def}}{=} \{\text{pref}(w) \mid w \in \mathcal{L}\}$ the prefix of $\mathcal{L}$. $\mathcal{L}$ is said to be *prefix-closed* whenever $\text{pref}(\mathcal{L}) = \mathcal{L}$ and *extension-closed* whenever $\mathcal{L} = \mathcal{L} \cdot A^*$. Given a tuple of symbols $e = (e_1, \ldots, e_n)$, $\Pi_i(e)$ is the projection of e on its ith element ($\Pi_i(e) \stackrel{\text{def}}{=} e_i$).

Timed languages. Let $\mathbb{R}_{\geq 0}$ denote the set of non negative real numbers, and Σ a finite alphabet of *actions*. A pair $(\delta, a) \in (\mathbb{R}_{\geq 0} \times \Sigma)$ is called an *event*. We note $\text{del}(\delta, a) = \delta$ and $\text{act}(\delta, a) = a$ the projections of events on delays and actions, respectively. A *timed word* over Σ is a finite sequence of events ranging over $(\mathbb{R}_{\geq 0} \times \Sigma)^*$. For $\sigma = (\delta_1, a_1)\cdot(\delta_2, a_2)\cdots(\delta_n, a_n)$, δ_i ($2 \leq i \leq n$) is the delay between a_{i-1} and a_i and δ_1 the time elapsed before the first action. Note that the alphabet is infinite in this case. Nevertheless, previous notions and notations defined above (related to length, concatenation, prefix, etc) naturally extend to timed words. *The sum of delays* of a timed word σ is noted $\text{time}(\sigma)$. Given $t \in \mathbb{R}_{\geq 0}$, and a timed word $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$, we define the *observation of σ at time t* as the timed word $\text{obs}(\sigma, t) \stackrel{\text{def}}{=} \max\{\sigma' \mid \sigma' \preccurlyeq \sigma \wedge \text{time}(\sigma') \leq t\}$, i.e., the longest prefix of σ with a sum of delays less than t. The *untimed projection* of σ is $\Pi_\Sigma(\sigma) \stackrel{\text{def}}{=} a_1 \cdot a_2 \cdots a_n$ in Σ^* (i.e., delays are ignored). A *timed language* is any subset $\mathcal{L} \subseteq (\mathbb{R}_{\geq 0} \times \Sigma)^*$. We define the following order on timed words: σ' *delays* σ (noted $\sigma' \preccurlyeq_d \sigma$) if $\Pi_\Sigma(\sigma') \preccurlyeq \Pi_\Sigma(\sigma)$ and $\forall i \leq |\sigma'|, \text{del}(\sigma(i)) \leq \text{del}(\sigma'(i))$.

Timed Automata. Let $X = \{X_1, \ldots, X_k\}$ be a finite set of *clocks*. A *clock valuation* for X is a function ν from X to $\mathbb{R}_{\geq 0}^X$ where $\mathbb{R}_{\geq 0}^X$ denotes the valuations of X. For $\nu \in \mathbb{R}_{\geq 0}^X$ and $\delta \in \mathbb{R}_{\geq 0}$, $\nu + \delta$ is the valuation assigning $\nu(X_i) + \delta$ to each clock X_i of X. Given a set of clocks $X' \subseteq X$, $\nu[X' \leftarrow 0]$ is the clock valuation ν where all clocks in X' are assigned to 0. $\mathcal{G}(X)$ denotes the set of clock constraints defined as boolean combinations of simple constraints of the form $X_i \bowtie c$ with $X_i \in X, c \in \mathbb{N}$ and $\bowtie \in \{<, \leq, =, \geq, >\}$. Given $g \in \mathcal{G}(X)$ and $\nu \in \mathbb{R}_{\geq 0}^X$, we write $\nu \models g$ when $g(\nu) \equiv \mathit{true}$.

Definition 1 (Timed automaton). *A* timed automaton *(TA) is a tuple* $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G\rangle$*, s.t.* L *is a finite set of* locations *with* $l_0 \in L$ *the* initial location*,* X *is a finite set of clocks,* Σ *is a finite set of* events*,* $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ *is the* transition relation*, and* $G \subseteq L$ *is a set of accepting locations.*

The semantics *of a TA is a timed transition system* $[\![\mathcal{A}]\!] = \langle Q, q_0, \Gamma, \rightarrow, F_G\rangle$ *where* $Q = L \times \mathbb{R}_{\geq 0}^X$ *is the (infinite) set of* states*,* $q_0 = (l_0, \nu_0)$ *is the initial state where* ν_0 *is the valuation that maps every clock to* 0*,* $F_G = G \times \mathbb{R}_{\geq 0}^X$ *is the set of of accepting states,* $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$ *is the set of transition* labels*, i.e., pairs composed of a delay and an action. The transition relation* $\rightarrow \subseteq Q \times \Gamma \times Q$ *is a set of transitions of the form* $(l, \nu)\xrightarrow{(\delta,a)}(l', \nu')$ *with* $\nu' = (\nu + \delta)[Y \leftarrow 0]$ *whenever there exists* $(l, g, a, Y, l') \in \Delta$ *s.t.* $\nu + \delta \models g$ *for* $\delta \geq 0$*.*

In the following, we consider a timed automaton $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G\rangle$ with its semantics $[\![\mathcal{A}]\!]$. $\mathcal{A}$ is *deterministic* whenever for any (l, g_1, a, Y_1, l'_1) and (l, g_2, a, Y_2, l'_2) in Δ, $g_1 \wedge g_2$ is *false*. $\mathcal{A}$ is *complete* whenever for any location $l \in L$ and every event $a \in \Sigma$, the disjunction of the guards of the transitions leaving l and labeled by a is *true*. In the remainder of this paper, we shall consider only deterministic timed automata.

A *run* ρ from $q \in Q$ is a sequence of moves in $[\![\mathcal{A}]\!]$ of the form: $\rho = q_0 \xrightarrow{(\delta_1,a_1)} q_1 \cdots q_{n-1} \xrightarrow{(\delta_n,a_n)} q_n$. The set of runs from $q_0 \in Q$ is denoted $Run(\mathcal{A})$ and $Run_{F_G}(\mathcal{A})$ denotes the subset of runs *accepted* by $\mathcal{A}$, i.e., ending in F_G. The *trace* of a run ρ is the timed word $(\delta_1, a_1) \cdot (\delta_2, a_2) \cdots (\delta_n, a_n)$. We note $\mathcal{L}(\mathcal{A})$ the set of traces of $Run(\mathcal{A})$. We extend this notation to $\mathcal{L}_{F_G}(\mathcal{A})$ in a natural way.

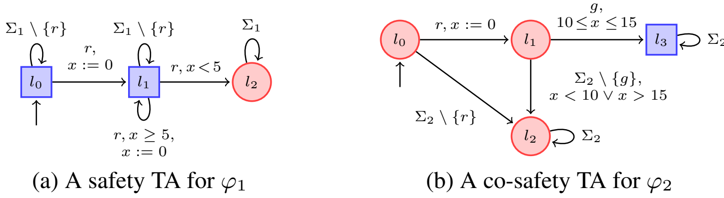
Timed Properties. A timed property is defined by a timed language $\varphi \subseteq (\mathbb{R}_{\geq 0} \times \Sigma)^*$. Given a timed word $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$, we say that σ satisfies φ (noted $\sigma \models \varphi$) if $\sigma \in \varphi$. In the sequel, we shall be interested in safety and co-safety timed properties. Informally, safety (resp. co-safety) properties state that "nothing bad should ever happen" (resp. "something good should happen within a finite amount of time"). Safety (resp. co-safety) properties can be characterized by prefix-closed (resp. extension-closed) languages. We consider only the sets of safety and co-safety properties that can be represented by timed automata (Definition 1).

Definition 2 (Safety and Co-safety TA). *A complete and deterministic TA* $\langle L, l_0, X, \Sigma, \Delta, G\rangle$*, where* $G \subseteq L$ *is the set of accepting locations, is said to be:*

- *a* safety *TA if* $\nexists\langle l, g, a, Y, l'\rangle \in \Delta, l \in L \setminus G \wedge l' \in G$*;*
- *a* co-safety *TA if* $\nexists\langle l, g, a, Y, l'\rangle \in \Delta, l \in G \wedge l' \in L \setminus G$*.*

It is easy to check that safety and co-safety TAs define safety and co-safety properties.

Example 1 (Safety and co-safety TA). Fig. 1a and 1b present two properties formalized with safety and co-safety TA. Accepting locations are represented by squares. The safety TA formalizes the property φ_1 defined over $\Sigma_1 = \{a, r\}$: *"There should be a delay of at least 5 time units between any two user requests (*r*)"*. The co-safety TA formalizes the property φ_2 defined over $\Sigma_2 = \{r, g, a\}$: *"The user can perform an action* a *only after a successful authentication, i.e., after sending a request* r *and receiving a grant* g*. After an* r*,* g *should occur between 10 and 15 time units"*.



(a) A safety TA for φ_1 (b) A co-safety TA for φ_2

Fig. 1. Example of Timed Properties

3 Enforcement Monitoring in a Timed Context

Roughly speaking, both in the timed and untimed settings, the purpose of an enforcement monitor (EM) is to read some (possibly incorrect) input sequence σ produced by a running system (input to the enforcer), and to transform it into an output sequence o that is correct w.r.t. a property φ, here modeled by a TA. From an abstract point of view, an enforcement monitor realizes an enforcement function E that transforms timed words into timed words according to global time.

Definition 3. *For a given property φ, an* enforcement function *is a function E from* $(\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0}$ *to* $(\mathbb{R}_{\geq 0} \times \Sigma)^*$.

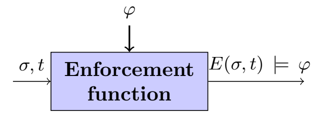


Fig. 2. Enforcement function E

An enforcement function E transforms some timed word σ given as input and possibly incorrect w.r.t. the desired property (see Fig. 2). The resulting output $E(\sigma, t)$ at time t is a timed word with same actions, but possibly increased delays between actions so that it satisfies the property. Similar to the untimed setting, additional constraints on $E(\sigma, t)$, namely *soundness* and *transparency*, are required on actions. However, in the timed setting, those constraints also depend on both delays between events and the class of the enforced property, as we shall discuss later.

An enforcement function E is realized by an *enforcement monitor* EM. This monitor is equipped with a memory and a set of enforcement operations used to store and dump some timed events to and from the memory, respectively. The memory of an EM is basically a queue containing a timed word, the received actions with increased delays that have not been released yet. In addition, the EM also keeps track of the state of the TA modeling the property, satisfaction of the property using a Boolean variable, and some variables indicating the clock values used to count time between input and output events.

The specific operations of the EM are the *Store* operation which stores in memory the received action together with a possibly modified delay; the *Dump* operation which releases the first action from the memory; and the optional *Halt* operation which stops the enforcer, i.e., blocks the input sequence and stops producing outputs. *Off* operation which turns off the enforcer. The *Off* and *Halt* operations can be added for optimization. The *Off* can be used when we observe that the property will be satisfied for any future input events. The *Halt* operation is useful if the property cannot be satisfied anymore.

In the following sections, we will present enforcement monitors for both safety and co-safety properties and analyze constraints on the associated enforcement functions.

4 Enforcement of Safety Properties

In this section we focus on the enforcement of a safety property φ specified by a safety automaton $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G\rangle$ and its associated semantics $[\![\mathcal{A}]\!] = \langle Q, q_0, \Gamma, \rightarrow, F_G\rangle$. Without loss of generality, we assume that the set of locations $L \setminus G$ is reduced to a singleton $\{Bad\}$. Given φ, and a timed word σ, an enforcement function E for φ should satisfy the following soundness, transparency and optimality conditions.

Definition 4 (Soundness, transparency and optimality). *Let* $E : (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \rightarrow (\mathbb{R}_{\geq 0} \times \Sigma)^*$ *be an enforcement function for a safety property* φ*.* E *is:*
- sound *if* $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}$, $E(\sigma, t) \models \varphi$;
- transparent *if* $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}$, $E(\sigma, t) \preccurlyeq_d \mathrm{obs}(\sigma, t) \wedge \mathrm{time}(E(\sigma, t)) \leq t$.

If E *is both sound and transparent, we say that it is* optimal *if, for any input* $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$*, at any time* $t \in \mathbb{R}_{\geq 0}$*, the following constraints hold:*
(Op1) $\nexists \omega', \omega' \models \varphi \;\wedge\; \omega' \preccurlyeq_d obs(\sigma, t) \;\wedge\; |\omega'| > |E(\sigma, t)|$

(Op2) $$\begin{aligned}\forall i \in [1, |E(\sigma,t)|], \nexists \delta'' \in \mathbb{R}_{\geq 0},\ &\mathrm{del}(\mathrm{obs}(\sigma,t)(i)) \leq \delta'' \leq \mathrm{del}(E(\sigma,t)(i)) \\ &\wedge E(\sigma,t)_{[\cdots i-1]} \cdot (\delta'', \mathrm{act}(E(\sigma,t)(i))) \models \varphi\end{aligned}$$

Soundness means that, at any time t, the produced timed word should satisfy the property φ. Transparency means that, at any time instant t, the output $E(\sigma, t)$ delays the input $\mathrm{obs}(\sigma, t)$: the enforcement function should not modify the order of events, should not reduce the delays between consecutive events, and should not produce outputs faster than inputs. Optimality means that the enforcement function should provide the output as soon as possible. The optimality condition **(Op1)** extends the requirement on the output sequences of the enforcement function in the untimed case (cf. [9]): at any time instant t, the output sequence $E(\sigma, t)$ should be the longest correct timed word delaying the input sequence $\mathrm{obs}(\sigma, t)$. Here, taking physical time into account, **(Op2)** requires that the input and output sequences are as close as possible w.r.t. physical observation, i.e., every prefix of $E(\sigma, t)$ has the shortest possible last delay.

We now design an enforcement monitor whose semantics effectively realizes the enforcement function as described Definition 4.

Definition 5 (Enforcement Monitor for safety). *An enforcement monitor for* φ *is a transition system* $EM = \langle C, C_0, \Gamma_{EM}, \hookrightarrow\rangle$ *s.t.:*

- $C = (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{B} \times Q$ *is the set of configurations;*
- *the initial configuration is* $C_0 = \langle \epsilon, 0, 0, \mathtt{tt}, q_0\rangle \in C$*;*
- $\Gamma_{EM} = \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ *is the input-operation-output alphabet, where* $Op = \{\mathrm{store}(\cdot), \mathrm{dump}(\cdot), \mathrm{del}(\cdot)\}$*;*
- $\hookrightarrow \subseteq C \times \Gamma_{EM} \times C$ *is the transition relation defined as the smallest relation obtained by the following rules applied in the following order:*

- *store:* $\langle \sigma_s, \delta, d, \mathtt{tt}, q\rangle \overset{(\delta,a)/\mathrm{store}(\delta',a)/\epsilon}{\hookrightarrow} \langle \sigma_s \cdot (\delta', a), 0, d, (\delta' \neq \infty), q'\rangle$ *with:*
  - $*$ $\delta' = \mathrm{update_s}(q, a, \delta)$, *where* $\mathrm{update_s}$[1] *is the function defined as:*

$$Q \times \Sigma \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}_{\geq 0}$$
$$(q, a, \delta) \mapsto \begin{cases} \infty & \text{if } \forall \delta' \in \mathbb{R}_{\geq 0}, \forall q_1 \in Q, (\delta' \geq \delta \wedge q \overset{(\delta',a)}{\rightarrow} q_1) \Rightarrow q_1 \notin F_G \\ \min\{\delta' \in \mathbb{R}_{\geq 0} \mid \exists q_1 \in F_G, q \overset{(\delta',a)}{\rightarrow} q_1 \wedge \delta' \geq \delta\} \end{cases}$$

  - $*$ q' *is defined as* $q \overset{(\delta',a)}{\rightarrow} q'$ *if* $\delta' < \infty$ *and* $q' = q$ *otherwise;*
- *dump:* $\big\langle (\delta, a) \cdot \sigma_s, s, \delta, b, q\big\rangle \overset{\epsilon/\mathrm{dump}(\delta,a)/(\delta,a)}{\hookrightarrow} \langle \sigma_s, s, 0, b, q\rangle$ *if* $\delta \neq \infty$*;*
- *delay:* $\langle \sigma_s, s, d, b, q\rangle \overset{\epsilon/\mathrm{del}(\delta)/\epsilon}{\hookrightarrow} \langle \sigma_s, s + \delta, d + \delta, b, q\rangle$.

A configuration $\langle \sigma_s, s, d, b, q\rangle$ of the EM consists of the current stored sequence (i.e., the memory content) σ_s, two clock values s and d indicating respectively the time elapsed since the last store and dump operations, a Boolean b indicating whether the underlying enforced property is satisfied or not on the output sequence, and q the current state of $[\![\mathcal{A}]\!]$ reached after processing the sequence already released followed by the timed word in memory. Regarding its alphabet, in the input (resp. output) sequence, the EM either lets time elapse and no event is read or released, or reads and stores (resp. dumps and releases) a symbol event after some delays. Semantics rules can be understood as follows:

- The *store* rule is executed upon the reception of an event (δ, a). The timed event (δ', a) is appended to the memory content, where δ' is the minimal delay that has to be waited so that the property remains satisfied – if such a delay exists. The value of s is then reinitialized to 0. If a delay can be found through the $\mathrm{update_s}$ function, q is updated to the state that will be reached by appending the timed event (δ', a) to the output sequence concatenated with the contents of the memory, and b remains $\mathtt{tt}$ and becomes $\mathtt{ff}$ otherwise.
- The *dump* rule is executed when the value of d is equal to the delay of the first timed event in the memory. The value of d is then reinitialized to 0. The first event in memory is suppressed (and released from the enforcer). Other elements of the configuration remain unchanged.
- The *delay* rule adds the time elapsed δ to the current values of s and d when no store nor dump operation is possible.

Remark 1. The model of enforcement monitor presented in Definition 5 can be easily extended by relaxing two hypothesis: in the *store* rule, we check whether there is a delay greater than δ allowing the output sequence to stay in the accepting states of the property ($\delta' = \infty$). Of course, this condition can be adapted to a given time bound in $\mathbb{R}_{\geq 0}$. More complex conditions are also possible according to some desired quality of service. Similarly, processing input and output actions is assumed to be done in zero time. Some delay (either fixed or depending on additional parameters) can be considered for this action by modifying the *store* rule.

[1] The $\mathrm{update_s}$ function computes the minimal delay $\delta' \geq \delta$, such that the safety-property automaton still remains in an accepting state after processing the action a.

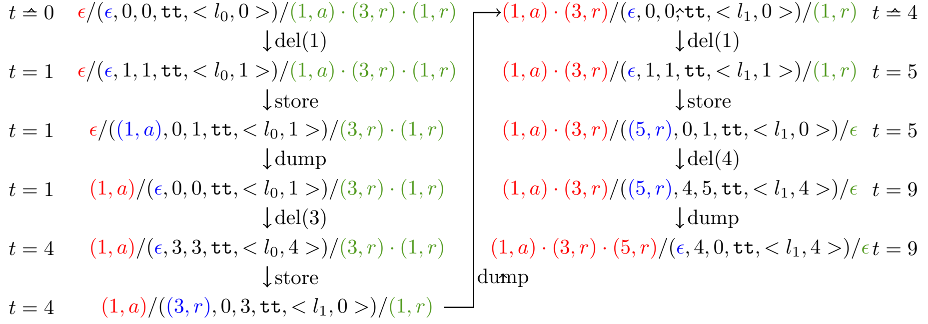


Fig. 3. Enforcer configuration evolution

We define the language of runs of an enforcement monitor EM:

$$\mathcal{L}(EM) \subseteq (\Gamma_{EM})^* = \Big(\big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)\Big)^*$$

It is worth noticing that enforcement monitors are deterministic. Hence, given $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$ and $t \in \mathbb{R}_{\geq 0}$, let $w \in \mathcal{L}(EM)$ be the unique maximal sequence such that

$$\Pi_\epsilon\Big(\bigodot_{i\in[1,|w|]} (\Pi_1(w(i)))\Big) = \text{obs}(\sigma, t),$$

where Π_ϵ is the projection that erases ϵ from words in $((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\})^*$.

Now, we define the enforcement function E associated to EM as

$$\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*,\ \forall t \in \mathbb{R}_{\geq 0},\ E(\sigma, t) = \Pi_\epsilon\Big(\bigodot_{i\in[1,|w|]} (\Pi_3(w(i)))\Big) \tag{1}$$

Proposition 1. *Given an enforcement monitor EM for a safety property φ and E defined as in Eq. (1), E verifies the soundness, transparency and optimality conditions of Definition 4.*

Example 2. Let us illustrate how these rules are applied to enforce φ_1 (represented by the TA in Fig. 1a with $\Sigma_1 = \{a, r\}$). Let us consider the input timed word $\sigma = (1, a) \cdot (3, r) \cdot (1, r)$. Figure 3 shows how successive rules are applied and the evolution of the configurations of the EM. The variable t describes global time. The input is represented on the right-hand (resp. left-hand) side of the configuration.

5 Enforcement of Co-safety Properties

Let us now focus on the enforcement of co-safety properties. We assume a co-safety property φ specified by a co-safety timed automaton $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G\rangle$ and its associated semantics $[\![\mathcal{A}]\!] = \langle Q, q_0, \Gamma, \rightarrow, F_G\rangle$. An enforcer function E for a co-safety property φ should satisfy new soundness, transparency and optimality conditions.

Before defining those constraints, the notion of a sequence delaying another has to be modified in the context of co-safety properties. Let $\sigma, \sigma' \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$ be two timed sequences, we note $\sigma' \preccurlyeq_c \sigma$ for $\Pi_\Sigma(\sigma') = \Pi_\Sigma(\sigma) \wedge \forall i \leq |\sigma'|, \mathrm{del}(\sigma'(i)) \geq \mathrm{del}(\sigma(i))$. This order between timed words shall be used in the transparency and optimality conditions below to constrain the sequences produced by an enforcer. We define $\gamma(\sigma) \stackrel{\text{def}}{=} \{\sigma' \preccurlyeq_c \sigma \mid \sigma' \models \varphi\}$, the set of sequences delaying σ and satisfying the property φ and $\gamma_t(\sigma) \stackrel{\text{def}}{=} \{\mathrm{time}(\sigma') \mid \sigma' \in \gamma(\sigma)\}$ the set of sums of delays of these sequences.

Definition 6 (Soundness, transparency, optimality). *An enforcement function* $E : (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \to (\mathbb{R}_{\geq 0} \times \Sigma)^*$ *for a co-safety property* φ *is*
- *sound if* $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}, E(\sigma, t) \neq \epsilon \Rightarrow (\exists t' \geq t, E(\sigma, t') \models \varphi)$.
- *transparent if* $\forall \sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*, \forall t \in \mathbb{R}_{\geq 0}, E(\sigma, t) \neq \epsilon \Rightarrow (\exists t' \geq t, E(\sigma, t') \preccurlyeq_c \mathrm{obs}(\sigma, t))$.

If E *is sound and transparent, it is optimal if for any input* $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^*$*, at any time* $t \in \mathbb{R}_{\geq 0}$*, the following constraints hold:*

(Op1) $\gamma\big(\mathrm{obs}(\sigma, t)\big) \neq \emptyset \wedge \forall t' < t, \gamma(obs(\sigma, t') = \emptyset) \Rightarrow \big(\exists t' \geq t, |E(\sigma, t')| = |obs(\sigma, t)| \wedge t' = t + time(E(\sigma, t'))\big)$;
(Op2) $E(\sigma, t) \neq \epsilon \Rightarrow (1) \wedge (2)$*, where*

let $E(\sigma, t)_{[\cdots n]}$ *be the smallest prefix of* $E(\sigma, t)$ *s.t.* $E(\sigma, t)_{[\cdots n]} \models \varphi$ *in*

(1) $\nexists(\delta'_1, \ldots, \delta'_n), \sum_{i=1}^{n} \delta_i{}' \leq \sum_{i=1}^{n} \mathrm{del}(E(\sigma, t)(i)) \wedge \bigodot_{i \in [1,n]}(\delta_i{}', \mathrm{act}((\sigma(i)))) \models \varphi \wedge \forall i \in [1, n], \mathrm{del}(\sigma(i)) \leq \delta_i{}'$

(2) $E(\sigma, t) \models \varphi \Rightarrow \big(E(\sigma, t) = E(\sigma, t)_{[\cdots n]} \cdot \mathrm{obs}(\sigma, t)_{[n+1 \cdots |E(\sigma,t)|]}\big)$

Soundness means that if a timed word is released by the enforcement function, in the future, the output timed word of the enforcement function should satisfy the property φ. [2] Transparency means that the enforcement function should not change the order of events, and the delay between any two consecutive events cannot be reduced.

Optimality means that the output is produced as soon as possible: $\mathbf{Op1}$ means that if t is the first time instant at which there is a timed word that delays $\mathrm{obs}(\sigma, t)$ and satisfies φ, then, in the future at time $t' = t + \mathrm{time}(E(\sigma, t'))$, the enforcement monitor should have output exactly all the observed events until time t. $\mathbf{Op2}$-1 means that if $E(\sigma, t) \neq \epsilon$ is released by the enforcement function, for the smallest prefix $E(\sigma, t)_{[\cdots n]}$ that satisfies φ, the total amount of time spent to trigger $E(\sigma, t)_{[\cdots n]}$ should be minimal.$\mathbf{Op2}$-2 means that the delay between the remaining actions $\mathrm{obs}(\sigma, t)_{[n+1 \cdots]}$ (i.e., when the property is satisfied) should not be changed. Similarly to safety properties, we expect the enforcement function to minimally alter the initial sequence: after correcting an incorrect prefix, the remainder of the sequence should be the same for events and delays between them.

Before presenting the definition of enforcement monitor, we introduce $\mathrm{update_c}$ as a function from $(\mathbb{R}_{\geq 0} \times \Sigma)^+ \to \mathbb{R}^+_{\geq 0} \times \mathbb{B}$ such that for $\sigma \in (\mathbb{R}_{\geq 0} \times \Sigma)^+$

[2] As usual in runtime enforcement, either it is assumed that the empty sequence ϵ does belong to the property or the soundness constraint does not take ϵ into account.

$$\text{update}_c(\sigma) \stackrel{\text{def}}{=} \begin{cases} \big((\delta_1, \ldots, \delta_{|\sigma|}), tt\big) \text{ s.t. } \sum_{i=1}^{|\sigma|} \delta_i = \min\{\gamma_t(\sigma)\}, \text{ if } \gamma(\sigma) \neq \emptyset \\ \big((\text{del}(\sigma(1)), \ldots, \text{del}(\sigma(|\sigma|))), \texttt{ff}\big), \text{otherwise} \end{cases}$$

Definition 7 (Enforcement Monitor for co-safety properties). *An enforcement monitor EM for φ is a transition system $\langle C, C_0, \Gamma, \hookrightarrow\rangle$ s.t.:*

- $C = (\mathbb{R}_{\geq 0} \times \Sigma)^* \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{B}$ *is the set of configurations and the initial configuration is* $C_0 = \langle \varepsilon, 0, 0, \texttt{ff}\rangle \in C$*;*
- $\Gamma_{EM} = (\mathbb{R}_{\geq 0} \times \Sigma) \times Op \times (\mathbb{R}_{\geq 0} \times \Sigma)$ *is the "input-operation-output" alphabet, where* $Op = \{\text{store-}\overline{\varphi}(\cdot), \text{store-}\varphi_{\text{init}}(\cdot), \text{store-}\varphi(\cdot), \text{dump}(\cdot), \text{delay}(\cdot)\}$*;*
- $\hookrightarrow \subseteq C \times \Gamma_{EM} \times C$ *is the transition relation defined as the smallest relation obtained by the following rules applied with the priority order below:*
  1. store-$\overline{\varphi}$: $\langle \sigma_s, \delta, d, \texttt{ff}\rangle \overset{(\delta,a)/\text{store}-\overline{\varphi}(\delta,a)/\epsilon}{\hookrightarrow} \langle \sigma_s \cdot (\delta, a), 0, d, \texttt{ff}\rangle$
     if $\Pi_2\big(\text{update}_c(\sigma_s \cdot (\delta, a))\big) = \texttt{ff}$
  2. store-φ_{init}: $\langle \sigma_s, \delta, d, \texttt{ff}\rangle \overset{(\delta,a)/\text{store}-\varphi_{\text{init}}(\delta',a)/\epsilon}{\hookrightarrow} \langle \sigma'_s, 0, 0, \texttt{tt}\rangle$
     if $\Pi_2\big(\text{update}_c(\sigma_s \cdot (\delta, a))\big) = \texttt{tt}$ *with*
     - $\delta' = \Pi_1\big(\text{update}_c(\sigma_s \cdot (\delta, a))\big)$
     - $\sigma'_s = \bigodot_{i \in [1, |\sigma_s|]} (\Pi_i(\delta'), \text{act}(\sigma_s(i))) \cdot (\delta'_{|\sigma_s|+1}, a)$
  3. store-φ: $\langle \sigma_s, \delta, d, \texttt{tt}\rangle \overset{(\delta,a)/\text{store}-\varphi(\delta,a)/\epsilon}{\hookrightarrow} \langle \sigma_s \cdot (\delta, a), 0, d, \texttt{tt}\rangle$
  4. *dump:* $\big\langle (\delta, a) \cdot \sigma_s, s, \delta, \texttt{tt}\big\rangle \overset{\epsilon/\text{dump}(\delta,a)/(\delta,a)}{\hookrightarrow} \langle \sigma_s, s, 0, \texttt{tt}\rangle$
  5. *delay:* $\langle \sigma_s, s, d, b\rangle \overset{\epsilon/\text{delay}(\delta)/\epsilon}{\hookrightarrow} \langle \sigma_s, s + \delta, d + \delta, b\rangle$.

The EM either lets time elapse when no event is read or released as output, or reads and stores (resp. dumps and outputs) an event after some delay. Semantic rules can be understood as follows:

- Upon reception of an event (δ, a), one of the three store rules is executed. The rule store-$\overline{\varphi}$ is executed if b ff and the property still remains unsatisfied after this new event (i.e., when the update_c function returns ff). If the update_c function returns tt (indicating that the φ can now be satisfied), then the rule store-φ_{init} is executed. When executing this rule, d is reset to 0, indicating that the enforcer can start outputting events. The rule store-φ is executed if the Boolean in the current configuration is tt, which indicates that the property is already satisfied by the inputs received earlier. So, in this case, it is not necessary to invoke the update_c function, and the event (δ, a) is appended to the memory.
- The *dump* rule is similar to the one of the enforcement of safety properties except that we wait that the Boolean indicating property satisfaction becomes tt.
- The *delay* rule adds the time elapsed to the current clock values s and d.

Note that, in this case, time measured in output starts elapsing upon property satisfaction by the memory content (contrarily to the safety case, where it starts with the enforcer).

As was the case in the previous section, from EM, we can define an enforcement function E as in Eq. (1), such that the following proposition holds:

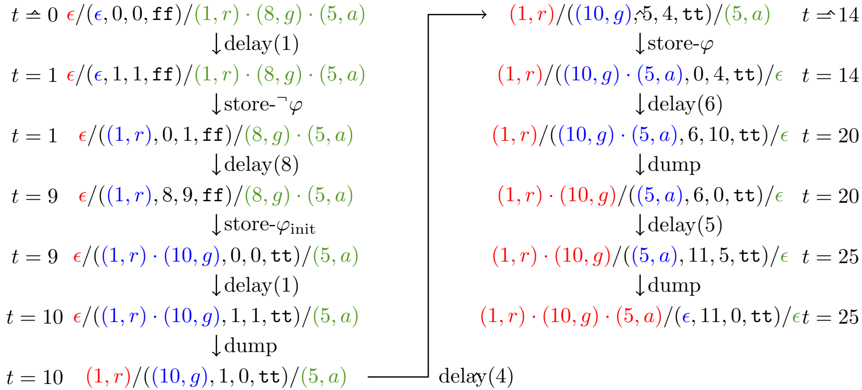


Fig. 4. Enforcer configuration evolution

Proposition 2. *Given an enforcement monitor EM for a co-safety property and E defined as in Eq (1), E is sound, transparent and optimal as per Definition 6.*

Example 3. Let us illustrate how these rules are applied to enforce φ_2 (Fig. 1b), with $\Sigma_2 = \{r, g, a\}$. Let us consider the input timed word $\sigma = (1, r) \cdot (8, g) \cdot (5, a)$. Figure 4 shows how semantic rules are applied, and the evolution of the configurations of the EM. The input is shown on the right of the configuration, and the output is presented on the left. The variable t describes global time. The resulting output is $E(\sigma) = (1, r) \cdot (10, g) \cdot (5, a)$, which satisfies the property φ presented in Fig. 1b.

6 Implementation

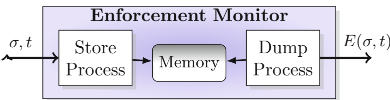


Fig. 5. Realizing an EM

Let us now provide the algorithms showing how enforcement monitors can be implemented. As shown in Fig. 5, the implementation of an enforcement monitor (EM) consists of two processes running concurrently (Store and Dump) and a memory. The Store process models the store rules. The memory contains the timed words σ_s. The Dump process reads events stored in the memory and releases them as output after the required amount of time. To define the enforcement monitors, the following algorithms assume a TA $\mathcal{A} = \langle L, l_0, X, \Sigma, \Delta, G \rangle$.

We now describe these processes for safety properties.

- The $\mathsf{DumpProcess_{safety}}$ algorithm (see Algorithm 1) is an infinite loop that scrutinizes the memory and proceeds as follows: Initially, d is set to 0. If the memory is empty ($|\sigma_s| = 0$), it waits until a new element (δ, a) is stored in memory, otherwise it proceeds with the first element in memory. Meanwhile, d keeps track of the time elapsed since the last dump operation. The $\mathsf{DumpProcess_{safety}}$ waits for $(\delta - d)$ time units before releasing the action a and resets d.

Algorithm 1. $\mathsf{DumpProcess}_{\mathsf{safety}}$

```
d ← 0
while tt do
    await (|σ_s| ≥ 1)
    (δ, a) ← dequeue (σ_s)
    wait (δ − d)
    dump (a)
    d ← 0
end while
```

Algorithm 2. $\mathsf{StoreProcess}_{\mathsf{safety}}$

```
(l, X) ← (l_0, [X ← 0])
while tt do
    (δ, a) ← await event
    if (post(l, X, a, δ) ∉ G) then
        δ' ← update(l, X, a, δ)
        if δ' = ∞ then
            terminate StoreProcess
        end if
    else
        δ' ← δ
    end if
    (l, X) ← post(l, a, X, δ')
    enqueue (δ', a)
end while
```

Algorithm 3. $\mathsf{DumpProcess}_{\mathsf{co-safety}}$

```
await startDump
d ← 0
while tt do
    await (|σ_s| ≥ 1)
    (δ, a) ← dequeue (σ_s)
    wait (δ − d)
    dump (a)
    d ← 0
end while
```

Algorithm 4. $\mathsf{StoreProcess}_{\mathsf{co-safety}}$

```
goalReached ← ff
while tt do
    (δ, a) ← await (event)
    enqueue(δ, a)
    if goalReached = ff then
        (newDelays, R) ← update_c(σ_s)
        if R = tt then
            modify delays
            goalReached ← tt
            notify (startDump)
        end if
    end if
end while
```

- The $\mathsf{StoreProcess}_{\mathsf{safety}}$ algorithm (see Algorithm 2) is an infinite loop that scrutinizes the system for input events. It proceeds as follows. Let (l, X) be the state of the property automaton, where l represents the location and X is the current clock values initialized to $(l_0, 0)$. The function post takes a state of the property automaton (l, X), an event (δ, a), and computes the state reached by the property automaton. The update function computes a new delay δ' such that the property automaton will reach an accepting state in an optimal way by triggering (δ', a).

  We now describe these processes for co-safety properties.
- The $\mathsf{DumpProcess}_{\mathsf{co-safety}}$ algorithm for co-safety properties (see Algorithm 3) resembles the one of the safety case. The only difference is that the infinite loop starts only after receiving the *startDump* notification from the $\mathsf{StoreProcess}_{\mathsf{co-safety}}$.
- In the $\mathsf{StoreProcess}_{\mathsf{co-safety}}$ algorithm (see Algorithm 4), *goalReached* is a Boolean, used to indicate if the goal location is visited by the input events which were already processed. It is initialized to `ff`. The update_c function takes all events stored in the enforcer memory, and returns new delays and if the goal location is reachable.

startDump is a notification message sent to the $\mathsf{DumpProcess}_{\mathsf{co-safety}}$, to indicate that it can start dumping the events stored in the memory. Note that the update_c can be easily implemented using the optimal path routine of UPPAAL.

7 Evaluation

Enforcement monitors for safety and co-safety properties, based on the algorithms presented in the previous section, have been implemented in prototype tool of 500 LOC using Python. The tool also uses UPPAAL [12] as a library to implement the update function and the pyuppaal library to parse UPPAAL models written in XML.

We present some performance evaluation on a simulated system where the input timed trace is generated. As described in Sec. 6, enforcement monitors for safety and co-safety properties are implemented by two concurrent processes. The TA representing the property is a UPPAAL model, and is an input to the enforcement monitor. The UPPAAL model also contains another automaton representing the sequence of events received by the enforcement monitor. The update function of the StoreProcess uses UPPAAL. Experiments were conducted on an Intel Core i7-2720QM at 2.20GHz CPU, and 4 GB RAM running on Ubuntu 12.04 LTS. Note that the implementation is a prototype, and there is still scope for improving the performance.

Results of the performance analysis of our running example properties are presented in Tables 1a and 1b. The values are presented in seconds. Average values are computed over multiple runs. The length of the input trace is denoted by $|tr|$. The entry t_tr represents the time taken by the system simulator process to generate the trace. The entry t_update (resp. t_Post) indicates the time taken for one call to the update (resp. post) function when the last event of the input trace is received. The entry t_EM presents the total time from the start of the simulation until the last event is dumped by the enforcer. The throughput shows how many events can be processed by the enforcer ($|\mathrm{tr}|/\mathrm{t_EM}$).

We observe that the throughput decreases with the length of the input trace. This unexpected behavior stems from the external invocation of UPPAAL to realize post and update functions. Indeed, after each event, the length of the automaton representing the trace grows, and, as indicated in Table 1a, the time taken by update and post functions also increases, unnecessarily starting the computation from the initial location each time an event is received. Future implementations will avoid this by realizing the post and update functions online from the current state. Performance and throughput shall be independent from the trace length. Further experiments have been carried out on different examples similarly demonstrating feasibility and scalability.

For co-safety properties, regarding the total time t_EM, note that the most expensive operation update is called upon each event. Moreover, examining the column t_update in Table 1b, the time taken by the update function increases with the number of events. This behavior is expected for co-safety properties, as we check for an optimal output from the initial state after each event. Please note that in case of a co-safety property, once the property is satisfied (a good location is reached), it is not necessary to invoke the update function. From that point onwards, the increase in total time t_EM per event will be very less (since we just add the received event to the output queue), and t_update will be zero for the events received later on.

Table 1. Performance analysis of enforcement monitors

(a) For φ_1

$\|tr\|$	t_update	t_post	t_tr	t_EM	throughput
100	0.0433	0.0383	0.00483	2.648	37
200	0.08196	0.07158	0.0087	9.135	21.89
300	0.121	0.1065	0.0118	19.42	15.46
400	0.1696	0.1525	0.0133	34.314	11.65
500	0.2148	0.1891	0.0142	53.110	9.41
600	0.2668	0.2334	0.0166	77.428	7.75
700	0.3164	0.2789	0.0178	107.61	6.50
800	0.3669	0.3289	0.0198	143.53	5.57
900	0.4256	0.3810	0.0237	181.06	4.97
1000	0.4878	0.4352	0.0259	229.12	4.36

(b) For φ_2

|tr|	t_update	t_tr	t_EM
100	0.063	0.0026	1.28
200	0.17	0.0065	8
300	0.33	0.0081	25
400	0.54	0.0115	58
500	0.79	0.0131	109
600	1.11	0.0157	186
700	1.50	0.0186	297
800	1.96	0.0209	462
900	2.40	0.0234	623
1000	2.84	0.0341	852

8 Related Work

This work is by no means the first to address monitoring of timed properties. Matteucci inspires from partial-model checking techniques to synthesize controller operations to enforce safety and information-flow properties using process-algebra [13]. Monitors are close to Schneider's security automata [7]. The approach targets discrete-time properties and systems are modelled as timed processes expressed in CCS. Compared to our approach, the description of enforcement mechanisms remains abstract, directly restricts the monitored system, and no description of monitor implementation is proposed.

Other research efforts aim to mainly runtime verify timed properties and we shall categorize them into i) rather theoretical efforts aiming at synthesizing monitors, and ii) tools for runtime monitoring of timed properties.

Synthesis of timed automata from timed logical formalisms. Bauer et al. propose an approach to runtime verify timed-bounded properties expressed in a variant of Timed Linear Temporal Logic [4]. Contrarily to TLTL, the considered logic, TLTL_3, processes finite timed words and the truth-values of this logic are suitable for monitoring. After reading some timed word u, the monitor synthesized for a TLTL_3 formula φ state the verdict $\top$ (resp. $\perp$) when there is no infinite timed continuation w such that $u \cdot w$ satisfy (resp. does not satisfy) φ. Another variant of LTL in a timed context is the metric temporal logic (MTL), a dense extension of LTL. Nickovic et al. [3, 14] proposed a translation of MTL to timed automata. The translation is defined under the bounded variability assumption stating that, in a finite interval, a bounded number of events can arrive to the monitor. Still for MTL, Thati et al. propose an online monitoring algorithm by rewriting of the monitored formula and study its complexity [1]. Later, Basin et al. propose an improvement of this approach having a better complexity but considering only the past fragment of MTL [5].

Runtime enforcement of timed properties as presented in this paper is compatible with the previously described approaches. These approaches synthesize automata-based

decision procedures for logical formalisms. Decision procedures synthesized for safety and co-safety properties could be used as input to our framework.

Tools for runtime monitoring of timed properties. The Analog Monitoring Tool [10] is a tool for monitoring specifications over continuous signals. The input logic of AMT is STL/PSL where continuous signals are abstracted into propositions and operations are defined over signals. Input signal traces can be monitored in an offline or incremental fashion (i.e., online monitoring with periodic trace accumulation).

LARVA [11, 15] takes as input properties expressed in several notations, e.g., Lustre, duration calculus. Properties are translated to DATE (Dynamic Automata with Timers and Events) which basically resemble timed automata with stop watches but also feature resets, pauses, and can be composed into networks. Transitions are augmented with code that modify the internal system state. DATE target only safety properties. In addition, LARVA is able to compute an upper-bound on the overhead induced on the target system. The authors also identify a subset of duration calculus, called counter-examples traces, where properties are insensitive to monitoring [16].

Our monitors not only differ by their objectives but also by how they are interfaced with the system. We propose a less restrictive framework where monitors asynchronously read the outputs of the target system. We do not assume our monitors to be able to modify the internal state of the target program. The objective of our monitors is rather to correct the timed sequence of output events before this sequence is released to the environment (i.e., outside the system augmented with a monitor).

9 Conclusion and Future Work

This paper introduces runtime enforcement for timed properties and provides a complete framework. We consider safety and co-safety properties described by timed automata. We propose adapted notions of enforcement monitors with the possibility to delay some input actions in order to satisfy the required property. For this purpose, the enforcement monitor can store some actions during a certain time period. We propose a set of enforcement rules ensuring that outputs not only satisfy the required property (if possible), but also with the "best" delay according to the current situation. We describe how to realize the enforcement monitor using concurrent processes, how it has been prototyped and experimented. This paper introduced the first steps to runtime enforcement of (continuous) timed properties. However, several research questions remain open. As this approach targets explicitly safety and co-safety properties, it seems desirable to investigate whether more expressive properties can be enforced, and if so, propose enforcement mechanisms for them. We expect to extend our approach to Boolean combinations of timed safety and co-safety properties, and more general properties. The question requires further investigation since the update function would have to be adapted. A precise characterization of *enforceable timed properties* would thus be possible, as was the case in the untimed setting [4, 17]. Also related to expressiveness is the question of how the set of timed enforceable properties is impacted when the underlying memory is limited and/or the primitives operations endowed to the monitor are modified. A more practical research perspective is to study the implementability of the approach proposed in this paper, e.g., using *robustness* of timed automata.

References

1. Thati, P., Rosu, G.: Monitoring algorithms for metric temporal logic specifications. Electr. Notes Theor. Comput. Sci. 113, 145–162 (2005)
2. Chen, F., Roşu, G.: Parametric Trace Slicing and Monitoring. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 246–261. Springer, Heidelberg (2009)
3. Ničković, D., Piterman, N.: From MTL to Deterministic Timed Automata. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 152–167. Springer, Heidelberg (2010)
4. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology 20, 14 (2011)
5. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for Monitoring Real-Time Properties. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 260–275. Springer, Heidelberg (2012)
6. Barringer, H., Falcone, Y., Havelund, K., Reger, G., Rydeheard, D.: Quantified Event Automata: Towards Expressive and Efficient Runtime Monitors. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 68–84. Springer, Heidelberg (2012)
7. Schneider, F.B.: Enforceable security policies. ACM Transactions on Information and System Security 3 (2000)
8. Ligatti, J., Bauer, L., Walker, D.: Run-time enforcement of nonsafety policies. ACM Transaction Information System Security 12 (2009)
9. Falcone, Y.: You Should Better Enforce Than Verify. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 89–105. Springer, Heidelberg (2010)
10. Nickovic, D., Maler, O.: AMT: A Property-Based Monitoring Tool for Analog Systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 304–319. Springer, Heidelberg (2007)
11. Colombo, C., Pace, G.J., Schneider, G.: LARVA — safer monitoring of real-time java programs (tool paper). In: SEFM, pp. 33–37 (2009)
12. Larsen, K., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1, 134–152 (1997)
13. Matteucci, I.: Automated synthesis of enforcing mechanisms for security properties in a timed setting. Electron. Notes Theor. Comput. Sci. 186, 101–120 (2007)
14. Maler, O., Nickovic, D., Pnueli, A.: From MITL to Timed Automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 274–289. Springer, Heidelberg (2006)
15. Colombo, C., Pace, G.J., Schneider, G.: Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 135–149. Springer, Heidelberg (2009)
16. Colombo, C., Pace, G.J., Schneider, G.: Safe Runtime Verification of Real-Time Properties. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 103–117. Springer, Heidelberg (2009)
17. Falcone, Y., Fernandez, J.C., Mounier, L.: What can you verify and enforce at runtime? STTT 14, 349–382 (2012)

Monitoring Dense-Time, Continuous-Semantics, Metric Temporal Logic

Kevin Baldor[1,2] and Jianwei Niu[1]

[1] University of Texas at San Antonio, USA
{kbaldor,niu}@cs.utsa.edu
[2] Southwest Research Institute, San Antonio, USA

Abstract. The continuous semantics and dense time model most closely model the intuitive meaning of properties specified in metric temporal logic (MTL). To date, monitoring algorithms for MTL with dense time and continuous semantics lacked the simplicity the standard algorithms for discrete time and pointwise semantics. In this paper, we present a novel, transition-based, representation of dense-time boolean signals that lends itself to the construction of efficient monitors for safety properties defined in metric temporal logic with continuous semantics. Using this representation, we present a simple lookup-table-based algorithm for monitoring formulas consisting of arbitrarily nested MTL operators. We examine computational and space complexity of this monitoring algorithm for the past-only, restricted-future, and unrestricted-future temporal operators.

1 Introduction

Program monitoring has attracted interest as an alternative to model checking or theorem proving as these become impractical due to the size of the state space of a full program. A dynamic analysis, monitoring is limited to the detection of property violations that are actually observed in a program's execution, and more fundamentally, to so called *safety properties*, those that can be falsified given only a finite number of program events.

Temporal logics such as linear temporal logic (LTL) [7] and computation tree logic (CTL) [3] provide an effective formal description for desired or undesired program behavior and are commonly used in monitoring applications. Each of these logics specify constraints on the order of occurrence of events. For example, they can state that "after event p, event q must take place at some point in the future". This is not an enforceable safety property but would become one if modified to state that q must occur within a certain period of time. To do so, these logics must be augmented with an explicit notion of time.

One such augmentation is metric temporal logic (MTL) [5]. It introduces limits on the periods of time over which a logical connective operates. For example, the property described in the preceding paragraph may be specified as $p \rightarrow \Diamond_{[0,5]}\ q$. The subscript on the *eventually* operator ($\Diamond$), is an interval – relative to the

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 245–259, 2013.
© Springer-Verlag Berlin Heidelberg 2013

current time – in which q must hold in order for the statement to be true at the current time. Some notations support a number of subscript forms, but without loss of generality, we restrict our presentation to the use of intervals that may be closed or open on either end. Additionally, we admit intervals of the form $[a, a]$, though their use incurs a potential space penalty.

The runtime-verification community employs two time models for MTL: discrete and dense. Within the dense-time model, there are two semantics: point-based and continuous [8] [2]. We concentrate on the latter, as in [2], Basin et al. assert that "Real-time logics based on a dense, interval-based time model are more natural and general than their counterparts based on a discrete or point-based model". But in it, they present a monitoring algorithm that they describe as "conceptually simpler" for the point-based semantics than for the interval-based (continuous) semantics.

Our contribution with this paper is the introduction of a transition-based – rather than interval-based [2] – representation for the dense-time boolean signals that are a feature of the continuous semantics. With this representation, the output of all MTL connectives can be expressed as a simple lookup-table indexed by the input. Using this representation, we present a conceptually simple MTL monitoring algorithm modeled on the transducer-approach of [6] that reduces to something like the LTL-monitoring algorithm of [4] for past-only operators. We then observe the increase in space complexity of its extension to future MTL expressions.

2 Background

2.1 LTL and MTL

Logical expressions use the connectives for disjunction ($\vee$), *logical or*; conjunction ($\wedge$), *logical and*; and negation ($\neg$) to describe the relationship between logical statements at the current time. Linear Temporal Logic (LTL) augments them with a number of logical connectives that describe the relationship between logical expressions over time.

The past-only operator *historically* ($\boxminus\phi$) indicates that the expression ϕ has been true since time zero, *once* ($\diamondminus\phi$) indicates that ϕ must have been true at some point in time since time zero, and *since* ($\phi\ \mathcal{S}\ \psi$) indicates that at some point in the past ψ must have been true and that ϕ must at least have been true at every point after that until the current time;

The future-only operator *globally* ($\Box\phi$) indicates that the expression ϕ is true now and will be true at all points in the future, *eventually* ($\Diamond\phi$) indicates that ϕ must be true at the current time or at some point in the future, and *until* ($\phi\,\mathcal{U}\ \psi$) indicates that at the current time or at some point in the future ψ must be true and that that ϕ must at least have been true at every point between the current time and that point.

The semantics of LTL operate on a trace, a countably infinite sequence of truth values of atomic elements. LTL expressions are only interpreted as having a truth value at the instants of time corresponding to the elements of the trace. Beyond the order of events, the actual time at which the events plays no role in determining the truth of the LTL expressions.

The pointwise semantics of MTL are a natural extension of the semantics of LTL in that, while they augment the order constraints of LTL with true time constraints, the truth of an expression is only defined at discrete points in time. When used for monitoring, an MTL expression might be evaluated only when an input event arrives. This is more efficient than periodically re-evaluating expressions, but can lead to counter-intuitive results. In [2], Basin et al. discuss a number of such results. Perhaps most striking is that under pointwise semantics $\diamondsuit_{[0,1]}\diamondsuit_{[0,1]}\phi$ is not logically equivalent to $\diamondsuit_{[0,2]}\phi$. This is illustrated in the case that ϕ is true at time $\tau = 0$, and the next observation of the system takes place at time $\tau = 2$, $\diamondsuit_{[0,2]}\phi$ is *true* at time $\tau = 2$, but $\diamondsuit_{[0,1]}\diamondsuit_{[0,1]}\phi$ is *false* since the observations lack the 'bridge' at time $\tau = 1$ (for which $\diamondsuit_{[0,1]}\phi$ would evaluate to *true*) needed to declare $\diamondsuit_{[0,1]}\diamondsuit_{[0,1]}\phi$ to be *true* at time $\tau = 2$.

As observed by the authors of [2], adding additional sample points can restore the equivalence of these expressions at the cost of additional computation by the monitor. In a discrete-valued-time system, this can be taken to the extreme of evaluating all expressions with each 'clock tick'. Beyond the computational cost, this cannot be extended to the dense-time representation that best models external events for which there is no shared clock.

2.2 Continuous Semantics and Boolean Signals

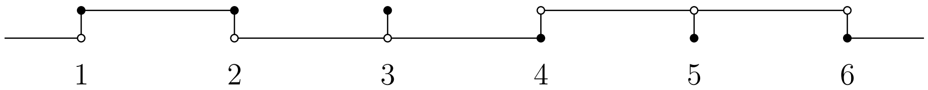
Under continuous semantics, we avoid the ambiguity introduced by the selection of sample points. Loosely, the continuous semantics (MTL) assign a truth value to any expression for any point of time greater than zero. A more formal definition is presented in [2], but we will present enough here for the purpose of discussion. Essentially, the notion of a trace used in LTL and the pointwise semantics of MTL is replaced by a mapping from time $\tau \in \mathbb{R}_{\geq 0}$ to $\{$*true*, *false*$\}$ that the authors term a boolean signal. In their formulation, the boolean signal for the expression ϕ, denoted γ_ϕ, is the set of all points in time for which ϕ evaluates to true. In Figure 1 – expanded from the definitions given in [2] to include $\mathcal{U}_I$, the set of all signals in a model is written $\hat{\gamma}$, $\tau \in \mathbb{R}_{\geq 0}$ denotes the time for which the statement applies, and the subscript I is an interval on $\mathbb{R}$ for which the operator applies.

$$
\begin{array}{lll}
\hat{\gamma}, \tau \models p & \text{iff} & \tau \in \gamma_p \\
\hat{\gamma}, \tau \models \neg\phi & \text{iff} & \hat{\gamma}, \tau \not\models \phi \\
\hat{\gamma}, \tau \models \phi \wedge \psi & \text{iff} & \hat{\gamma}, \tau \models \phi \text{ and } \hat{\gamma}, \tau \models \psi \\
\hat{\gamma}, \tau \models \phi\ \mathcal{S}_I\ \psi & \text{iff} & \exists\, \tau' \in [0,\tau] \text{ such that } \tau - \tau' \in I,\ \hat{\gamma}, \tau' \models \psi, \text{ and } \hat{\gamma}, \kappa \models \phi\ \forall \kappa \in (\tau', \tau] \text{ or}^{1} \\
 & & \exists\, \tau'' \in [0,\tau') \text{ such that } \tau - \tau'' \in I,\ \hat{\gamma}, \kappa \models \psi\ \forall \kappa \in [\tau'', \tau') \text{ and } \hat{\gamma}, \kappa \models \phi\ \forall \kappa \in [\tau', \tau] \\
\hat{\gamma}, \tau \models \phi\ \mathcal{U}_I{}^{2}\ \psi & \text{iff} & \exists\, \tau' \geq \tau \text{ such that } \tau' - \tau \in I,\ \hat{\gamma}, \tau' \models \psi, \text{ and } \hat{\gamma}, \kappa \models \phi\ \forall \kappa \in [\tau, \tau') \text{ or} \\
 & & \exists\, \tau'' > \tau' \text{ such that } \tau'' - \tau \in I,\ \hat{\gamma}, \kappa \models \psi\ \forall \kappa \in (\tau', \tau''] \text{ and } \hat{\gamma}, \kappa \models \phi\ \forall \kappa \in [\tau, \tau']
\end{array}
$$

Fig. 1. Continuous Semantics of MTL

3 Modeling Dense-Time Boolean Signals as Event Sequences

The dense model of time precludes a representation consisting of every value for which a boolean signal is true as there may be uncountably many such points. However, boolean signals are defined as satisfying the finite-variability condition that on any bounded interval there exists a finite number of non-overlapping intervals over which the signal is *true*. This lends itself to the representation used in [2] an at-most-countably-infinite set of non-overlapping intervals. Our representation differs in that we model boolean signals not as a series of intervals, but as a sequence of timed events denoting a transition from one truth value to another. For example, a signal described with the intervals $\{[1,2],[3,3],(4,5),(5,6)\}$ might be illustrated as



where the higher line indicates *true* and the lower, *false*. The dots at the transition indicate whether the signal is considered *true* at the transition point. We represent this signal as the series of transitions

$$\{(\uparrow,1),(\downarrow,2)\,(\updownarrow,3)\,(\uparrow,4),(\updownarrow,5)\,(\downarrow,6)\}$$

This example exhausts all of the transition types required to describe boolean signals. In the following sections the additional events $(_,\tau)$ and $(^-,\tau)$ are used to indicate the lack of transition on one of the inputs of a binary operator. They are not strictly required to unambiguously describe a boolean signal, but are convenient for the implementation of the monitor.

1 This clause was added to capture another boundary condition introduced by the dense time model.

2 We introduce the *until* operator using the presentation of [2] to relate the definition of future operators in the standard definitions of the runtime-verification community.

Definition 1. *A boolean signal is described by an infinite sequence of timed transitions* (δ_i, τ_i) *for which* $\delta_0 \in \{_, \overline{\ }, \uparrow^{\circ}_{\bullet}, \downarrow^{\bullet}_{\circ}\}$, $\delta_i \in \{\uparrow^{\bullet}_{\circ}, \uparrow^{\circ}_{\bullet}, \bot^{\bullet}_{\circ}, \downarrow^{\circ}_{\bullet}, \downarrow^{\bullet}_{\circ}, \top^{\circ}_{\bullet}\} \forall\ i > 0$, *and* $\tau_i \in \mathbb{R}_{\geq 0}$, $\tau_0 = 0$, *and* $\tau_{i+1} > \tau_i \forall i \geq 0$. *The timed transitions are subject to the further constraint types of adjacent transitions must agree in the sense that the incoming value of each transition must match the outgoing value of the previous transition. More formally,*

$$\delta_i \in \{\overline{\ }, \uparrow^{\bullet}_{\circ}, \uparrow^{\circ}_{\bullet}, \top^{\circ}_{\bullet}\} \rightarrow \delta_{i+1} \in \{\downarrow^{\circ}_{\bullet}, \downarrow^{\bullet}_{\circ}, \top^{\circ}_{\bullet}\} \textit{ and}$$
$$\delta_i \in \{_, \downarrow^{\circ}_{\bullet}, \downarrow^{\bullet}_{\circ}, \bot^{\bullet}_{\circ}\} \rightarrow \delta_{i+1} \in \{\uparrow^{\bullet}_{\circ}, \uparrow^{\circ}_{\bullet}, \bot^{\bullet}_{\circ}\}.$$

Definition 2. *The truth of a signal* γ *at time* τ *is given by*

$$\tau \in \gamma \doteq \begin{cases} \delta_k \in \{\overline{\ }, \uparrow^{\bullet}_{\circ}, \downarrow^{\bullet}_{\circ}, \bot^{\bullet}_{\circ}\} & \exists k : \tau_k = \tau \\ \delta_k \in \{\overline{\ }, \uparrow^{\bullet}_{\circ}, \uparrow^{\circ}_{\bullet}, \bot^{\bullet}_{\circ}\} & \exists k : \tau_k < \tau \wedge \ (\tau_{k+1} > \tau \vee k = |\gamma|) \end{cases}$$

4 Monitor Construction

4.1 Supported Temporal Operators

Although the timed *until* and *since* are sufficient to capture MTL semantics, the treatment of their transitions is sufficiently complicated that we follow the approach of [6] and introduce timed *eventually* to enable the treatment of only the non-metric *until* and *since*. This is accomplished by exploiting the fact that timed *since* and *until* are redundant given timed *historically* ($\boxminus_I$) , *once* ($\diamondminus_I$), *henceforth* ($\Box_I$), and *future* ($\Diamond_I$) since

$$\phi\ \mathcal{S}_{[a,b]}\psi \leftrightarrow \boxminus_{[0,a]}(\phi\ \mathcal{S}\ \psi) \wedge\ \diamondminus_{[a,b]}\psi \text{ and}$$
$$\phi\ \mathcal{U}_{[a,b]}\psi \leftrightarrow \Box_{[0,a]}(\phi\ \mathcal{U}\ \psi) \wedge\ \Diamond_{[a,b]}\psi$$

and that $\boxminus_I$ and $\Box_I$ are redundant since

$$\boxminus_I \phi \leftrightarrow \neg \diamondminus_I \neg \phi$$
$$\Box_I \phi \leftrightarrow \neg \Diamond_I \neg \phi$$

Further, we observe that for the same input the output of $\Diamond_{[a,a+\Delta]}$ and $\diamondminus_{[b,b+\Delta]}$ are both simply a time-shifted version of the output of $\Diamond_{[0,\Delta]}$. By generalizing the intervals to support negative indices, we obtain

$$\diamondminus_{[a,b]}\phi \leftrightarrow \Diamond_{[-b,-a]}\phi$$

As a result, we can monitor both future and past MTL using only the transducers for the operators $\neg$, $\wedge$, $\mathcal{S}$, $\mathcal{U}$, and $\Diamond_I$, the formal definitions for which are given in Figure 2.

$$\begin{array}{lll}
\hat{\gamma},\tau \models \neg\phi & \text{iff} & \hat{\gamma},\tau \not\models \phi \\
\hat{\gamma},\tau \models \phi \wedge \psi & \text{iff} & \hat{\gamma},\tau \models \phi \text{ and } \hat{\gamma},\tau \models \psi \\
\hat{\gamma},\tau \models \Diamond_I\, \phi & \text{iff} & \exists\, \tau' \text{such that } \tau' - \tau \in\ I,\ \hat{\gamma},\tau' \models\ \phi \\
\hat{\gamma},\tau \models \phi\ \mathcal{S}\ \psi & \text{iff} & \exists\, \tau' \in\ [0,\tau] \text{ such that } \hat{\gamma},\tau' \models \psi \text{ and } \hat{\gamma},\kappa \models \phi\ \forall\kappa \in (\tau',\tau] \text{ or} \\
 & & \quad \exists\, \tau'' \in\ [0,\tau') \text{ such that } \hat{\gamma},\kappa \models \psi\ \forall\kappa \in [\tau'',\tau') \text{ and } \hat{\gamma},\kappa \models \phi\ \forall\kappa \in [\tau',\tau] \\
\hat{\gamma},\tau \models \phi\ \mathcal{U}\ \psi & \text{iff} & \exists\, \tau' \geq \tau \text{ such that } \hat{\gamma},\tau' \models \psi \text{ and} \hat{\gamma},\kappa \models \phi\ \forall\kappa \in (\tau',\tau] \text{ or} \\
 & & \quad \exists\, \tau'' > \tau' \text{ such that } \hat{\gamma},\kappa \models \psi\ \forall\kappa \in (\tau',\tau''] \text{ and } \hat{\gamma},\kappa \models \phi\ \forall\kappa \in [\tau,\tau']
\end{array}$$

Fig. 2. Semantics of Monitored MTL Connectives

4.2 Monitoring Algorithm

To monitor a formula ϕ, we begin by converting it into a parse tree. From this, we construct an array Φ consisting of one transducer for each node of the parse tree in reverse-topological-sort order. That is, for any node in the parse tree, its children appear before it in Φ. The transducers maintain some operation-specific fields, but each contains at least $\langle op, inputs, Q, I_{\text{valid}}\rangle$ where *op* identifies the operation, *inputs* contains a pointer to the elements of Φ upon which it depends, Q is a queue containing the output of the transducer, and I_{valid} indicates the time interval over which the output of the transducer is valid.

The valid interval allows a transducer to 'stop time' while its state is undetermined. For example, the transducer for *eventually* uses this to apply a constant offset to all output transitions, whereas the *until* transducer may delay its output for an indeterminate period. Even the simple transducers such as negation and conjunction must be able to specify a valid interval since their input might do so.

Some transducers define additional state variables in addition to those mentioned above. The *since* transducer maintains a state indicating whether or not the latest transition left its output in the UP state; The *eventually* transducer maintains a timer that is used by the monitoring algorithm to call UPDATE again at some point in the future.

In the following pseudocode, the UPDATE procedures for each transducer employs transition tables such as $\text{FUTURE}_{\text{a}}[\delta]$ for the *eventually* transducer. Their contents are given in the following sections.

The monitoring procedure consists of gathering an ensemble of simultaneous transition events for the external inputs to the monitor and storing them in a container, Δ, that maps input variable names to transition types. If a timer expires, UPDATE may be called with no input transitions. The UPDATE function returns the next timer expiration time so that the monitor may call it when it has expired. The following pseudocode describes the most general version of the

update operation and the UPDATE procedures for the more interesting transducers[3]. Subsequent sections will describe the simplifications that are possible when supporting subsets of MTL.

function UPDATE(Φ, Δ, τ)
  for $\varphi \in \Phi$ **do**
    if φ.op $\in$ *input variables* **then**
      if φ.op.id $\in \Delta$ **then**
        ENQUEUE(φ_Q,$\Delta[\varphi$.op.id$]$, τ)
        $\varphi.I_{\text{valid}} \leftarrow [0, \tau]$
    else
      while $(\delta^*, \tau) \leftarrow$ SYNC(φ.inputs) **do**
        UPDATE$_{\varphi.\text{op}}(\varphi, \delta^*, \tau)$
    UPDATEVALIDINTERVAL$_{\varphi.\text{op}}(\varphi)$
  return $\min(\{\varphi.\tau_{\text{timer}}$ for $\varphi \in \Phi\})$

function UPDATE$_{\diamond_I}(\varphi, \delta, \tau)$
  $\tau' \leftarrow \tau - b$ ▷ output time offset
  if $\tau = 0$ **then**
    ENQUEUE($\varphi.Q$, FUTURE$_{\text{INIT}}[\delta], \tau'$)
  else if $\tau = \varphi.\tau_{\text{timer}}$ **then**
    if $\delta = \varnothing$ **then**
      ENQUEUE($\varphi.Q$, $\varphi.\delta_\downarrow, \tau'$)
    else if $\varphi.\delta_\downarrow =$ ↲ **then**
      ENQUEUE($\varphi.Q$, FUTURE$_{\text{c}}[\delta], \tau'$)
    $\varphi.\delta_\downarrow \leftarrow \varnothing$
    $\varphi.\tau_{\text{timer}} \leftarrow \varnothing$
  else if $\delta \in$ FUTURE$_{\text{a}}$ **then**
    ENQUEUE($\varphi.Q$, FUTURE$_{\text{a}}[\delta], \tau'$)
  if $\delta \in$ FUTURE$_{\text{b}}$ **then** ▷ down transition
    $\varphi.\delta_\downarrow \leftarrow$ FUTURE$_{\text{b}}[\delta]$
    $\varphi.\tau_{\text{timer}} \leftarrow \tau + b - a$

function UPDATEVALIDINTERVAL$_{\diamond_I}(\varphi)$
  $\phi \leftarrow \varphi$.inputs
  switch $\phi.I_{\text{valid}}$
    case $[0, i]$ $\varphi.I_{\text{valid}} \leftarrow [0, i - b]$
    case $[0, i)$ $\varphi.I_{\text{valid}} \leftarrow [0, i - b)$

function UPDATE$_{\mathcal{U}}(\varphi, \delta_\phi, \delta_\psi, \tau)$
  if $\tau = 0$ **then**
    ENQUEUE($\varphi.Q$, UNTIL$_{\text{INIT}_\text{a}}[\delta_\phi, \delta_\psi], \tau$)
    $\varphi.\delta_\uparrow \leftarrow$ UNTIL$_{\text{INIT}_\text{b}}[\delta_\phi, \delta_\psi]$
    $\varphi.\delta_\downarrow \leftarrow$ UNTIL$_{\text{INIT}_\text{c}}[\delta_\phi, \delta_\psi]$
    $\varphi.\tau_{\text{pending}} \leftarrow \tau$
  else
    switch UNTIL$_{\text{a}}[\delta_\phi, \delta_\psi]$
      $\tau' \leftarrow \varphi.\tau_{\text{pending}}$
      case ↱
        ENQUEUE($\varphi.Q$, $\varphi.\delta_\uparrow$, τ')
      case ↲
        ENQUEUE($\varphi.Q$, $\varphi.\delta_\downarrow$, τ')
    ENQUEUE($\varphi.Q$, UNTIL$_{\text{b}}[\delta_\phi, \delta_\psi], \tau$)
    $\varphi.\delta_\uparrow \leftarrow$ UNTIL$_{\text{c}}[\delta_\phi, \delta_\psi]$
    $\varphi.\delta_\downarrow \leftarrow$ UNTIL$_{\text{INIT}_\text{d}}[\delta_\phi, \delta_\psi]$
    $\varphi.\tau_{\text{pending}} \leftarrow \tau$
  if $\neg(\varphi.\delta_\uparrow = \varphi.\delta_\downarrow = \varnothing)$ **then**
    $\varphi.I_{\text{valid}} \leftarrow [0, \tau)$

function UPDATEVALIDINTERVAL$_{\mathcal{U}}(\varphi)$
  $\phi, \psi \leftarrow \varphi$.inputs
  if $\varphi.\delta_\uparrow = \varphi.\delta_\downarrow = \varnothing$ **then**
    $\varphi.I_{\text{valid}} \leftarrow \phi.I_{\text{valid}} \cap \psi.I_{\text{valid}}$

[3] For the complete pseudocode, see the full version of this paper [1]

```
function Enqueue(Q, δ, τ)
    if δ = ∅ then return
    case τ < 0
        Clear(Q)
        if δ ∈ {↑, ↑, ↑} then
            Append(Q,(‾,0))
        else
            Append(Q(_,0))
    case τ = 0
        Clear(Q)
        case δ = ↑
            Append(Q(‾,0))
        case δ ∈ {↑, ↑}
            Append(Q(↑,0))
        case δ ∈ {↓, ↓}
            Append(Q(↓,0))
        case δ = ↓
            Append(Q(_,0))
    case τ > 0
        if Empty(Q) ∧ Q.value = ∅ then
            if δ ∈ {_, ‾} then
                Q.append((δ, 0))
                return
            else if δ ∈ {↑, ↑, ↑} then
                Q.append((_, 0))
            else
                Q.append((‾, 0))
        else
            Q.append((δ, τ))
```

```
function Dequeue(Q)
    (δ, τ) ← RemoveFirst(Q)
    case δ ∈ {‾, ↑, ↑, ↑}
        Q.value ← ‾
    case δ ∈ {_, ↓, ↓, ↓}
        Q.value ← _
    return (δ, τ)
```

```
function Sync(inputs)
    φ, ψ ← inputs
    if ψ = ∅ then                                         ▷ one input
        if Empty(φ.Q) then
            return ∅
        return Dequeue(φ.Q)
    else                                                  ▷ two inputs
        I_valid ← φ.I_valid ∩ ψ.I_valid
        e_φ ← Head(φ.Q) if Head(φ.Q).τ ∈ I_valid
        e_ψ ← Head(ψ.Q) if Head(ψ.Q).τ ∈ I_valid
        if e_φ ≠ ∅ ∧ e_ψ ≠ ∅ then
            if e_φ.τ = e_ψ.τ then
                Dequeue(φ.Q)
                Dequeue(ψ.Q)
                return ((e_φ.δ, e_ψ.δ), e_φ.τ)
            if e_φ.τ < e_ψ.τ then
                e_ψ ← ∅
            else
                e_φ ← ∅
        if e_φ ≠ ∅ then
            Dequeue(φ.Q)
            return ((e_φ.δ, ψ.Q.value), e_φ.τ)
        else
            Dequeue(ψ.Q)
            return ((φ.Q.value, e_ψ.δ), e_φ.τ)
```

Missing from this elided version of the pseudocode are the Update procedures for negation, conjunction, and the *since* operator. They are simpler than $\Diamond_I$ and $\mathcal{U}$ and the general sense of their operation is given in the sections describing their transducer tables.

The functions on the second page support the Update procedures and simplify their logic. For example, in addition to its obvious purpose, Enqueue ensures that the outputs obey the transition rules described in definition 1. Dequeue ensures that the the value of a boolean signal can be obtained at times for which there is no transition event per definition 2. Using this behavior of the Dequeue procedure, the Sync procedure produces from two Boolean signals an ordered stream of events for all time points at which either signal exhibits a transition event. This is needed to drive the transducer tables for those operators that have two inputs.

5 Signal Transducer Tables

5.1 Negation and Conjunction

Figure 3 contains the transition tables for negation and conjunction. The INIT versions are used for the first transition and the regular version for all subsequent transitions. The non-transition states of the inputs are illustrated, but table entries for which there is no transition are omitted in the interest of readability. Note that for the INIT versions of the tables, the non-transition outputs (‾ and _) are shown because they are actually produced for the initial entry of a boolean signal.

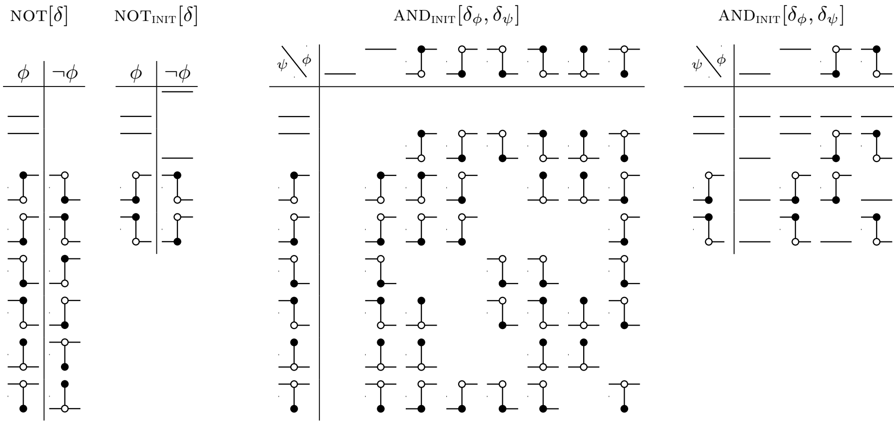


Fig. 3. Transition Tables for the Negation and Conjunction Operators

5.2 Since

The transducer for $\phi \mathcal{S} \psi$ is somewhat more complicated. The output transition for a given set of input transitions is influenced by whether or not the since operator is currently in the down or the up state. As in the previous section, omitted entries indicate states for which there is no transition. Some of the omitted entries indicate states that cannot be reached, but these are not specially indicated – nor is any special handling required to deal with unreachable states.

The transducer begins by applying the table SINCE$_{\text{INIT}}[\delta_\phi, \delta_\psi]$ for the initial transition, afterward the tables SINCE$_{\text{UP}}[\delta_\phi, \delta_\psi]$ and SINCE$_{\text{DOWN}}[\delta_\phi, \delta_\psi]$ are used. Which of the tables is to be used is determined by the type of transition last emitted. If it is in {‾, ⌶, ⌶, ⌶}, then the UP table is used, otherwise, it is the DOWN table.

5.3 Eventually

The metric eventually operator uses the tables from Figure 5. Its transducer is distinguished from those introduced thus far by the addition of a timer used

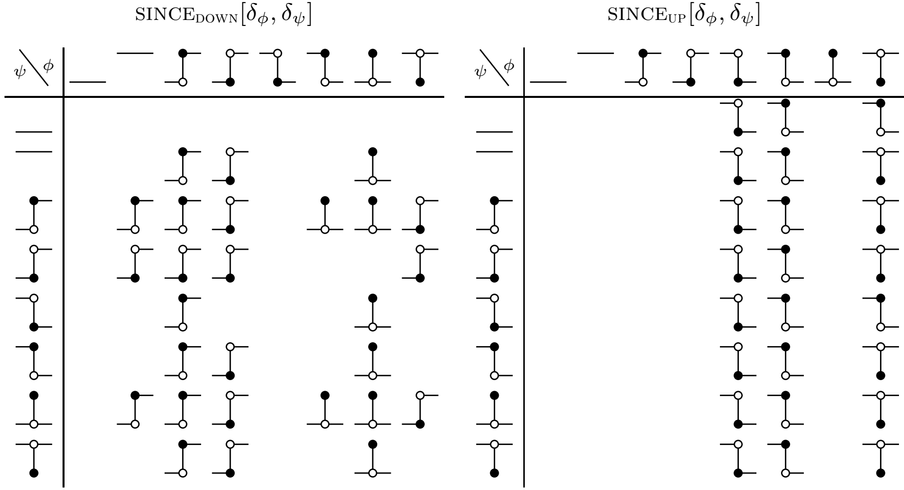


Fig. 4. Transition Tables for the *Since* Operator, $\phi\ \mathcal{S}\ \psi$

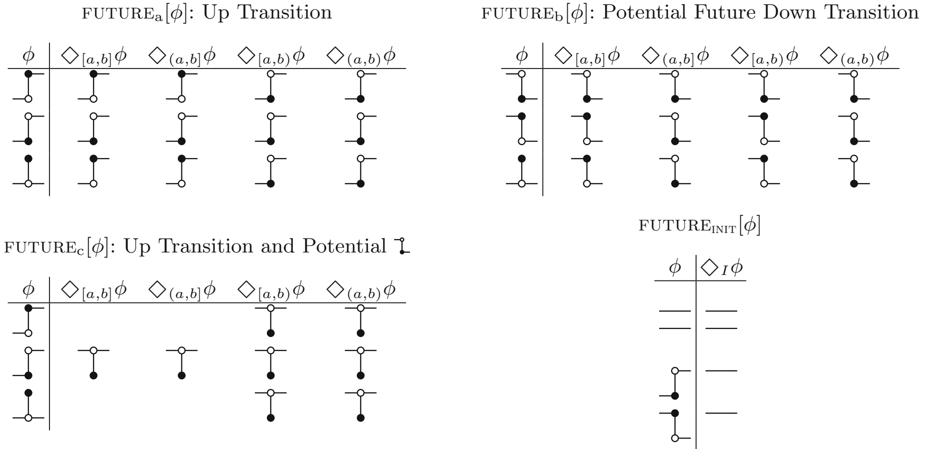


Fig. 5. Transition Tables for the *Eventually* Operator

to generate a down event $b - a$ time units after the input transitions to the down state. All events emitted in response to an event at time τ are emitted at time $\tau - b$. Also, it can enter an indeterminate state in response to a down transition. When a down transition occurs, the transducer stores the potential down transition – the type of which is determined by the interval type – in the state variable $\delta_\downarrow$. The actual transition emitted can be affected if an up transition occurs before or simultaneous with the timer expiration.

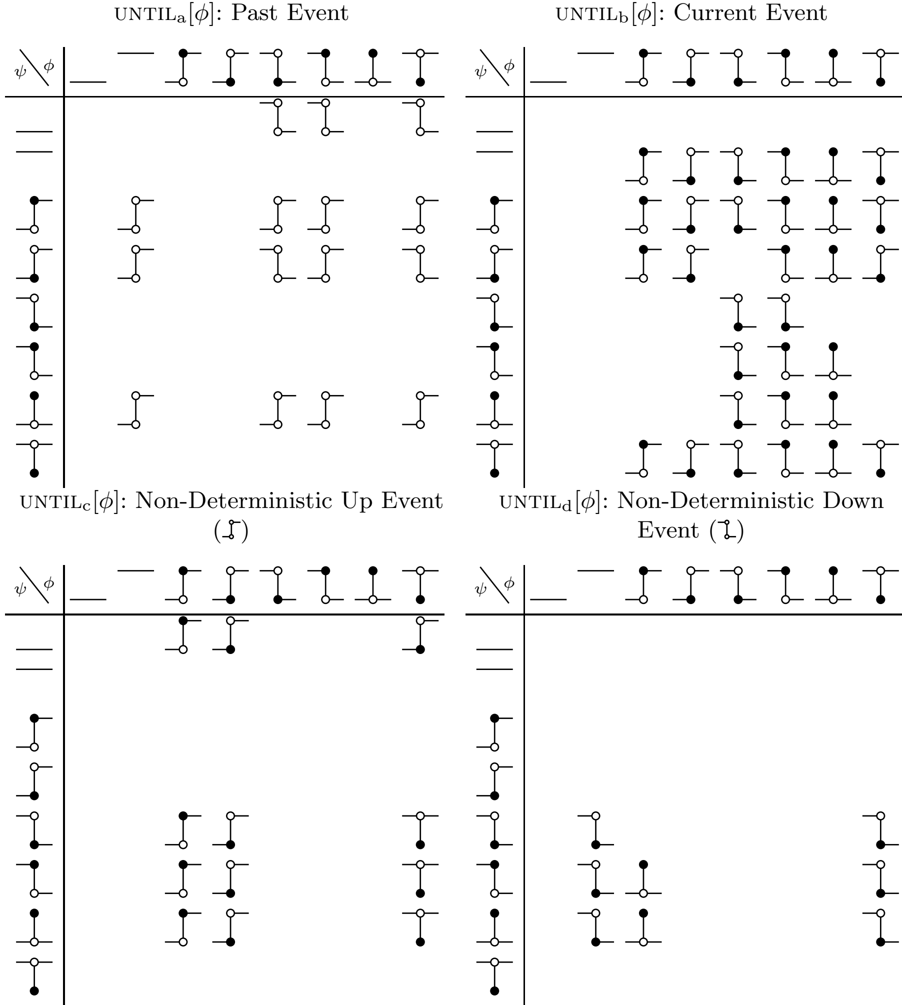


Fig. 6. Transition Tables for the *Until* Operator, $\phi\,\mathcal{U}\,\psi$

5.4 Until

The transducer for the *until* operator can enter a more complicated indeterminate state than that of the non-metric *eventually* operator; It can maintain a different potential transition depending on whether it is ultimately found to be *true* or *false* at the point in time at which its output became uncertain. The simplest example occurs when monitoring $\phi\,\mathcal{U}\,\psi$ and ϕ becomes *true* with transition type ⌐ at a time τ when ψ is *false*. If ψ becomes *true* before ϕ becomes *false*, then the transducer should emit the transition event (⌐, τ); If ϕ becomes *true* before ψ becomes *false*, this potential transition is abandoned and the output remains *false* up to and including the current time. It is possible that the

transducer will maintain two such potential transitions, $\delta_\downarrow$ or $\delta_\uparrow$. We introduce the notations *non-deterministic up* ↥ and *non-deterministic down* ↧ in UNTIL_a to denote which of the potential transitions is to be emitted upon the arrival of input events.

6 Correctness

Theorem 1. [4] *The* UPDATE *procedure applied to the above transducer tables correctly models the semantics of Figure 1.*

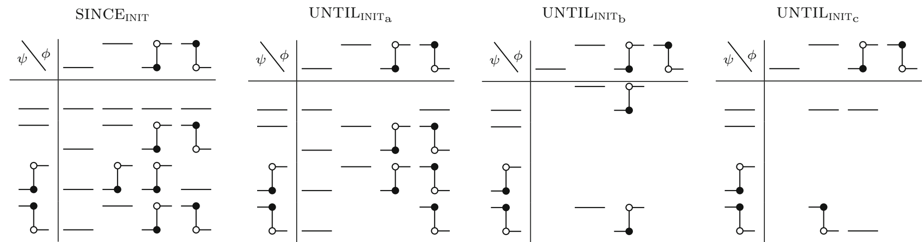


Fig. 7. Initialization Tables for the *Since* and *Until* Operators

7 Monitoring Algorithm Complexity

7.1 Instantaneous Transducers

The instantaneous transducers are defined as those for which all emitted transitions take place at the current time, that is, with the same τ as that of the event that caused them. They comprise $\neg$, $\wedge$, $\mathcal{S}$, and $\Diamond_{[-a,0]}$.

The UPDATE procedure can be simplified in that there is no need to keep track of the *valid* intervals. Without the potential for delayed output, the queues will never grow larger than one element and can be replaced with single values.

Theorem 2. *The runtime to monitor expression ϕ that consists of only instantaneous operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for n = the sum of the number of transitions on all inputs.*

Proof. The proof is provided in the full version of this paper [1], but is reasonably clear from the pseudocode if it is given that ENQUEUE, DEQUEUE, and SYNC run in constant time.

Theorem 3. *The space required to monitor expression ϕ that consists of only instantaneous operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|)$, for n = the sum of the number of transitions on all inputs.*

[4] The proof of the theorems in the following sections are provided in the full version of this paper [1].

Proof. The proof is provided in the full version of this paper [1] and centers on a proof that the queues for the instantaneous transducers do not grow larger than one element.

7.2 Strictly Past Transducer

The transducer for the operator $\Diamond_{[a,b]}$ for $a <= b < 0$ operates identically to that of $\Diamond_{[b-a,0]}$ except that the timestamp of the output that results from an event at time τ is $\tau - b$. The corresponding statement is true for other intervals with open bounds as well. This introduces the need to maintain valid ranges and queues to store the output of the intermediate stages to support operators with multiple inputs.

Theorem 4. *The runtime to monitor expression ϕ that consists of only past-time operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for $n =$ the sum of the number of transitions on all inputs.*

Theorem 5. *The space required to monitor expression ϕ that consists of only past-time operators with $\Diamond_{[a,b]}$ where $a < b < 0$ on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi| \left\lfloor \frac{a}{b-a} \right\rfloor)$, for $n =$ the sum of the number of transitions on all inputs.*

Theorem 6. *The space required to monitor expression ϕ that consists of only past-time operators with $\Diamond_{[a,a]}$ where $a < 0$ on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for $n =$ the sum of the number of transitions on all inputs.*

7.3 Restricted Future

The next increment in monitor complexity introduces the metric eventually operator $\Diamond_{[a,b]}$ with $b > 0$. From the UPDATE procedure, we see that it introduces a delay in its output events relative to the input events that produces them. This adds no computational complexity, but reduces the guarantees that can be made about space complexity even when $a \neq b$.

Theorem 7. *The runtime to monitor expression ϕ that consists of past-time and restricted-future operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for $n =$ the sum of the number of transitions on all inputs.*

Theorem 8. *The space required to monitor expression ϕ that consists of only past-time operators with $\Diamond_{[a,b]}$ where $a <= b$ and $b > 0$ on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for $n =$ the sum of the number of transitions on all inputs.*

7.4 Unrestricted Future

The introduction of the *until* operator presents two challenges related to the fact that it may remain in an indeterminate state for an arbitrary length of time. The unchanged space complexity belies the fact that whereas a bound may be placed on the growth of the size of the monitor for restricted-future operators if a limit can be placed on the number of transitions within any time interval, no such limit can be placed on the size of the monitor for unrestricted-future operators. Also, it is possible to construct liveness properties that can not be falsified by a monitoring procedure.

Theorem 9. *The runtime to monitor expression ϕ that consists of past-time and restricted-future operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for n = the sum of the number of transitions on all inputs.*

Theorem 10. *The space required to monitor expression ϕ that consists of only past-time operators with $\Diamond_{[a,b]}$ where $a <= b$ and $b > 0$ on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for n = the sum of the number of transitions on all inputs.*

8 Conclusion

We have presented straightforward procedures for monitoring dense-time continuous-semantics MTL formulae as well as the tradeoffs in runtime and space complexity incurred as the expressiveness of the supported formulae increases.

We have included the unrestricted-future operators to demonstrate support for full MTL but also because policy writers may find them to be the most natural way of representing the policy that they wish to enforce. That said, they must be used with care as they introduce the ability to describe pure liveness properties for which no truth value will ever be determined, such as $\neg\Diamond\neg\Diamond\phi$.

Future work may include mechanisms for trimming the boolean signals of subexpressions that cannot affect the truth of the full monitored expression as well as the augmentation to the unrestricted-future monitoring algorithm to support the extension of the valid-interval for binary operators in cases for which its output value can be determined based on only the one of its inputs for which the valid-interval extends further in time. For example, a conjunction for which one of its input is unknown beyond τ), but the other is known to be *false* over the entire interval $[\tau,$ *current time*$)$.

More immediately, we intend to pursue a VHDL implementation of the subset of MTL for which size restrictions can be guaranteed.

Acknowlegements. Jianwei Niu is supported in part by NSF award CNS-0964710 and the UTSA research award TRAC-2008.

References

1. Baldor, K., Niu, J.: Monitoring metric temporal logic with continuous semantics. Technical Report CS-TR-2012-11, UTSA (2012)
2. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for Monitoring Real-Time Properties. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 260–275. Springer, Heidelberg (2012)
3. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)
4. Havelund, K., Roşu, G.: Synthesizing Monitors for Safety Properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
5. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2, 255–299 (1990), doi:10.1007/BF01995674
6. Maler, O., Nickovic, D., Pnueli, A.: From MITL to Timed Automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 274–289. Springer, Heidelberg (2006)
7. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57 (1977)
8. Prabhakar, P., D'Souza, D.: On the Expressiveness of MTL with Past Operators. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 322–336. Springer, Heidelberg (2006)

Rewrite-Based Statistical Model Checking of WMTL*

Peter Bulychev[1], Alexandre David[1], Kim G. Larsen[1], Axel Legay[2], Guangyuan Li[3], and Danny Bøgsted Poulsen[1]

[1] Aalborg University, Denmark
[2] INRIA Rennes – Bretagne Atlantique, France
[3] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, P.R. of China

Abstract. We present a new technique for verifying Weighted Metric Temporal Logic (WMTL) properties of Weighted Timed Automata. Our approach relies on Statistical Model Checking combined with a new monitoring algorithm based on rewriting rules. Contrary to existing monitoring approaches for WMTL ours is exact. The technique has been implemented in the statistical model checking engine of UPPAAL and experiments indicate that the technique performs faster than existing approaches and leads to more accurate results.

1 Introduction

Runtime verification (RV) [11,1] is an emerging paradigm used to design a series of techniques whose main objective is to instrument the specification of a system (code, ...) in order to prove/disprove potentially complex properties at the execution level. Over the last years, RV has received a lot of interest and has been implemented in several toolsets. Such tools have been successfully applied on several real-life case studies.

The main problem with RV is that, contrary to classical verification techniques, it does not permit to assess the overall correctness of the entire system. *Statistical model checking* (SMC) [4,19,17] extends runtime verification capabilities by exploiting statistical algorithms to get evidence that a given system satisfies some property. The core idea of the approach is to monitor several executions of the system. The results are then used together with algorithms from statistics to decide whether the system satisfies the property with a probability greater than some threshold. Statistical model checking techniques can also be used to estimate the probability that a system satisfies a given property [12]. In contrast to classical exhaustive formal verification approaches, a simulation-based solution does of course not guarantee a result with 100% confidence. However, it is possible to bound the probability of making an error. Simulation-based methods are known to be far less memory and time intensive than exhaustive ones,

* Work partially supported by the VKR Centre of Excellence MT-LAB and the Sino-Danish Basic Research Center IDEA4CPS.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 260–275, 2013.
© Springer-Verlag Berlin Heidelberg 2013

and are sometimes the only option [20]. Statistical model checking, which clearly complements RV, is widely accepted in various research areas such as software engineering, in particular for industrial applications, or even for solving problems originating from systems biology [13,10].

To get a more accurate intuition, Fig. 1 provides a schematic view of a statistical model checker and its interaction with RV procedures.

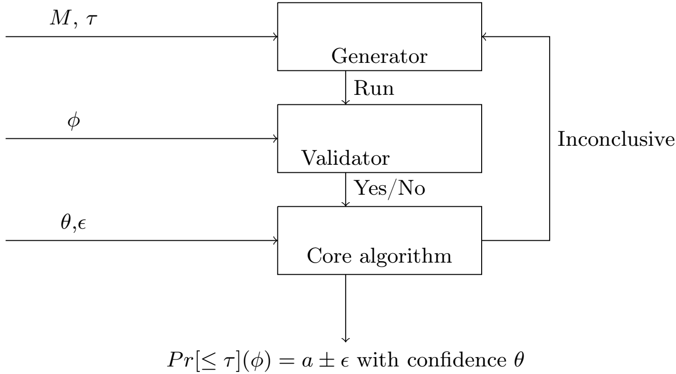


Fig. 1. A statistical model checker. The run generator first generates a run of M, which is propagated into the run validator. The run validator then validates if the run satisfies the property φ and returns Yes or No to the core algorithm. Afterwards the core algorithm decides if another run is needed or if it, based on the accumulated knowledge, can draw a conclusion.

The run generator is responsible for generating runs of the model under verification and the run validator, which corresponds to the runtime verification part of the effort, validates if a run satisfy the property or not. The core algorithm collects the simulation results until sufficient samples has been obtained to provide an overall result. The core algorithm is computationally lightweight compared to the remaining two. An optimisation of SMC is therefore most easily obtained by optimising either the run generation or the run validation. In this paper, we focus on the run validation part.

In our work, we consider combining RV and SMC techniques in order to verify complex quantitative properties (performance evaluation, scheduling, ..) over rich systems. More precisely, we are interested in computing the probability that a random run of a Weighted Timed Automata(WTA)[3] satisfies a formula written in Weighted Metric Temporal Logic (WMTL)[5]. WTA is a rich formalism capable of capturing (quantitative) non-linear hybrid systems, while WMTL corresponds to the real-time extension of the linear temporal logic equipped with cost operators. In this paper, due to the use of SMC, we assume that the scope of the temporal operators is bounded, i.e., that one can decide whether a run satisfies a formula only by looking at a finite prefix. Unfortunately, it is known that, due to the expressivity of the automata-based model, the problem of verifying

WMTL with respect to WTA is undecidable [6] – hence it cannot be tackled with existing formal techniques such as model checking. Another drawback is that it is known that, even for the case where temporal operators are bounded, WMTL is more expressive than the class of deterministic timed automata [15]. This latter result implies that there is no automata-based runtime monitoring procedure for WMTL, even for the case where the scope of the temporal operators is finitely bounded. A first solution to the above problems could be to use a three-valued logic [2]. However, the absence of decision results is often unsatisfactory from an engineering point of view, especially when dealing with performance analysis.

In [8], we proposed the first SMC-based verification procedures for the eventually and always fragments of WMTL. Our work relies on a natural stochastic semantic for WTA. The work was implemented in UPPAAL-SMC and applied to a wide range of case studies. However, our original work does not consider nested temporal operators for which a solution was first proposed by Clarke et al. in [21]. While the approach in [21] is of clear interest, it only works for a subset of MTL where the temporal operators can only be upwards bounded, i.e., the lower bound is 0. In [7], we proposed another approach that relies on monitoring automata representing over and under approximations of solutions to the WMTL formula. This approach, which has been implemented in CASAAL and UPPAAL-SMC, exploits confidence levels obtained on both approximations in order to estimate the probability to satisfy the formula. The first drawback with the approach in [7] is that both the under and over approximation depend on some precision that has an influence on the confidence level returned by the SMC algorithms. The second drawback is that automata-based monitors may be of large size, hence intractable.

In this paper, we propose a new monitoring approach for WMTL formulas. Contrary to existing approaches that work by first constructing a monitor for the property, ours exploit a graph-grammar procedure that rewrite the formula on-the-fly until a decision can be taken. The approach extends that of [16] to a timed logic. Contrary to existing off-line monitoring approaches [9], ours stops as soon as the formula is proved/disproved, which allows to save computation time and hence drastically improve both memory and time performances. Our approach has been implemented in UPPAAL-SMC and evaluated on several case studies, from random large-size formulas to concrete applications. As expected, there are many situations where we clearly outperform [7] while being more precise!

Outline. In section 2 we introduce our modelling formalism Networks of Weighted Timed Automata. Later in section 3 we define the WMTL logic, and section 4 describes our rewrite-based algorithm for monitoring of WMTL properties. The experiments are described in section 5.

2 Networks of Priced Timed Automata

In this paper, we briefly recap the formalism of networks of Weighted Timed Automata[3].

Let $\mathcal{X}$ be a set of variables called clocks. A clock valuation over $\mathcal{X}$ is a function $v : \mathcal{X} \to \mathbb{R}$ that assigns a real-valued number to each clock. We let $V(\mathcal{X})$ denote all possible valuations over $\mathcal{X}$ and let $\overline{0}$ be the valuation that assign zero to all clocks. An *upper bound* (resp. lower bound) over $\mathcal{X}$ is of the form $x \bowtie m$ where $x \in \mathcal{X}$, $m \in \mathbb{N}$, and $\bowtie \in \{<, \leq\}$ (resp. $\bowtie \in \{>, \geq\}$). We denote by $B^{\leq}(\mathcal{X})$ (resp. $B^{\geq}(\mathcal{X})$) the set of upper bounds (resp. lower bounds) over X. We let $B(\mathcal{X}) = B^{\leq}(\mathcal{X}) \cup B^{\geq}(\mathcal{X})$. Let v be a valuation over $\mathcal{X}$ and let $g \subseteq B(\mathcal{X})$ then we write $v \models g$ if for all $(x \bowtie m) \in g$, $v(x) \bowtie m$. For a valuation $v \in V(\mathcal{X})$, a function $r : \mathcal{X} \to \mathbb{Q}$ and a $\tau \in \mathbb{R}$ we let $v + r \cdot \tau$ be the valuation over $\mathcal{X}$ such that $(v + r \cdot \tau)(x) = v(x) + r(x) \cdot \tau$ for every clock $x \in \mathcal{X}$. Let $\mathcal{X}_2 \subseteq \mathcal{X}$ then $v[\mathcal{X}_2 = 0]$ is the valuation that assigns zero to every clock in $\mathcal{X}_2$ and agress with v on all other clocks. For two valuations v_1 and v_2 we let $v_2 - v_1$ be the valuation v' where $v'(x) = v_2(x) - v_1(x)$ for every clock $x \in \mathcal{X}$.

Definition 1. *A Weighted Timed Automaton over the finite set of actions Σ and the set of propositions $\mathcal{P}$ is a tuple $(\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{X}_O^i, E, \mathcal{I}, \mathcal{R}, \mathcal{X}^{\mathcal{R}}, P)$, where*

- *$\mathcal{L}$ is a finite set of locations,*
- *$\ell_0 \in \mathcal{L}$ is the initial location,*
- *$\mathcal{X}$ is a finite set of clocks,*
- *$\mathcal{X}_O \subseteq \mathcal{X}$ is a finite set of observable clocks*
- *$E \subseteq \mathcal{L} \times 2^{B^{\geq}(\mathcal{X})} \times \Sigma \times \mathcal{X} \times \mathcal{L}$ is a set of edges,*
- *$\mathcal{I} : \mathcal{L} \to 2^{B^{\leq}(\mathcal{X})}$ assigns invariants to the locations,*
- *$\mathcal{R} : \mathcal{L} \to \mathbb{Q}$ assigns transition rates to locations,*
- *$\mathcal{X}^{\mathcal{R}} : \mathcal{L} \to \mathcal{X} \to \mathbb{Q}$ assign rates to the clocks of the WTA and*
- *$P : \mathcal{L} \to 2^{\mathcal{P}}$ assign propositions to the locations of the WTA.*

The semantics of a WTA $A = (\mathcal{L}, \ell_0, \mathcal{X}, \mathcal{X}_O^i, E, \mathcal{I}, \mathcal{R}, \mathcal{X}^{\mathcal{R}}, P)$ is given as a timed transition system with state space $\mathcal{L} \times V(\mathcal{X})$ (denoted $(SP(A))$ and initial state $(\ell_0, \overline{0})$ (denoted $init(A)$). For consistency, we require $\overline{0} \models \mathcal{I}(\ell_0)$. Furthermore we require that the rates of the observable clocks in any location is greater than 0. The transition rules are given below

- *delay*: $(\ell, v) \xrightarrow{d} (\ell, v')$ where $d \in \mathbb{R}_{\geq 0}$, if $v' = v + \mathcal{X}^{\mathcal{R}}(\ell) \cdot d$ and $v' \models \mathcal{I}(\ell)$
- *discrete transition*: $(\ell, v) \xrightarrow{a} (\ell', v')$ if there exists $(\ell, g, a, \mathcal{Y}, \ell') \in E$ such that $v \models g$, $v' = v[\mathcal{Y} = 0]$ and $v' \models \mathcal{I}(\ell')$.

To prepare for composition of WTAs we assume that the set of actions Σ is partitioned into a set of input actions Σ_i and output actions Σ_o. Also we assume the WTA is input-enabled for the input actions Σ_i, i.e. that for any $a \in \Sigma_i$ and any state (ℓ, v) there exists a transition $(\ell, v) \xrightarrow{a} (\ell', v')$. A WTA is deterministic for $\Sigma' \subseteq \Sigma$ if there exists at most one transition for each $a \in \Sigma$. In the paper, we let $A^i = (\mathcal{L}^i, \ell_0^i, \mathcal{X}^i, \mathcal{X}_O^i, E^i, \mathcal{I}^i, \mathcal{R}^i, \mathcal{X}^{\mathcal{R}^i}, P^i)$.

Network of WTAs. A network of WTAs (NWTA) is a set of WTAs executing in parallel. The automata communicate via *broadcast synchronisation.*

Let $A^1, A^2, \ldots, A^n$ be WTAs over the common set of actions Σ. Furthermore, let $\Sigma^1, \Sigma^2 \ldots \Sigma^n$ be mutually disjoint subsets of Σ and for all i let A^i be deterministic and input-enabled with respect to $\Sigma \setminus \Sigma^i$ and deterministic with respect to Σ^i. Then we call $N = A^1|A^2|\ldots|A^n$ a network of WTAs over Σ where Σ^i is the output actions of A^i and $\Sigma \setminus \Sigma^i$ is its input actions.

The semantics of the network of WTAs is a timed transition system with the state space $SP(N) = SP(A^1) \times SP(A^2) \times \cdots \times SP(A^n)$ and the initial state $(init(A^1), init(A^2), \ldots, init(A^n))$. We refer to an element $\boldsymbol{s} = (s_1, s_2, \ldots, s_n) \in SP(N)$ as a state vector of the network and let $\boldsymbol{s}_i = s_i$. The transition rules of a network is given as

- $(\boldsymbol{s}) \xrightarrow{d} (\boldsymbol{s}')$ if for all $i, 1 \leq i \leq n$, $\boldsymbol{s}_i \xrightarrow{d} \boldsymbol{s}'_i$, and $d \in \mathbb{R}_{\geq 0}$
- $(\boldsymbol{s}) \xrightarrow{a} (\boldsymbol{s}')$ if for all $i, 1 \leq i \leq n$ $\boldsymbol{s}_i \xrightarrow{a} \boldsymbol{s}'_i$, and $a \in \Sigma$.

Consider WTAs given in Fig. 2. WTAs (a) and (b) are competing to force (c) to go either to location *Left* or to location *Right*. Initially both competitors are waiting for between 3 and 5 time units whereafter one of them moves the (c) to either *Left* or *Right*. Afterwards both competitors have a period where time progresses and nothing occurs. Indeed, when one of the competitors returns it must wait for between 3 and 5 time units again and choose to either move (c) or let it be and enter a waiting period again. The primary difference between (a) and (b) is that (b) rushes to return to a position from which it can change (c) and (a) returns within 5 time units.

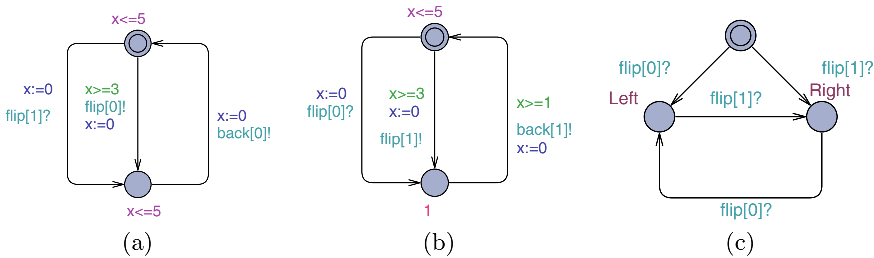


Fig. 2. Network of Timed Automata

Let $\boldsymbol{s} = ((\ell_1, v_1), (\ell_2, v_2), \ldots, (\ell_n, v_n))$ be a state vector for $A^1|A^2|\ldots|A^n$. Then we let $P(\boldsymbol{s}) = \bigcup_{i=1}^{n} P^i(\ell_i)$. Let $x \in \mathcal{X}_i$ for som i then $V(\boldsymbol{s}, x) = v_i(x)$.

Definition 2 (Run). *Let $A^1|A^2|\ldots|A^n$ be a network of WTAs. A run of the network is an infinite weighted word $(\mathcal{P}_0, v_0)(\mathcal{P}_1, v_1)\ldots$ where for all i, v_i is a valuation over $\mathcal{Y} = \bigcup_{i\in\{1,2,\ldots,n\}} \mathcal{X}^i_O$ and*

- $v_0 = \overline{0}$,
- *there exists an alternating sequence of delays and discrete transitions* $\boldsymbol{s}_0 \xrightarrow{d_0} \boldsymbol{s}'_0 \xrightarrow{a_0} \boldsymbol{s}_1 \xrightarrow{d_1} \ldots$, *where for all* i, $0 < i$, *and for all* $x \in \mathcal{Y}$ $v_i(x) = v_{i-1}(x) + (V(\boldsymbol{s}'_{i-1}, x) - V(\boldsymbol{s}_{i-1}, x))$.
- $\boldsymbol{s}_0 = (init(A^1), init(A^2), \ldots, init(A^n))$ *and*
- *for all* $j, j \geq 0, \mathcal{P}_j = P(s_j)$.

For a run $\omega = (\mathcal{P}_0, v_0)(\mathcal{P}_1, v_1)\ldots$, we let $\omega^i = (\mathcal{P}_i, v_i)(\mathcal{P}_{i+1}, v_{i+1})\ldots$. A run ω is called *diverging* for clock x if for any i there exists a j such that $v_j(x) > v_i(x)+1$. A run is diverging if it is diverging for all clocks. In what follows, we assume that there always exists a clock τ in a WTA, and this clock always have a rate of 1 and is never reset, i.e. τ measures the time length of a run.

Stochastic Semantics. In [8] we introduced the stochastic semantics for NWTAs, i.e. proposed a probability measure on the set of all runs of a network and described an algorithm for generating *a random run.* Roughly speaking, the stochastic semantics of WTA components associates probability distributions on both the delays one can spend in a given state as well as on the transition between states. In UPPAAL-SMC uniform distributions are applied for bounded delays and exponential distributions for the case where a component has unbounded delay. In a network of WTAs the components repeatedly race against each other, i.e. they independently and stochastically decide on their own how much to delay before outputting, with the "winner" being the component that chooses the minimum delay.

Statistical Model Checking. As said in the introduction, we use SMC [4,19,17] to compute the probability for a network of WTAs to satisfy a given property. Given a program B and a trace-based property[1] ϕ , SMC refers to a series of simulation-based techniques that can be used to answer two questions: (1) *qualitative:* is the probability for B to satisfy ϕ greater or equal to a certain threshold θ (or greater or equal to the probability to satisfy another property ϕ') [19]? and (2) *quantitative:* what is the probability for B to satisfy ϕ [12]? In both cases, the answer is correct up to some confidence level, i.e., probability that the algorithm does not make mistake, whose value can be configured by the user. For the quantitative approach, which we will intensively use in this paper, the method computes a confidence interval that is an interval of probabilities that contains the true probability to satisfy the property. The confidence level is interpreted as the probability for the algorithm to compute a confidence interval that indeeds contains the probability to satisfy the property.

Our UPPAAL-SMC toolset implements a wide range of SMC algorithms for WTAs. In addition, the tool offers several features to visualize and reason on the results. Until now, the monitoring procedure for WMTL relies on a technique that computes over and under approximation monitors for the formulas. In this paper, we go one big step further and propose a more efficient and precise monitoring procedure.

[1] i.e. a property with semantics defined on traces.

3 Weighted Metric Temporal Logic

In this section we review the syntax and semantics of Weighted Metric Temporal Logic (WMTL) [5]. The syntax is defined as follows.

Definition 3. *A WMTL formula over the propositions $\mathcal{P}$ and the clocks $\mathcal{X}$ is generated by the grammar:*

$$\varphi, \varphi_1, \varphi_2 ::= \top \mid \bot \mid p \mid \neg p \mid O\,\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 U^x_{[a,b]} \varphi_2 \mid \varphi_1 R^x_{[a,b]} \varphi_2$$

where $a, b \in \mathbb{Q}$, $a \leq b$, $p \in \mathcal{P}$ and $x \in \mathcal{X}$.

As one can see in the syntax, we restrict to a fragment of WMTL where temporal operators are bounded. As stated in the introduction, this fragment is sufficient to break any decidability results. Observe that WMTL is an extension of Metric Temporal Logic (MTL) [14] in which U and R can also be bounded for arbitrary clocks. As an example, bounding U and R over arbitrary clock allows one to express that a communication device should recover from a state without spending more than x units of energy. This can be accomplished by adding an observable clock, that measures the energy consumption, to the model and bound the U and R modalities over this clock.

We interpret WMTL formulas over runs of WTAs. Informally, the WMTL formula $\varphi_1 U^x_{[a,b]} \varphi_2$ is satisfied by a run if φ_1 is satisfied on the run until φ_2 is satisfied, and this should happen before the value of the clock x increases with more than b units starting from the beginning of the run, and after it increases for more than a units. Formula $O\,\varphi$ means that φ should be satisfied starting from the next observation of the run. The logical operators are defined as usual, and the *release* opearator R is dual to U, and $\varphi_1 R^x_{[a,b]} \varphi_2 \equiv \neg(\neg\varphi_1 U^x_{[a,b]} \neg\varphi_2)$.

Formally, let $\omega = (\mathcal{P}_0, v_0)(\mathcal{P}_1, v_1)\ldots$ be a timed run. The satisfaction relation is inductively defined as

- $\omega \models \top$
- $\omega \models p$ if $p \in \mathcal{P}_0$
- $\omega \models \neg p$ if $p \notin \mathcal{P}_0$
- $\omega \models O\,\varphi$ if $\omega^1 \models \varphi$
- $\omega \models \varphi_1 \vee \varphi_2$ if $\omega \models \varphi_1$ or $\omega \models \varphi_2$
- $\omega \models \varphi_1 \wedge \varphi_2$ if $\omega \models \varphi_1$ and $\omega \models \varphi_2$
- $\omega \models \varphi_1 U^x_{[a,b]} \varphi_2$ if there exists i such that $a \leq v_i(x) - v_0(x) \leq b, \omega^i \models \varphi_2$ and for all $j < i$ we have $\omega^j \models \varphi_1$
- $\omega \models \varphi_1 R^x_{[a,b]} \varphi_2$ if there exists i such that $a \leq v_i(x) - v_0(x) \leq b, \omega^i \models \varphi_1$ and for all $j \leq i, \omega^j \models \varphi_2$, or for all i such that $v_i(x) - v_0(x) \leq b$ we have $\omega^i \models \varphi_2$

In the rest of the paper, we use the following equivalences: $\Diamond^x_{[a,b]} \varphi = \top U^x_{[a,b]} \varphi$ and $\Box^x_{[a,b]} \varphi = \bot R^x_{[a,b]} \varphi$. We also use $\Box_{[a,b]} \varphi$ instead of $\Box^\tau_{[a,b]} \varphi$ for the case τ grows with rate 1.

Example 1. Consider again the WTAs in Fig. 2 and assume that the winner of the competition is the one who managed to have (c) located in its designated location for 8 consecutive time units. To express that (a) wins within 100 time units we need to state that (c) stays in *Left* for 8 consecutive time units at some point and that it has not stayed in *Right* for 8 consecutive time units before that point. Using WMTL this can be expressed like

$$(\neg Left \vee \Diamond_{[0,8]} Right) U_{[0,92]} (\Box_{[0,8]} Right).$$

We now focus on deciding a WMTL formula φ on a finite prefix of an infinite diverging run $\omega = (\mathcal{P}_0, v_0)(\mathcal{P}_1, v_1)\dots$. We first define the bound function $N(w, \varphi)$ inductively as follows:

$$\begin{aligned}
N(\omega,\top) = N(\omega,\bot) = N(\omega,p) &= 0\\
N(\omega,\neg p) &= 0\\
N(\omega,\varphi_1 \wedge \varphi_2) &= \max\{N(\omega,\varphi_1), N(\omega,\varphi_2)\}\\
N(\omega,\varphi_1 \vee \varphi_2) &= \max\{N(\omega,\varphi_1), N(\omega,\varphi_2)\}\\
N(\omega,O(\varphi)) &= 1 + N(\omega,\varphi)\\
N(\omega,\varphi_1 U^x_{[a;b]} \varphi_2) &= \max_{i.a\le v_i(x)-v_0(x)\le b}\big(\max\{i + N(\omega^i,\varphi_2),\\
&\qquad \max_{j<i}\{j + N(\omega^j,\varphi_1\}\}\big)\\
N(\omega,\varphi_1 R^x_{[a;b]} \varphi_2) &= \max_{i.a\le v_i(x)-v_0(x)\le b}\big(\max\{i+1, i + N(\omega^i,\varphi_2),\\
&\qquad \max_{j\le i}\{j + N(\omega^j,\varphi_1\}\}\big)
\end{aligned}$$

The bound function characterises the maximal prefix of ω that one needs to observe to decide φ. Observe that, contrary to [21], the bound depends not only on the formula but also on the run itself. The latter is due to the introduction of the next operator that is absent in [21].

We say that two infinite runs $\omega_1 = (\mathcal{P}_0^1, v_0^1)(\mathcal{P}_1^1, v_1^1)\dots$ and $\omega_2 = (\mathcal{P}_0^2, v_0^2)$ $(\mathcal{P}_1^2, v_1^2)\dots$ are n-equivalent, denoted $\omega_1 \equiv_n \omega_2$, if for all $i \le n$ $\mathcal{P}_i^1 = \mathcal{P}_i^2$ and $v_i^1 = v_i^2$. We say that ω n-boundly satisfies φ, denoted $\omega \models^n \varphi$, iff for all ω' where $\omega \equiv_n \omega'$, $\omega' \models \varphi$. We say that run n-boundly violate φ if for all ω' where $\omega \equiv_n \omega'$, $\omega' \not\models \varphi$. It is easy to see that $\omega \models^n \varphi \implies \omega \models \varphi$ and $\omega \not\models^n \varphi \implies \omega \not\models \varphi$. We can now conclude with the following theorem that shows that any WMTL property can be decided on a finite prefix of the run.

Theorem 1. *Let ω be an infinite run and φ be a WMTL formula. Then $\omega \models \varphi$ if and only if $\omega \models^{N(\omega,\varphi)} \varphi$ and $\omega \not\models \varphi$ if and only if $\omega \not\models^{N(\omega,\varphi)} \varphi$.*

4 Monitoring WMTL Properties

We present an efficient online monitoring algorithm for checking if a given infinite run ω of a WTA satisfies a given WMTL property φ.

Algorithm 1. WMTL formula satisfiability checking

```
// Input: MTL formula φ and weighted word ω
// Output: true iff ω ⊨ φ, false otherwise
i:=0
while φ ≠ ⊤ ∧ φ ≠ ⊥ do
    φ:=β(γ(φ, P_i, v_{i+1} − v_i))
    i:=i+1
end
if φ == ⊤ then
    return true
end
if φ == ⊥ then
    return false
end
```

The pseudo code of our algorithm is presented in Algorithm 1. Intuitively, the algorithm reads the elements of the input run one-by-one and rewrites the formula after reading each new element. The algorithm stops when the formula becomes $\top$ or $\bot$ meaning that any continuation of the finite prefix read so far will be accepted (or rejected) by the original formula φ. The rewriting step is performed by first applying the function γ, that *updates* the formula according to a new observation, and then applying β function, that *simplifies* the formula and tries to reduce it to $\top$ or $\bot$.

The *rewrite* function γ is defined by the following recursive rules where v is a function that gives the change of the clock variables since the last element of the run:

- $\gamma(p, \mathcal{P}, v) = \begin{cases} \top, & \text{if } p \in \mathcal{P} \\ \bot, & \text{if } p \notin \mathcal{P} \end{cases}$
- $\gamma(\neg p, \mathcal{P}, v) = \begin{cases} \bot, & \text{if } p \in \mathcal{P} \\ \top, & \text{if } p \notin \mathcal{P} \end{cases}$
- $\gamma(\varphi_1 \wedge \varphi_2, \mathcal{P}, v) = \gamma(\varphi_1, \mathcal{P}, v) \wedge \gamma(\varphi_2, \mathcal{P}, v)$
- $\gamma(\varphi_1 \vee \varphi_2, \mathcal{P}, v) = \gamma(\varphi_1, \mathcal{P}, v) \vee \gamma(\varphi_2, \mathcal{P}, v)$
- $\gamma(O\, \varphi, \mathcal{P}, v) = \varphi$
- $\gamma(\varphi_1 U^x_{[a,\, b]} \varphi_2, \mathcal{P}, v) =$
$$\begin{cases} \gamma(\varphi_1, \mathcal{P}, v) \wedge \varphi_1 U^x_{[min(a-v(x),\, 0),\, b-v(x)]} \varphi_2\,, & \text{if } a > 0 \wedge v(x) \leq b \\ \gamma(\varphi_2, \mathcal{P}, v) \vee (\gamma(\varphi_1, \mathcal{P}, v) \wedge \varphi_1 U^x_{[0,\, b-v(x)]} \varphi_2)\,, & \text{if } a = 0 \wedge v(x) \leq b \\ \gamma(\varphi_2, \mathcal{P}, v), & \text{if } a = 0 \wedge v(x) > b \\ \bot, & \text{if } a > 0 \wedge v(x) > b \end{cases}$$
- $\gamma(\varphi_1 R^x_{[a,\, b]} \varphi_2, \mathcal{P}, v) =$
$$\begin{cases} \gamma(\varphi_2, \mathcal{P}, v) \wedge \varphi_1 R^x_{[min(a-v(x),\, 0),\, b-v(x)]} \varphi_2\,, & \text{if } a > 0 \wedge v(x) \leq b \\ \gamma(\varphi_2) \wedge (\gamma(\varphi_1, \mathcal{P}, v) \vee \varphi_1 R^x_{[0,\, b-v(x)]} \varphi_2)\,, & \text{if } a = 0 \wedge v(x) \leq b \\ \gamma(\varphi_2, \mathcal{P}, v), & \text{if } v(x) > b \end{cases}$$

The omitted cases are all rewritten into themselves. The *simplify* function β is defined by the following recursive rules:

– $\beta(\varphi_1 \wedge \varphi_2) = \begin{cases} \bot, & \text{if } \beta(\varphi_1) = \bot \text{ or } \beta(\varphi_2) = \bot \\ \beta(\varphi_1), & \text{if } \beta(\varphi_2) = \top \\ \beta(\varphi_2), & \text{if } \beta(\varphi_1) = \top \\ \beta(\varphi_1) \wedge \beta(\varphi_2), & \text{otherwise.} \end{cases}$

– $\beta(\varphi_1 \vee \varphi_2) = \begin{cases} \top, & \text{if } \beta(\varphi_1) = \top \text{ or } \beta(\varphi_2) = \top \\ \beta(\varphi_1), & \text{if } \beta(\varphi_2) = \bot \\ \beta(\varphi_2), & \text{if } \beta(\varphi_1) = \bot \\ \beta(\varphi_1) \vee \beta(\varphi_2), & \text{otherwise.} \end{cases}$

– $\beta(\varphi) = \varphi$ in rest of the cases.

The *simplify* function β takes into account only the logical equivalences, namely $\varphi \wedge \top \equiv \varphi$, $\varphi \wedge \bot \equiv \bot$, $\varphi \vee \top \equiv \top$, $\varphi \vee \bot \equiv \varphi$.

The correctness and termination of our algorithm is proved by the following two theorems:

Theorem 2. *Let $\omega = (\mathcal{P}_0, v_0), (\mathcal{P}_1, v_1), \ldots$ be an infinite weighted word, and φ be a WMTL formula. Then $\omega^i \models \varphi$ if and only if $\omega^{i+1} \models \gamma(\varphi, \mathcal{P}_i, v_{i+1} - v_i)$.*

Theorem 3. *Let $\omega = (\mathcal{P}_0, v_0), (\mathcal{P}_1, v_1), \ldots$ be an infinite weighted word that diverges for every clock used in a WMTL formula φ. Let $\varphi_0 = \varphi, \varphi_1, \ldots$ be a sequence of WMTL formulas such that for all $i > 0$ $\varphi_{i+1} = \beta(\gamma(\varphi_i, \mathcal{P}_i, v_{i+1} - v_i))$. Then there exists $k \geq 0$ such that $\varphi_k = \top$ if and only if $\omega \models \varphi_0$. Similarly, there exists $k \geq 0$ such that $\varphi_k = \bot$ if and only of $\omega \not\models \varphi_0$.*

Example 2. Consider the run $(\{a\}, \{\tau \mapsto 0\})(\{a\}, \{\tau \mapsto 2.5\})(\{b\}, \{\tau \mapsto 3\})$ $(\{a\}, \{\tau \mapsto 3.2\})(\{b, c\}, \{\tau \mapsto 5\})(\{a\}, \{\tau \mapsto 6\}) \ldots$ and the WMTL formula $(aU_{[0,4]}b)U_{[0,10]}c$. Our algorithms will produce the following sequence of rewriting rules. The sequence results in $\top$ thus the formula is satisfied by the run.

$$\begin{aligned} (aU_{[0,4]}b)U_{[0,10]}c &\xrightarrow{\{a\},\{\tau\mapsto 2.5\}} (aU_{[0,1.5]}b) \wedge (aU_{[0,4]}b)U_{[0,7.5]}c \\ &\xrightarrow{\{a\},\{\tau\mapsto 0.5\}} (aU_{[0,1.0]}b) \wedge ((aU_{[0;3.5]}b) \wedge (aU_{[0,4]}b)U_{[0,7.0]}c) \\ &\xrightarrow{\{b\},\{\tau\mapsto 0.2\}} (aU_{[0,4]}b)U_{[0,6.8]}c \\ &\xrightarrow{\{a\},\{\tau\mapsto 1.8\}} (aU_{[0,3.2]}b) \wedge ((aU_{[0,4]}b)U_{[0,5.0]}c) \\ &\xrightarrow{\{b,c\},\{\tau\mapsto 1\}} \top. \end{aligned}$$

5 Experiments

Our approach has been implemented in UPPAAL-SMC. We now illustrate the technique and compare it with the one in [7] that relies on automata-based

monitors. If there is no deterministic automaton for the corresponding formula, [7] builds a deterministic under/over approximation that may strongly impact the confidence interval computed by SMC.

5.1 Size of Intermediate Formulas and Precision

Our rewriting rules are recursive in the structure of the formula, which means that the performance of the technique is highly dependent on the size of the intermediate formulas. In the following example, we show how the size of the intermediate formulas vary. We also show that our technique is often much more accurate than the one of [7].

We first study the evolution of the size of the intermediate formula generated by our technique during the monitoring of several randomly generated formulas. We also study the precision of the confidence interval returned by the SMC algorithm in case [7] uses an over or under approximation of the monitor. We also exploit an encoding in Uppaal to show how the size of the formula varies over time for a validation of a single run. In both cases runs are randomly generated by automata. This is done by choosing a delay with respect to an exponential distribution with rate parameter r and after the delay with a discrete probabilistic choice set one of the propositions to true or false.

Random Formulas. We compute the average size of the largest intermediary formula generated in the rewriting process of different formulas. We verified each formula with a confidence level of 0.05. The results of the test are shown in Table 2(a) and Table 2(b). We also give the verifiation time and the time used in total for the monitor based approach, i.e. both the time to construct the monitor and to verify. The results show that the intermediate formula size depends on the transition rate of the model and as a result so does the validation. The monitor based approach, on the other hand, does not depend on this and the time used remain constant for all the models - due to the most significant part of the monitor based approach is constructing the monitor. However, the rewrite technique is significantly faster than the monitoring technique in all cases. For the results in Table 2(a) the monitors are tight approximations thus we gain time and not precision. However, results in Table 2(b) show that we can obtain much more accurate confidence intervals with our new technique. This is due to the monitors might be a large over approximations/a small under approximation. The variance in Table 2(a) is rather high due to the runs being random.

Modeling Uppaal inside Uppaal. In order to obtain a more in-depth view on how the size of formulas change over time, we have encoded the rules as Uppaal timed automata. The objective being to use the visualisation features of the tool to see how the number of automata evolve over time. Our construction is recursive in the structure of the formula in the sense that a network of observing automata for ϕ is obtained as one automata for ϕ and at least one automaton for each of the sub-formulas of ϕ.

Table 1. Result of the random formula test. The r column is the rate at which the run was generated. The $\#U/R$ column contains the number of until or release modalities that was in the formula. The E_{Max} column is the average largest size of formula and σ^2_{Max} is the variance thereof. τ_M and τ_R is the verification time for the monitoring technique and the rewrite technique, respectively. The verification time for the monitors are the time to construct the monitors and use them - in all the cases the monitors were not exact and both the under and over approximation was used. The R_R and R_M columns contain the number of runs each method required to establish the verification result. The $\%_M$ and $\%_R$ columns refer to the confidence interval obtained by the monitoring and the rewrite process respectively.

Formula	r	$\#U/R$	Largest	E_{Max}	σ^2_{Max}	τ_R	τ_M	R_R	R_M
Random1	1	11	14	6.81	3.16	$0.19s$	$5.70s$	738	1748
Random1	4	11	18	7.03	4.92	$0.22s$	$5.83s$	738	1748
Random1	8	11	21	7.06	4.74	$0.23s$	$5.78s$	738	1748
Random2	1	8	17	8.52	5.33	$0.19s$	$6.13s$	738	1748
Random2	4	8	21	11.05	4.71	$0.34s$	$6.17s$	738	1748
Random2	8	8	27	12.79	7.16	$0.58s$	$6.26s$	738	1748
Random3	1	11	21	11.51	4.74	0.50	$10.99s$	738	1748
Random3	4	11	40	13.58	16.53	1.08	$11.06s$	738	1748
Random3	8	11	36	14.00	18.16	1.52	$11.38s$	738	1748

(a)

Formula	$\#U/R$	r	$\%_R$	$\%_M$	R_R	R_M	τ_R	τ_M
random4	15	4	[0.57; 0.67]	[0.57; 0.83]	738	1748	0.34s	7.77s
random5	15	4	[0.00; 0.05]	[0.00; 0.97]	738	1748	0.94s	2.83s
random6	15	4	[0.00; 0.05]	[0.00; 0.72]	738	1748	0.81s	3.18s
random7	15	4	[0.00, 0.07]	[0.00; 0.43]	738	1748	2.36s	26.61s

(b)

The automaton for ϕ starts its sub-automata through a designated *init*-channel and the sub-automata informs the ϕ-automaton that their sub-formula has been rewritten to $\top$ or $\bot$ through designated channels. The automata for until and release rely on having multiple automata for their sub-formulas that they can start one of after each observation. If there are insufficient sub-automata an error state is reached - because of this the encoding is an under-approximation of the WMTL formula in question.

Example 3. Consider the run $(p, \{\tau \mapsto t_0\})(p, \{\tau \mapsto t_1\})(p, \{\tau \mapsto t_2\})(\neg p, \{\tau \mapsto t_3\})(p, \{\tau \mapsto t_4\})(p, \{\tau \mapsto t_5\})(p, \{\tau \mapsto t_6\})(p, \{\tau \mapsto t_7\})(?, \{\tau \mapsto t_8\})$ where we do not know if the proposition p is true at time t_8 and let $t_8 - t_0 > 10$. In Fig. 3 we provide a snapshot of the set of active automata at time t_7. At the top we have an automaton that monitors the expression $\Diamond_{[0;10]}\Box_{[4;15]}p$ which has been active since t_0 thus it has $10 - (t_7 - t_0)$ time units left before its expression has been violated.

Below this automaton are automata observing the subexpression $\Box_{[4;15]}p$. These automata have been started at times t_4, t_5 and t_6 respectively and will

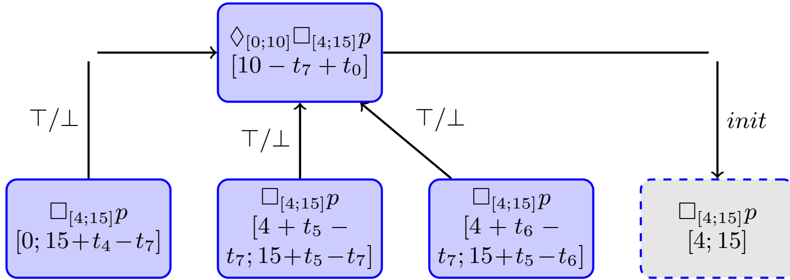
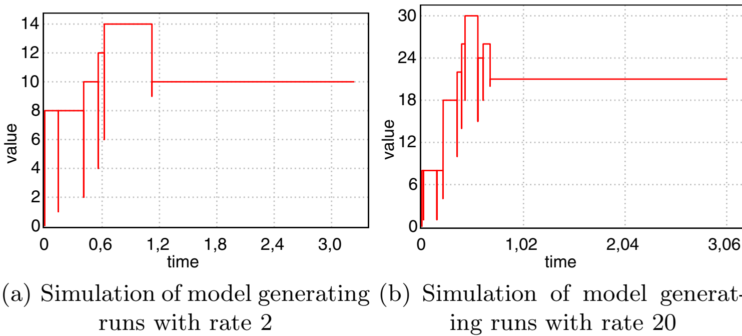


Fig. 3. Snapshot at time t_7 with 3 active automata and one being started.



(a) Simulation of model generating runs with rate 2 (b) Simulation of model generating runs with rate 20

Fig. 4. Plots of how the size of the formula varies over time. On the y-axis is plotted the number of active automata and the x-axis contain the time.

report ⊤ to the parent automaton at the moment they have oberved p for 15 time units or ⊥ if they observe $\neg p$. Notice that all the automata started before t_4 are no longer active since $\neg p$ was true at time t_3. Also, there is one automaton (the gray one with dashed borders) that is being started by $\Diamond_{[0;10]}\Box_{[4;15]}p$ through its *init*-channel. Since $t_8 - t_0 > 0$ the top level automaton will not start any sub-automata at time t_8. Instead it will merely wait for the already started automata to return either ⊤ or ⊥. If one of them return ⊤ then the top-level automaton will return ⊤. In case all of the sub-automata return ⊥ then the top-level automata will return ⊥.

We encoded the formula $\Diamond_{[0;1]}(p \wedge \Box_{[0;1]}(\neg r) \wedge \Diamond_{[0;1]}(q))$, and put the resulting automata in parallel with an automata generating random runs and an automaton incrementing a counter whenever an automaton was started or decremented the counter, whenever an automaton stopped. We did this for transition rates 2 and 20 of the random run generating automaton and used the simulate query, `simulate 1 [<=3] size`.

In Fig. 4 we show the plots we obtain for runs generated with varying transition rates. One can easily see that the number of automata does not increase exponentially. We have observed the phenomena on various case studies.

5.2 IEEE 802.15.4 CSMA/CA Protocol

IEEE 802.15.4 standard [18] specifies the physical and media access control layers for low-cost and low-rate wireless personal area networks. Devices operating in such networks share the same wireless medium and can possibly corrupt the transmission of each other by sending data at the same time. We applied our technique to the analysis of Carrier Sense Multiple Access/Collision Avoidance (CSMA/CA) network contention protocol that is used in IEEE 802.15.4 to minimise the number of collisions.

Our objective is to estimate the probability that if a collision occurs, then all nodes participating in it will recover from the collision within a given time bound. This can be specified with $\wedge_{i=1..N}\varphi_i$, where N is a number of network nodes and φ_i specifies the behavior of a single node:

$$\varphi_i \equiv \Box_{\leq 10000}(collision_i \rightarrow \Diamond_{\leq 4000} send_i)$$

The monitor built by [7] is precise. We are thus not interested to reason on precision of the confidence interval, but rather on the evolution of computation time. In Fig. 5 we observe that both the size of intermediary formulas used to rewrite φ and the computation time grow linearly as the number of components N increases. On the other hands, both the size of monitor and the computation time with the approach in [7] grow exponentially and cannot be applied to real-life deployments of CSMA.

Number of nodes	2	3	4	5	6
Monitor-based approach (time)	<1s	3s	57s	20m2s	-
Size of the monitor	230	2049	16306	123800	-
Rewrite-based approach (time)	55s	2m85s	4m11s	6m32s	9m21.47s
Average formula size	8.98	13.76	19.24	24	30.34

Fig. 5. Results for the CSMA/CA protocol

Although the monitor-based approach is faster for smaller N, for larger N it quickly becomes intractable, while the rewrite-based approach scales well.

6 Conclusion

We presented a new monitoring procedure for WMTL formulas. The technique relies on a series of rewriting step for the formula and is guaranteed to terminate. Contrary to automata-based approaches, ours is precise in the sense that it does not depend on over and under approximation of the formula. We have implemented our approach in Uppaal-smc. Our results outperform those of the monitor-based approaches.

References

1. Bauer, A., Leucker, M., Schallhart, C.: Comparing ltl semantics for runtime verification. J. Log. Comput. 20(3), 651–674 (2010)
2. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM TESMy 20(4), 14 (2011)
3. Behrmann, G., Fehnker, A., Hune, T., Larsen, K.G., Pettersson, P., Romijn, J., Vaandrager, F.W.: Minimum-Cost Reachability for Priced Timed Automata. In: Di Benedetto, M.D., Sangiovanni-Vincentelli, A.L. (eds.) HSCC 2001. LNCS, vol. 2034, pp. 147–161. Springer, Heidelberg (2001)
4. Legay, A., Delahaye, B., Bensalem, S.: Statistical Model Checking: An Overview. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 122–135. Springer, Heidelberg (2010)
5. Bouyer, P., Larsen, K.G., Markey, N.: Model checking one-clock priced timed automata. Logical Methods in Computer Science 4(2) (2008)
6. Bouyer, P., Markey, N.: Costs Are Expensive! In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 53–68. Springer, Heidelberg (2007)
7. Bulychev, P.E., David, A., Larsen, K.G., Legay, A., Li, G., Poulsen, D.B., Stainer, A.: Monitor-Based Statistical Model Checking for Weighted Metric Temporal Logic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18 2012. LNCS, vol. 7180, pp. 168–182. Springer, Heidelberg (2012)
8. David, A., Larsen, K.G., Legay, A., Mikučionis, M., Poulsen, D.B., van Vliet, J., Wang, Z.: Statistical Model Checking for Networks of Priced Timed Automata. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 80–96. Springer, Heidelberg (2011)
9. Drusinsky, D.: The Temporal Rover and the ATG Rover. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 323–330. Springer, Heidelberg (2000)
10. Gong, H., Zuliani, P., Komuravelli, A., Faeder, J.R., Clarke, E.M.: Computational Modeling and Verification of Signaling Pathways in Cancer. In: Horimoto, K., Nakatsui, M., Popov, N. (eds.) ANB 2010. LNCS, vol. 6479, pp. 117–135. Springer, Heidelberg (2012)
11. Havelund, K., Roşu, G.: Synthesizing Monitors for Safety Properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
12. Hérault, T., Lassaigne, R., Magniette, F., Peyronnet, S.: Approximate Probabilistic Model Checking. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 73–84. Springer, Heidelberg (2004)
13. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A Bayesian Approach to Model Checking Biological Systems. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 218–234. Springer, Heidelberg (2009)
14. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2(4), 255–299 (1990)
15. Maler, O., Nickovic, D., Pnueli, A.: Real Time Temporal Logic: Past, Present, Future. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 2–16. Springer, Heidelberg (2005)
16. Rosu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Autom. Softw. Eng. 12(2), 151–197 (2005)

17. Sen, K., Viswanathan, M., Agha, G.: Statistical Model Checking of Black-Box Probabilistic Systems. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 202–215. Springer, Heidelberg (2004)
18. I. C. Society. Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs) (2003)
19. Younes, H.L.S.: Ymer: A Statistical Model Checker. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 429–433. Springer, Heidelberg (2005)
20. Younes, H.L.S., Kwiatkowska, M.Z., Norman, G., Parker, D.: Numerical vs. statistical probabilistic model checking. STTT 8(3), 216–228 (2006)
21. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to simulink/stateflow verification. In: HSCC 2010, pp. 243–252. ACM, New York (2010)

From Runtime Verification to Runtime Intervention and Adaptation

Martin Rinard

Massachussetts Institute of Technology

Abstract. Runtime verification monitors the execution of a program to determine if it satisfies (typically specified) correctness properties. But what happens when the program violates the correctness properties? The standard view is that continued execution may be unsafe, so the execution must be terminated. We present a variety of intervention techniques that enable software systems to exhibit remarkable flexibility and resilience in the face of errors and faults. These techniques can deliver safe continued execution that offers significant benefits over termination. We also present techniques that build on this malleability to purposefully modify the computation to adapt to changing needs, delivering benefits such as improved performance and reduced power consumption. These results place the advantages of runtime intervention and adaptation clearly on display. They also point the way to a future in which developers produce not the final version of the program which the system blindly executes, but instead a starting point for further modification and evolution as the system adapts to dynamically observed events and conditions.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, p. 276, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Certifying Solutions for Numerical Constraints

Eva Darulova and Viktor Kuncak*

EPFL
{eva.darulova,viktor.kuncak}@epfl.ch

Abstract. A large portion of software is used for numerical computation in mathematics, physics and engineering. Among the aspects that make verification in this domain difficult is the need to quantify numerical errors, such as roundoff errors and errors due to the use of approximate numerical methods. Much of numerical software uses self-stabilizing iterative algorithms, for example, to find solutions of nonlinear equations.

To support such algorithms, we present a runtime verification technique that checks, given a nonlinear equation and a tentative solution, whether this value is indeed a solution to within a specified precision.

Our technique combines runtime verification approaches with information about the analytical equation being solved. It is independent of the algorithm used for finding the solution and is therefore applicable to a wide range of problems. We have implemented our technique for the Scala programming language using our affine arithmetic library and the macro facility of Scala 2.10.

Keywords: solution verification, numerical computation, error estimation, affine arithmetic.

1 Introduction

Software manipulating numerical quantities has numerous applications in decision making, science, and technology. Such software is difficult to validate by any method—manual inspection, testing, or static analysis. One of the core challenges in each case is the gap between the approximate nature of numerical computations and the idealized mathematical models that form their foundation and specification. Specialized programming languages inside commercial computer algebra systems aim to simplify working with numerical computations. However, their precision and soundness guarantees compared to the mathematical meaning are not well documented, and many of the implementations are closed source. Much of the real-world computation is done in general-purpose languages, supported by many numerical software libraries written for them. The work on this paper builds on open-source general-purpose infrastructures, providing a next step in validated numerical computation for Scala [14].

Existing validation of numerical computations supports estimation of roundoff errors [7,1]; we have previously incorporated computation of roundoff errors in

* This research is supported by the Swiss NSF Grant #200021_132176.

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 277–291, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Scala using affine arithmetic [5]. Going a step further, we present automated estimation of not only roundoff errors, but also *method errors*, which arise, for example, when using numerical methods to iteratively solve equations. Such methods are used to solve equations that have no symbolic closed-form solution, which is often the case in practice. Even if symbolic solutions exist, iterative approaches can be faster or better-behaved with respect to roundoff errors.

1.1 Contributions

To understand the notion of method errors we address, consider an iterative method that performs a search for the solution of $f(x) = 0$ by computing a sequence of approximations $x_0, x_1, x_2, \ldots$ One common stopping criterion for an iteration is finding x_k for which $|f(x_k)| < \varepsilon$, for a given error tolerance ε. From a validation point of view, however, we are ultimately interested not in ε but in τ such that $|x - x_k| < \tau$, where x is the actual solution in real numbers. Fortunately, we can estimate τ from ε using a bound on the derivative of f in an interval conservatively enclosing x and x_k.

A tempting approach is to perform the entire computation of x_k using interval [12] or affine arithmetic. However, this approach would be inefficient, and would give too pessimistic error bounds. Instead, our method uses a runtime checking approach. We allow any standard non-validated floating point code to compute the approximation x_k. We perform only the final validation of an individual candidate solution x_k using a range-based computation. In this way we achieve efficiency and reusability of existing numerical routines, while still providing rigorous bounds on the total error. The bounds certified by our system are always sound for the given execution.

Our system thus realizes a new kind of assertion, appropriate for numerical computation: an assertion that verifies "this was precise enough" in a way that takes into account both the numerical algorithm and floating point semantics.

To perform such sound computation, our approach uses static information about the function and computes derivatives at compile time. For this purpose it uses the macro facility of Scala, our implementation of symbolic differentiation, and a method to compute bounds of a function over an interval. A technical challenge that arises in rigorously estimating the error is that mean value theorems (the foundation for error estimation), refer to an arbitrary point between the approximate and the unknown exact solution. It is therefore not clear over which interval one needs to estimate the error. We solve this circularity through a simple design, which expects a bound on the argument error as the input, and verifies whether this bound indeed holds. This allows us to perform an estimation using very narrow intervals, contributing to the precision of our approach.

We integrated our method into the Scala programming language (Section 4). We demonstrate its applicability and usefulness on a number of examples (sections 2 and 5). Among the consequences of this development is a Scala framework that can check runtime assertions in a way consistent with mathematical reals, while executing on the standard virtual machine, soundly taking into account the concrete semantics of floating point operations and iterative numerical methods.

2 Examples

We motivate our contribution with examples that model physical processes, taken from [18,4,15]. These examples illustrate the applicability of our techniques and introduce the main features of our library. For space reasons we abbreviate the Scala Double type with D (the code snippets remain valid Scala code using the rename-on-import Scala feature). We include variable type declarations for expository purposes, even though the Scala compiler can infer all but the function parameter types. A function that maps x into $e(x)$ is denoted in Scala by $x \Rightarrow e(x)$. Method names printed in bold (e.g., **jacobian**, **assertBound**) are parts of the public interface of our library for certifying solutions of numerical computations.

Stress on a Turbine Rotor. We illustrate the basic features of our library on the following system of three non-linear equations with three unknowns (v, ω, r). An engineer may need to solve such a system to compute the stress on a turbine rotor [18].

$$3 + \frac{2}{r^2} - \frac{1}{8}\frac{(3-2v)}{1-v}\omega^2 r^2 = 4.5 \qquad 6v - \frac{1}{2}\frac{v}{1-v}\omega^2 r^2 = 2.5$$
$$3 - \frac{2}{r^2} - \frac{1}{8}\frac{(1+2v)}{1-v}\omega^2 r^2 = 0.5 \tag{1}$$

Given a numerical routine computeRoot and our library for certifying solutions, the engineer can directly map the above equations into the following code:

```
val f1 = (v:D,w:D,r:D) ⇒ 3 + 2/(r*r) − 0.125*(3−2*v)*(w*w*r*r)/(1−v)−4.5
val f2 = (v:D,w:D,r:D) ⇒ 6*v − 0.5 * v * (w*w*r*r) / (1−v)−2.5
val f3 = (v:D,w:D,r:D) ⇒ 3 − 2/(r*r) − 0.125*(1+2*v)*(w*w*r*r) / (1−v)−0.5
```

The engineer can then solve the problem numerically using an off-the-shelf numerical routine that accepts the function and its derivative as an argument:

```
val x0 = Array(0.75, 0.5, 0.5) // initial value for iteration
val roots: Array[D] = computeRoot(Array(f1,f2,f3), jacobian(f1,f2,f3), x0, 1e−8)
```

Finally, the engineer can *certify* the solution using our library:

```
val errors:Array[Interval] = assertBound(f1,f2,f3, roots(0), roots(1), roots(2), 1e−8)
```

The method **assertBound** takes as input the three functions of our system of equations, the previously computed roots and a tolerance. It returns sound bounds on the true errors on the roots. In the case where these errors are larger than the tolerance specified, the method throws an exception and thus acts like an assertion. Our library also includes the method **jacobian**, which computes the Jacobian matrix of the functions f_1, f_2 and f_3 symbolically at compile time (Section 4.2). The true roots for v, w and r are $0.5, 1.0$ and 1.0 respectively. The roots and maximum absolute errors computed by the above code are

```
0.5, 1.0000000000018743, 0.9999999999970013
2.3684981521893e-15, 1.8806808806556e-12, 3.0005349681420e-12
```

Note that the error bounds computed are, in fact, orders of magnitude smaller than the tolerance $1e{-}8$ given to the numerical routine and to **assertBound**.

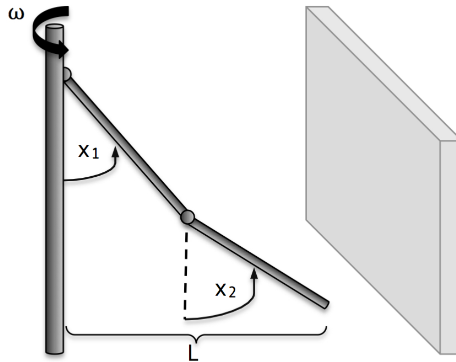


Fig. 1. A double pendulum standing close to an obstacle

Double Pendulum. The following example demonstrates how our library fits into a runtime assertion framework consistent with mathematical reals. A double pendulum rotates with angular velocity ω around a vertical axis, like a centrifugal regulator [4]. At equilibrium, the two pendulums make the angles x_1 and x_2 to the vertical axis. It can be shown that the angles are determined by the equations

$$\begin{aligned} \tan x_1 - k(2\sin x_1 + \sin x_2) = 0 \\ \tan x_2 - 2k(\sin x_1 + \sin x_2) = 0 \end{aligned} \tag{2}$$

where k depends on ω, the lengths of the rods and gravity. Suppose the pendulum is standing close to a wall (as in Figure 1) and we would like to verify that in the equilibrium position it cannot hit the wall. Also suppose that the distance to the center of the pendulum is given by a function distancePendulumWall. Then the following code fragment verifies that a collision is impossible in the real world, not just in a world with floating-points.

```
val distancePendulumWall : SmartFloat = ...
val length = ... //length of bars
val tolerance = 1e−13; val x0 = Array(0.18, 0.25)
val f1 = (x1: D, x2: D) ⇒ tan(x1) − k ∗ (2∗sin(x1) + sin(x2))
val f2 = (x1: D, x2: D) ⇒ tan(x2) − 2∗k ∗ (sin(x1) + sin(x2))
val r: Array[D] = computeRoot(Array(f1,f2), jacobian(f1,f2), x0, tolerance)
val roots: Array[SmartFloat] = certify(r, errorBound(f1, f2, r(0), r(1), tolerance))

val L: SmartFloat = _sin(roots(0)) ∗ length + _sin(roots(1)) ∗ length
if (certainly(L <= distancePendulumWall)) {
  // continue computation
} else {
  // reduce speed of the pendulum and repeat
}
```

To account for all sources of uncertainty, we use the SmartFloat data type developed previously [5]. SmartFloat performs a floating point computation while

additionally keeping track of different sources of errors, including floating point round-off errors, as well as errors arising from other sources, for example, due to the approximate nature of physical measurements.

In our example, distancePendulumWall and **certify** both return a SmartFloat; the first one captures the uncertainty on a physical quantity, and the second one the method error due to the approximate iterative method. If the comparison in line 9 succeeds, we can be sure the pendulum does not touch the wall. This guarantee takes into account roundoff errors committed during the calculation, as well as the error committed by the computeRoot method and their propagation throughout the computation.

State Equation of a Gas. Values of parameters may only be known within certain bounds but not exactly, for instance if we take inputs from measurements. Our library provides guarantees even in the presence of such uncertainties. Equation 3 below relates the volume V of a gas to the temperature T and the pressure p, given parameters a and b that depend on the specifics of the gas, N the number of molecules in the volume V and k the Boltzman constant [15].

$$[p + a(N/V)^2](V - Nb) = kNT \tag{3}$$

If T and p are given, one can solve the nonlinear Equation 3 to determine the volume occupied by the (very low-pressure) gas. Note however, that this is a cubic equation, for which closed-form solutions are non-trivial, and their approximate computation may incur substantial roundoff errors. Using an iterative method, whose result is verified by our library, is thus preferable:

```
val T = 300; val a = 0.401; val b = 42.7e−6;
val p = 3.5e7; val k = 1.3806503e−23; val x0 = 0.1
val N: Interval = 1000 +/− 5
val f = (V: D) ⇒ (p + a ∗ (N.mid/V) ∗ (N.mid/V)) ∗ (V − N.mid∗b) − k∗N.mid∗T
val V: D = computeRoot(f, derivative(f), x0, 1e−9)
val Vcert: SmartFloat = certify(V, assertBound(f, V, 0.0005))
```

We make the assumption that we cannot determine the number of molecules N exactly, but we are sure that our number is accurate at least to within ± 5 molecules (line 3). We compute the root as if we knew N exactly, using the middle value of the interval and the standard Newton's method. We only check *a posteriori* that the result is accurate up to $\pm 0.0005 m^3$, for all N in the interval $[995, 1005]$. Our library will confirm this providing us also with the (certified) bounds on V: `[0.0424713, 0.0429287]`.

3 Computing the Error

Our verification technique is based on several theorems from the area of validated numerics. It can verify roots of a system of nonlinear equations computed by an arbitrary black-box solution or estimation method.

In the following, we denote computed approximate solutions by $\tilde{x}$ and true roots by x. $\mathbb{IR}$ denotes the domain of intervals over the real numbers $\mathbb{R}$ and variables written in bold type, e.g. $\mathbf{X}$, denote interval quantities. For a function f, we define $f(\mathbf{X}) = \{f(x) \mid x \in \mathbf{X}\}$. All errors are given in absolute terms. Error tolerance, that is, the maximum acceptable value for $|\tilde{x} - x|$, will be denoted by τ or tolerance. We will use the term *range arithmetic* to mean either interval arithmetic [12] or affine arithmetic [6]. The material presented in this section is valid for any such "arithmetic", as long as it computes guaranteed enclosures containing the result that would be computed in real numbers. We wish to compute a guaranteed bound on the error of a computed solution, that is, determine an upper bound on $\Delta x = \tilde{x} - x$. Note that Δx is different from τ, because Δ considers the sign of the difference.

Unary Case. For expository purposes, consider first the unary case $f : \mathbb{R} \rightarrow \mathbb{R}$, f differentiable, and suppose that we wish to solve the equation $f(x) = 0$. Then, by the Mean Value Theorem

$$f(\tilde{x}) = f(x + \Delta x) = f(x) + f'(\xi)\Delta x \tag{4}$$

where $\xi \in \mathbf{X}$ and $\mathbf{X}$ is a range around $\tilde{x}$ sufficiently large to include the true root. Since $f(x) = 0$,

$$\Delta x \in \frac{f(\tilde{x})}{f'(\mathbf{X})} \tag{5}$$

The set membership instead of equality is because the right-hand side is now a range-valued expression, which takes into account the fact that ξ in the Mean Value Theorem is not known exactly. The following theorem (stated in the formulation from [17]) formalizes this idea.

Theorem 1. *Let a differentiable function $f : \mathbb{R} \rightarrow \mathbb{R}$, $\mathbf{X} = [x_1, x_2] \in \mathbb{IR}$ and $\tilde{x} \in \mathbf{X}$ be given, and suppose $0 \notin f'(\mathbf{X})$. Define*

$$N(\tilde{x}, \mathbf{X}) := \tilde{x} - f(\tilde{x})/f'(\mathbf{X}). \tag{6}$$

If $N(\tilde{x}, \mathbf{X}) \subseteq \mathbf{X}$, then $\mathbf{X}$ contains a unique root of f. If $N(\tilde{x}, \mathbf{X}) \cap \mathbf{X} = \varnothing$, then $f(x) \neq 0$ for all $x \in \mathbf{X}$.

Claim. If, following Equation 5, we compute an interval $\mathbf{\Delta x} = f(\tilde{x})/f'(\mathbf{X})$ enclosing the upper bound on the error Δx, and if $\mathbf{\Delta x} \subseteq [-\tau, \tau]$, then the approximately computed result $\tilde{x}$ is indeed within the specified precision τ.

Indeed, choose $\mathbf{X} = [\tilde{x} - \tau, \tilde{x} + \tau]$, i.e. the computed approximate solution plus or minus the tolerance we want to check, and compute $\mathbf{\Delta x} = \frac{f(\tilde{x})}{f'(\mathbf{X})}$. Then the condition $N(\tilde{x}, \mathbf{X}) \subseteq \mathbf{X}$ from Theorem 1 becomes

$$N(\tilde{x}, \mathbf{X}) = \tilde{x} - \mathbf{\Delta x} \subseteq \mathbf{X} = [\tilde{x} - \tau, \tilde{x} + \tau] \tag{7}$$

If $\mathbf{\Delta x} \subseteq [-\tau, \tau]$, this condition holds, and thus the computed result is within the specified precision.

```
def assertBound (Function, Derivative, xn, τ)
  X = [xn ± τ]
  error = Function(xn) / Derivative(X)
  if error ∩ [−τ, τ] = ∅ throw SolutionNotIncludedException
  if ¬(error ⊂ [−τ, τ]) throw SolutionCannotBeVerifiedException
  return error
```

Fig. 2. Procedure for computing errors in the unary case

Our assertion library uses the procedure in Figure 2 for unary problems. Note that we not only check that errors are within a certain error tolerance, but we also return the computed error bounds. As we show in Section 5, the computed error bounds tend to be much tighter than the user-required tolerance. As Section 4.3 illustrates, this error bound can be used in subsequent computations to track overall errors more precisely.

Multivariate Case. Our error estimates for the unary case follow from the Mean Value Theorem, which extends to n dimensions. Theorem 2 follows the interval formulation of [17] where J_f is the Jacobian matrix of f. If $\mathbf{D} = (\mathbf{x_1}, \ldots, \mathbf{x_n}) \in \mathbb{IR}^n$, let $\bar{\mathbf{D}}$ denote $\mathbf{x_1} \times \ldots \times \mathbf{x_n}$. For $a, b \in \bar{\mathbf{D}}$, define convex union as $a \underline{\cup} b = \{a + \lambda b \mid \lambda \in [0,1]\}$. For $A \subseteq \bar{\mathbf{D}}$, define $hull(A) := \bigcap\{\mathbf{Z} \in \mathbb{IR}^n \mid A \subseteq \mathbf{Z}\}$.

Theorem 2. *Let there be given a continuously differentiable $f : \bar{\mathbf{D}} \to \mathbb{R}^n$ with $\mathbf{D} \in \mathbb{IR}^n$ and $x, \tilde{x} \in \bar{\mathbf{D}}$. Then for $\mathbf{X} := hull(x \underline{\cup} \tilde{x})$*

$$f(x) \in f(\tilde{x}) + J_f(\mathbf{X})(x - \tilde{x}) \tag{8}$$

We extend our method for computing the error on each root in a similar manner:

$$\delta \in J_f^{-1}(\mathbf{X}) \cdot (-f(\tilde{x})) \tag{9}$$

where $\delta = x - \tilde{x}$ is the vector of errors on our tentative solution. Since we now must consider the Jacobian of f instead of a single derivative function, we can no longer solve for the errors by a simple scalar division. We wish to find the maximum possible error, so we need a way to compute an upper bound on the right-hand side of Equation 9. Computing the inverse of a Jacobian matrix in a range arithmetic typically does not yield a useful result, due to over-approximation. Instead, we use the following Theorem 3, which is originally due to [11], but we use the formulation by [17].

Theorem 3 ([17]). *Let $A, R \in \mathbb{R}^{n \times n}$, $b \in \mathbb{R}^n$ and $\mathbf{E} \in \mathbb{IR}^n$ be given, denote by I the identity matrix. Assume*

$$Rb + (I - RA)\mathbf{E} \subset int(\mathbf{E}). \tag{10}$$

where $int(\mathbf{E})$ denotes the interior of the set $\mathbf{E}$. Then the matrices A and R are non-singular and $A^{-1}b \in Rb + (I - RA)\mathbf{E}$.

We instantiate Theorem 3 with all possible matrices A such that $A \in J_f(\mathbf{X})$ and all possible vectors b such that $b \in -f(\tilde{x})$, where $J_f(\mathbf{X})$ and $-f(\tilde{x})$ are both evaluated in range arithmetic. Combining with Condition 9, we obtain

$$\delta \in J_f^{-1}(\mathbf{X}) * -f(\tilde{x}) \subseteq Rb + (I - RA)\mathbf{E}, \tag{11}$$

provided that Condition 10 is satisfied in range arithmetic.

Matrix R in Theorem 3 can be chosen arbitrarily as long as Condition 10 holds. A common choice is to use an approximate inverse of A. In our case, A is range-valued, so we first compute the matrix whose entries are the midpoints of the intervals of A, and use its inverse as R. It now remains to determine $\mathbf{X}$. We choose it to be the vector where the ith entry is the interval around $\tilde{x}_i$ and width τ. If we can then show that Condition 11 holds, we have proven that $\mathbf{X}$ indeed contains a solution. Moreover, we have computed a tighter upper bound on the error. We obtain the procedure in Figure 3 for computing error bounds for systems of equations. The variables Xn, A, b, E, errors are all range valued.

```
def assertBound (functions, Jacobian, xn, τ)
  Xn = [xn ± τ]
  A = Jacobian(Xn)
  b = − functions(xn) // goal is to certify that xn is a zero of 'functions' up to τ
  R = inverse(mid(A)) // calculated in ordinary floating points
  E = [0 ± τ]
  errors = R∗b + (I − RA)E // Theorem 3
  if errors ∩ [−τ, τ]^n = ∅^n throw SolutionNotIncludedException
  if ¬(errors ⊆ [−τ, τ]^n) throw SolutionCannotBeVerifiedException
  return errors
```

Fig. 3. Procedure for computing errors in the multivariate case

Our approach requires the derivatives to be non-zero, respectively the Jacobian to be non-singular, in the neighborhood of the root. This means that, at present, we can only verify single roots. Verifying multiple roots is an ill-conditioned problem by itself, and thus requires further approximation techniques, as well as dealing with complex values. We leave this for future work. Our library does distinguish the cases when an error is provably too large from the case when our method is unable to ensure the result: we use two different exceptions for this purpose.

4 Implementation

Given the theoretical building blocks described above, the next question is how to integrate them into a general-purpose programming language. Our goal is to obtain an assertion framework for real numbers that is intuitive to use and

efficient. Figures 2 and 3 require the computation of derivatives and their evaluation in range arithmetic, but we do not want the user having to provide two differently typed functions, one in Doubles for the solver and one in Intervals for our verification method. Also, the solver may not actually require derivatives or the Jacobian, so this computation should be performed automatically and symbolically at compile time. Fortunately, Scala facilitates this within the existing compiler framework using a notion of macros.

4.1 Scala Macros

Scala version 2.10 (release candidate) introduces a macro facility [3]. To a user, macros look like regular methods, but in fact, their code is executed at compile time and performs a transformation on the Scala compiler abstract syntax tree (AST). Thus, by passing a regular function to a macro, we can access its AST and perform transformations, such as computing a derivative of an expression. The type checker runs after macro expansion, so the resulting code retains all guarantees from Scala's strong static typing. Our library provides the following functions:

def errorBound(f: (Double ⇒ Double), x: Double, tol: Double): Interval
def assertBound(f: (Double ⇒ Double), x: Double, tol: Double): Interval
def certify(root: Double, error: Interval): SmartFloat

and similarly for functions of two, three, and more variables. The function **assertBound** computes the guaranteed bounds on the errors using the algorithms in figures 2 and 3. **errorBound** removes the assertion check and only provides the computed error; the programmer is then free to define individual assertions. **certify** wraps the computed root(s) including their associated errors into a value of the SmartFloat datatype, thus providing a link to our assertion-checking framework. We also expose the automatic symbolic derivative computation facility:

def derivative(f: Double ⇒ Double): (Double ⇒ Double)
def jacobian(f1: (Double, Double) ⇒ Double, f2: (Double, Double) ⇒ Double):
(Array[Array[(Double, Double) ⇒ Double]])

The functions passed to our macros have type (Double*) => Double and may be given as anonymous functions, or alternatively defined in the immediately enclosing method or class. The functions may use parameters, with the same restrictions on their original definitions. This is particularly attractive, as it allows us to write concise code as presented in the code snippets from Section 2. Source code including all examples can be downloaded from
`http://lara.epfl.ch/~darulova/cerres.zip`.

4.2 Computing Derivatives

In this section we explain how we compute derivatives and discuss the effects of our technique on efficiency and precision. Given the function ASTs, we compute

the derivatives or Jacobian matrices already at compile time, and thus need to do this symbolically. Our system computes derivatives with the standard derivation rules, such as the chain rule. Moreover, it performs the following expression simplifications:

- pull constants outside of multiplications (before differentiation);
- compact multiplications of the same terms into a power function (before differentiation);
- simplify multiplication and addition of zeros or ones arising from the differentiation (after differentiation);
- evaluate powers with integers by repeated multiplication (at runtime).

Overall, the effect is that the resulting expressions of derivatives do not grow too large. The second and third column of Table 3 show the impact of our optimizations on execution times: the cumulative improvement is 28%.

On the other hand, the syntactic algebraic form of the expressions affects the precision of evaluating them in floating-point, interval or affine arithmetic. To estimate this impact, we have compared the overall behavior of our system with our symbolic differentiation routine against the results obtained with manually provided derivatives. Manually means that the derivatives have the syntax one would compute by hand on paper. We did the comparison on our unary benchmark problems (Table 1), and it turns out that except for two instances, the errors computed are exactly the same. For the two other functions, our manually computed derivatives actually compute an error that is worse, but the precision is still sufficient to prove solutions are correct to within the given tolerance.

A possible alternative to our compile time differentiation is runtime automatic differentiation [9]. Because it does not perform the optimizations listed above, we would expect it to have performance similar to our unoptimized version. Although we have opted for symbolic differentiation at compile time, in principle one can use any type of function and differentiation method, as long as the function is differentiable in a sufficiently large neighborhood of the root and the differentiation method keeps track of the roundoff and method errors it commits.

4.3 Integration into a Roundoff Error Assertion Framework

We combine the current work with our existing library for tracking roundoff errors [5] into an assertion language that can be assumed to work with real numbers. That is, if no exceptions are thrown, the program would take the same path if real numbers were used instead of floating-points and the values computed are within the bounds computed by the SmartFloat datatype. This assertion language thus tracks two sources of errors

- quantization errors due to the discrete floating-point number representation (this has been implemented in [5]);
- method errors due to the approximate numerical method (this is a contribution of the present paper).

The bounds on computed values are ensured by using SmartFloats throughout the straight-line computations. Note that the numerical method still uses only Doubles since we verify the result a posteriori. Path consistency is ensured by the compare method of the SmartFloat datatype, which takes uncertainties into account. That is, if a comparison $x < y$ cannot be decided for sure due to uncertainties on the arguments, an exception is thrown. This behavior can be adjusted to a particular application by the methods

```
def certainly(b : ⇒ Boolean) : Boolean =
  try b catch { case e: SmartFloatComparisonUndetermined ⇒ false }
def possibly(b : ⇒ Boolean) : Boolean =
  try b catch { case e: SmartFloatComparisonUndetermined ⇒ true }
```

If we cannot be sure a boolean expression involving SmartFloats is true, we assume it is **false** in the case of **certainly**, and that it is **true** in the case of **possibly**. Hence, the following identity holds:

if (**certainly**(P)) T **else** E ⇔ **if** (**possibly**(!P)) E **else** T

4.4 Uncertain Parameters

Theorem 3 also holds for range-valued A and b. It is thus natural to extend our macro functions to also accept range-valued parameters. The SmartFloat datatype already has the facility to keep track of manually user-added errors so that we can track external uncertainties as a third source of errors. Consider again the gas state equation example from Section 2, especially the following two lines:

```
val N = 1000 +/− 5
val f = (V:D) ⇒ (p + a∗(N.mid/V)∗(N.mid/V))∗(V − N.mid∗b) − k∗N.mid∗T
```

The +/− method returns an Interval, which in turn defines the mid method. Thus, the function typechecks correctly and can be passed for example to a solver, but inside the macro we can use the interval version of the parameter.

5 Evaluation

Precision Evaluation. The theorems from Section 3 provide us with sound guarantees regarding upper bounds. In practice however, we also need our method to be precise. Because our library computes error bounds and not only binary answers for assertions, we are interested in obtaining as precise error estimates as possible. We have evaluated the precision of our approach in the following way. We compute a high-precision estimate of the root(s) using a quadruple precision library [10], which allows us to compute the true error on the computed solutions with high confidence. We compare this error to the one provided by our library. The results on a number of benchmark problems chosen from numerical analysis textbooks are presented in Tables 1 and 2. We are able to confirm the error bounds specified by the user in all cases. In fact, of all the examples we tried,

Table 1. Comparison of errors for unary functions. All numbers are rounded

Problem (tolerance specified)	certified (affine)	certified (interval)	true errors
system of rods (1e-10)	7.315e-13	**1.447e-13**	1.435e-13
Verhulst model (1-e9)	4.891e-10	**9.783e-11**	9.782e-11
predator-prey model (1e-10)	7.150e-11	**7.147e-11**	7.146e-11
carbon gas state equation (1e-12)	**1.422e-17**	2.082e-17	1.625e-26
Butler-Volmer equation (1e-10)	4.608e-15	**3.896e-15**	3.768e-17
$(x/2)^2 - sin(x)$ (1e-10)	7.4e-16	**5.879e-16**	1.297e-16
$e^x(x-1) - e^{-x}(x+1)$ (1e-8)	5.000e-10	5.000e-10	5.000e-10
degree 3 polynomial (1e-7)	7.204e-9	**1.441e-9**	1.441e-9
degree 6 polynomial (1e-5)	**2.741e-14**	3.538e-14	2.258e-14

Table 2. Comparison of errors for multivariate functions. All numbers are rounded

Problem (tolerance specified)	certified (affine)	certified (interval)	true errors
stress distribution (1e-10)	3.584e-11 4.147e-11	3.584e-11 4.147e-11	3.584e-11, 4.147e-11
sin-cosine system (1e-7)	6.689e-09 6.655e-09	6.689e-09 6.655e-09	6.689e-9 6.6545e-9
double pendulum (1e-13)	**4.661e-15** **6.409e-15**	5.454e-15 7.449e-15	5.617e-17 9.927e-17
circle-parabola intersection (1e-13)	**5.551e-17** 1.110e-16	1.110e-16 1.110e-16	8.0145e-51 5.373e-17
quadratic 2d system (1e-6)	**2.570e-12** 3.025e-09	3.326e-12 3.025e-09	2.192e-12 3.024e-9
turbine rotor (1e-12)	**1.517e-13** **1.707e-13** **1.908e-14**	1.523e-13 1.724e-13 1.955e-14	1.514e-13 1.703e-13 1.887e-14
quadratic 3d system (1e-10)	**4.314e-16** **5.997e-16** **4.349e-16**	6.795e-16 1.632e-15 5.127e-16	1.2134e-16 7.914e-17 7.441e-17

our library failed only in the case of a multiple root for the reasons explained in Section 3 and never for precision reasons. We split the evaluation between the unary case and the multivariate case because of their different characteristics. All numbers are the maximum absolute errors computed. The numbers in parentheses are the tolerances given to the solvers and have been chosen randomly to simulate the different demands of the real world. We highlight the better error estimates in bold.

Note that the precision of the error estimates we obtain is remarkably good. Another perhaps surprising result of our experiments is that using interval arithmetic is generally more precise (in the unary case) or not much worse (in the multivariate case) than affine arithmetic, although the latter is usually presented as

Table 3. Average runtimes for the benchmark problems from Tables 1 and 2. Averages are taken over 1000 runs.

Problem set	solution time only	affine	interval	interval w/o optimizations	quadruple precision
unary problems	0.032ms	2.170ms	0.459ms	0.733ms	17.196ms
2D problems	0.044ms	2.779ms	0.984ms	1.240ms	4.446ms
3D problems	0.183ms	3.563ms	1.063ms	1.515ms	16.605ms

Table 4. Runtimes for individual problems. Averages are taken over 1000 runs.

Problem	affine	interval
carbon gas state equation	0.272ms	0.084ms
double pendulum problem	0.784ms	0.228ms
turbine problem	2.643ms	0.644ms
degree 3 polynomial	0.116ms	0.044ms
quadratic 2d system	0.425ms	0.200ms
quadratic 3d system	0.943ms	0.460ms

the superior approach. Indeed, for the tracking of roundoff errors we have shown affine arithmetic to provide (sometimes much) better results than interval arithmetic [5]. The reason why intervals perform as well is that for transcendental functions they are able to compute a tighter range, since affine arithmetic has to compute a linear approximation of those functions. The exceptions in the unary case are the degree 6 polynomial and the carbon gas state equation example, which confirms our hypothesis, since in that case the dependency tracking of affine arithmetic can recover some of the imprecision in the long run.

For the multivariate case, affine arithmetic performs generally better because the computation consists to a large part of linear arithmetic. Due to the larger computation cost (see Section 5), however, we leave it as a choice for the user which arithmetic to use and select interval arithmetic as a default.

Performance Evaluation. Table 3 compares the performance of our implementation when using affine, interval arithmetic, or interval arithmetic without the differentiation optimizations listed in Section 4.2. Switching off the optimizations is similar to performing automatic differentiation. We can see that our optimizations actually make a big difference in the runtimes, improving by up to 37% for unary functions and 30% for our 3D problems over pure differentiation. On the other hand, the table clearly shows that affine arithmetic is much less efficient than interval arithmetic (factor 3-4.5 approx.), so it should only be used if precision is of importance. The first column shows the runtimes for computing the solutions with Newton's method without any kind of verification. We have also included the runtimes of re-computing the root(s) in quadruple precision [10]. That is we have used approximately 64 decimal digits for all calculations of the numerical method. The runtimes illustrate that this approach

for computing trustworthy results is unsuitable from the performance point of view, and would not actually provide any guarantees on errors.

Table 4 illustrates the dependence of runtimes on the complexity (operation count and dimension) of the problems. The first three problems are those from our example section 2 and the second set comprises relatively short polynomial equations. Runtimes depend both on the type of equations, as e.g. transcendental functions are more expensive, and on the size of the system of equations. We consider the increases appropriate given the increase of complexity of the problems.

6 Related Work

We are not aware of any work for general-purpose programming languages that could verify solutions of nonlinear constraints or that provides runtime assertions that are consistent with mathematical reals. Closest to our work are self-validated methods for solving systems of non-linear equations. [17] contains a fairly complete overview and an implementation exists in the INTLAB library [16]. The main difference to our work is that these methods are solution methods that use interval arithmetic throughout the computation. In contrast, we use the theorems from Section 3 as a *verification* method that accepts solutions computed by an arbitrary method. This allows us to leverage the generally good results and efficiency of numerical methods with sound results. Moreover, our implementation performs part of the computation already at compile time, and is thus more efficient.

In the case of systems of linear equations, one can use the linearity for optimizations [13]. The presented algorithm remains an iterative solver. [8] gives an iterative refinement algorithm for linear systems that uses higher precision arithmetic to compute the residual. The techniques cannot however be translated to nonlinear systems. Since we do not compute residuals that suffer heavily from cancellation errors in our approach, we believe that the additional cost of higher precision arithmetic is not warranted in order to achieve a slightly better precision. Another related area is that of approximate computation [19,2], which uses program transformations to trade accuracy for efficiency. The error bounds are generally provided by the user in form of trusted specifications or are determined by simulations. The results show a great potential for improving computation efficiency while retaining precision sufficient for the application.

7 Conclusion

We have shown how to integrate the theory of error estimation from numerical analysis into a general-purpose programming language. This allows us to estimate how close computed numerical quantities are from the corresponding values that would be computed using idealized operations on real numbers. As a result, it is now possible to use the well-developed theory of reals to reason about the programs manipulating floating points. The expectations of the programmer

can already be validated using runtime assertions that are easy and intuitive to use for developers. Static analysis approaches can complement our solution and can be built to use the same specification language.

References

1. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: Fifth International Joint Conference on Automated Reasoning (2010)
2. Baek, W., Chilimbi, T.M.: Green: a framework for supporting energy-conscious programming using controlled approximation. In: Proc. 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (2010)
3. Burmako, E., Odersky, M., Vogt, C., Zeiger, S., Moors, A.: Sip 16: Self-cleaning macros (2012), http://scalamacros.org/documentation/specification.html
4. Dahlquist, G., Björck, A.: Numerical Methods in Scientific Computing. Society for Industrial and Applied Mathematics (2008)
5. Darulova, E., Kuncak, V.: Trustworthy numerical computation in Scala. In: Proc. 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (2011)
6. de Figueiredo, L.H., Stolfi, J.: Self-Validated Numerical Methods and Applications. In: IMPA/CNPq, Brazil (1997)
7. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an Industrial Use of FLUCTUAT on Safety-Critical Avionics Software. In: Alpuente, M., Cook, B., Joubert, C. (eds.) FMICS 2009. LNCS, vol. 5825, pp. 53–69. Springer, Heidelberg (2009)
8. Demmel, J.W., Hida, Y., Kahan, W., Li, X.S., Mukherjee, S., Riedy, E.J.: Error Bounds from Extra Precise Iterative Refinement. Technical report, EECS Department, University of California, Berkeley (2005)
9. Griewank, A.: A mathematical view of automatic differentiation. Acta Numerica 12, 321–398 (2003)
10. Hida, Y., Xiaoye, S.L., Bailey, D.H., Kaiser, A.: Quad Double computation package (2012), http://crd-legacy.lbl.gov/~dhbailey/mpdist/
11. Krawczyk, R.: Newton-Algorithmen zur Bestimmung von Nullstellen mit Fehlerschranken. Computing 4, 187–201 (1969)
12. Moore, R.: Interval Analysis. Prentice-Hall (1966)
13. Nguyen, H.D., Revol, N.: Solving and Certifying the Solution of a Linear System. Reliable Computing (2011)
14. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide. Artima Incorporation (2008)
15. Quarteroni, A., Saleri, F., Gervasio, P.: Scientific Computing with MATLAB and Octave, 3rd edn. Springer (2010)
16. Rump, S.M.: INTLAB - INTerval LABoratory. In: Developments in Reliable Computing. Kluwer Academic Publishers (1999)
17. Rump, S.M.: Verification methods: rigorous results using floating-point arithmetic. In: Proc. 2010 International Symposium on Symbolic and Algebraic Computation, pp. 3–4 (2010)
18. Woodford, C., Phillips, C.: Numerical Methods with Worked Examples, vol. 2. Springer (2012)
19. Zhu, Z.A., Misailovic, S., Kelner, J.A., Rinard, M.: Randomized accuracy-aware program transformations for efficient approximate computations. In: Proc. 39th ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (2012)

Profiling Field Initialisation in Java

Stephen Nelson, David J. Pearce, and James Noble

Victoria University of Wellington
Wellington, New Zealand
{stephen,djp,kjx}@ecs.vuw.ac.nz

Abstract. Java encourages programmers to use constructor methods to initialise objects, supports **final** modifiers for documenting fields which are never modified and employs static checking to ensure such fields are only ever initialised inside constructors. Unkel and Lam observed that relatively few fields are actually declared final and showed using static analysis that many more fields have final behaviour, and even more fields are *stationary* (i.e. all writes occur before all reads). We present results from a runtime analysis of 14 real-world Java programs which not only replicates Unkel and Lam's results, but suggests their analysis may have under-approximated the true figure. Our results indicate a remarkable 72-82% of fields are stationary, that **final** is poorly utilised by Java programmers, and that initialisation of immutable fields frequently occurs after constructor return. This suggests that the **final** modifier for fields does a poor job of supporting common programming practices.

1 Introduction

The notion of immutability has been well-studied in the programming language community (e.g. [1,2,3]). Modern statically typed languages, such as Java and C#, typically support immutable fields (e.g. **final** in Java) though, curiously, explicit support for immutable classes is often missing. Bloch advises Java programmers to "Favour Immutability" as "Immutable classes are easier to design, implement, and use than mutable classes" [4]. This leads to a natural question of how well immutability modifiers (such as Java's **final**) match common programming idioms.

To examine this question, Unkel and Lam developed the term *stationary field* to describe fields which are never observed to change, that is, all writes precede all reads [5]. Their intuition was that programmers are often forced to perform *late initialisation* of objects (i.e initialisation after the constructor has returned), meaning some fields cannot be declared **final**. A common idiom where this happens is with the initialisation of cyclic data structures [6].

Unkel and Lam performed a *static analysis* over a corpus of 26 Java applications, and found that 40-60% of Java fields were stationary. Their analysis was necessarily conservative and, hence, under-reported the number of stationary fields. In this paper, we report on an experiment to identify stationary fields using *runtime profiling*. Our results from 14 Java applications indicates that

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 292–307, 2013.
© Springer-Verlag Berlin Heidelberg 2013

72-82% of fields are stationary — thereby supporting the conclusion of Unkel and Lam, but suggesting that it is an underestimate.

2 Background

Java provides a special modifier for immutable fields: **final**. Java itself ensures that fields annotated with **final** are only modified once and only within constructors. Java encourages programmers to use constructors for initialising fields and establishing invariants. Constructors are distinct from regular methods in one significant way: they can initialise **final** fields. Unfortunately, programmers are sometimes forced (or voluntarily choose) to initialise fields late (i.e. after the constructor has completed). This prevents such fields from being marked **final** *even when they are designed to be immutable.* The following illustrates a common patten which exemplifies this scenario:

```
abstract class Parent {
  private final Child child;
  public Parent(Child c) { this.child = c; }
}

abstract class Child {
  private Parent parent; // cannot be marked as final
  public void setParent(Parent p) { this.parent = p; }
}
```

The programmer intends that every Parent has a Child and vice-versa and, furthermore, *that these do not change for the life of the program.* He/she has marked the field Parent.child as **final** in an effort to enforce this. However, he/she is unable to mark the field Child.parent as **final** because one object must be constructed before the other. We have marked Parent and Child **abstract** to indicate the intention that different subclasses will be used.

In the above example, the method Child.setParent(Parent) is used as a *late initialiser.* This is a method which runs after the constructor has completed and before which the object is not considered properly initialised. Once the Child.parent field is late initialised by this method, the programmer does not intend that it will change again. Such a field — where all reads occur after all writes — is referred to as *stationary* [5]. Another common situation where this arises is for classes with many configurable parameters. In such case, the programmer is faced with providing a single large constructor and/or enumerating many constructors with different combinations of parameters. Typically, late initialisation offers a simpler and more elegant solution.

3 Implementation

We have developed a profiler called *rprof* to track (amongst other things) all reads and writes to object fields in an executing program. The key advantages of

rprof are that it runs on a commodity JVM (e.g. Oracle's HotSpot JVM), catches reads/writes to almost all objects (including those in the standard library), profiles large real-world applications with manageable overhead and processes event traces (containing potentially billions of events) using a parallel, distributed map/reduce computation. There are four main components:

- *Agent*: a JVMTI [7] C++ agent loaded by the JVM running the target application.
- *Profiler*: a Java application running in a separate JVM that performs bytecode rewriting, and provides other utility functions for the agent.
- *Workers*: Java applications that aggregate the event stream and handle storing the results.
- mongodb: a commercial *nosql* database server that the profiler and worker applications use to store persistent data.

The profiler and workers perform tasks in parallel using multiple threads on multiple computers, minimising overheads on the profiled application. To enable profiling, *rprof* performs bytecode rewriting on the target application. When the JVM loads a class, it is intercepted by the agent which inserts instrumentation using the ASM bytecode modification library [8]. To increase the range of objects which can be profiled, bytecode rewriting is performed by the profiler in a separate JVM from the target application. The agent passes classes to be rewritten to the profiler (potentially across the network) running in another JVM which rewrites them, and passes them back. Without this, classes needed by the ASM library could not be profiled as they would have to be loaded before rewriting.

Due to lack of space, unfortunately we cannot discuss every aspect of the *rprof* profiler. A more complete discussion of the operation of *rprof* can be found in [9]. We will now give a high-level overview of the main issues.

3.1 Object Tracking

The first challenge faced in *rprof* is the unique identification of objects in the target application. Three options exist for uniquely identifying objects within the JVM:

- *Using Object References.* This approach is commonly used with weak references to ensure garbage collection proceeds as normal (see e.g. [10,11]). Since we store profiling data in the mongodb database, we require a concrete ID rather than a reference. Unfortunately, Java itself provides no easy mechanism for converting references into IDs[1].

[1] Some works (e.g. [12,13]) employ System.identityHashCode(Object) to generate object IDs. The value returned from this method is derived from the object's physical address, and then stored for subsequent calls. Consequently, it is unsuitable for uniquely identifying objects because, in standard VMs (using generational garbage collectors) objects initially reside within the nursery. This is a relatively small region of memory and we found many live objects which shared the same identityHashCode().

- *Using Physical Memory Addresses.* Since object references correspond to physical memory addresses in the JVM, a logical option is to use them as unique IDs. Through the JVMTI it is possible to convert an object reference into a physical address. Whilst this may seem straightforward, it is fraught with difficulty since an object can change its physical location during garbage collection. In other words, we would need to intercept garbage collection events to determine which objects were moved and now have a new physical address (hence, ID).
- *Storing unique ID's with every object.* The JVMTI provides a mechanism whereby agents can associate a 64bit (long) tag with any object. The JVM maintains this tag and handles all issues related to garbage collection, etc.

In our context, the only viable solution is to associate unique IDs with objects via the JVMTI — which is the approach taken in *rprof*.

Tracking System Objects. Before the JVM loads native agents, it performs some basic bootstrapping including loading classes such as java.lang.Object and java.lang.String. Agents have a chance to modify previously loaded classes, but JVMTI facilities such as object tagging remain unavailable until the JVM reaches the end of its bootstrapping phase. Once the JVM has completed bootstrapping, the agent uses JVMTI to iterate over all the Java objects, and adds unique ID tags to them.

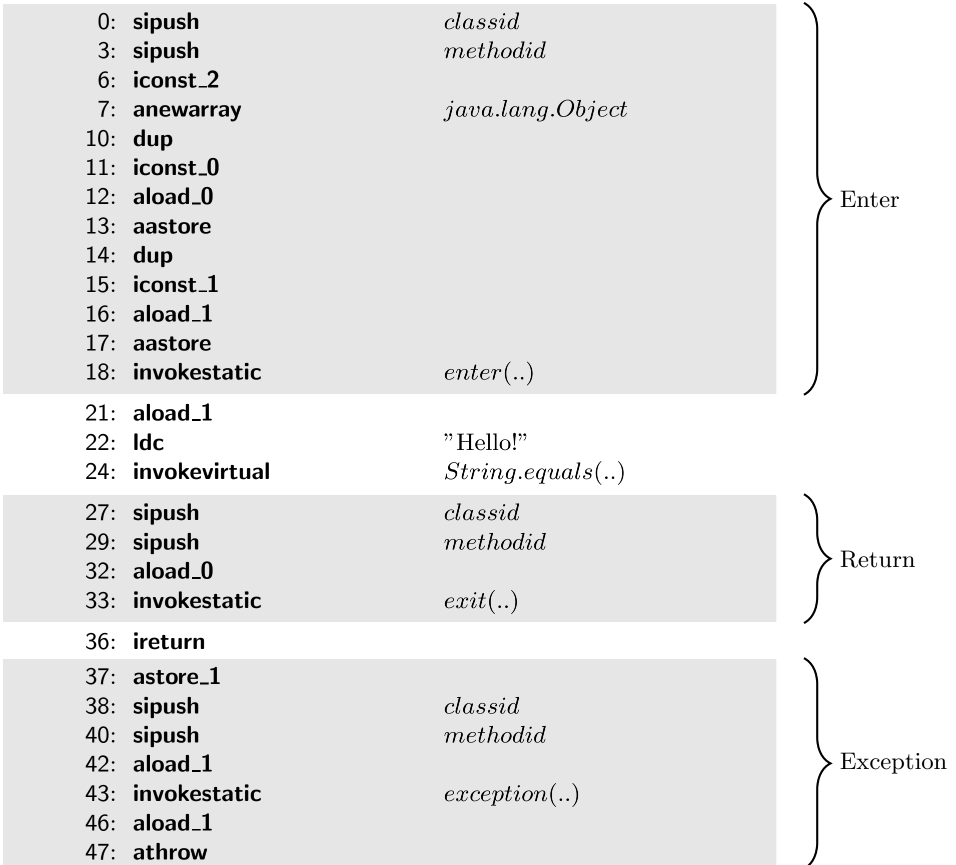
3.2 Tracking Methods and Constructors

rprof is capable of generating events for all method and constructor calls and returns (inc. exceptional returns), although this analysis only requires method return tracking for constructors. Since the JVMTI does not provide any facility for tracking method calls, *rprof* uses bytecode modification to instrument classes as the JVM loads them. Consider the following Java method:

```
public boolean isHello(String message) { return message.equals("Hello!"); }
```

This takes a string as input and returns true if the message is "Hello!". *rprof* can generate three different events for this method: a *'Method Enter'* event, a *'Method Return'* event, and/or an *'Exceptional Return'* to catch any thrown exceptions (e.g. because message is null). Figure 1 shows how *rprof* instruments this method by inserting bytecodes to generate these three events. For example, the inserted code at the beginning pushes class and method identifiers onto the stack, constructs an array containing the arguments to the method, then invokes a static *rprof* method for tracking method entry events. *rprof* handles method returns similarly to method calls. *rprof* generates exceptional returns by wrapping the method body in a try block, and inserting a finally handler at the end of the method which signals to *rprof* an exceptional return. *rprof* inserts the try block last so that any other catch or finally blocks run first. If a block consumes the exception and returns normally then *rprof*'s exceptional return handler will not run.

```
public boolean isHello(java.lang.String);
  flags: ACC_PUBLIC
  Code:
    stack=6, locals=2, args_size=2
```




```
  Exception table:
    from   to   target type
    0      37   37     Class java/lang/Exception
```

Fig. 1. The byte code resulting from modifying an example method to track method entry, exit, and exceptional return

3.3 Tracking Fields

rprof supports tracking both field writes and field reads. This is simpler than for methods, since the JVMTI provides a callback mechanism to notify agents of these events. The callback provides a reference to the object that owns the field, the value of the read or write, and the JVM-internal field ID. Unfortunately, JVM field IDs are unique to a given class, but are not guaranteed unique across all classes. *rprof* requires program-wide unique field IDs for persistence and, hence, maintains a map between JVM-internal field IDs and *rprof*'s persistent field IDs. Finally, *rprof* tracks all objects created within the JVM, but it is not able to track all fields. *rprof* does not track fields of the following classes:

– java.nio.charset.CharsetDecoder, java.nio.charset.CharsetEncoder, java.util.zip.ZipFile. These three classes use some form of JVM optimisation which causes segfaults if *rprof* tracks them.
– java.lang.Throwable. The JVM generates an additional field at runtime which causes off-by-one errors in our tracking code.
– java.lang.String. Excluded because it is so common: Strings are actually immutable but they use internal fields to track access behaviour and generated properties, resulting in a disproportionate number of events which are not interesting for this experiment.

3.4 Data Aggregation and Analysis

rprof event streams contain billions of events which would take days or even weeks to store on disk. Reducing this event stream to a more compact form suitable for analysis or visualisation is a computational challenge. To address this, *rprof* processes the event stream as it is generated using a map-reduce style computation. *rprof* parallelises this computation across a cluster of machines, dramatically reducing the time and space taken to store the results.

The worker processes operate on the event stream as it is being generated, reducing it to a form amenable for analysis or visualisation. This reduction is specific to the experiment being performed, and discards information not directly relevant. For example, in the experiments discussed in this paper, it is not necessary to count how many field accesses there were — we only need to note when the first and last field reads and writes occurred.

As the profiled application runs the agent generates *event records*. These are mapped and reduced to *instance records* which, in turn, are then mapped and reduced to *result records*. Figure 2 presents the information contained in these records (roughly speaking) for the experiment presented in this paper. On the left of the figure, we see the event records emitted by the agent. One of these is emitted for every event being monitored in the profiled application which includes *constructor method exit* and *field read/write*. These events are then mapped and reduced to a set of complete instance records (shown in the middle), which capture information about a given object relevant to this experiment. The complete instance records are then mapped and reduced again to form the final result records (shown on the right).

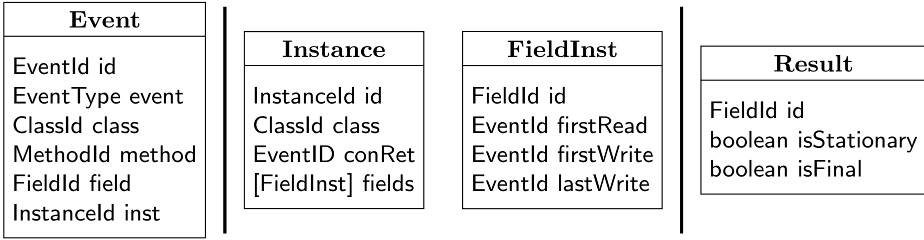


Fig. 2. Illustrating the event records emitted by the agent (left), which are mapped and reduced to instance records (middle) and then again into the final result records (right)

```
// Map event into instance record
void map(Event e) {
  InstanceId id = e.inst;
  Instance inst = new Instance();
  switch(e.type) {
    case FIELD_READ:
      FieldInst fi = new FieldInst(e.field);
      fi.firstRead = fi.lastRead = e.id;
      inst.fields.add(fi);
      break;
    case FIELD_WRITE:
      FieldInst fi = new FieldInst(e.field);
      fi.firstWrite = fi.lastWrite = e.id;
      inst.fields.add(fi);
      break;
    case METHOD_RETURN:
    case METHOD_EXCEPTION:
      if(isConstructor(e.clazz,e.method)) {
          inst.conRet = e.id;
          break;
      }
    default:
      return; // don't emit anything
  }
  emit(id, inst);
}
```

```
// Reduce two instances with same ID
Instance reduce(Instance l, Instance r) {
  // Update constructor return ID
  l.conRet = max(l.conRet,r.conRet);

  // Merge field instance records
  for(int i=0;i<l.fields.size();++i) {
    FieldInst fi = l.fields.get(i);
    for(int j=i+1;j<l.fields.size();++j) {
      FieldInst fj = l.fields.get(j);
      if(fi.id == fj.id) {
        fi.firstRead = min(fi.firstRead,fj.firstRead);
        fi.firstWrite = min(fi.firstWrite,fj.firstWrite);
        fi.lastWrite = max(fi.lastWrite,fj.lastWrite);
        l.fields.remove(j);
        j = j − 1;
      }
    }
  }

  return l;
}
```

Fig. 3. Illustrating how Event records are mapped to (incomplete) Instance records, which are then reduced to form complete records. Not every Event maps to an Instance record; for example, most method entry events are ignored, with only method return and exceptional return events on constructors emitting Instances. The methods min(EventID,EventID) and max(EventID,EventID) operated as expected — by return the earlier (resp. later) of the two parameters and handling null values correctly. Finally, please note that, in practice, these methods are further optimised for performance.

Consider the process of converting Event records into Instance records. Initially, each Event record is either ignored (if not relevant) or *mapped* to an (incomplete) Instance record. These Instance records are then *reduced* to form complete instance records. Here, Instance.conRet gives the EventID for the object's constructor return, whilst Instance.fields contains records for its fields. For each field, firstWrite and lastWrite give the EventIDs for the first and last write and, similarly, for firstRead. Figure 3 illustrates the map and reduce procedures.

Consider now the process for converting complete Instance records into complete Result records. This is similar to before. Given a complete instance record we can determine which of its fields were stationary (i.e. all writes before all reads) and/or final (i.e. one write which occurred before constructor return). We then reduce all Result records for a given class to determine which fields were stationary and final *across all instances*. The reduce procedure is thus:

```
// Reduce result records with same field ID
Result reduce(Result left, Result right) {
  left.isStationary &= right.isStationary;
  left.isFinal &= right.isFinal;
  return left;
}
```

Here, we see how two Result records with the same FieldID are reduced. All Result records for a given field are reduced to a single Result record capturing its stationary and final status across all instances.

4 Experimental Results

We now present our experimental results looking at final and stationary fields. We begin with a more detailed definition of these terms.

Final (F). A *Final* field is an *object instance field* which is modified once, before the object's constructor method returns. A field is not final if, for any object which reads the field, the field is written to after the object's constructor returns or the field is written to more than once. Final fields may be *Declared Final (dF)* or *Undeclared Final (uF)*. A declared final field is any field whose declaration in Java code is annotated with the final modifier. A field which is not annotated with this modifier but nevertheless conforms to this definition is undeclared final.

Stationary (S). A *Stationary* field is an *object instance field* which is not modified after it has been read. A field is not stationary if there exists an object which modifies that field after it has been read. The set of stationary fields has no relationship with that of declared or undeclared final fields. A field which has been declared final may have its state read before it is initialised (while it is in its default state). This is valid behaviour for declared final fields but not stationary fields. Likewise, stationary fields may be initialised after constructor return, which is not valid behaviour for final fields.

Table 1. List of programs in the *Dacapo* benchmarks suite (dacapo−9.12−bach) including a brief summary (from [14]). Also included are statistics on each benchmark obtained using *rprof* giving the number of classes loaded during the experiment run (including interfaces) and the number of methods they contained (including static).

Name	Description	Classes	Methods
avrora	Simulates a number of programs run on a grid of AVR microcontrollers.	999	10685
batik	Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.	1814	21710
eclipse	Executes some of the (non-gui) jdt performance tests for the Eclipse IDE.	2653	37949
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file.	1703	20133
h2	Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application.	934	14622
jython	Inteprets the pybench Python benchmark.	2953	34167
luindex	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible.	788	10887
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.	701	9640
pmd	Analyzes a set of Java classes for a range of source code problems.	1328	18281
sunflow	Renders a set of images using ray tracing.	907	12854
tomcat	Runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages.	2373	32369
tradebeans	Runs the daytrader benchmark via a Jave Beans to a GERONIMO backend with an in memory h2 as the underlying database.	8155	96250
tradesoap	Runs the daytrader benchmark via a SOAP to a GERONIMO backend with in memory h2 as the underlying database.	8246	97026
xalan	Transforms XML documents into HTML.	1125	14260

4.1 Benchmarks

Unkel and Lam analysed a selection of Java programs and also the SpecJVM98 benchmark suite. Unlike their static analysis, our dynamic analysis needs to execute each program with a set of inputs that will exercise the program's functionality in a reproducible manner. We decided to analyse the *dacapo* benchmark suite, a compilation of non-trivial real world Java applications designed for benchmarking that includes non-trivial inputs for each application [14]. The dacapo suite consists of the 14 applications listed in Table 1. Each benchmark

was executed using the default input size for a single iteration. We used the following command to execute benchmarks:

```
java [rprof−opts] −Xint −Xmx1024m −jar dacapo−9.12−bach.jar −n 1 −t 1 [benchmark]
```

4.2 Experimental Setup

The benchmarks were profiled executing on an Opteron 254 (2.8GHz) dual-CPU machine with 4GB of memory running Ubuntu 10.04.3 LTS (64 bit server) using OpenJDK 1.8.0-ea-b37[2]. We used the preview Java 8 build because our analysis is not stable on previous JDK versions. OpenJDK 1.8.0-ea-b37 includes a bug fix for a problem with JVMTI which we identified and reported[3].

4.3 Results

The results in this section are directly comparable to the Unkel and Lam's work on stationary fields so we use a consistent format to present our results [5]. Figure 4 presents the results of our analysis for all fields. The first column shows the name of the benchmark, the second shows the total number of unique fields contributed by each benchmark (i.e. which were read or written at least once during the run), all other columns show the percentage of the total number of fields in that category (rounded to whole numbers). Results are separated broadly into stationary and non-stationary fields, then into *declared final* (dF), *undeclared final* (uF) and *not final* (¬F). The final three columns show summary information: the total number of *declared final* (dF) fields, *final* fields (F = dF ∪ uF), and *stationary* fields (S).

Figure 4 shows that between 70% and 86% of the fields declared in dacapo benchmarks are *stationary* (S) and between 50% and 68% are *final* (F). The number of final fields which are declared final (dF) varies between benchmarks but, in most cases, is less than half the number of fields whose behaviour was observed to be final.

Finally, Figures 5 and 6 show the results across fields of reference type and fields of primitive type. The format is largely the same as before, comparing final field behaviour between stationary and non-stationary fields in each case.

4.4 Discussion

Comparing our results to Unkel and Lam's [5], we can make the following main observations:

1. **Declared v Undeclared Final.** Consistent with the findings of Unkel and Lam, we find many undeclared final fields. This suggests that programmers

[2] The batik benchmark which relies on a proprietary jpeg class which is not included in the pre-release JDK 8 build. For this benchmark we used Oracle JDK 1.7.0_03-b04.

[3] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7162645

Program	Total	S (%) dF	S (%) uF	S (%) ¬F	¬S (%) dF	¬S (%) uF	¬S (%) ¬F	% dF	% F	% S
avrora	1,702	36	31	16	0	0	17	36	67	82
batik	3,272	8	52	17	0	0	23	8	60	77
eclipse	6,779	10	37	25	0	0	28	10	46	72
fop	3,390	9	43	28	0	0	19	9	53	81
h2	2,014	17	39	22	0	0	22	17	56	78
jython	2,501	14	46	19	0	0	22	14	60	78
luindex	1,674	18	38	16	0	0	26	18	57	73
lusearch	1,303	14	43	21	0	0	21	14	58	78
pmd	1,962	14	41	24	0	0	20	14	55	79
sunflow	1,831	14	43	20	0	0	23	14	57	77
tradebeans	15,404	25	35	21	0	0	19	25	60	81
tradesoap	15,611	25	35	21	0	0	19	25	60	81
tomcat	6,279	9	40	25	0	0	26	9	50	74
xalan	2,063	11	41	27	0	0	21	11	52	78
Total	65,785	18	38	22	0	0	21	18	57	78

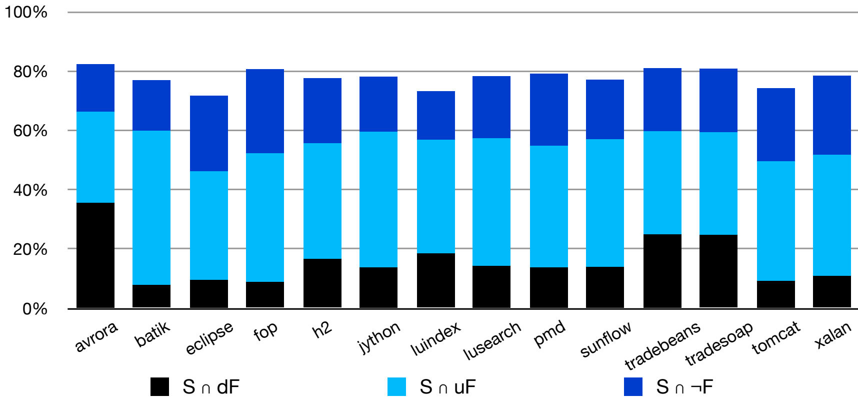


Fig. 4. Stationary vs Final for All Fields

are not making good use of the final modifier. The most obvious reason it that some programmers may simply be "lazy" and choose not to use it even when they could. However, other possibilities exist. For example, such undeclared final fields may be protected, where the programmer anticipated subclasses that would mutate them but which never eventuated in the given application.

2. **Stationary v Final.** Our results show a significantly higher proportion of stationary fields than reported by Unkel and Lam. Furthermore, we detect far fewer fields which are final but not stationary[4] — indeed, fewer than 0.5% in Figure 4. These results suggests that the final modifier is doing a poor job of supporting common programming practices.

[4] Such behaviour is possible in Java if a field read occurs indirectly via dynamic dispatch from a super-constructor [6].

Program	Total	S (%)			¬S (%)			%	%	%
		dF	uF	¬F	dF	uF	¬F	dF	F	S
avrora	880	47	31	12	0	0	9	47	78	90
batik	1,593	9	54	17	0	1	19	9	64	81
eclipse	3,324	15	43	20	0	0	22	15	58	78
fop	1,760	12	45	28	0	0	14	12	58	86
h2	990	21	41	20	0	0	18	21	62	82
jython	1,354	16	49	18	0	0	16	16	66	83
luindex	812	28	42	16	0	0	14	28	70	85
lusearch	598	20	48	18	0	1	14	20	69	86
pmd	1,048	18	45	22	0	1	15	18	63	85
sunflow	831	18	47	20	0	0	15	18	66	85
tradebeans	9,451	33	34	19	0	0	14	33	67	86
tradesoap	9,619	33	34	19	0	0	14	33	67	86
tomcat	3,393	12	43	26	0	0	18	12	55	82
xalan	1,095	14	46	24	0	1	16	14	60	83
Total	36,748	25	39	20	0	0	16	25	64	84

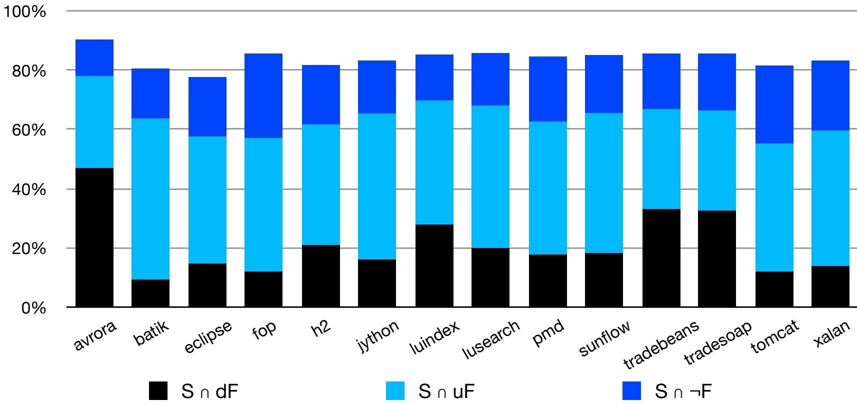


Fig. 5. Stationary versus Final for fields with reference type

3. **Reference v Primitives.** Our results show that reference fields are more likely than primitive fields to be declared final, undeclared final, and/or stationary. This is consistent with Unkel and Lam, though the differences between groups is more modest in our results. This suggests a distinct difference in the way programmers treat reference and primitive fields.

4.5 Threats to Validity

Any experiment of this nature has limitations with respect to the scope of the experiment itself. We now identify the main limitations:

- **Benchmark Inputs.** As discussed in Section 4.1, each of our benchmarks was profiled using the workflow provided by Dacapo. This ensures that our results are reproducible, but effectively constitutes running the benchmark

Program	Total	S (%)			¬S (%)			%	%	%
		dF	uF	¬F	dF	uF	¬F	dF	F	S
avrora	822	23	30	20	0	0	26	23	54	74
batik	1,679	6	50	17	0	0	26	6	56	74
eclipse	3,455	5	30	31	0	0	34	5	35	66
fop	1,630	6	42	28	0	0	25	6	47	75
h2	1,024	12	38	23	0	0	26	12	50	73
jython	1,147	11	41	19	0	0	28	11	53	72
luindex	862	10	35	17	0	0	38	10	45	62
lusearch	705	10	38	24	0	0	28	10	48	72
pmd	914	9	36	27	0	0	27	9	46	73
sunflow	1000	11	40	20	0	0	29	11	50	70
tradebeans	5,953	12	36	25	0	0	26	12	49	73
tradesoap	5,992	12	36	25	0	0	27	12	48	73
tomcat	2,886	6	37	22	0	0	34	6	43	65
xalan	968	8	36	30	0	0	27	8	43	73
Total	29,037	10	37	25	0	0	29	10	47	71

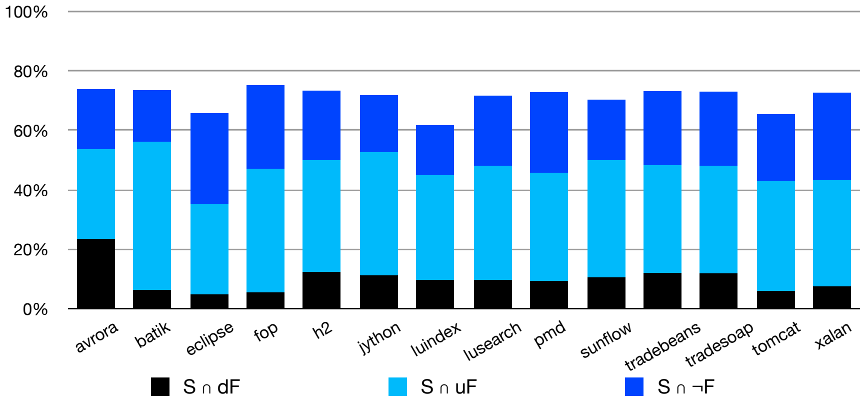


Fig. 6. Stationary versus Final for fields with primitive type

using a single set of inputs. Clearly, we cannot generalise program behaviour from one set of inputs to all possible inputs, and it is possible that the Dacapo inputs are not representative of the benchmark program's general behaviour. In particular, fields identified as undeclared final or stationary may receive different classifications for different inputs. This contrasts with the work of Unkel and Lam, whose static analysis was a conservative approximation of all possible program behaviours.

– **Benchmark Scope.** The Dacapo benchmark suit is well-known and widely used for experiments such as this. However, it remains unclear how representative Dacapo is of the general population consisting of all programs. Indeed, there is work which suggests Dacapo programs do have observably different behaviour from other benchmark suites [15].

Despite these limitations, we believe our work compliments that of Unkel and Lam. Being a conservative static analysis, their results necessarily *under approximate* the true number of stationary fields. In contrast, being a runtime analysis our results necessarily *over approximate* the true number of stationary fields. This provides insight into how conservative the results of Unkel and Lam were.

5 Related Work

Various OO languages have support for immutability via, for example, *final* or *const* fields. CLU [16] also supports immutable versions of primitive data structures — although clusters (classes) are always mutable. A similar design has been adopted in Scala, where the library provides mutable and immutable versions of most collections [17].

As discussed already, Unkel and Lam also examined stationary fields [5]. Unlike us, they employed a static analysis which is necessarily conservative. For a corpus of 26 Java applications, they found that 40-60% of Java fields were stationary which is a similar, but consistently lower, figure than we have found. Given that their result is an under-approximation and ours an over-approximation, it seems reasonable to conclude that the true figure lies somewhere inbetween. Earlier, Porat et al. [18] conducted a similar analysis looking for "deeply immutable" fields (where neither the field itself nor any object reachable from that field is modified after the object's constructor completes) and found that around 60% of `static` fields were immutable. These results compare with our (dynamic) profile finding that a large fraction of Java objects are immutable after full construction. Previously, we examined object behaviours and found significant differences depending on whether or not they entered a Java collection [19]. This is particularly relevant for collections such as e.g. HashSet and HashMap which restrict how contained objects may be modified.

Pechtchanski and Sarkar present an interesting study of field immutability [20]. Their work includes a framework for specifying and verifying both shallow and deep field immutability, as well as a runtime study that found that at least 61% of field accesses were immutable, a similar property to stationary fields. Their analysis computed exhaustive lists of field and array read and write operations using a modified Jikes JVM, but their analysis was limited to much smaller programs than the Dacapo suite. Nevertheless, they find similar results to ours and additionally use those results for performance optimisations, yielding 5-10% speedups for some benchmarks.

Several works have looked at permitting type-safe late initialisation of objects in a programming language. Summers and Müller presented a lightweight system for type checking delayed object initialiation which is sufficiently expressive to handle cyclic initialisation [6]. Fähndrich and Xia's Delayed Types [2] use dynamically nested regions in an ownership-style type system to represent this post-construction initialisation phase, and ensure that programs do not access uninitialised fields. Haack and Poll [1] have shown how these techniques can be applied specifically to immutability, and Leino et al. [3] show how ownership transfer (rather than nesting) can achieve a similar result. Qi and Myers'

Masked Types [21] use type-states to address this problem by incorporating a list of uninitialised fields ("masked fields") into object types. Gil and Shragai [22] address the related problem of ensuring correct initialisation between subclass and superclass constructors within individual objects. Based on our results, we would expect such type systems to be of benefit to real programs.

6 Conclusion

We have reported the results from an experiment examining final and stationary fields across 14 real-world benchmarks. Our work compliments the earlier work of Unkel and Lam which employed static analysis, and supports their general conclusions. However, our findings indicate a larger proportion of stationary fields which, in part at least, stems from the differences between our approaches.

Like Unkel and Lam, we conclude that final fields annotations are used far less often than they could be, while a *stationary* annotation could be used even more. The extremely high number of stationary fields that we found (around 80%) suggests that language authors should make fields stationary by default, while VM authors should optimise for immutability. These results also support the use of type systems for immutability, e.g. Masked Types [21] could be used to track fields requiring additional initialisation.

Finally, there are many additional studies that are motivated from these results. For example, it would be interesting to examine whether protection modifiers (e.g. public, protected, private) had any bearing on the likelihood of a field being declared or undeclared final. It would also be interesting to extend our analysis to detect deep stationary behaviour, similar to the smaller analysis of Pechtchanski and Sarkar [20].

References

1. Haack, C., Poll, E.: Type-based object immutability with flexible initialization. Technical Report ICIS-R09001, Radboud University Nijmegen (January 2009)
2. Fähndrich, M., Xia, S.: Establishing object invariants with delayed types. In: OOPSLA, pp. 337–350 (2007)
3. Leino, K.R.M., Müller, P., Wallenburg, A.: Flexible Immutability with Frozen Objects. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 192–208. Springer, Heidelberg (2008)
4. Bloch, J.: Effective Java. Prentice Hall PTR (2008)
5. Unkel, C., Lam, M.S.: Automatic inference of stationary fields: a generalization of Java's final fields. In: POPL, pp. 183–195 (2008)
6. Summers, A.J., Müller, P.: Freedom before commitment: a lightweight type system for object initialisation. In: OOPSLA, pp. 1013–1032. ACM (2011)
7. Jdk 6 java virtual machine tool interface (JVMTI) (2008)
8. Bruneton, E.: Asm 3.0 a java bytecode engineering library (2007), http://download.forge.objectweb.org/asm/asmguide.pdf
9. Nelson, S.: Measuring Equality and Immutability in Object-Oriented Programs. PhD thesis, School of Engineering and Computer Science, Victoria University of Wellington, NZ (submitted, 2012)

10. Agesen, O., Garthwaite, A.: Efficient object sampling via weak references. In: Proc. ISMM, pp. 121–126 (2000)
11. Pearce, D.J., Webster, M., Berry, R., Kelly, P.H.J.: Profiling with AspectJ. Software: Pracice and Experience 37(7), 747–777 (2007)
12. Goldberg, A., Havelund, K.: Instrumentation of java bytecode for runtime analysis. In: FTfJP (2003)
13. Xu, G.H., Rountev, A.: Precise memory leak detection for java software using container profiling. In: ICSE, pp. 151–160. ACM (2008)
14. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: java benchmarking development and analysis, pp. 169–190 (2006)
15. Mitchell, N.: The Runtime Structure of Object Ownership. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 74–98. Springer, Heidelberg (2006)
16. Liskov, B., Guttag, J.V.: Abstraction and Specification in Program Development. MIT Press/McGraw-Hill (1986)
17. Odersky, M.: Programming in Scala. Artima, Inc. (2008)
18. Porat, S., Biberstein, M., Koved, L., Mendelson, B.: Automatic detection of immutable fields in Java. In: Proc. CASCON (1990)
19. Nelson, S., Pearce, D.J., Noble, J.: Understanding the Impact of Collection Contracts on Design. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 61–78. Springer, Heidelberg (2010)
20. Pechtchanski, I., Sarkar, V.: Immutability specification and its applications. Concurrency and Computation: Practice and Experience, 639–662 (2005)
21. Qi, X., Myers, A.C.: Masked types for sound object initialization. In: POPL, pp. 53–65 (2009)
22. Gil, J(Y.), Shragai, T.: Are We Ready for a Safer Construction Environment? In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 495–519. Springer, Heidelberg (2009)

Defense against Stack-Based Attacks Using Speculative Stack Layout Transformation

Benjamin D. Rodes, Anh Nguyen-Tuong, Jason D. Hiser, John C. Knight, Michele Co, and Jack W. Davidson

Department of Computer Science, University of Virginia,
85 Engineer's Way, P.O. Box 400740, Charlottesville, VA 22904
{bdr7fv,nguyen,hiser,jck,mc2zk,jwd}@virginia.edu
http://www.cs.virginia.edu

Abstract. This paper describes a novel technique to defend binaries against intra-frame stack-based attacks, including overflows into local variables, when source code is unavailable. The technique infers a specification of a function's stack layout, i.e., variable locations and boundaries, and then seeks to apply a combination of transformations, including variable reordering, random-sized padding between variables, and placement of canaries. To overcome the imprecision of static binary analysis, yet be as aggressive as possible in the transformations applied to the stack layout, the technique is speculative. A stack frame is aggressively transformed based on static analysis, and the validity of inferred stack layout is assessed through regression testing. If a transformation changes a program's semantics because of imprecision in the inference of the stack layout, a less aggressive layout is inferred until the transformed program passes the supplied regression tests. We present an overview of the technique and preliminary results of its feasibility and security effectiveness.

Keywords: artificial diversity, stack layout transformation, run-time verification, buffer overflow, non-control-data attacks, security attacks.

1 Introduction

We present a technique, Stack Layout Transformation (SLX), for the runtime transformation of function stack layouts to protect against stack-based attacks, including intra-frame overflows and non-control data attacks. SLX is designed to operate directly on binaries and transforms the stack layout using a combination of random-length padding between variables, reordering of variables, and placement of canaries (Figure 1).

If source code were available, fine-grained transformations would be relatively simple to effect since the location, type, and size of variables are directly specified [2,3,8]. For example, by default, the gcc compiler reorders buffers to be above local variables and inserts a canary to protect the frame pointer and return address. Unfortunately, precise stack structure information is not completely recoverable once a program is compiled into its binary form. Thus, existing methods

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 308–313, 2013.
© Springer-Verlag Berlin Heidelberg 2013

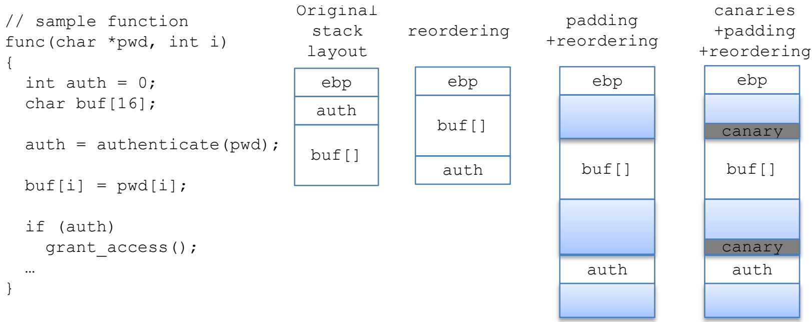


Fig. 1. Stack layout transformations

to protect the stack that operate directly on binaries typically do so at a coarse level, unable to protect individual variables within a stack frame [1,5,7].

The novelties of SLX lie in the fact that it does not rely on source and in its speculative approach to generating and validating hypothesized stack layouts to enable fine-grained transformations such as those shown in Figure 1. The end result of the SLX transformation process is a binary where each function is transformed as aggressively as possible based on a validation process. In many cases, SLX is able to infer variable information at a sufficient level of granularity to protect against intra-frame stack-based attacks such as overflows into local variables and non-control-data attacks [4].

The next section presents an overview of the technique and preliminary experimental assessments of its efficacy and overhead.

2 Stack Layout Transformation (SLX)

The current SLX prototype has three phases: (a) candidate function selection, (b) hypothesis generation/refinement, and (c) hypothesis validation. SLX only attempts to transform functions for which it can find common compiler-generated code patterns such as stack allocation and deallocation patterns.

Stack layout hypotheses for a function are generated based on static analysis of the code for the function and a set of stack layout inference heuristics. Currently, SLX uses four variable layout inference heuristics to produce stack layout hypotheses. These heuristics are based on relatively naïve memory access pattern observations in the disassembly: each offset into stack memory is assumed to be a potential variable boundary. Future implementations will incorporate more sophisticated heuristics.

The most aggressive inference heuristic, the All Offsets Inference (AOI), uses each stack offset in the candidate function to derive a variable boundary. For example, in Figure 2, lines 3, 5, and 6 are used to derive boundaries at ebp-0xc and ebp-0x1c. The Direct Offset Inference (DOI), uses only direct memory accesses, and derives a boundary at ebp-0xc. Likewise, the Scaled Offset Inference

```
(0) push   %ebp
(1) mov    %esp,%ebp
(2) sub    $0x38,%esp                                      # stack allocation
(3) movl   $0x0,-0xc(%ebp)          ; auth = 0             # direct offset
(4) …
(5) mov    %dl,-0x1c(%ebp,%eax,1)   ; buf[i] = pwd[i]      # scaled offset
(6) cmpl   $0x0,-0xc(%ebp)          ; if (auth)            # direct offset
(7) …
(8) leave                                                  # stack deallocation
(9) ret
```

Fig. 2. Disassembly code fragment for source code shown in Figure 1

(SOI), uses only scaled memory accesses and derives a boundary at ebp-0x1c. As a final catch-all inference, ESI, the Entire Stack Inference, treats the entire local variable region as one variable. ESI does not facilitate fine-grained stack protections. Nevertheless, it does allow for transformations to protect against common attacks to the frame pointer and return address.

Each of the four heuristics is applied to each function and hypotheses about the stack layout are generated using the resulting inferences. The generated hypotheses are ordered by the number of variables inferred (a hypothesis with more variables will be attempted first, and then successively relaxed layouts will be attempted).

Hypothesis validation is accomplished using regression testing. We do not specify the origin of test inputs, but they may be provided, manually generated, automatically generated through concolic or fuzz test case generation, etc. SLX uses the hypothesis to create a stack layout transformation that is a combination of: (1) variable layout randomization, (2) insertion of random-sized padding between variables, and (3) placement of canaries (random values placed on the stack and checked at runtime to detect overflows). Of these three, only canaries dynamically detect attacks. The other two obfuscate the attack surface. During validation, all transformations are combined to validate the layout inference. The candidate transformation is applied to the subject function and is validated using regression testing. If regression testing fails, SLX rejects that hypothesized variable layout, and tries the next, more relaxed, hypothesis. Note that other interpretations of regression testing failure are possible, e.g., erroneous disassembly, but the current SLX prototype focuses on relaxing hypothesized stack layouts. The process repeats until a candidate stack layout hypothesis successfully passes the regression tests, in which case a final transformation is chosen and committed. The final transformation can be any combination of transformations to allow the user to balance security with performance, however the current prototype selects all three transformations. If no candidate hypothesis passes the regression tests, the function is left in its original form.

Transformations require instruction modification, and insertion of new instructions. SLX achieves both by using the Strata software dynamic translator [6]. Software dynamic translation permits program rewriting in a relatively simple and comprehensive manner.

3 Evaluation

To assess the feasibility of SLX, an experiment was conducted using binaries compiled with gcc and -O3 optimization for eleven of the Unix core utilities on Ubuntu 10.04 LTS. Only statically-linked functions were considered for transformation. We omit dynamically linked libraries as they should be transformed and evaluated separately. Libraries only need to be transformed once, after which they can be reused by any number of binaries. The suite of eleven Unix coreutility programs comes with a comprehensive set of test cases provided by the developers, and these were used for the validation step in SLX. Such a set of tests provides a more desirable situation than SLX is likely to encounter in practice. Nevertheless, the test suite simplified the experiment as we assume the tests are all valid and fully test the program.

On average, 195 functions per program were found in the eleven sample programs. SLX determined that 53% of the functions were candidate transformable functions, meaning that SLX found patterns indicating that the function had local variables as well as stack allocation and deallocation patterns. Of the candidate functions, only one function, `term_proc`, could not be transformed by SLX due to the static analyzer's inability to identify the stack deallocation point in the function.

AOI, the most aggressive layout inference, was used to transform 94% of the candidate functions successfully, with an average of four variables per stack frame. SOI was used on 4%, with an average of 1.8 variables found per stack frame. ESI was used on the remaining 2%. These preliminary results are quite encouraging as they demonstrate the feasibility of the SLX approach towards inferring stack layouts to drive security transformations.

The execution-time overhead incurred by SLX was assessed using a set of six programs from the SPEC 2006 benchmarking suite. Randomizing the order of variables on the stack and introducing padding between variables incurred an average increase in execution time of 9%. The addition of canaries, which require run-time checks, increased the total overhead by an average 35%, for a total average overhead of 44% when all three transformations are applied for every function. These overheads include the dynamic translation time. In principle, overhead for canaries can be reduced by more selective placement of canaries.

The security effectiveness of SLX was evaluated by applying SLX to the Wilander buffer overflow test suite [9]. This suite contains twelve stack-based buffer vulnerabilities in six functions and is used to evaluate buffer overflow protection techniques. The overflows in the Wilander suite include: (a) classic overflows that overwrite the return address and stack frame base pointer, and (b) overflows to various local variables. To assess the security potential of SLX alone, we selected the best generated stack layout hypotheses possible based on manual inspection of the six functions in the Wilander binary. Five functions were transformed using AOI, and the remaining function was transformed with SOI. Because SLX transformations are stochastic, SLX was applied to the Wilander suite ten separate times. With the exception of one vulnerability, the transformations produced by SLX placed the target data out of the path of the overflow,

detected the attack at runtime, or transformed the attack surface to such a degree that the attack caused the program to terminate. In most cases, 92%, the attack was detected or the target data was not in the path of the overflow.

One attack succeeded for all ten transformations. The vulnerable function was transformed using SOI, but SOI did not infer the boundary between the vulnerable buffer and the target data. Thus, in this case, the transformation provided by SLX was unable to thwart the attack.

4 Conclusions

SLX is a technique for transforming binary programs to provide run-time protection against stack-based attacks, including intra-frame overflows, when source code is not available. SLX uses a speculative technique to overcome the imprecision of static binary analysis, yet is as aggressive as possible in the randomization transformations applied to the stack. Preliminary results obtained using an SLX prototype show that the technique can transform the stack layout for a wide variety of functions at a finer level of resolution than most existing binary-based techniques.

Acknowledgments. This research is supported by National Science Foundation (NSF) grant CNS-0811689, the Army Research Office (ARO) grant W911-10-0131, the Air Force Research Laboratory (AFRL) contract FA8650-10-C-7025, and DoD AFOSR MURI grant FA9550-07-1-0532. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the NSF, AFRL, ARO, DoD, or the U.S. Government.

References

1. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proceedings of the 12th Conference on USENIX Security Symposium, vol. 12, pp. 105–120. USENIX Association, Berkeley (2003)
2. Bhatkar, S., Sekar, R.: Data Space Randomization. In: Zamboni, D. (ed.) DIMVA 2008. LNCS, vol. 5137, pp. 1–22. Springer, Heidelberg (2008)
3. Bhatkar, S., Sekar, R., DuVarney, D.C.: Efficient techniques for comprehensive protection from memory error exploits. In: Proceedings of the 14th Conference on USENIX Security Symposium, vol. 14, pp. 255–270. USENIX Association, Berkeley (2005)
4. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: USENIX Security Symposium, pp. 177–192 (2005)
5. Hiser, J.D., Coleman, C.L., Co, M., Davidson, J.W.: MEDS: The Memory Error Detection System. In: Massacci, F., Redwine Jr., S.T., Zannone, N. (eds.) ESSoS 2009. LNCS, vol. 5429, pp. 164–179. Springer, Heidelberg (2009)

6. Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J.W., Soffa, M.L.: Retargetable and reconfigurable software dynamic translation. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization, CGO 2003, pp. 36–47. IEEE Computer Society, Washington, DC (2003)
7. The PAX Team, http://pax.grsecurity.net
8. Van Acker, S., Nikiforakis, N., Philippaerts, P., Younan, Y., Piessens, F.: Value-Guard: Protection of Native Applications against Data-Only Buffer Overflows. In: Jha, S., Mathuria, A. (eds.) ICISS 2010. LNCS, vol. 6503, pp. 156–170. Springer, Heidelberg (2010)
9. Wilander, J., Kamkar, M.: A comparison of publicly available tools for dynamic buffer overflow prevention. In: Proceedings of the Network and Distributed System Security Symposium, NDSS. The Internet Society (2003)

Incremental Runtime Verification of Probabilistic Systems

Vojtěch Forejt[1], Marta Kwiatkowska[1], David Parker[2], Hongyang Qu[1], and Mateusz Ujma[1]

[1] Department of Computer Science, University of Oxford, Oxford, UK
[2] School of Computer Science, University of Birmingham, Birmingham, UK

Abstract. Probabilistic verification techniques have been proposed for runtime analysis of adaptive software systems, with the verification results being used to steer the system so that it satisfies certain Quality-of-Service requirements. Since systems evolve over time, and verification results are required promptly, efficiency is an essential issue. To address this, we present incremental verification techniques, which exploit the results of previous analyses. We target systems modelled as Markov decision processes, developing incremental methods for constructing models from high-level system descriptions and for numerical solution using policy iteration based on strongly connected components. A prototype implementation, based on the PRISM model checker, demonstrates performance improvements on a range of case studies.

1 Introduction

Probabilistic systems are prevalent in our daily life: physical devices may fail, communication media are lossy and protocols use randomisation. Formally verifying that such systems behave correctly and reliably requires quantitative approaches, such as *probabilistic model checking*, which can assure, e.g., "the web service successfully delivers a response within 5ms with probability at least 0.99".

It has recently been proposed to use these techniques for *runtime* verification of *adaptive* systems [2], where quantitative verification is used to steer a system such that it satisfies formally specified Quality-of-Service (QoS) requirements. The framework of [2] comprises a computer system exhibiting probabilistic behaviour, a monitoring module that observes its behaviour and a reconfiguration component, which issues it instructions. Requirements to be fulfilled are verified against a high-level model of its behaviour, which is parameterised using data from the monitoring module. The results of verification are then forwarded to the reconfiguration module, which directs the system accordingly.

As a real world example of the applicability of these techniques (which we will return to later), consider a network containing a dynamic set of devices in which joining devices establish a local IP address using the Zeroconf protocol. A suitable QoS requirement for the network would be to minimise the probability of nodes choosing conflicting IP addresses. Parameters influencing this probability

S. Qadeer and S. Tasiran (Eds.): RV 2012, LNCS 7687, pp. 314–319, 2013.
© Springer-Verlag Berlin Heidelberg 2013

include the number of hosts in the network and the number of probes (query messages) that are broadcast before claiming a given IP address.

In this paper, our aim is to optimise the performance of runtime verification for probabilistic systems. Since the systems being verified change dynamically and the results of verification are needed promptly to steer the system, efficiency is essential. We consider *incremental* verification techniques, which exploit the results of previous analyses following a small change to the system being verified.

We target systems modelled as Markov decision processes (MDPs), a widely used model for systems exhibiting both probabilistic and nondeterministic behaviour. We present incremental techniques for the two main phases of probabilistic verification: *model construction*, which exhaustively constructs an MDP from a high-level model description, and *quantitative verification*, which applies numerical techniques to determine the correctness of a system requirement, formally specified in temporal logic. For the former, we propose a technique that infers all states that have to be visited in the incremental step. For the latter, we use policy iteration, optimised using a decomposition of the system into strongly connected components, and performed incrementally by re-using policies between verification runs. We have implemented our techniques in a prototype extension of the PRISM model checker [6], and illustrate the benefits of our approach on a set of benchmark models.

An extended version of this paper with additional details is available as [5].

Related Work. Various techniques have been developed that use model checking at runtime; see [1] for a discussion and further references. There is also increasing interest in incremental model checking techniques. Of particular relevance here are those for probabilistic systems. Wongpiromsarn et al. [8] studied incremental model construction for increasing numbers of system components. In contrast, we focus on changes within a fixed set of components. Filieri et al. [4] presented efficient incremental verification for the simpler model of discrete-time Markov chains using parametric techniques, but their method is subject to an exponential blow up when applied to MDPs and does not handle structural model changes. Kwiatkowska et al. [7] proposed incremental methods for MDPs based on a decomposition into strongly connected components. We consider model changes at the modelling language description level, which [7] does not, and also permit changes in model structure, rather than just transition probabilities.

2 Incremental Model Construction

We first consider incremental techniques for model construction. In this paper, we work with systems specified in the PRISM modelling language, a textual formalism based on guarded commands. Our incremental techniques are designed to operate after relatively small runtime changes to the structure of the MDP. At the level of the modelling language description, we assume that these changes are made by altering *parameters*: constants from the model description whose value is not determined until runtime. We only consider changes in parameters that occur in *guards* of commands, which is a common scenario in practice.

```
4:   const int N; // number of abstract hosts
5:   const int K; // number of probes to send
     ...
20:  module host0
       ...
26:    // send probe
27:    [send] l=2 & x=2 & probes<K → (x'=0) & (probes'=probes + 1);
28:    // sent K probes and waited 2 seconds
29:    [] l=2 & x=2 & probes=K → (l'=3) & (probes'=0) & (coll'=0) & (x'=0);
       ...
33:  endmodule
```

Fig. 1. Fragments of a PRISM model of the Zeroconf protocol

For simplicity, we do not consider parameters that affect transition probabilities values. Such changes could be handled using the techniques described in [7].

We work with an explicit-state implementation. Building an MDP from a PRISM model requires a systematic state-space exploration, the most costly parts of which are the evaluation of all commands in each state, and subsequent creation of new states found. The basic idea of our incremental method is to infer the subset of states needing to be rebuilt, reducing the number of commands to be re-evaluated. Full details of the algorithm are in [5]; here we give an informal description using an example.

Figure 1 shows a fragment of a PRISM model for the previously mentioned Zeroconf protocol example. We assume that parameter N is fixed and K varies. We consider a scenario where we have already built a model $\mathcal{M}_1$ for $K{=}k_1$ and need to construct a new model $\mathcal{M}_2$ for $K{=}k_2$. We start by identifying guards that contain K: we find them in lines 27 and 29; for convenience, call them g_1, g_2 (in the example, these commands have probability 1, but this is not a limitation of our approach). In each guard, there is a variable compared to the parameter K, in this case *probes* in both guards. The key observation is that, to build model M_2, we do not need to re-evaluate commands in all states: it is sufficient to examine states from $\mathcal{M}_1$ that satisfied g_1, g_2 for $K{=}k_1$ but no longer do for $K{=}k_2$, and states that now satisfy g_1, g_2 for $K{=}k_2$. To find such states, we need to compute bounds on the values of *probes* for $K{=}k_1$ and $K{=}k_2$.

For the majority of PRISM models (whose guards involve just linear arithmetic), we can accomplish this using an SMT solver. In fact, for many common classes of expressions (such as this example) we can extract the bounds directly. In our example, for $K{=}k_1$, we obtain $probes \in [0, k_1)$ for g_1 and $probes \in [k_1, k_1]$ for g_2. For $K{=}k_2$, we get $probes \in [0, k_2)$ for g_1 and $probes \in [k_2, k_2]$ for g_2. Taking the intersection identifies states in $\mathcal{M}_1$ that satisfy g_1 for both $K{=}k_1$ and $K{=}k_2$, giving $probes \in [0, k_1) \cap [0, k_2)$. The union, i.e. $probes \in [0, k_1) \cup [0, k_2)$, gives all states of $\mathcal{M}_1$ that may satisfy g_1. The states that need to be re-evaluated in the context of g_1 are then found by performing a state space exploration from states with variable $probes \in ([0, k_1) \cup [0, k_2)) \setminus ([0, k_1) \cap [0, k_2))$. The same process is subsequently repeated for guard g_2. The efficiency of performing these steps can be improved considerably by keeping the state space of $\mathcal{M}_1$ sorted, with respect to variable *probes*, and using binary search when looking for the states

satisfying a given bound. Finally, we remove from model $\mathcal{M}_2$ any states that are no longer reachable from its initial state using standard reachability algorithms.

3 Incremental Quantitative Verification

Next, we consider incremental techniques for quantitative verification of MDPs, the key part of which is the numerical computation of either the *minimum* or *maximum* probability of reaching a set of target states, over all possible *adversaries* of the MDP (an adversary represents one way of resolving all nondeterminism in the model). Common methods for computing these probabilities are *value iteration*, which is an approximate iterative numerical solution method, and *policy iteration*, which analyses a sequence of adversaries with increasing/decreasing probabilities.

Previous incremental verification techniques for MDPs [7] were based on the use of value iteration, applied to a decomposition of the the model into its strongly connected components (SCCs) [3]. These methods are not directly applicable to the scenarios we consider in this paper since, unlike [7], we permit structural changes to be made to the MDP. Instead, we propose an SCC-based version of policy iteration. Like [7], we first decompose the MDP into its SCCs and determine their topological ordering; next, we solve each SCC separately, working through them backwards according to the topological ordering. Here, however, we use policy iteration to compute the probabilities for each SCC.

For incremental verification, the key benefit from using policy iteration comes when we select the initial adversary used to start the computation. For this, rather than taking the usual approach of selecting an arbitrary adversary, we adapt the optimal adversary from the previous run of verification. Let $\mathcal{M}_1$ be the previous MDP and $\mathcal{M}_2$ be the new one. An adversary for an MDP resolves the nondeterminism in each of its states. To construct the initial adversary for solving $\mathcal{M}_2$, we identify all the states that are present in both $\mathcal{M}_1$ and $\mathcal{M}_2$, and which have the same nondeterministic choices in both models; we then re-use the choices made by the old adversary for $\mathcal{M}_1$ in the one for $\mathcal{M}_2$.

4 Experiments

We implemented our techniques in an extension of PRISM [6], using its explicit-state model checking engine, and evaluated them on 4 existing benchmarks: *zeroconf*, *mer*, *consensus* and *firewire*. Details of these examples and additional results are in [5]. We ran a series of verification instances, varying a particular model parameter. Table 1 shows the parameter ranges, the sizes of the resulting models, and the total time required to perform model construction and verification for all models, in both a non-incremental and incremental fashion. The overhead on memory usage is negligible for both algorithms, and thus omitted.

For incremental model construction, we obtained speed-ups in all cases, up to a 10-fold improvement for the *mer* example. The key factor for performance is the number of states added between each verification run. For incremental verification due to space restrictions we do not show results for original algorithms,

Table 1. Performance comparison for incremental techniques

Model			Time (s)			
Name [parameters]	Parameter values	States [10^3]	Original model constr.	Incremental model constr.	SCC-based policy iteration	Incremental policy iteration
zeroconf [N, K]	10,1-5	32-496	15.9	12.3	10.6	8.5
	10,10-20	3002-5812	859.7	320.2	1859.1	1329.2
	60000,1-5	32-496	16.2	11.6	49.7	50.9
	60000,10-20	3002-5812	853.9	313.7	9333.9	4218.4
mer [N]	1-100	8-592	429.5	44.3	70.5	65.5
	200-300	1183-1774	2352.4	192.5	400.6	369.7
	400-500	2364-2955	4375.7	358.3	695.9	683.3
consensus [N, K]	2,1-40	1-5	0.8	0.4	33.6	23.2
	2,80-120	10-15	2.3	0.9	1235.8	900.1
	4,1-20	12-20	15.5	4.8	1029.1	666.5
firewire [*deadline*]	1000-1050	369-398	62.3	10.3	38.5	37.7
	2000-2050	970-1000	160.5	25.7	99.7	97
	3000-3050	1571-1601	265.7	42.4	174	181.6

which we outperform for each case study. Incremental policy iteration is quicker than SCC-based policy iteration in some, but not all, cases. The best results are those for the *zeroconf* example, with a 2-fold speedup. The performance of this phase is mostly influenced by the structure of the state space.

5 Conclusions

We have described ongoing work to develop incremental verification techniques for Markov decision processes, aimed at improving the efficiency of runtime methods for systems with probabilistic behaviour. Future directions include evaluating presented techniques on a deployed adaptive system and improving system reconfiguration using policies obtained from model checking.

Acknowledgements. The authors are part supported by ERC Advanced Grant VERIWARE, EU FP7 project CONNECT and EPSRC project EP/F001096/1.

References

1. Calinescu, R.: When the requirements for adaptation and high integrity meet. In: Proc. 8th Workshop on Assurances for Self-Adaptive Systems, pp. 1–4 (2011)
2. Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R., Tamburrelli, G.: Dynamic QoS management and optimisation in service-based systems. IEEE Transactions on Software Engineering 37(3), 387–409 (2011)
3. Ciesinski, F., Baier, C., Größer, M., Klein, J.: Reduction techniques for model checking Markov decision processes. In: Proc. QEST 2008, pp. 45–54. IEEE (2008)
4. Filieri, A., Ghezzi, C., Tamburrelli, G.: Run-time efficient probabilistic model checking. In: Proc. ICSE 2011, pp. 341–350. ACM, New York (2011)

5. Forejt, V., Kwiatkowska, M., Parker, D., Qu, H., Ujma, M.: Incremental runtime verification of probabilistic systems. Tech. Rep. RR-12-05, Department of Computer Science, University of Oxford (2012)
6. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
7. Kwiatkowska, M., Parker, D., Qu, H.: Incremental quantitative verification for Markov decision processes. In: Proc. DSN-PDS 2011, pp. 359–370. IEEE (2011)
8. Wongpiromsarn, T., Ulusoy, A., Belta, C., Frazzoli, E., Rus, D.: Incremental temporal logic synthesis of control policies for robots interacting with dynamic agents. In: Proc. IROS 2012 (to appear, 2012)

Author Index

Ahmed, Samatar 126

Baldor, Kevin 245
Barre, Benjamin 184
Bartocci, Ezio 168
Basin, David 151
Beaudet, Éric 126
Bonakdarpour, Borzoo 199
Brauer, Jörg 110
Bryant, Randy 19
Bucur, Doina 96
Bulychev, Peter 260

Cadar, Cristian 2
Chen, Feng 136
Co, Michele 308
Colombo, Christian 214
Cristal, Adrián 42
Currea, Sebastian 88

Darulova, Eva 277
David, Alexandre 260
Davidson, Jack W. 308

Erickson, John 1
Eyolfson, Jon 49

Falcone, Yliès 88, 229
Fiedor, Jan 35
Fischmeister, Sebastian 199
Forejt, Vojtěch 314
Freund, Stephen 1
Függer, Matthias 110

Ganai, Malay K. 3
Gibson, Garth 19
Grosu, Radu 168

Hallé, Sylvain 126, 184
Hickey, Jason 19
Hiser, Jason D. 308

Jaber, Mohamad 88
Jéron, Thierry 229

Kabitzsch, Klaus 131
Karmarkar, Atul 168
Kestor, Gokcen 42
Klaedtke, Felix 151
Klein, Mathieu 184
Knight, John C. 308
Kuncak, Viktor 277
Kuru, Ismail 42
Kwiatkowska, Marta 314

Lam, Patrick 49
Larsen, Kim G. 260
Larus, Jim R. 48
Legay, Axel 260
Leucker, Martin 82
Li, Guangyuan 260
Li, Wenchao 64

Marchand, Hervé 229
Marinovic, Srdjan 151
Matar, Hassan Salehe 42
Mrad, Aouatef 126
Musuvathi, Madanlal 1

Navabpour, Samaneh 199
Nelson, Stephen 292
Nguena Timo, Omer Landry 229
Nguyen-Tuong, Anh 308
Niu, Jianwei 245
Noble, James 292

Ollivier, Pierre-Antoine 184

Pace, Gordon J. 214
Parker, David 314
Pearce, David J. 292
Pinisetty, Srinivas 229
Poulsen, Danny Bøgsted 260

Qu, Hongyang 314

Reinbacher, Thomas 110
Richter, Andreas 131
Rinard, Martin 276
Rodes, Benjamin D. 308
Rollet, Antoine 229
Roșu, Grigore 136

Sen, Koushik 2
Șerbănuță, Traian Florin 136
Seshia, Sanjit A. 64
Seyster, Justin 168
Simsa, Jiri 19
Smolka, Scott A. 168
Soucy-Boivin, Maxime 184
Stoller, Scott D. 168

Ujma, Mateusz 314
Unsal, Osman 42

Vigna, Giovanni 183
Vojnar, Tomáš 35

Zadok, Erez 168
Zălinescu, Eugen 151