Pre-proceedings of the
15th International Workshop on
Automated Verification of Critical Systems

https://sites.google.com/site/avocs15/

Gudmund Grov, Andrew Ireland (Editors)

School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, UK
Table of Contents

Foreword

Invited Talks

THREADSAFE: Static Analysis for Java Concurrency
Robert Atkey and Donald Sannella

Verifying Critical Cyber-Physical Systems After Deployment
Colin O’Halloran

Regular Papers

Estimating Rewards & Rare Events in Nondeterministic Systems
Sean Sedwards, Axel Legay and Louis-Marie Traonouez

Conditional Lemma Discovery and Recursion Induction in Hipster
Irene Lobo Valbuena and Moa Johansson

Computing Bounds for Counter Automata
Maximilien Colange, Dimitri Racordon and Didier Buchs

Transforming Event-B Models to Dafny Contracts
Mohammadsadegh Dalvandi, Michael Butler and Abdolbaghi Rezazadeh

Loop Patterns in C Programs
Thomas Pani, Helmut Veith and Florian Zuleger

Studying Verification Conditions for Imperative Programs
Claúdio Belo Lourenço, Si-Mohamed Lamraoui, Shin Nakajima and Jorge Sousa Pinto

Automated Verification of Asynchronous Communicating Systems with TLA+
Florent Chevrou, Aurélie Hurault and Philippe Quéinnec

KriQL: A Language for Query-based Diagnosis of Transition Systems
Khaoula Es-Salhi, Siham Rim Boudaoud, Ciprian Teodorov, Vincent Ribaude and Zoé Drey

Approximate Active Learning of Nondeterministic Input Output Transition Systems
Michele Volpato and Jan Tretmans

Formal Specification, Verification, and Implementation of Fault-Tolerant Systems
Vincent Rahli, David Guaspari, Mark Bickford and Robert Constable

First-order logic for safety verification of hedge rewriting systems
Alexei Lisitsa

Distributed Verification of Rare Properties using Importance Splitting Observers
Sean Sedwards, Axel Legay, Louis-Marie Traonouez and Cyrille Jegourel
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model-based WCET Analysis with Invariants</td>
<td>229</td>
</tr>
<tr>
<td>Bojan Nokovic and Emil Sekerinski</td>
<td></td>
</tr>
<tr>
<td>State Distribution Policy for Distributed Model Checking of Actor Models</td>
<td>245</td>
</tr>
<tr>
<td>Ehsan Khamespanah, Marjan Sirjani, Mohammadreza Mousavi, Zeinab Sabahi Kaviani and Mohammadreza Razzazi</td>
<td></td>
</tr>
<tr>
<td>Permissive strategies in timed automata and games</td>
<td>263</td>
</tr>
<tr>
<td>Patricia Bouyer, Erwin Fang and Nicolas Markey</td>
<td></td>
</tr>
</tbody>
</table>

**Research Idea Papers**

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automated Hazard Analysis with pMAX-SMT for Automobile Systems</td>
<td>281</td>
</tr>
<tr>
<td>Shogo Hattori, Shoji Yuen, Hiroyuki Seki and Shuichi Sato</td>
<td></td>
</tr>
<tr>
<td>Model Transformation of high-level requirements in a D into a Formal Specification Language</td>
<td>283</td>
</tr>
<tr>
<td>Suraj Ajit, Oyindamola Olajubu, Scott Thomson and Mark Edwards</td>
<td></td>
</tr>
<tr>
<td>A Verified Virtual Machine for Safety-Critical Java</td>
<td>285</td>
</tr>
<tr>
<td>James Baxter and Ana Cavalcanti</td>
<td></td>
</tr>
<tr>
<td>Structural Resolution for Automated Verification</td>
<td>287</td>
</tr>
<tr>
<td>Ekaterina Komendantskaya, Peng Fu and Patricia Johann</td>
<td></td>
</tr>
<tr>
<td>Model Checking Web Applications using SPIN and UPPAAL</td>
<td>291</td>
</tr>
<tr>
<td>Mohammed Alzahrani</td>
<td></td>
</tr>
<tr>
<td>Revisiting modelling and analysing railway control systems in CSP</td>
<td>295</td>
</tr>
<tr>
<td>Mike Smith, Hoang Nga Nguyen, Markus Roggenbach</td>
<td></td>
</tr>
</tbody>
</table>
Foreword

The 15th International Workshop on Automated Verification of Critical Systems (AVoCS 2015) will be hosted by the School of Mathematical and Computer Sciences within Heriot-Watt University on September 2 – 4, 2015. The workshop will take place in headquarters of the International Centre for Mathematical Sciences (ICMS), located in the historic old town of the Edinburgh. The 6th AI4FM workshop, which will be held on 1 September, is co-located with AVoCS 2015.

The workshop programme consists of regular papers, research ideas papers and two keynote speakers – Don Sannella (Contemplate Ltd & the University of Edinburgh, UK) and Colin O’Halloran (D-RisQ Software Systems & the University of Oxford, UK). We are particularly grateful to Don and Colin for their contributions to AVoCS 2015.

We are grateful for the sponsorship provided by Altran, D-RisQ Software Systems, Formal Methods Europe (FME) and the Scottish Informatics and Computer Science Alliance (SICSA). Sponsorships from Altran, FME and SICSA was directly used to support students attending AVoCS.

We received 24 submitted papers, in which 16 were accepted for presentation at the workshop and to appear in this pre-proceedings. 13 of the accepted papers have been accepted for the post-proceedings, to be published as an EASST series, while 3 of the papers have been conditionally accepted. We also received 8 research ideas papers, and 6 of them will be presented at the workshop. These are part of the pre-proceedings but will not appear in the EASST volume. New to AVoCS 2015 is that the research ideas will be organised within two discussion-based sessions.

We would like to thank our excellent programme committee and additional reviewers for their efforts in reviewing the contributed papers in order to ensure a high quality workshop, and the AVoCS steering committee for giving us the opportunity to host AVoCS 2015 in Edinburgh.

We would like to thank Andrius Velykis and Leo Freitas, who, together with Gudmund Grov, are the organisers of the AI4FM 2015 workshop. We would also like to thank: the organisers of AVoCS 2014 for all the help and use of their \LaTeX template; staff within the School of MACS for all the support; and particularly Audrey Brown at ICMS, for all the help with local arrangements.

Finally, special thanks go to Yuhui Lin. We are greatly indebted to all the work that he has done. He has far exceeded the remit of a Local Arrangements and Publicity Chair.

Workshop Programme Chairs

August 2015

Heriot-Watt University

Gudmund Grov

Andrew Ireland
Programme Committee

- Ernie Cohen, University of Pennsylvania, USA
- Ewen Denney, NASA Ames, USA
- Jean-Christophe Filliètre, CNRS, France
- Michael Goldsmith, University of Oxford, UK
- Gudmund Grov, Heriot-Watt University, UK (co-chair)
- Keijo Heljanko, Aalto University, Finland
- Mike Hinchey, University of Limerick, Ireland
- Marieke Huisman, University of Twente, Netherlands
- Andrew Ireland, Heriot-Watt University, UK (co-chair)
- Gerwin Klein, NICTA/UNSW, Australia
- Thierry Lecomte, ClearSy, France
- Yuhui Lin, Heriot-Watt University, UK
- Peter Gorm Larsen, Aarhus University, Denmark
- Panagiotis (Pete) Manolios, Northeastern University, USA
- Stephan Merz, INRIA Nancy & LORIA, France
- Hoang Nga Nguyen, University of Nottingham, UK
- Jaco van de Pol, University of Twente, Netherlands
- Markus Roggenbach, Swansea University, UK
- Marco Roveri, FBK, Italy
- Thomas Santen, Microsoft Research, Germany
- Bernard Steffen, Technical University Dortmund, Germany
- Jan Strejček, Masaryk University, Czech Republic
- Jun Sun, Singapore University of Technology and Design, Singapore
- Tayssir Touili, CNRS & LIPN, France
- Helen Treharne, University of Surrey, UK
- Laurent Voisin, Systerel, France
- Angela Wallenburg, Altran, UK
- John Wickerson, Imperial College London, UK
- Peter Ölveczky, University of Oslo, Norway

Additional Reviewers

- Sylvain Conchon  
- František Blahoudek  
- Stefan Hallerstede  

- Marcio Diaz  
- Jiří Barnat  
- Christine Rizkallah

Steering Committee

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

Local Organisers

- Gudmund Grov
- Andrew Ireland
- Yuhui Lin (local arrangement & publicity chair)
Invited Talks
**THREADSAFE: Static Analysis for Java Concurrency**

Robert Atkey\(^1,3\) and Donald Sannella\(^2,3\)

\(^1\) bob.atkey@gmail.com, bentnib.org/
Computer & Information Sciences, University of Strathclyde
Glasgow, United Kingdom

\(^2\) dts@inf.ed.ac.uk, homepages.inf.ed.ac.uk/dts/
School of Informatics, University of Edinburgh
Edinburgh, United Kingdom

\(^3\) www.contemplateltd.com/
Contemplate Ltd
Edinburgh, United Kingdom

**Abstract:** THREADSAFE is a commercial static analysis tool that focuses on detection of Java concurrency defects. THREADSAFE’s bug-finding capabilities and its look and feel are presented through examples of bugs found in the codebases of two widely-used open source projects.

**Keywords:** static analysis, concurrency, Java

1 Introduction

It is widely acknowledged that developing reliable concurrent software is very difficult, see e.g. [Goe06, Lee06, HS12]. Concurrency-related defects like data races and deadlocks can be subtle and hard to understand. The fact that concurrent software is inherently non-deterministic means that reliance on testing for software quality assurance is inappropriate and dangerous; intermittent bugs that show up once in a million runs are not uncommon. At the same time, requirements for improved performance force the increased use of application-level concurrency in order to exploit multicore hardware.

Use of static analysis to discover and diagnose concurrency defects during software development is a cost-effective solution to this problem. Static analysis can take all possible execution paths into consideration, not just the ones that are explored by specific test data for a specific scheduling of threads. Static analysis can be applied to a codebase while it is still under construction, long before testing is possible. Bugs that are detected early in the development process are easier and cheaper to fix.

THREADSAFE is a commercial static analysis tool that focuses on detection of Java concurrency defects. By focusing on concurrency bugs, THREADSAFE can find bugs that other static analysis tools, both commercial and freely available, miss or are not designed to look for.

THREADSAFE is fully automatic. It requires no annotations to be added to codebases, but if @GuardedBy annotations [Goe06] are present then they are checked for accuracy. Its findings are generated from information produced by a class-by-class flow-sensitive, path-sensitive,
context-sensitive points-to and lock analysis. Heuristics are used to tune the analysis to avoid false positives. Knowledge of concurrency aspects of the Java library and of some aspects of frameworks including Android have been built in. It has been successfully applied to codebases of more than a million lines. It is being used in a companies across a wide range of industry sectors and in university teaching, for example [Ses14].

In the following sections we give an outline of how THREADSAFE works and present some examples of bugs that it finds in two widely-used open source projects. We then present some of our experiences with developing and using THREADSAFE, discuss how its performance can be measured, and give a comparison of its performance against FindBugs on a Java concurrency benchmark.

2 How THREADSAFE operates

THREADSAFE analyses .class files containing JVM bytecode for evidence of concurrency errors, which are reported back to the user. In theory, because THREADSAFE analyses JVM bytecode, it could be used to analyse the output of any compiler that targets the JVM. However, due to the need for heuristics that encode assumptions about the nature of the code being analysed, and in-built knowledge about compilers, idioms, libraries and frameworks that are common when programming with Java as the source language, THREADSAFE only really operates effectively on the output of a Java compiler.

The general operation of THREADSAFE can be thought of as a kind of compiler for JVM bytecode, only instead of producing optimised machine code, it produces static analysis warnings. The front-end portion is the same: JVM .class files are read in, translated into a simpler form for analysis, and then analysed. Then, instead of using the results of the analysis to optimise, checkers are run over the analysis results to find evidence of code that is problematic from a concurrency point of view. In the following, we explain in more details the three main stages of THREADSAFE’s operation, and discuss the trade-offs that have been made.

Although THREADSAFE is primarily targeted at finding concurrency flaws, there is nothing particularly concurrency focused about the analysis it performs (other than tracking lock acquisitions) before the checkers are run. Therefore, it would be relatively straightforward to write new checkers that discover, for example, resource handling errors, or information leakage errors, on top of the existing analysis.

2.1 Class loading and preprocessing

In order to be scalable to very large code bases, THREADSAFE analyses .class files on a by-package basis, only referring to .class files in other packages as needed to fill in details in the inheritance hierarchy, or to analyse inherited methods. Class files are read in lazily, as required by the analysis described below, and the bytecode in the body of each method is preprocessed in two phases.

Firstly, the JVM’s native stack-based bytecode is translated into a register-based bytecode. This translation enables the interprocedural dataflow analysis, described below, to be much more efficient, as the local variables in each method can be modelled as a flat array of abstract values,
rather than a dynamic stack. The translation also makes all control flow in each method body explicit, turning the flat arrays of bytecode instructions into an explicit control flow graph, with one instruction per node. In the process, in-method subroutines, implemented by the JVM's `jsr` and `ret` instructions, are inlined. The `jsr` and `ret` instructions complicate analysis, since they result in the same segment of bytecode being invoked in several contexts within the same method. Since recursive subroutines via `jsr/ret` are not possible, it is safe to inline bytecode subroutines into their callsites, making the dataflow analysis in the next stage simpler.

Secondly in the preprocessing phase, accessor methods are inlined into their callers. Accessor methods are methods generated by the Java compiler that allow inner (or outer) classes to access private fields and methods in their outer (resp. inner) classes. Normally, the access control enforced by the JVM would not allow this (the JVM knows nothing about inner or outer classes), so the compiler generates a public “synthetic method” that provides direct access to the underlying private field or method. This results in the actual field or method access appearing in a different bytecode method to the one that appears in the source code. If one is not careful, this can lead to confusing reports from a static analyser that naively reports which method a problematic field access appears in. Also, while it would be possible to just let the normal interprocedural analysis machinery handle accessor methods, it is more efficient to just inline the methods into their callers. Therefore, THREADSAFE inlines all accessor methods into their callers. This process is not entirely straightforward, because the compilation scheme for accessor methods is not specified by the Java Language Specification, and different compilers have slightly different strategies. THREADSAFE handles the schemes used by the standard OpenJDK `javac` compiler, and Eclipse’s `ecj` compiler.

### 2.2 Interprocedural per-class analysis

After translation, a context sensitive interprocedural shape and lock analysis is run on each public entry point of each instantiable class. The idea is that each class is a self-contained entity whose instances can be invoked arbitrarily via its public entry points. A method is deemed to be a “public entry point” if it is either a public method, or has been specially marked for the analysis because it will be called by some framework or library method. An example of a non-public “public entry point” is the protected `doInBackground` method from the Android `AsyncTask` class, which is invoked by the framework code as an asynchronous background task.

This analysis is interprocedural, but to keep the overall analysis scalable, only calls to private and protected methods on the same class are followed. All other calls are treated symbolically. This again follows the idea that each public method of a class is special in that it represents how classes are used by other classes, while private and protected methods are part of a class’s implementation. A disadvantage of this approach is that it relies heavily on the programmer having good taste when it comes to assigning access qualifiers to methods. A class whose methods are all public will act the same during execution, but will produce very different analysis results. A possible method to ameliorate this, which we have not yet tried, is to run a whole-program analysis to infer a tightest possible access qualifier for each method, ignoring the programmer’s annotations.

The analysis computes, for each object-manipulating instruction, an approximation of the ob-
jects that will be accessed and the set of locks that are held when that instruction is executed. Objects (whether locks or directly manipulated) are represented as paths relative to the symbolic `this` reference pointing to the instance of the class currently under analysis, the parameters of the public entry point method being analysed, or to references acquired from globally accessible static fields and methods.

The abstract lock sets computed for each instruction consist of representations of monitors, i.e., the JVM’s intrinsic lock associated with every object, and first-class locks as represented by implementations of the `java.util.concurrent.locks.Lock` interface. Monitors are slightly simpler because they must strictly nest with respect to method calls, so it is easier to accurately track whether or not they are held through recursive method calls.

The shape and lock analysis are intentionally unsound in their handling of aliasing and mutation. For example, the sequence of code `this.f.lock(); ... this.f.unlock();` is assumed to lock and unlock the same object, even if the field `f` could be updated in between. A later checker checks to see whether fields containing locks are ever mutated while the corresponding lock is held, because this is often a mistake by the programmer.

Once the analysis of each public entry point of each instantiable class is complete, some general summary information is collected: fields that contain collection objects are identified, and classified according to whether the collection object is known to be suitable for concurrent use or not. For example, the concurrent collections from the `java.util.concurrent` package are known to be safe for concurrent use (though see the description of the Get-Check-Put checker below, while a instances of a collection class like `java.util.ArrayList` are definitely known to be unsafe it accessed concurrently without synchronisation.

### 2.3 Checkers

Finally, to produce the analysis warnings that are presented to the user, so-called checkers are run over the information gathered in the previous pass. THREADSAFE contains over a dozen checkers that look for specific instances of wrong or problematic code that uses concurrency. Two of the checkers are the Inconsistent Synchronisation checker and the Get-Check-Put checker.

**Inconsistent synchronisation** The Inconsistent Synchronisation checker examines, for each field, all the accesses to that field, and determines if they all share a common lock. If either: all accesses are locked, but do not share a common lock; or some accesses are locked and some are not, with the ratio of locked/total being over some threshold, then a warning is produced. This analysis was originally based on the inconsistent synchronisation analysis in the FindBugs tool, but has been extended to have a more precise determination of which locks are held, and to handle accesses to thread unsafe collections stored in fields, as well as direct accesses to fields. In particular, THREADSAFE’s analysis can handle cases where all accesses are locked, but with different locks.

**Get-Check-Put** The Get-Check-Put checker looks for sequences of method calls on an instance of concurrent collection, e.g., a `ConcurrentHashMap`, that check the status of the collection, and then mutate the collection based on the result of the check. If other threads are not prevented from mutating the collection between the check and the mutation, then there is
a potential race condition, because the information gathered from the check will have become outdated. The name “Get-Check-Put” comes from the common pattern where a .get(key) method is invoked to find out whether key is in the map, the result is compared to null (the “check”), and if so, the .put method is used to associate key with a new value in the map. Such a pattern is often used to represent lazy caches of objects that are expensive to create. Instances of Get-Check-Put are detected by performing another interprocedural analysis over the results of the first interprocedural analysis, tracking the evolution of method invocations on each collection object using a typestate-style analysis.

3 Examples

The following sections give a few examples of bugs and potential bugs that THREADSAFE finds in open source codebases. As far as we are aware, none of these bugs is found by any other static analysis tool. Our aim in presenting these examples is to give an impression of the kinds of bugs that THREADSAFE is able to find as well as a feeling for the overall look and feel of the tool.

3.1 Example: incorrect synchronization of collection accesses

Java provides a rich assortment of collection classes, each with its own requirements on whether or not synchronization is required for concurrent access to the collection.

Inconsistent synchronization on collections can be particularly harmful to program behaviour. While incorrectly synchronizing accesses to a field may “only” result in missed updates or stale information, incorrectly synchronizing accesses to collections that have not been designed for concurrent use can lead to violations of the collections’ internal invariants. This may not immediately cause visible effects, but may cause odd behaviour, including infinite loops or corrupted data, at a later point in the program’s execution. See [Tym09, Ora12] for an example.

An example of inconsistent use of synchronization when accessing a shared collection is present in version 2.10 of Apache JMeter, an open source tool for testing application performance under load. THREADSAFE produces 44 findings for JMeter, including the one shown below in THREADSAFE’s Eclipse plug-in.
This finding suggests a possible mistake in synchronizing accesses to the collection stored in the field `RespTimeGraphVisualizer.internalList`.

We can click on the “Problem location” to display the declaration of this field or on any of the locations in the list of accesses. We have clicked above on the location of the unsynchronized read access and the relevant line of code is highlighted. Detailed documentation is available under “Rule description” on the finding, the risks that it raises and their potential severity, and remediation options.

We can ask THREADSAFE to show us the accesses to this field along with the locks that are held, by clicking on “Accesses and locks”:

Now we can see that there are three methods that access the collection stored in the field `internalList`. One of these methods is `actionPerformed`, which will be invoked by the Swing GUI framework on the UI thread.

Another method that accesses the collection stored in `internalList` is `add()`. By investigating the possible callers of this method in the call hierarchy, we can see that it is indeed called from the `run()` method of a thread that is not the main UI Thread of the application, indicating that synchronization ought to have been used.
3.2 Example: potential deadlock

K9Mail is an Android email client, written in Java, that uses concurrent threads to prevent the user interface from becoming unresponsive during communication with the network.

K9Mail consists of over 1000 classes. It would be impractical to go through all of them to look for potential deadlock cycles. Running the THREADSAFE Eclipse plug-in on K9Mail produces the following deadlock warning:

From the finding report, we can see that the intrinsic locks associated with objects of the Preferences and Account classes form a circular relationship. Investigating this finding in the call hierarchy, we learn that the circularity can arise from the following pieces of code:

1. In the synchronized method Preferences.getAvailableAccounts(), there is a call on line 95 to the synchronized method Account.isEnabled().

2. In the synchronized method Account.save(Preferences), there is a call on line 679 to the synchronized method Preferences.getAccounts().

If two threads invoke the Preferences.getAvailableAccounts() method and the Account.save() method concurrently, then there is a real chance that a deadlock will result.

It will take further investigation to determine whether or not this scenario is actually possible, but THREADSAFE has successfully narrowed down the number of lines of code that we have to examine.

---

1 The version used here and in Subsection 3.3 was taken from github.com/k9mail/k-9 and has commit SHA1 id cc8353d25572b5f1c19047c0c093371f5ac721b4.
3.3 Example: missing synchronization in Android

The concurrent environment in which an application may run is almost never under the complete control of the application developer. Frameworks invoke various parts of applications in response to user, network, or other external events, and often have implicit requirements about which methods may be invoked on which threads.

An example of the incorrect use of frameworks is visible in K9Mail. Running THREADSAFE on K9Mail yields the following finding, indicating that the field `mDraftId` appears to be accessed from an Android background thread, and another thread, with no synchronization.

![Unsynchronized access to field from asynchronously invoked method](image)

Clicking on “Accesses and locks”, we can see that the field `mDraftId` has been accessed from a `doInBackground` method.

![Guards for access to field MessageCompose.mDraftId: long](image)

The `doInBackground` method is part of the Android framework’s `AsyncTask` facilities for running time-consuming tasks in the background separately from the main UI thread. Correct use of `AsyncTask.doInBackground(..)` can ensure that Android applications remain responsive to user input, but care must be taken to ensure that interaction between the background thread and the main UI thread is correctly synchronized.

Investigating further in Eclipse’s call hierarchy, we discover that the `onDiscard()` method, which also accesses the `mDraftId` field, is called by the `onBackPressed()` method. This method is in turn always invoked by the Android framework on the main UI thread, not the background thread for running `AsyncTasks`, indicating the presence of a potential concurrency defect.
4 Experiences with developing and using THREADSAFE

4.1 What is a concurrency bug, and how do developers react to static analysis?

In our experience, the primary difficulty in developing an unsound and incomplete static analysis like THREADSAFE is in determining exactly what ought to be reported, given the information that we are able to compute statically from the program.

For example, the Inconsistent Synchronisation analysis described above only discovers places where the programmer has been inconsistent in their use of locks when accessing a given field. It does not attempt to determine whether or not the two accesses can actually happen in parallel. Nor does it attempt to determine any particular severity of the potential race condition. There may be several reasons why accesses to a field use locks inconsistently:

1. The programmer has genuinely made a mistake, and the unlocked, or incorrectly locked accesses, represent real data races that will manifest at runtime;

2. The programmer has genuinely made a mistake, but the unlocked, or incorrectly locked accesses are extremely rare at runtime, and are very unlikely to produce a real reliability issue when the code is in production;

3. The field is only ever accessed by one thread, but other fields are accessed by multiple threads, and the locks used to protect them are irrelevantly being held while accessing the single-thread field, giving the impression that it has been locked unnecessarily;

4. The methods that access the fields without locking are never executed in parallel with those that do. For example, initialisation methods or shutdown methods may be guaranteed to have exclusive access to their object;

5. The programmer has been able to determine via other means that this particular field access is safe, perhaps by reasoning based on the Java Memory Model.

In case 2, some programmers were loath to change their code to assuage the analysis. Changing code has costs of its own, and if the bug is extremely unlikely, then why bother? Other programmers take a longer view, reasoning that a bug that is unlikely now may become more likely in the future after the structure and intended purpose of the code evolves. Leaving concurrency bugs hidden inside code that is believed to be reliable creates technical debt that obstructs future development.

In cases 3 and 5, it could be argued that the design of the code is at fault. If the locking strategy used by the code is sufficiently complex that subtle reasoning is required to determine when access to a field requires a lock or not, then the code ought to be changed.

In our experience, different developers differ widely in their responses to static analysis results. Some actively dislike false positives, treating them as noise; some appreciate the warnings, but require a reliable way of turning them off for code they have hand-checked to be safe; while others will rewrite code to remove static analysis warnings, perhaps in the belief that code that confuses a static analyser will confuse a human too. In general, however, we observed a lean towards conservatism in most developers: unless a particular static analysis warning obviously indicates a real, costly, bug, then it was usually felt better to leave apparently working and tested
code alone until time could be set aside to work on it properly. Performing quick fixes piecemeal on complex code, following the warnings produced by a static analyser, may actually decrease the reliability of a program if it is done without a clear understanding of what the code is meant to do.

At a level above data races on individual fields, it can be very difficult to determine the difference between intended behaviour and unintended behaviour. In some cases, non-deterministic behaviour may be exactly what the programmer intends to happen in a concurrency environment, but in other cases it may be a mistake. In general, THREADSAFE works on code that has no formal specification describing what the intended behaviour is. Arguably, the most damaging concurrency errors are not the low-level data races and race conditions on concurrent collections, but high-level design errors that lead to user-visible data inconsistencies. However, in the absence of information about what ought to happen, these bugs are the most difficult for a static analysis tool to find.

4.2 Testing a static analyser

Beyond the basic question of what kinds of errors THREADSAFE ought to catch, or not catch, a practical aspect of the development of any heuristic static analysis is how to maintain and track the behaviour of the analysis as it is added to and bugs are fixed. It is all too easy to fix a bug that removes some false positive, and at the same time also remove many more valuable true positives. Whether or not such trade-offs are a net positive is one of the hardest parts of developing a static analyser intended for a mass audience. But in order to be able to make judgements about such tradeoffs, we need to have visibility into the effect of changes we make during development of THREADSAFE.

To ensure that development of THREADSAFE does not accidentally introduce new false positives, or remove true positives, we maintain a large test suite of over 530 tests identifying specific true/false positives and negatives, and for each one whether it is expected to be reported or not. Since most of the checkers in THREADSAFE rely to a greater or lesser extent on heuristics, the only practical way to document and maintain the desirable behaviour of checkers is via testing — there is no formal specification of how THREADSAFE ought to behave.

Since the individual test cases are not representative of real Java projects, THREADSAFE is also systematically tested against a portfolio of open source and proprietary Java code bases. The analysis is tested against the benchmark suite before and after each change determine the impact of altered heuristics, or new checkers. We do not attempt to classify all the findings of THREADSAFE on the benchmark suite — this turns out to be too difficult and time consuming for a small company, see What is a Concurrency Bug? above — but we do inspect the difference in the analysis results before and after the change, to determine whether the proposed change has an overall positive impact.


5 Performance

5.1 Measuring static analysis tool performance

The performance of a static analysis tool like THREADSAFE can be measured in a number of ways. For example:

- What proportion of the real bugs in a given codebase does the tool find (true positives) and miss (false negatives)?
- What is the potential impact of the bugs found versus the ones missed?
- How difficult would it be to find the same bugs by other means?
- What percentage of findings produced are false alarms (false positives)?
- How easy is it to investigate a finding to discover if it is a genuine bug and how to fix it?
- How much time and space does the tool consume?
- How well does it scale for use with very large codebases?
- How smoothly is the tool integrated with a software developer’s existing workflow?

Many of these are difficult to measure. For instance: in any very large real-world codebase, computing the proportion of real bugs found assumes knowledge of all of the real bugs that it contains. Determining whether or not a finding reports a real bug or is a false alarm, and the potential impact of a real bug, often relies on expert knowledge of the architecture of the system and/or the way that it is intended to be used. Some aspects are somewhat subjective and a matter of taste or attitude. Finally, measurements of performance on small or textbook examples may bear little relation to performance on large real-world codebases.

For a commercial static analysis tool like THREADSAFE, intended for use in industrial software development, the tradeoffs between these factors are different from what an academic researcher might expect. When working with a million-line codebase, developers are primarily interested in finding and fixing very high-impact bugs quickly. Their experience tells them that some bugs will remain, no matter what they do. And since fixing a bug might introduce a new bug, as explained earlier, it might well be prudent to leave a low-impact bug unfixed. A tool that reports large numbers of low-impact bugs alongside a few high-impact bugs may therefore be less useful than one that reports only the high-impact bugs. But filtering lists of findings according to type will sometimes allow likely high-impact bugs to be separated from low-impact bugs.

Any static analysis tool needs to balance false positives against false negatives: higher sensitivity will tend to reveal more likely bugs, but some of those found will turn out to be false alarms. If the aim is to eliminate high-impact bugs quickly, then a high level of false positives is more damaging than a moderate level of false negatives. Developers will quickly lose patience with a tool if they need to investigate a long list of false alarms before encountering a genuine high-impact bug.
More interesting than the absolute performance of a tool is its performance in comparison to competing tools. Unfortunately, a head-to-head comparison between THREADSAFE and competing commercial tools is not possible because their licenses forbid their use for this purpose. Our experiments suggest that THREADSAFE outperforms other commercial static analysis tools with respect to detection of Java concurrency defects, and users of other tools are invited to undertake a comparison themselves. We provide a comparison below between THREADSAFE and the non-commercial FindBugs system [Fin15], the most widely-used static analysis tool for Java.

5.2 Performance on IBM benchmark

The IBM concurrency bugs benchmark [EHSU07, ETU08, IBM15] is a collection of Java programs with concurrency bugs, meant for use in comparing bug-finding tools; it is the most widely-used benchmark for this purpose. Most are small programs produced by students at the University of Haifa as coursework for a software testing course and the bugs vary in complexity. An example of a simple bug is a use of the double-checked locking anti-pattern [Pug00] in the program DCL. An example of a bug that seems well beyond the reach of an automatic static-analysis tool is an atomicity bug in the program Lottery that leads to violation to an implicit logical invariant. An earlier version of the benchmark was annotated with information about the location(s) and type(s) of bugs in each program, but this information and some of the programs in the benchmark have been lost [TB15].

THREADSAFE and FindBugs (with only “multithreaded correctness” checkers enabled) were both applied to each program, first with default settings (first column) and then with sensitivity increased (second column). For THREADSAFE, sensitivity was increased by reducing the threshold for inconsistent synchronisation from 70% to 50%. For FindBugs, sensitivity was increased by setting analysis effort to “maximal” and setting the minimum rank to report so that all findings would be reported. Findings produced by both tools were manually classified as true positives or false positives and the number of concurrency bugs in each program was counted. The TP/FP classification and count of bugs are admittedly somewhat subjective.

The results produced by THREADSAFE and FindBugs on this benchmark are given in Table 1. There are many bugs in the benchmark, including the atomicity bug in Lottery mentioned above, that neither tool is able to detect. THREADSAFE finds about twice as many bugs as FindBugs does, with about the same false positive rate.

There are a few bugs that FindBugs detects that THREADSAFE, with its more sophisticated analysis, misses. An example is the bug in Piper, where there is a call to java.lang.Object.wait() that is not in a loop. FindBugs contains a checker for exactly this bug pattern. In developing THREADSAFE, we decided not to duplicate checkers in FindBugs unless we were able to significantly improve on its results, since it is easy to use FindBugs alongside THREADSAFE. This is just such a case.

Spathoulas [Spa14] did a similar comparison of THREADSAFE with FindBugs on the IBM benchmark and on concurrency examples from The CERT Oracle Secure Coding Standard for Java [LMS+11], but without classifying findings into true and false positives, comparing THREADSAFE with default sensitivity against FindBugs with increased sensitivity. He also compared the tools’ accuracy on these examples with very simple obfuscation methods applied, as a first step towards assessing their performance when faced with more complex and hidden bugs. He found
<table>
<thead>
<tr>
<th>Class</th>
<th>THREADSAFE default TP / FP</th>
<th>THREADSAFE sensitive TP / FP</th>
<th>FindBugs default TP / FP</th>
<th>FindBugs sensitive TP / FP</th>
<th># bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_B_push_pop</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>Account</td>
<td>0 / 0</td>
<td>1 / 0</td>
<td>0 / 0</td>
<td>1 / 0</td>
<td>1</td>
</tr>
<tr>
<td>Airlines Tickets</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 1</td>
<td>0 / 1</td>
<td>1</td>
</tr>
<tr>
<td>AllocationVector</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>BoundedBuffer</td>
<td>0 / 3</td>
<td>0 / 4</td>
<td>0 / 0</td>
<td>0 / 3</td>
<td>1</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>1</td>
</tr>
<tr>
<td>BubbleSort2</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>BufWriter</td>
<td>1 / 3</td>
<td>0 / 4</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>2</td>
</tr>
<tr>
<td>Critical</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 1</td>
<td>0 / 1</td>
<td>1</td>
</tr>
<tr>
<td>DCL</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>1</td>
</tr>
<tr>
<td>Deadlock</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 1</td>
<td>0 / 1</td>
<td>1</td>
</tr>
<tr>
<td>DeadlockException</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>Fiforeadwritelock</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>FileWriter</td>
<td>0 / 0</td>
<td>2 / 0</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>1</td>
</tr>
<tr>
<td>GarageManager</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>2</td>
</tr>
<tr>
<td>Hierarchy Example</td>
<td>0 / 1</td>
<td>0 / 1</td>
<td>0 / 0</td>
<td>0 / 1</td>
<td>1</td>
</tr>
<tr>
<td>LinkedList</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>Liveness</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>Lottery</td>
<td>1 / 0</td>
<td>2 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>2</td>
</tr>
<tr>
<td>Manager</td>
<td>2 / 3</td>
<td>2 / 3</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>2</td>
</tr>
<tr>
<td>MergeSort</td>
<td>0 / 1</td>
<td>0 / 1</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>MergeSortBug</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>PingPong</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>Piper</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>1</td>
</tr>
<tr>
<td>ProducerConsumer</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>Shop</td>
<td>1 / 0</td>
<td>1 / 0</td>
<td>0 / 0</td>
<td>1 / 0</td>
<td>2</td>
</tr>
<tr>
<td>Suns Account</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>0 / 0</td>
<td>1</td>
</tr>
<tr>
<td>XtangoAnimation</td>
<td>5 / 0</td>
<td>5 / 0</td>
<td>2 / 3</td>
<td>2 / 5</td>
<td>5</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>14 / 11</strong></td>
<td><strong>19 / 16</strong></td>
<td><strong>7 / 5</strong></td>
<td><strong>9 / 11</strong></td>
<td><strong>37</strong></td>
</tr>
</tbody>
</table>

False positive rate: FP / (FP+TP) 44% 46% 42% 55%
Percentage of bugs found 38% 51% 19% 24%

Table 1: THREADSAFE and FindBugs applied to the IBM benchmark
that THREADSAFE was much better than FindBugs at resisting obfuscation, which is to be expected because of the simple bug-pattern detection methods used by FindBugs.

Although comparison of performance on small examples like the ones in the IBM benchmark provides a useful data point, is not a reliable guide to performance on large real-world examples. At least, that is our experience with deadlock detection. The heuristics that THREADSAFE uses to achieve a good rate of deadlock detection in reasonable time sometimes yield poor performance on small “textbook-style” examples. For real industrial-scale examples, its results tend to be much better. THREADSAFE’s heuristics could be adjusted to perform better on the small examples, but this would increase the false positive rate on larger examples. We have not yet found a way to achieve good results in both cases.

6 Packaging

THREADSAFE is tightly integrated with Eclipse, as shown above, and with the SonarQube quality platform. It can also be used from the command line to generate an HTML report containing information similar to what is available in Eclipse.

Contemplate’s website http://www.contemplatetltd.com/threadsafe provides more information about THREADSAFE and access to free two-week trials.

Bibliography


http://bugs.java.com/view_bug.do?bug_id=7027300

[Pug00] W. Pugh. The ”double-checked locking is broken” declaration. 2000. 

http://www.itu.dk/people/sestoft/itu/PCPP/E2014/

http://homepages.inf.ed.ac.uk/dts/students/spathoulas/spathoulas.pdf

http://mailinator.blogspot.co.uk/2009/06/beautiful-race-condition.html

Verifying Critical Cyber-Physical Systems After Deployment

Colin O’Halloran

University of Oxford

Abstract: Cyber-Physical Systems (CPS) are increasingly novel hardware and software compositions creating smart, autonomously acting devices, enabling efficient end-to-end workflows and new forms of user-machine interaction. The heterogeneous, evolving and distributed nature of CPS means that there is little chance of performing a top down development or anticipating all critical requirements such devices will need to satisfy individually and collectively. This paper describes an approach to verifying system requirements, when they become known, by performing an automated refinement check of its composed components abstracted from the actual implementation.

Keywords: Autonomy, Cyber-Physical Systems, Internet of Things, Verification, Validation, Security, Safety

1 Introduction

Autonomy does not reside in one particular software system or Electronic Control Unit (ECU). Autonomy arises out of the collection of ECUs responsible for: processing sensor inputs, i.e. perceiving; deciding what to do; and acting upon the decision made. Establishing the safety of an autonomous system, therefore, means assuring the composition, of these Perceive-Decide-Act functions, is safe. There is currently a lot of work on the development of driverless cars as well as other autonomous air and maritime vehicles. In the automotive sector a manufacturer might well develop the decision-making software themselves, but their business model depends upon buying and integrating third-party ECUs from their supply chain with decision-making software. For example, Bosch supplies systems to many car manufacturers and their software design is their protected IP. This means that autonomy relies upon third-party software for which limited assurance is available. To varying degrees the same is true of the aerospace and maritime sectors.

More generally Cyber-Physical Systems (CPS) are increasingly novel hardware and software compositions creating smart, autonomously acting devices, enabling efficient end-to-end workflows and new forms of user-machine interaction. Currently CPS range from large supervisory control and data acquisition (SCADA) systems that manage physical infrastructure to medical devices such as pacemakers and insulin pumps, to vehicles such as airplanes and satellites In the future they will include low power small devices that will make up the Internet of Things.

Clearly resilience, safety, security and privacy are of paramount concern. However, the heterogeneous, evolving and distributed nature of CPS means that there is little chance of performing a top down development or anticipating all critical requirements such devices will need to satisfy individually and collectively.
The looming problem is therefore how to address the verification and validation of such systems when they are critical and rely upon other systems that have limited assurance? Worse still, how can CPS be verified when the requirements will not be known until after they are deployed and composed with other CPS to form unanticipated critical services?

An approach to addressing this problem is to automatically extract the behaviours of such deployed systems that are composed together through well-defined architectures, or protocols. At this point system requirements can be articulated because the overall system of systems has been constructed for some purpose. This means that a system level representation of the relevant behaviours is needed.

The importance of pushing the representation up to the level of system properties is that it is the natural level for people to articulate requirements, it is often difficult to express these as lower level specifications for components unless it is part of a top down development. Such top down development has already been excluded from the problem space being addressed. Verifying components against specified lower level properties that are generally desirable is rather hit and miss. The history of developing secure systems is strewn with examples of systems where low level security properties to hold, but the system is insecure. Conversely it is also possible for the system level to be secure, but low level vulnerabilities to be present – but not exploitable at the system level. There is no reason to believe that the same is not true for systems that are required to be safe.

Another advantage of working with system level requirements is that they can guide what behaviours are relevant to the system level requirements, i.e. they drive relevant abstractions that can be performed. For example a sub-system on a CAN bus might have no relevant critical behaviour except that it does not flood the CAN bus with messages when critical messages are being passed between other sub-systems.

In the past the approach advocated in this paper would have been perceived to have too many obstacles for any chance of success. Advances in various areas of automated reasoning and model checking now, perhaps, make such an approach possible. In the rest of the paper a small example representing encrypted communication to a hobbyist quadcopter is given to illustrate the approach. The software written in C is abstracted to a system level and then verified against a property that encodes the sort of attack a penetration tester might try. Finally future work that employs de-compilation of executable binary into a mathematical representation is outlined along with related work.

2 The System

The example of a system requirement used to illustrate the approach is that of absence from the system’s behaviour of one type of penetration tester attack. The simplified example system is based upon an early version of the SMACMMPilot [1] developed under the DARPA High Assurance Cyber Military Systems (HACMS) project [2]. SMACMMPilot is an open-source autopilot software for small unmanned aerial vehicles (UAVs) using new high-assurance software methods, the simple C code used in the example is not from SMACMMPilot.
The relevant part of the system for the requirement of interest is that of encrypted pairwise communication between a ground station and SMACMMPilot and a safety controller and SMACMMPilot, illustrated in Figure 1.

Within the quadcopter’s software there are two channels: Stream1 and Stream2. Stream1 consists of: an internal buffer (for the packets transmitted from the ground station), denoted GCSXMIT; a decryption function; and a process that assembles the message for processing according to the MAVLINK protocol [3]. A diagram representing Stream1 and Stream2 appears in Figure 2.
Stream2 is a copy of Stream1. The implementation of Stream1 is in terms of 4 reactive processes that use 2 system services (send and receive) to communicate with each other. The 4 processes are GCS, which models the ground station transmitting data; DLR, which takes the data from GCS and sends it onward to be decrypted; DECRYPT, which decrypts the data and sends it onward to be dealt with by RECVR before sending onwards to the MAVLINK protocol software.

In the process algebra of Communicating Sequential Processes [4], CSP, the architectural composition of the streams of Figure 2 is shown in Figure 3.

\[(\text{STREAM1} \parallel \{\{\text{GCSXMIT}\}\} \parallel \text{STREAM2}) \parallel \{\{\text{R2M}\}\} \parallel \text{MAVLINK(\text{R2M})}\]

**Figure 3: CSP composition of communication streams with MAVLINK**

STREAM1 and STREAM2 communicate through the messages over the channel GCSXMIT. MAVLINK communicates with STREAM1 and STREAM2 through the messages over R2M. The messages over GCSXMIT consist of symbolic values for a key, a sequence number and data. The messages over R2M consist of a sequence number and data. In CSP the composition of STREAM1 is in terms of the 4 processes GCS, DLR, DECRYPT and RECVR is shown in Figure 4.

**Figure 4: Composition of STREAM1**

The messages over D2D (like GCSXMIT) consist of the symbolic values for a key, a sequence number and data. The C code for the function GCS, is shown in Figure 5.

```c
void GCS(int Key, chan* out) {
    SMsg GCS_local_message;
    GCS_local_message.Key = Key;
    GCS_local_message.cmsg.SeqNo = 0;
    GCS_local_message.cmsg.msg.val = 0;
    for (;;) {
        GCS_local_message.cmsg.SeqNo = (GCS_local_message.cmsg.SeqNo+1) % 4;
        send(&GCS_local_message, sizeof(GCS_local_message), out);
    }
    return;
}
```

**Figure 5: C function GCS**
Note that the function GCS does not terminate because the ‘for’ loop, which calls a service to send a packet onward, does not terminate. The simplifications of functionality do not detract from the points this paper makes about architectural composition, abstraction and verification of system properties.

The composition of STREAM2 is similar to STREAM1 (shown in Figure 4), but a process modelling an intruder’s behaviour replaces the process GCS in Figure 6.

STREAM2 =
INTRUDER(GCSXMIT, ICPY)
[ICPY<->ICPY]
( DLR(ICPY, D2D)
[D2D <-> D2D]
( DECRYPT(Sk, D2D, D2R) [D2R <-> D2R] RECVR(0, D2R, R2M.CMsgStruct.1) )
)

Figure 6: Composition of STREAM2

The intruder’s behaviour is modelled abstractly as listening to the transmissions from the ground station and replaying them, or non-deterministically ignoring a transmission. The CSP process modelling this behaviour is shown in Figure 7.

INTRUDER(in, out) =
in?x -> (out!x -> INTRUDER(in, out))
|~|
in?_ -> INTRUDER(in, out)

Figure 7: Intruder behaviours

The behaviours of GCS, DLR, DECRYPT, RECVR and MAVLINK are determined by abstracting the code to a predicate representing initial and final states. The abstracted predicate is then translated into CSP enabling the overall system composition, shown in Figure 3, to be verified with respect to a property of robustness against a certain form of external attack.

3 Abstracting the implementation into CSP

MAVLINK is used to illustrate the abstraction of the C code (shown in Figure 8) to a specification and then translated into CSP.

int MAVLINK(chan* in) {
    SMsg MAVLINK_local_x;
    do
        {recv((SMsg*) &MAVLINK_local_x, sizeof(MAVLINK_local_x), in);}  
    while (1);
    return 1;
}

Figure 8: C code for MAVLINK
The code simply acts as a sink for the decrypted messages, the extra functionality for a real implementation would simply be incorporated into the abstracted predicate.

A tool, called FSG and developed by D-RisQ under the DARPA HACMS project, essentially generates the strongest post-condition for a C function. The tool, implemented on top of ProofPower’s QCZ [5] tool, generates as output the C compliance notation - a wide spectrum language that includes specifications in the Z language as well as C code. When FSG is run on the C function of Figure 8 it generates the specification statement shown in Figure 9.

Figure 9: FSG output for MAVLINK

The specification states that if the “do loop” within MAVLINK doesn’t terminate then the result is a relational closure called ‘DoLoop0’. Note that neither parameter to ‘DoLoop0’ is changed by ‘DoLoop0’. The point of the predicate is not to reason about it directly to establish some assertion; it is there to record sufficient information in order to translate into CSP.

Figure 10: First refinement step

The soundness of the abstraction is established by reversing it into refinement steps that generate verification conditions that, if proven by ProofPower, mean that the original C code is
correct with respect to the specification statement. For example, Figure 10 is a refinement of Figure 9; and Figure 11 is a refinement of Figure 10. ProofPower’s QCZ tool processes these refinement steps to generate verification conditions that can then be proven by ProofPower.

Figure 11: Second refinement step

Returning to Figure 9, it is translated into CSP as the process MAVLINK defined in Figure 12.

\[
\text{MAVLINK(in)} = \text{DoLoop0(in, ([], SKIP)}
\]

Figure 12: Definition of the CSP process MAVLINK

The communication channel ‘in’ is a parameter of MAVLINK, reflecting the channel ‘in’ passed as a parameter to the C function MAVLINK in Figure 9.

The definition of DoLoop0 is a relational composition of a function called DoLoopBody0 with its own reflexive transitive closure, i.e. the loop body sequentially composed with a “guarded” set of all finite iterations of the loop body. The guard takes the form of domain restriction. The co-domain anti-restriction characterises the state on loop termination, if the loop does not terminate then this is the empty set.

\[
\forall in_V : VALUE_C \times LOC_C, MAVLINK_{\text{local}}.x_L : LOC_C
\]
\[
\bullet \text{DoLoop0} (in_V, MAVLINK_{\text{local}}.x_L) = (\text{DoLoopBody0} \triangle (\text{DoLoopTest0} \triangleright \text{DoLoopBody0}) \triangleright \text{DoLoopTest0})
\]

Figure 13: FSG generated Z axiomatic definition of DoLoop0

As the name implies, the “do loop” in the C code executes the body of the loop before the loop guard is evaluated. In this example the loop guard is true and the body is executed indefinitely giving the behaviour of a reactive system because of the call of the external function ‘recv’ used as part of the definition of DoLoopBody0 in Figure 17. The translation of DoLoop0 into machine readable CSP, called CSPm, is shown in Figure 14. It models predicate equalities by use of a let clause and then calls the externally provided function of a receive service that takes the packets of information decrypted by streams 1 and 2.
DoLoop0(in_V, Indirections,Q) =
  let
  recv__x = MAVLINK_local_x_L
  recv__y = SizeOf(SMsg)
  recv__queue = in_V
  within
  EXTERNAL_recv(recv__x,recv__queue,Loop0,in_V, Indirections,Q)

Figure 14: CSP translation of DoLoop0

The external function has two arguments that in the CSP are augmented by three more that are used to supply continuation information for subsequent behaviour. The augmented definition of ‘EXTERNAL__recv’ is shown in Figure 15.

EXTERNAL_recv(x_v',queue,P,i,Indirections,Q) =
  let
  Input_Channel = queue
  within
  Input_Channel?Input_Value ->
  let
  Indirections1 = mapUpdate(Indirections,x_v',Input_Value)
  within
  P(i,Indirections1,Q)

Figure 15: Pre-defined definition of the external service recv

The argument ‘Indirections’ is a map function of the Haskell like functional language that augments the process algebra of CSP that is CSPm. The map function ‘Indirections’ models simple pointers, such as those used for efficient parameter passing. The process EXTERNAL_recv is defined by a human as part of the model of the CSPm model of the system architecture. It simply takes an input value received and updates the map function that models pointers. It then behaves like the process P supplied as an argument to EXTERNAL_recv with P’s required arguments, which also includes another continuation process Q. The actual parameter, Loop0, substituted for P in EXTERNAL_recv is shown in Figure 14.

Loop0(in_V, Indirections,Q) =
  if true
  then
  DoBody0_Star( in_V, Indirections,Loop0,in_V,Indirections,Q)
  else
  let
  MAVLINK_output = true
  within
  Q

Figure 16: Process supplied as part of a continuation parameter in DoLoop0

The continuation process Loop0, defined in Figure 16, corresponds to the translation of the reflexive transitive closure of the loop body. The loop guard is true therefore the loop never
terminates and hence, in the process definition, the ‘else’ part can never be taken. In general if a loop terminates the process Q takes over the flow of execution. The CSPm definition of DoBody0_Star is translated from the FSG generated axiomatic definition of DoLoopBody0 (used in the definition of DoLoop0 in Figure 14); DoLoopBody0 is shown in Figure 17.

\[
\text{DoLoopBody0} : \text{VALUE}_C \times \text{LOC}_C \rightarrow \text{VALUE}_C \times \text{LOC}_C
\]

\[
\forall \text{in}_V : \text{VALUE}_C; \text{MAVLINK\_local\_x}_L : \text{LOC}_C
| \text{B}_C (\forall \text{recv\_queue}, \text{recv\_recv}, \text{recv\_x}, \text{recv\_y} : \text{VALUE}_C; \sigma, \sigma' : \text{STORE}_C
\]

\[
\land \text{recv\_y} = \text{IntVal}_C
\]

\[
(\text{SizeOf}_C
\]

\[
(\text{StructType}_C
\]

\[
\langle\langle "\text{Key}\" , \text{ArithmeticType}_C (\text{Signed}_C \text{Int}_C) , \langle "\text{msg}\" , \text{StructType}_C
\]

\[
\langle\langle "\text{SeqNo}\" , \text{ArithmeticType}_C (\text{Signed}_C \text{Int}_C) , \langle "\text{msg}\" , \text{StructType}_C
\]

\[
\langle\langle "\text{val}\" , \text{ArithmeticType}_C (\text{Signed}_C \text{Int}_C) \rangle\rangle\rangle
\]

\[
\land \text{recv\_queue} = \text{in}_V
\]

\[
\bullet (\text{recv\_post})
\]

\[
[\text{recv\_queue}/\text{queue}_v, \text{recv\_recv}/\text{recv}_v!, \text{recv}_v/x_v, \text{recv\_y}/y_v)]
\]

\[
\neq \text{IntVal}_C 0
\]

\[
\bullet \text{DoLoopBody0} (\text{in}_V, \text{MAVLINK\_local\_x}_L) = (\text{in}_V, \text{MAVLINK\_local\_x}_L)
\]

Figure 17: FSG generated Z axiomatic definition of the loop body

Although it looks (at first sight) to be complicated, Figure 17 is simply setting the abbreviations for expressions that are substituted for formal parameters of the external function ‘recv’. For example the identifier ‘recv\_x’ is an abbreviation for the address of a pointer and ‘recv\_y’ is an abbreviation for the size of the structure passed to ‘recv’. The identifier recv\_post is the schema predicate that is the specification for the external function ‘recv’ and is defined by a human like the corresponding CSPm process.
The translation into CSPm of DoLoopBody0 of Figure 17 is shown in Figure 18.

\[
\text{DoBody0\_Star(in\_V, Indirections,P,i,j,Q) =}
\]
\[
\quad \text{let}
\]
\[
\quad \quad \text{recv\_x = MAVLINK\_local\_x\_L}
\]
\[
\quad \quad \text{recv\_y = SizeOf(SMsg)}
\]
\[
\quad \quad \text{recv\_queue = in\_V}
\]
\[
\quad \text{within}
\]
\[
\quad \quad \text{EXTERNAL\_recv(recv\_x,recv\_queue,P,i,j,Q)}
\]

Figure 18: Translation of DoLoopBody0 into CSPm

The definition of the process DoBody0\_Star is similar to that of DoLoop0 defined in Figure 14 and reflects the axiomatic definition of the loop body in Figure 17. The required recursion is achieved by supplying ‘Loop0’ as the continuation P (along with Loop0’s arguments ‘in\_y’ for ‘i’ and ‘Indirection’ for ‘j’) that is supplied to EXTERNAL\_recv in Figure 16.

The translation of the reactive process MAVLINK into CSPm along with the similarly translated C code implementations of the reactive processes, which make up Stream1 and Stream2, can be plugged into CSPm system architecture definition of Figure 3.

4 Verification of a System Property

The encrypted communication over two streams (one for a ground control station and one potentially for a safety controller) is represented in CSPm as

\[
\text{SYSTEM = (STREAM1 \[|\{|GCSXMIT|\}|\] STREAM2) \[|\{|R2M\}|\] MAVLINK(R2M)}
\]

Figure 19: CSPm SYSTEM definition

The process MAVLINK is represented by the CSPm process

\[
\text{MAVLINK(in) = DoLoop0(in, (| |), SKIP )}
\]

where the definition of DoLoop is derived from the specification generated from the FSG tool that abstracts the C code implementation, as described in section 3. Thus the behaviour of MAVLINK instantiated with the channel R2M can be plugged into the SYSTEM definition above. Similarly the behaviours of the process that make up STREAM1 and STREAM2 can be derived from their implementation in a similar manner to that described in section 3.

The quadcopter is a Cyber–Physical system that was initially developed without reference to potential penetration tester type attacks. A form of external attack is to interfere with the commands sent from the ground control station by replaying them through the second communication stream. A specification for robustness against this type of attack is that once started with a particular input stream, it must not be possible to receive messages from the other stream. The specification can be specified operationally as the CSPm process in Figure 20.
Performing a refinement check using the FDR3 model checker of SYSTEM (shown in Figure 19) against the specification in Figure 20 returns a failure with a counterexample of undesirable system behaviour. The weakness is that because the implementation of Stream2 is a copy of Stream1 it can accept replayed messages from the other stream. The same system representation can be used to check other system security properties specified in CSPm.

5 Related Work

The High Assurance Cyber Military Systems (HACMS) project [2] is creating technology for the construction of high-assurance Cyber-Physical Systems, where high assurance is concerned with functional correctness and satisfaction of appropriate safety and security properties. The work described in this paper is part of the “Red Team” effort that is seeking to use “traditional” penetration testing techniques allied to analysing the system using automated verification. In section 6, ongoing work is discussed that seeks to de-compile executable binary into the C compliance notation from which a specification can be generated and translated into CSPm. Thus the aim is to use the C compliance notation as an intermediate representation generated from a de-compiler.

Recent advances indicate that the time is ripe for verifying properties of systems. A number of projects on systems verification have emerged, most notably, the L4 verified OS Kernel project [6]. The project developed and formally verified a high-performance microkernel on ARM and x86 architectures. Such microkernels are a critical core component of modern embedded systems architectures. They are the piece of software that has the most privileged access to hardware and regulates access to that hardware for the rest of the system. Virtually every modern smart-phone runs a microkernel quite similar to seL4.

In the rest of the HACMS project a clean-slate approach has been adopted using formal methods-based approaches to enable semi-automated code synthesis from executable, formal specifications. In addition to generating code, HACMS seeks a synthesizer capable of producing a machine-checkable proof that the generated code satisfies functional specifications as well as security and safety policies. Work on extending the verified L4 kernel to an RTOS is taking place within HACMS. All the preceding work described involves top-down development of software from scratch where the safety and security requirements are known. The approach advocated in this paper aims to address software that already exists and when requirements only emerge later.

Another notable example is the Verisoft and Verisoft XT projects [7]. They verify special-purpose operating systems and a hypervisor, at the level of source code but without I/O. There have also been systems verifications that focus on safety properties, most notably, Yang and Hawblitzel’s type-safe operating system [8]. New supporting technology, such as programming logics for system source code, have been developed, e.g. by Shao's group at the
University of Yale [9]. In all of the above lines of work, verification assumes availability of the source code.

The Research Institute in Automated Program Analysis and Verification [10] in the UK has two related research projects. The first is “Compositional Security Analysis for Binaries” that is translating binaries to C code in order to apply software analysis tools like CMBC [11]. However this approach is focussed on low-level properties, like zeroing of released memory, rather than system-level properties. The second project is “Program Verification Techniques for Understanding Security Properties”, focussed on source code rather than binaries, and is therefore not applicable to software that has already been developed.

The VATES project [12] aims at verifying behaviour of embedded software at the level of an LLVM intermediate representation. It uses a top-down approach by inventing an abstract CSP-based specification and then formally relating a model of a concrete system implementation, given in the form of the LLVM intermediate representation. This demonstrated that well-established formal verification techniques for declarative process-algebraic specifications are applicable to low-level software programs. The VATES project developed a formal operational semantics of LLVM and established a bisimulation relation between LLVM and CSP models in order to prove that a given LLVM program is a correct implementation of a given CSP model. Again a top-down development was assumed and the translation from the intermediate representation down to assembly is just assumed to be correct.

The most relevant work has been that of Magnus Myreen who developed an approach to formal verification of machine-code [15] that has been exploited to verify binary code [16]. It has been subsequently extended by Fox to deal with more ISAs [17]. Myreen and Fox’s work has so far targeted ARM, Intel x86 and IBM PowerPC processors with a proof-producing de-compiler, which given some machine code for any of the processors mentioned above will automatically, via proof, derive a functional program from the machine code. This functional program is a record of the state change the machine code can perform. Although the approach only provides a low level description in Higher Order Logic it could form the basis for connecting binary executables to system architectures.

6 Ongoing and Future Work

Within the HACMS project, the lead organisation for the Red Team is the Charles Stark Draper Laboratories, or Draper Labs. Draper Labs are working on a de-compiler tool under HACMS. The HACMS de-compiler, called the Fracture tool, converts Executable Object Code for an Instruction Set Architecture (ISA) into an intermediate representation called LLVM [13].

Current work on the HAMCS de-compiler is attempting to emit a representation, in the C compliance notation, from LLVM that is abstracted into a predicate by FSG and then translated into CSPm for the refinement model checker, FDR3 [14], to verify against system security properties. Work is also ongoing on the formal specification of a translator from the specifications, generated by the FSG tool, to CSPm.
Myreen’s de-compilation technology provides advantages over the use of Fracture and FSG. For example the Fracture tool does not provide any assurance of correctness against the semantics of the ISA used. The manipulation of LLVM and the translation into the C compliance notation is similarly unassured. Similarly there is no planned verification of the translation from the output of FSG to CSPm. As a consequence any reported violations of system properties may be false positives and satisfaction of the properties cannot be relied upon. Such limitations make its use questionable when safety is required, although its role within the Red Team’s assessment of systems is still useful.

If Myreen’s de-compilation technology could be allied to a system representation, as illustrated by this paper, then system safety and security requirements could be established with high assurance and automatically. The gap between functional representations of binary executable in Higher Order Logic and CSPm can be addressed through the Unified Theories of Programming (UTP) [18]. An assured translator could be built on top of a current tool [19] that supports UTP and is built on top of Isabelle.

To employ the FDR3 model checker for properties that were sensitive to data values would require a two pronged approach. One is to build data abstraction into the translation from the representation in Higher Order Logic to CSPm. The other is the development of a symbolic model checker for CSPm to assure such data abstractions. The capability of FDR3 to harness cloud resources to check over large state spaces means that it is plausible that embedded systems could be within the reach of automated verification against system properties. For example FDR3 checked a trillion states using Amazon’s EC2 web service in a few hours at a cost of approximately $70 [15].

The only part that could not be automated is the human understanding of architectural patterns used for Cyber-Physical Systems. Work on identifying such patterns is ongoing by Draper Labs. The specification of the compositional framework would have to be done by a human, but it is conjectured that there are a relatively small number of such patterns. It is envisioned that a library of re-usable compositions, like the one described in this paper, could be created by human effort and then automatically incorporated into a system representation for verification.

7 Acknowledgements

I am hugely indebted to Neil Brock from Draper Labs for discussions about this problem area and for providing me with the C code and architectural composition used in this paper.

8 References

1. http://smacmpilot.org/
3. https://github.com/mavlink
10. https://verificationinstitute.org/
Regular Papers
Estimating Rewards & Rare Events in Nondeterministic Systems

Axel Legay, Sean Sedwards and Louis-Marie Traonouez

Inria Rennes – Bretagne Atlantique

Abstract: Exhaustive verification can quantify critical behaviour arising from concurrency in nondeterministic models. Rare events typically entail no additional challenge, but complex systems are generally intractable. Recent work on Markov decision processes allows the extremal probabilities of a property to be estimated using Monte Carlo techniques, offering the potential to handle much larger models. Here we present algorithms to estimate extremal rewards and consider the challenges posed by rarity. We find that rewards require a different interpretation of confidence and that reachability rewards require the introduction of an auxiliary hypothesis test. We show how importance sampling can significantly improve estimation when probabilities are low, but find it is not a panacea for rare schedulers.

Keywords: Nondeterminism, Markov decision process, rewards, rare events, Monte Carlo, importance sampling, statistical model checking.

1 Introduction

Complex systems often contain implicit elements of uncertainty, arising from human interactions or unknown deployment configurations. Such uncertainty may be expressed as nondeterminism in formal models and then analysed using formal verification. In this work we focus on Markov decision processes (MDP), although the techniques we present are adaptable to other formalisms that include nondeterminism.

MDPs interleave nondeterministic actions and probabilistic transitions and may be seen as comprising probabilistic subsystems whose transitions depend on the states of the other subsystems. Assigning rewards to actions, MDPs have proven useful in many real optimisation problems [Whi93]. Rewards are a useful decoration to model quantitative aspects of systems that are not expressible as probabilities, such as information, energy and financial cost.

Given a system represented as an MDP, model checking [BK08] may be used to detect and optimise its critical performance, but exhaustive numerical algorithms have complexity related to the size of the system’s state space and scale exponentially with respect to the number of independent variables in the model. The application of symmetry reductions and compositional approaches may help, but real systems are typically intractable due to their scale and heterogeneity.

Statistical model checking (SMC) describes a collection of Monte Carlo techniques that approximate the results of numerical model checking and aim to avoid considering the entire state space. States are generated on the fly during simulation and results are given with statistical confidence. Monte Carlo techniques may be divided linearly on parallel computation architectures and are generally insensitive to the number of states of a model, but the number of simulations required for a given confidence scales quadratically with the rarity of the event being quantified.
Since systems are designed to function correctly, failure is typically very rare and critically important. Rare event techniques, such as importance sampling [Sha94] and importance splitting, have therefore been applied to SMC.

SMC requires a means to sample execution traces, but MDPs deliberately avoid completely specifying how a system executes. The notion of a scheduler is thus used to transform an MDP into a discrete time Markov chain that can be executed using stochastic simulation. The verification problem becomes one of finding optimal schedulers—schedulers that optimise some quantitative property.

Early attempts to apply SMC to MDPs store schedulers explicitly, but schedulers typically have the same complexity as the system as a whole, so these approaches tend to scale no better than numerical algorithms. In [LST14, DLST15] the authors use pseudo-random number generators to define the possibly infinite behaviour of schedulers in constant memory, allowing them to be sampled at random and tested individually. While sampling offers a significant computational saving over enumeration, it raises the problems of rare schedulers and rare events. The problem of rare schedulers has been partially ameliorated by ‘smart sampling’ [DLST15], but properties with low probability make even non-rare schedulers difficult to find.

In this work we enhance the ideas of [LST14, DLST15] with importance sampling, to significantly increase the chance of finding optimal schedulers when properties have low probability. Since rewards properties have not yet been implemented in the sampling context of [LST14, DLST15], we also present an algorithm to find schedulers that approximately maximise or minimise expected rewards. Unlike probabilities, rewards have no inherent a priori bounds, so the standard statistical techniques to bound the absolute error do not apply. Our solution is to use a relative bound, based on a generalisation of the Chernoff-Hoeffding bound often used in SMC [Oka58]. A further challenge is that the commonly used ‘reachability rewards’ [KNP07] assume the probability of a property is known with absolute certainty and define the reward of properties with probability less than 1 to be infinite. With no additional information about the property, this definition induces an unknown distribution with potentially infinite variance. As such its properties cannot be directly estimated by sampling. Our solution is to introduce an auxiliary hypothesis test to assert that the probability of the property is 1. The estimation results can then be said to lie within the specified confidence bounds given that the hypothesis is true, while the confidence of the hypothesis test can be made arbitrarily high.

In Section 2 we provide the notational definitions used in the sequel. In Section 3 we review the notions of lightweight verification that underlie our approach. In Section 4 we briefly describe smart sampling [DLST15] and show how it may be profitably extended with importance sampling. In Section 5 we present our algorithm for rewards properties. In Section 7 we present experimental results that demonstrate the effectiveness and limitations of our algorithms. Section 8 concludes the paper.

Related Work

The Kearns algorithm [KMN02] is a well known sampling algorithm for discounted MDPs which approximately optimises rewards over infinite horizons. The rewards used in model checking, however, are typically not discounted and are defined over finite horizons [KNP07].

In [LP12] the authors use an adaptation of the Kearns algorithm to find a memoryless scheduler
that is near optimal with respect to a discounted reward scheme. The resulting scheduler induces a Markov chain whose properties may be verified with standard SMC, but these are not rewards properties of the original MDP.

In [BFHH11, HT13] the authors present algorithms to remove ‘spurious’ nondeterminism on the fly, so that standard SMC may be used. This approach is limited to the class of MDPs whose nondeterminism is not affected by scheduling.

In [HMZ+12] the authors count the occurrence of state-actions in simulations, to iteratively improve a probabilistic scheduler that is assessed using sequential hypothesis testing. If an example that satisfies the hypothesis is found it is correct, but the frequency of state-actions is not in general indicative of global optimality.

In [BCC+14] the authors present learning algorithms to bound the maximum probability of (unbounded) reachability properties. The algorithms refine upper and lower bounds associated to state-actions, according to the contribution of individual simulations. The algorithms converge very slowly and may not converge to the global optimum.

In [LST14] the authors define the behaviour of a scheduler by combining a pseudo-random number generator with an incremental hash function. Schedulers are selected at random and SMC is applied to each of the discrete time Markov chains they induce, to find one that is approximately optimal. This simple sampling approach is made more efficient by ‘smart sampling’ in [DLST15]: an initial candidate set of randomly selected schedulers is progressively refined by discarding those which are sub-optimal and re-assigning their simulation budget to the schedulers that remain. The present work builds on these results, which are reviewed in more detail in Sections 3 and 4.

Importance sampling has been widely applied to quantify failure in ‘highly reliable’ systems (e.g., [Sha94]) and more recently in the specific context of SMC (e.g., [JLS12]). We believe the present work is the first to apply importance sampling to MDPs.

2 Preliminaries

In this work an MDP comprises a possibly infinite set of states $S$, a finite set of actions $A$, a finite set of probabilities $Q$ and a relation $T : S \times A \times S \times Q$, such that $\forall s \in S$ and $\forall a \in A$, $\sum_{s' \in S} T(s,a,s') = r$, where $r \in \{0,1\}$. The execution of an MDP proceeds by a sequence of transitions between states, starting from an initial state, inducing a set of possible traces $\Omega = S^+$. Given an MDP in state $s$, an action $a$ is chosen nondeterministically from the set $\{d' \in A : \sum_{s' \in S} T(s,d',s') = 1\}$. A new state $d \in S$ is then chosen at random with probability $T(s,a,d)$. We assume that rewards are defined by some function $R : S^+ \rightarrow Q$ or $R : A^+ \rightarrow Q$ that maps a sequence of states or a sequence of actions to a total reward. In what follows we abuse the notation and simply write $R(\omega)$ to mean the total reward assigned to trace $\omega \in \Omega$ according to an arbitrary reward scheme.

Our algorithms find deterministic schedulers that approximately maximise or minimise expected rewards and probabilities for an MDP. A history-dependent scheduler is a function $\mathcal{S} : \Omega \rightarrow A$. A memoryless scheduler is a function $\mathcal{M} : S \rightarrow A$. Intuitively, at each state in the course of an execution, a history-dependent scheduler chooses an action based on the sequence of previous states and a memoryless scheduler chooses an action based only on the current state.
History-dependent schedulers therefore include memoryless schedulers. In the context of SMC we consider finite simulation traces of bounded length, hence $\mathcal{E}$ and $\mathcal{M}$ are finite.

A scheduler applied to an MDP induces a Markov chain over which there is a probabilistic measure $F : \Omega \to \mathbb{R}$ such that $\int_{\Omega} dF = 1$. Given a set of paths $\omega \in \Omega$ that satisfy some bounded linear temporal logic property $\varphi$, denoted $\omega \models \varphi$, the probability of $\varphi$ is given by $p = \int_{\Omega} 1(\omega \models \varphi) dF$, where $1 : \{true, false\} \to \{0, 1\}$ is an indicator function that returns 1 if its argument is true and 0 otherwise. To estimate $p$, SMC algorithms typically construct an automaton to decide the truth of the statement $\omega_i \models \varphi$, i.e., whether concrete simulation trace $\omega_i$ satisfies property $\varphi$. The estimated probability of $\varphi$ is then given by

$$\hat{p} = \frac{1}{N} \sum_{i=1}^{N} 1(\omega_i \models \varphi) \quad \omega_i \sim F,$$

where $\omega_1, \ldots, \omega_N$ are $N$ statistically independent random simulation traces distributed according to $F$, denoted $\omega_i \sim F$, and $\hat{p}$ denotes the estimate of $p$. To bound the estimation error it is common to use the “Chernoff” bound of [Oka58]. The user specifies an absolute error $\varepsilon$ and a probability $\delta$ to define the bound $P(|\hat{p} - p| \geq \varepsilon) \leq \delta$. The bound is guaranteed if $N$ satisfies the relation

$$N \geq \left[ \frac{(\ln 2 - \ln \delta)}{(2\varepsilon^2)} \right].$$

3 Lightweight Verification of Markov Decision Processes

We recall here the lightweight verification techniques of [LST14] that underlie our approach.

Schedulers as Pseudo-Random Number Generators To avoid storing schedulers as explicit mappings, we construct schedulers on the fly using uniform pseudo-random number generators (PRNG) that are initialised by a seed and iterated to generate the next pseudo-random value. Our technique uses two independent PRNGs that respectively resolve probabilistic and nondeterministic choices. The first is used in the conventional way to make pseudo-random choices during a simulation experiment. The second PRNG is used to choose actions such that the choices are consistent between different simulations in the same experiment. Given multiple simulation experiments, the further role of the second PRNG is to range uniformly over all possible sets of choices. The seed of the second PRNG can be seen as the identifier of a specific scheduler.

To estimate the probability of a property under a scheduler, we generate multiple probabilistic simulation traces by fixing the seed of the PRNG for nondeterministic choices while choosing random seeds for the PRNG for probabilistic choices. To ensure that we sample from history-dependent schedulers, we construct a per-step PRNG seed that is a hash of a large integer representing the sequence of states up to the present and a specific scheduler identifier [LST14].

A Hash Code to Identify a Trace We assume that the state of an MDP is an assignment of values to a vector of $n$ system variables $v_i, i \in \{1, \ldots, n\}$, with each $v_i$ represented by a number of bits $b_i$. The state can thus be represented by the concatenation of the bits of the system variables, while a sequence of states (a trace) may be represented by the concatenation of the bits of all
the states. We interpret such a sequence of states as an integer of $\sum_{i=1}^{n} b_i$ bits, denoted $\bar{s}$, and refer to this as the trace vector. A scheduler is identified by an integer $\sigma$ of $b_\sigma$ bits, which is concatenated to $\bar{s}$ (denoted $\sigma : \bar{s}$) to uniquely identify a trace and a scheduler. Our approach is to generate a hash code $h = H(\sigma : \bar{s})$ and to use $h$ as the seed of a PRNG that resolves the next nondeterministic choice. In this way we can approximate the scheduler functions $\mathcal{S}$ and $\mathcal{M}$: $H$ maps $\sigma : \bar{s}$ to a seed that is deterministically dependent on the trace and the scheduler; the PRNG maps the seed to a value that is uniformly distributed but also deterministically dependent on the trace and the scheduler. Algorithm 1 implements these ideas as a simulation function that returns a trace, given a scheduler and bounded temporal property as input. The uniformity of scheduler selection is demonstrated by the accuracy of the estimates labelled ‘uniform prob’ in Figs. 1–3.

### Algorithm 1: Simulate

**Input:**

- $\mathcal{M}$: an MDP with initial state $s_0$
- $\varphi$: a bounded temporal logic property
- $\sigma$: an integer identifying a scheduler

**Output:**

- $\omega$: a simulation trace

1. Let $U_{\text{prob}}, U_{\text{nondet}}$ be uniform PRNGs with respective samples $r_{pr}, r_{nd}$
2. Let $H$ be a hash function
3. Let $s$ denote a state, initialised $s \leftarrow s_0$
4. Let $\omega$ denote a trace, initialised $\omega \leftarrow s$
5. Let $\bar{s}$ be the trace vector, initially empty
6. Select seed of $U_{\text{prob}}$ randomly
7. **while** $\omega \models \varphi$ is not decided **do**
8. \[ \bar{s} \leftarrow \bar{s} : s \]
9. Set seed of $U_{\text{nondet}}$ to $H(\sigma : \bar{s})$
10. Iterate $U_{\text{nondet}}$ to generate $r_{nd}$ and use to resolve nondeterministic choice
11. Iterate $U_{\text{prob}}$ to generate $r_{pr}$ and use to resolve probabilistic choice
12. Set $s$ to the next state
13. $\omega \leftarrow \omega : s$

### An Efficient Incremental Hash Function

To implement our approach we use an efficient hash function that constructs seeds incrementally using standard precision mathematical operations. The function is based on modular division, such that $h = (\sigma : \bar{s}) \mod m$, where $m$ is a large prime not close to a power of 2 [CLRS09, Ch. 11]. Since $\bar{s}$ is typically very large, we use Horner’s method [CLRS09, Ch. 30] to generate $h$: we set $h_0 = \sigma$ and find $h \equiv h_n (n \text{ as above})$ by iterating the recurrence relation

$$h_i = (h_{i-1}2^{b_i} + v_i) \mod m. \quad (3)$$

Equation (3) allows us to generate a hash code knowing only the current state and the hash code from the previous step. When considering memoryless schedulers we need only know
the current state. Using suitable congruences \cite{LST14}, the following equation allows (3) to be implemented using efficient native arithmetic:

\[(h_{i-1}2^j) \mod m = (h_{i-1}2^{j-1}) \mod m + (h_{i-1}2^{j-1}) \mod m\]

In a typical implementation on current hardware, a hash function and PRNG may span around $10^{19}$ schedulers. This is usually many orders of magnitude more than the number of schedulers sampled. There is no advantage in sampling from a larger set of schedulers until the number of samples drawn approaches the size of the sample space.

**Confidence with Multiple Estimates** To avoid the cumulative error when choosing a single probability estimate from a number of alternatives, \cite{LST14} defines the following Chernoff bound for multiple estimates:

\[N \geq \left\lceil \left( \ln 2 - \ln \left( 1 - \sqrt{1 - \delta} \right) \right) / (2\varepsilon^2) \right\rceil.\] (4)

Given $M$ estimates \{$\hat{p}_1, \ldots, \hat{p}_M$\} of corresponding true probabilities \{$p_1, \ldots, p_M$\} each generated with $N$ samples, (4) asserts that for any estimate $\hat{p}_i$, in particular the minimum or maximum, $P(\left| \hat{p}_i - p_i \right| \geq \varepsilon) \leq \delta$. Note that when $M = 1$, (4) degenerates to (2).

### 4 Smart Sampling with Importance Sampling

Smart sampling \cite{DLST15} builds on the foregoing techniques to maximise the probability of finding an optimal scheduler with a finite simulation budget. It works by iteratively eliminating sub-optimal schedulers from a candidate set and re-allocating their budget to those that remain.

The problem of finding optimal schedulers by sampling has two independent components: the probability of near-optimal schedulers (denoted $p_g$) and the average probability of the property under near-optimal schedulers (denoted $p\overline{g}$). A near-optimal scheduler is one whose reward or probability (depending on the context) is within some $\varepsilon$ of the optimal value. If we select $M$ schedulers at random and verify each with $N$ simulations, the expected number of traces that satisfy the property using a near-optimal scheduler is thus $M p_g N p\overline{g}$. The probability of seeing a trace that satisfies the property using a near-optimal scheduler is the cumulative probability

\[\left( 1 - (1 - p_g)^M \right) \left( 1 - (1 - p\overline{g})^N \right)\] (5)

To maximise the chance of finding a good scheduler with a simulation budget of $N_{\text{max}} = NM$, $N$ and $M$ should be chosen to maximise (5). Then, following a sampling experiment using these values, any scheduler that produces at least one trace that satisfies the property becomes a candidate for further investigation. Since the values of $p_g$ and $p\overline{g}$ are usually unknown a priori, it is necessary to perform an initial uninformed sampling experiment to estimate them, setting $N = M = \lceil \sqrt{N_{\text{max}}} \rceil$. The results can be used to numerically optimise (5), however an effective heuristic is to set $N = \lceil 1 / \hat{p}_\overline{g} \rceil$, where $\hat{p}_\overline{g}$ is the maximum observed estimate (or minimum non-zero estimate in the case of finding minimising schedulers).
The best scheduler is found by iteratively refining the candidate set. At each iteration the per-iteration simulation budget \( N_{\text{max}} \) is divided between the remaining candidates, simulations are performed and the average probability or reward for each scheduler is estimated. Schedulers whose estimates fall into the “worst” quantile (lower or upper half, depending on context) are discarded. Refinement continues until estimates are known with specified confidence, according to (4). With a per-iteration budget satisfying (2), the algorithm is guaranteed to terminate with a valid estimate.

Smart sampling offers a potentially exponential improvement in performance over the simple sampling strategy described in Sect. 3, but if the property is a rare event, i.e., \( p_\varphi \) is small, finding a good scheduler may remain challenging even if schedulers are not rare. To address this we enhance the basic algorithm with the variance reduction technique of importance sampling, to make \( p_\varphi \approx 1 \) and thus maximise the number of schedulers considered.

Importance sampling is a standard variance reduction technique that works by weighting the executable model of a probabilistic system to make a rare event occur more frequently in simulations. The proportion of simulations in which the event occurs using the weighted model overestimates the true probability, but the estimate may be exactly compensated by the weights. This arises from the equality \( p = \int_G 1(\omega \models \varphi) dF = \int_G 1(\omega \models \varphi) \frac{dF}{dG} dG \), where \( F \) and \( G \) are measures over \( \Omega \) such that \( 1(\omega \models \varphi) \frac{dF}{dG} > 0, \forall \omega \models \varphi \). \( F \) is the original (unweighted) measure, \( G \) is the importance sampling (weighted) measure and \( \frac{dF}{dG} \) is the so-called likelihood ratio. From this we construct the importance sampling counterpart of (1):

\[
\hat{p} = \frac{1}{N} \sum_{i=1}^{N} 1(\omega_i \models \varphi) \frac{dF(\omega_i)}{dG(\omega_i)} \quad \omega_i \sim G. \tag{6}
\]

\( N \) simulation traces \( \omega_i \) are generated under measure \( G \) and their contribution is compensated by the likelihood ratio, which is calculated on the fly. If \( G \) is constructed to strongly favour \( \varphi \), a high percentage of traces will satisfy \( \varphi \) and (6) will converge much faster than (1).

We use this phenomenon to significantly improve the performance of smart sampling when \( p_\varphi \ll 1 \). If \( p_\varphi \) denotes the fraction of traces we expect to satisfy \( \varphi \) using near-optimal schedulers under \( F \) and \( p_\varphi' \) is the fraction of traces we expect to satisfy \( \varphi \) using near-optimal schedulers under \( G \), then importance sampling will increase the expectation of seeing a near-optimal scheduler by \( p_\varphi'/p_\varphi \). Typically, \( p_\varphi' \approx 1 \), so the improvement is \( \approx 1/p_\varphi \).

The smart sampling algorithm with importance sampling follows the procedure described above, but substitutes \( p_\varphi', \hat{p}_\varphi \) for \( p_\varphi, \hat{p}_\varphi \) to optimise (5) and generate the initial candidate set of schedulers. Thereafter, the algorithm uses the corrected estimates, i.e., according to (6), to order and refine the candidate set.

Given \( F \) generated by an optimal scheduler, the theoretically optimal importance sampling distribution \( G^* \) is defined by \( dG^* = 1(\omega \models \varphi) dF/p \), where \( p \) is the probability of \( \varphi \). To maintain the advantages of sampling, \( G \) is typically generated by a syntactic re-parametrisation of \( F \) and constructed by efficient means that do not iterate over the entire state space, such as “failure biasing” [Sha94] and cross-entropy minimisation [JLS12]. These techniques generally find \( G \) that only approximate \( G^* \), but are nevertheless very effective. We make use of failure biasing in one of our case studies in Section 7 and leave cross-entropy minimisation for future work.

AVoCS 2015
A well-known limitation of importance sampling is that the standard statistical confidence bounds do not apply. In our application, this is less important because the principal goal is to make near-optimal schedulers more visible. Once such schedulers have been found, we are at liberty to use other methods to estimate the probability of the property, such as standard Monte Carlo. The values estimated by importance sampling may nevertheless provide useful non-zero bounds when standard Monte Carlo returns a zero estimate because the probability is too low.

5 Statistical Model Checking of MDPs with Rewards

In the classic context, rewards are assigned to actions [Put94]. In the context of formal verification, rewards are often assigned to states or transitions between states [KNP07]. In both cases the rewards are summed over the length of a trace and the expected reward is calculated by averaging the total reward with respect to the probability of the trace. In this work we focus on MDPs in the context of formal verification, but the mechanism of accumulating rewards is unimportant to our algorithms and we simply assume that a total reward is assigned to a finite trace.

The notions of probability estimation used in standard SMC can be adapted to estimate the expected reward of a trace. Given a function \( R(\omega) \in [a, b] \), \( a, b \) finite, that assigns a total reward to simulation trace \( \omega \), the expected reward may be estimated by \( \frac{1}{N} \sum_{i=1}^{N} R(\omega_i) \), where \( \omega_1, \ldots, \omega_N \) are statistically independent simulation traces. Since rewards may take values outside \([0, 1]\), we must use Hoeffding’s generalisation of (2) to bound the errors [Hoe63]. To guarantee \( P(|\hat{r} - r| \geq \varepsilon) \leq \delta \), where \( r \) and \( \hat{r} \) are respectively the true and estimated values of expected reward, \( N \) is required to satisfy the relation

\[
N \geq \left\lceil \ln(2/\delta) \times (a - b)^2/(2\varepsilon^2) \right\rceil.
\]

(7)

For non-trivial problems the values of \( a \) and \( b \) are usually not known, while guaranteed a priori bounds (e.g., by assuming maximum or minimum possible rewards on each step) may be too conservative to be useful. Although it is possible to develop a strategy using a posteriori estimates of \( a \) and \( b \), i.e., based on \( \max_{i \in \{1, \ldots, N\}} (R(\omega_i)) \) and \( \min_{i \in \{1, \ldots, N\}} (R(\omega_i)) \), we see that \( N \) depends on the ratio of the absolute error \( \varepsilon \) to the range of values \( (a - b) \). The confidence of estimates of rewards may therefore be specified a priori as a percentage of the maximum range of the support of \( R \). We adopt this idea in Algorithm 2, where we use (2) and (4) and assume that \( \varepsilon \) expresses a percentage as a fraction of 1.

Rewards Properties

The rewards properties commonly used in numerical model checking are based on an extension of the logic PCTL [KNP07]. This extension defines ‘instantaneous’ rewards (the average reward assigned to the \( k^{th} \) state of all traces, denoted \( I^k \)), ‘cumulative’ rewards (the average total reward accumulated up to the \( k^{th} \) state of all traces, denoted \( C^k \)) and ‘reachability’ rewards (the average accumulated reward of traces that eventually satisfy property \( \varphi \), denoted \( F\varphi \)). Instantaneous and cumulative rewards are based on finite traces and can be immediately approximated by sampling, using (2) and (4) to bound the errors. Reachability rewards are based on unbounded reachability (\( F \)) and require additional consideration.
By the definition of reachability rewards [KPN07], properties that are not satisfied with probability 1 are assigned infinite reward. The rationale behind this is that if $P(F\varphi) < 1$, there must exist an infinite path that does not satisfy $\varphi$, whose rewards will accumulate infinitely. This definition is somewhat arbitrary, since rewards are not constrained to be positive nor to have a minimum value. An infinite sum of positive and negative values can equate to zero and it is also possible for an infinite sum of positive values to converge, as in the case of discounted rewards.

The definition of reachability rewards makes sense in the context of numerical model checking, where paths are not considered explicitly and unbounded properties can be quantified with certainty, but it causes problems for sampling. In particular, using sampling alone it is not possible to say with certainty whether $P(F\varphi) = 1$, even if every observed trace of finitely many satisfies $\varphi$. Without additional guarantees, the random variable from which samples are drawn could include the value infinity, giving it infinite variance. Statistical error bounds, which generally rely on an underlying assumption of finite variance, will therefore not be correct without additional measures.

To accommodate the standard definition of reachability rewards, our solution is to implement $F\varphi$ as $F^k\varphi$, i.e., bounded reachability where $\varphi$ must be true within $k$ steps, with an auxiliary hypothesis test to assert that $P(F\varphi) = 1$ is true. A positive result is thus an estimate within user-specified confidence plus an accepted hypothesis within other user-specified confidence. A negative result is a similar estimate, but with an hypothesis that is not accepted. This approach is consistent with intuition and with the SMC ethos to provide results within statistical confidence bounds. The hypothesis test may be implemented in any number of standard ways. Our implementation uses a convenient normal approximation model, which we describe in Section 6.

In practice, the bound $k$ for reachability rewards is set much longer than it is supposed necessary to satisfy $\varphi$ and the hypothesis is of the form $P(F^k\varphi) \geq p_0$, $p_0 \approx 1$. Intuitively, the more traces of length $\leq k$ that satisfy $\varphi$, the more confident we are that $P(F\varphi) \geq p_0$ is true. Traces that fail to satisfy $\varphi$ after $k$ steps may nevertheless satisfy $\varphi$ if allowed to continue, hence the value of $p_0$ defines how certain we wish to be after $k$ steps. If the hypothesis is rejected, we may either conclude that the average reward is infinite (by definition), accept the calculated average reward as a lower bound or increase $k$ and try again.

Our SMC engine quits as soon as a property is satisfied or falsified, so there is very little penalty in setting $k$ large when we require high confidence, i.e., when $p_0 \approx 1$. Simulations that satisfy the property will only take as many steps as necessary, independent of a much larger value of $k$, while those that do not satisfy the property will be few because the auxiliary hypothesis is falsified quickly when $p_0$ is close to 1.

For the standard rewards properties described above, the value of $p_\varphi$ in (5) is effectively 1. In the case of instantaneous and cumulative rewards, traces are not filtered with respect to a property, so the probability of acceptance is 1. In the case of reachability rewards, either nearly all traces satisfy the property (‘nearly’ because the auxiliary hypothesis test allows for the case that not all traces satisfy the property) or the reward is assumed to be infinite. Hence, the case of probabilities significantly less than 1 does not have to be quantified, just detected. The consequence of this is that there is no need for an undirected simulation experiment to estimate $p_\varphi$ and the initial candidate set will contain the maximum number of schedulers for the specified budget, i.e., $N = 1$ and $M = N_{max}$.
6 Smart Reward Estimation Algorithm

Algorithm 2 builds on Algorithm 4 in [DLST15] to find schedulers that maximise rewards. The algorithm to minimise rewards follows intuitively: replace instances of ‘max’ with ‘min’ in lines 16, 17, 21 and the Output line, and replace line 20 with \( S \leftarrow \{ \sigma \in S \mid \sigma = Q(n) \land n \in \{1, \ldots, |S|/2\} \} \).

The reward property \( \rho \) may be of type instantaneous, cumulative or reachability, which are denoted \( \text{I}^k \varphi \), \( \text{C}^k \varphi \) and \( \text{F}^k \varphi \), respectively, to unify the description. The reward function \( \mathcal{R}_\rho : \mathbb{N} \times \Omega \to \mathbb{Q} \) maps the identifier of a scheduler and a trace to a reward, given reward property \( \rho \).

In the case of \( \text{I}^k \varphi \) and \( \text{C}^k \varphi \), \( k \) is the standard user-specified parameter for these rewards and \( \varphi \) is implicitly \( \text{G}^k \text{true} \) (i.e., \( \text{true} \) in the initial state and for \( k \) steps). In the case of \( \text{F}^k \varphi \), \( \varphi \) is user-specified and \( k \) is set as large as feasible to satisfy the hypothesis \( P(\text{F}^k \varphi) \geq p_0 \), with confidence defined by \( \alpha \) (described below). Given that our actual requirement is that \( P(\text{F} \varphi) = 1 \), both \( p_0 \) and \( \alpha \) will typically be close to 1, such that very few traces will be necessary to falsify the hypothesis.

The initial candidate set of schedulers and corresponding estimates are generated in lines 1 to 4. Applying (5), 1 simulation is performed using each of \( N_{\text{max}} \) schedulers chosen at random. The function \( Q \) maps schedulers to their current estimate. A number of initialisations take place in lines 5 to 6.

The function \( \text{trues} \) is used by the auxiliary hypothesis test and counts the total number of traces per scheduler that satisfy the property. The variable \( \text{samples} \) is also used by the auxiliary hypothesis test and counts the total number of traces used per scheduler. The value of \( \text{conf} \), initialised to 1 to ensure at least one iteration, is the probability that the estimates exceed their specified bounds (defined by \( \varepsilon \)), given the current number of simulations. The main loop (lines 6 to 20) terminates when \( \text{conf} \) is less than or equal to the specified probability \( \delta \). Typically, the per-iteration budget will be such that the required confidence is reached according to (4) before the candidate set is reduced to a single element. Lines 10 to 14 contain the main simulation loop, which quits as soon as the required confidence is reached. Lines 15 to 20 order the results by estimated reward and select the upper quantile of schedulers.

The auxiliary hypothesis test necessary for reachability rewards is provided in lines 21 to 23. To test \( P(\text{F}^k \varphi) \geq p_0 \), it considers the error statistic \( Z = \text{samples} \times (\hat{p}_\varphi - p_0) / \sqrt{\text{samples} \times p_0(1 - p_0)} \), where \( \hat{p}_\varphi = \text{trues}(\sigma_{\text{max}}) / \text{samples} \) is the estimate of \( P(\text{F}^k \varphi) \). For typical values of \( \text{samples} \), the distribution of \( Z \) is well approximated by a normal with mean 0 and variance 1 when the expectation of \( \hat{p}_\varphi \), denoted \( E(\hat{p}_\varphi) \), is equal to \( p_0 \). To test the hypothesis with confidence \( \alpha \), the algorithm compares the statistic \( Z \) with the standard normal quantile of order \( \alpha \), denoted \( z(\alpha) \).

The value of \( z(\alpha) \) may be drawn from a table or approximated numerically. If \( E(\hat{p}_\varphi) \geq p_0 \), the value of \( Z \) will be \( \geq z(\alpha) \) with probability \( \geq \alpha \).

To simplify the presentation, the auxiliary hypothesis test is also used by the instantaneous and cumulative rewards. In these latter cases, however, the test is guaranteed to always be satisfied.

7 Case Studies

The following results demonstrate typical performance on a selection of standard case studies. We necessarily use models whose expected rewards can be calculated or inferred using numerical...
Algorithm 2: Reward Estimation

Input:
\( \mathcal{M} \): an MDP
\( \rho \in \{I_k\varphi, C_k\varphi, F_k\varphi\} \): a reward property
\( \mathcal{R}_\rho \): the reward function for \( \rho \)
\( H_0 : P(F_k\varphi) \geq p_0 \): the auxiliary hypothesis
\( z(\alpha) \): confidence of \( H_0 \), the normal quantile of order \( \alpha \)
\( \varepsilon, \delta \): the reward estimation Chernoff bound
\( N_{\text{max}} > \ln(2/\delta)/(2\varepsilon^2) \): the per-iteration budget

Output:
\( \hat{r}_{\text{max}} \approx r_{\text{max}} \), where \( r_{\text{max}} \approx r_{\text{max}} \) and \( P(|r_{\text{max}} - \hat{r}_{\text{max}}| \geq \varepsilon) \leq \delta \)

\( N \leftarrow 1, M \leftarrow N_{\text{max}} \)
\( S \leftarrow \{M \} \) seeds chosen uniformly at random

\( \forall \sigma \in S, \forall j \in \{1, \ldots, N\} : \omega^j_\sigma \leftarrow \text{Simulate}(\mathcal{M}, \varphi, \sigma) \)

\( Q \leftarrow \{(\sigma, q) \mid \sigma \in S \land Q \ni q = \sum_{j=1}^{N} \mathcal{R}_\rho(\sigma, \omega^j_\sigma)/N\} \)

\( \forall \sigma \in S : \text{trues}(\sigma) \leftarrow 0 \)

\( \text{samples} \leftarrow 0, \text{conf} \leftarrow 1, i \leftarrow 0 \)

\textbf{while} conf > \delta \land S \neq \emptyset \textbf{do}

\( i \leftarrow i + 1 \)

\( M_i \leftarrow |S|, N_i \leftarrow 0 \)

\textbf{while} conf > \delta \land N_i < \lfloor N_{\text{max}}/M_i \rfloor \textbf{do}

\( N_i \leftarrow N_i + 1 \)

\( \text{conf} \leftarrow 1 - (1 - e^{-2\varepsilon^2 N_i})^{M_i} \)

\( \forall \sigma \in S : \omega^{N_i}_\sigma \leftarrow \text{Simulate}(\mathcal{M}, \varphi, \sigma) \)

\( \text{samples} \leftarrow \text{samples} + 1 \)

\( Q \leftarrow \{(\sigma, q) \mid \sigma \in S \land Q \ni q = \sum_{j=1}^{N_i} \mathcal{R}_\rho(\sigma, \omega^j_\sigma)/N_i\} \)

\( \sigma_{\text{max}} \leftarrow \arg\max_{\sigma \in S} Q(\sigma) \)

\( \hat{r}_{\text{max}} \leftarrow Q(\sigma_{\text{max}}) \)

\( \forall \sigma \in S, j \in \{1, \ldots, N_i\} : \text{trues}(\sigma) = \text{trues}(\sigma) + 1(\omega^j_\sigma \models \varphi) \)

\( Q' : \{1, \ldots, |S|\} \rightarrow S \) is an injective function s.t.

\( \forall (n, \sigma), (n', \sigma') \in Q' : n > n' \implies Q(\sigma) \geq Q(\sigma') \)

\( S \leftarrow \{\sigma \in S \mid \sigma = Q'(n) \land n \in \{|S|/2, \ldots, |S|\}\} \)

\( Z \leftarrow (\text{trues}(\sigma_{\text{max}}) - \text{samples} \times p_0)/\sqrt{\text{samples} \times p_0(1 - p_0)} \)

\textbf{if} Z \leq z(\alpha) \textbf{then}

\( H_0 \) is rejected
Network Virus Infection  This case study is based on [KNPV09] and comprises a network of linked sets of nodes. Initially, there is a set containing one node infected by a virus, a set with no infected nodes and a set of uninfected barrier nodes that divides the first two sets. A virus chooses which node to infect nondeterministically and infects it with probability 0.5. A node detects a virus probabilistically. Figure 1 plots the results of using a rewards property to estimate the expected number of attacks before a particular node is infected, varying the probability that a barrier node detects a virus as a parameter. Each point required approximately 15 seconds.
of simulation time. All estimates are within ±1% of the true values. Figure 2 plots the results of using importance sampling to estimate the probability of infection when the probability of a virus infecting a node is reduced to 0.01. Our importance sampling distribution is generated by the simple expedient of setting this parameter to its original value of 0.5. The plotted estimates are the average of 10 individual estimates generated using a per-iteration budget of just $10^4$ simulations. In most cases this budget is orders of magnitude less than the expected number of simulations necessary to see a single trace that satisfies the property.

**Choice Coordination** To demonstrate the scalability of our approach we consider instances of the choice coordination model of [NM10] with $BOUND = 100$. This value makes most of the models intractable to numerical model checking, however it is possible to infer the correct values of rewards from tractable instances. The chosen reachability property gives the expected minimum number of rounds necessary for a group of tourists to meet. The following table summarises the results:

<table>
<thead>
<tr>
<th>Number of tourists</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Estimated minimum rounds to converge</td>
<td>4.0</td>
<td>5.0</td>
<td>7.0</td>
<td>8.0</td>
<td>10.0</td>
<td>11.0</td>
<td>12.0</td>
<td>13.0</td>
<td>14.0</td>
</tr>
</tbody>
</table>

All the estimates are exactly correct, while the average time to generate each result was just 8 seconds.

**Gossip Protocol** The gossip protocol of [KNP08] uses local connectivity to propagate information globally. Using a reachability reward property, our algorithms accurately estimate the expected minimum and maximum number of rounds necessary for the network to become connected as 1.486 and 4.5, compared to correct values 1.5 and 4.5. The average simulation time per estimate was approximately 1 minute.

Figure 3 plots the maximum and minimum estimated path length at different time steps, using an instantaneous reward property. We see that the estimates are accurate up to about 75 time steps, but less so above this value. Since $p_5$ in (5) is effectively 1, this problem cannot be overcome by importance sampling. The estimates are nevertheless guaranteed by (4) not to exceed the true values by more than a factor of $1 + \varepsilon$ with probability $\delta$.

### 8 Prospects and Challenges

In this work we have focused on estimating the expected values of rewards and rare events. We believe the same techniques may be immediately extended to sequential hypothesis testing, as.
in [LST14] and [DLST15]. Ongoing work suggests that estimating rewards in continuous time models will also be feasible.

With respect to rewards, our case studies demonstrate that our approach is effective and can be efficient with state space that is intractable to numerical methods. We do not yet provide numerical confidence with respect to optimality, but our techniques generate useful conservative bounds with correct statistical guarantees of accuracy: the estimate will be greater (less) than the true maximum (minimum) expected reward by a factor of $\geq 1 + \varepsilon$ with probability $\leq \delta$.

The use of importance sampling can improve the performance of smart sampling by a factor of $1/p_{\bar{\pi}}$. We have demonstrated this technique using an importance sampling distribution generated by simple failure biasing, but as future work we propose to develop a cross-entropy minimisation algorithm to find an optimal parametrised distribution, along the lines of [JLS12].

In the case of standard rewards properties, $p_{\bar{\pi}} = 1$ and importance sampling cannot help. Figure 3 illustrates circumstances where our chosen per-iteration budget of $10^5$ is apparently not sufficient to explore the tails of the distribution of schedulers. Merely increasing the budget will not in general be adequate, since near-optimal schedulers may be arbitrarily rare. Our proposed future solution is to combine sampling with learning to construct composite schedulers.

Acknowledgement

This work was partially supported by the European Union Seventh Framework Programme under grant agreement no. 295261 (MEALS).

Bibliography


Conditional Lemma Discovery and Recursion Induction in Hipster

Irene Lobo Valbuena and Moa Johansson

lobo@student.chalmers.se, moa.johansson@chalmers.se
Department of Computer Science and Engineering
Chalmers University of Technology, Gothenburg, Sweden.

Abstract: Hipster is a theory exploration tool for the proof assistant Isabelle/HOL. It automatically discovers lemmas about given recursive functions and datatypes and proves them by induction. Previously, only equational properties could be discovered. Conditional lemmas, for example required when reasoning about sorting, has been beyond the scope of theory exploration. In this paper we describe an extension to Hipster to also support discovery and proof of conditional lemmas.

We also present a new automated tactic, which uses recursion induction. Recursion induction follows the recursive structure of a function definition through its termination order, as opposed to structural induction, which follows that of the datatype. We find that the addition of recursion induction increases the number of proofs completed automatically, both for conditional and equational statements.

Keywords: theory exploration, automated induction, interactive theorem proving

1 Introduction

Theory exploration is a technique for automatically discovering new interesting lemmas in a formal mathematical theory development. These lemmas are intended to help constructing a richer background theory about the concepts at hand (e.g. functions and datatypes) which can be useful both to enhance the power of automation as well as being of use in interactive proofs [MBA07, JDB11, MMDB12]. Theory exploration has proved particularly useful for automation of inductive proofs [CJRS13]. This work builds on Hipster [JRSC14], an interactive theory exploration system for the proof assistant Isabelle/HOL [NPW02]. It can be used in two modes, either exploratory mode to generate a set of basic lemmas about given datatypes and functions, or in proof mode, where it assists the user by searching for missing lemmas needed to prove the current subgoal. To generate conjectures, Hipster uses as a backend the HipSpec system, a theory explorer for Haskell programs [CJRS13]. Proofs are then performed by specialised tactics in Isabelle/HOL. Hipster has been shown capable of discovering and proving standard lemmas about recursive functions, thus speeding up theory development in Isabelle. However, lemma discovery by theory exploration has previously been restricted to equational properties. In this paper we take the first steps towards lifting this restriction and exploring also conditional conjectures. Conditional lemmas are necessary if we for example want to prove properties about sorting algorithms. As an example, consider the proof of correctness for insertion sort:

```
theorem isortSorts: "sorted (isort xs)"
```
To prove this theorem by induction will in the step-case require a lemma telling us that if a list is sorted, it remains so after an additional element is inserted:

\[
\text{lemma "sorted } xs \implies \text{sorted (insert x xs)"}
\]

Discovering this kind of conditional lemmas introduces a big challenge for theory exploration. First of all, the search space greatly increases: what statements should be picked as potentially interesting side-conditions to explore? Secondly, as our theory exploration system relies on generation of random test-cases, we also need to ensure that we perform tests where the condition evaluates to true, otherwise the system might miss some conditional equations (Example 2, p. 4).

As Hipster is designed as an interactive system, we avoid the first problem by asking the user to specify under which condition theory exploration should occur. In the example above, this would require the user to tell Hipster that the predicate \text{sorted} is an interesting pre-condition, in addition to which function symbols should be explored in the bodies of lemmas. The rest of the process is however automatic. We describe it in more detail in §3.

The second contribution of this paper is a new automated tactic for recursion induction (see e.g. §3.5.4 of [NPW02]). Previously, Hipster only supported structural induction over the datatypes, but has now been extended with a new tactic that uses recursion induction, following the termination order of function definitions instead of the datatype. This has shown to be useful for many proofs that previously failed, but can also provide shorter proofs in some cases. The new recursion induction tactic is described in §3.2. It is used by Hipster during automated theory exploration, but can equally well be applied as a powerful regular tactic by a human user working in Isabelle.

2 Hipster

This section provides a description of how Hipster works and how its subsystem QuickSpec generates conjectures.

2.1 Theory Exploration in Hipster

Figure 1 gives an overview of the Hipster system. Starting from an Isabelle theory file that defines a set of datatypes and functions, the user calls Hipster on a list of functions about which she is interested in finding lemmas. The workings of Hipster can be divided up into three stages:

1. Generation of Haskell code.
2. Theory exploration in Haskell.
3. Proof in Isabelle.

Hipster uses Isabelle’s code generator [HN10], to translate the theory to a Haskell program. Hipster then employs the theory exploration system HipSpec as a backend for generating conjectures. While HipSpec can be used also as a fully fledged theorem prover, Hipster only uses its conjecture generation subsystem QuickSpec [CSH10], and performs proofs inside Isabelle. Isabelle is an LCF-style prover, which means that it is based on a small core of trusted axioms,
upon which subsequent proofs must be built. Therefore, any proofs found outside Isabelle, e.g. by HipSpec, would have to be reconstructed inside Isabelle anyway. Hence it is easier for Hipster to simply use Isabelle for proofs in the first place.

Not all conjectures returned from QuickSpec are interesting. Hipster is parametrised by two tactics, which can be set by the user: one for \textit{routine reasoning} and one for \textit{difficult reasoning}. Conjectures solved by routine reasoning are deemed trivial and discarded, while those requiring more difficult reasoning are displayed to the user and included in the Isabelle theory so they can be used in subsequent proofs if necessary. In the context of this paper, routine reasoning is first-order equational reasoning and simplification, whilst difficult reasoning involves some kind of induction. If a conjecture is not immediately provable, Hipster will place it at the end of the list of open conjectures and will try it again if it has found some additional lemmas. Occasionally, Hipster might discover some conjecture which it does not manage to prove automatically, because not even its tactic for difficult reasoning is strong enough. Such an open conjecture would also be displayed to the user, who can then choose to perform an interactive proof in Isabelle, perhaps employing other tactics or lemmas than those currently available to Hipster.

2.2 Conjecture Generation in QuickSpec

QuickSpec takes as input a set of functions and variables (by default three per type), and generates all type-correct terms up to a given limit (by default depth three). The number of variables and term-depth limit can be adjusted by the user. QuickSpec then proceeds to divide the generated terms into equivalence classes, so that each equivalence class eventually represents a set of equations. Initially, all terms of the same type are in the same equivalence class. QuickSpec then uses QuickCheck \cite{CH00}, to generate random ground values for the variables in the terms, and evaluates the result. If two terms in an equivalence class turn out to evaluate differently, the equivalence class is split accordingly. The process is then repeated until the equivalence classes stabilise (after several hundred different random tests), which means that we usually have quite a
high confidence in that the conjectures produced are probably true, even though they are not yet proved.

**Example 1.** As a small example, consider a theory exploration attempt where we have asked Hipster for lemmas about a function `isort` implementing insertion sort. Among the terms generated by QuickSpec are those in the table below. Initially, all terms are placed in the same equivalence class. Suppose QuickSpec generates the random value `xs \rightarrow [3,1]`.

<table>
<thead>
<tr>
<th>Term</th>
<th>Ground Instance</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>isort xs</code></td>
<td><code>isort [3,1]</code></td>
<td><code>[1,3]</code></td>
</tr>
<tr>
<td><code>isort (isort xs)</code></td>
<td><code>isort (isort [3,1])</code></td>
<td><code>[1,3]</code></td>
</tr>
<tr>
<td><code>xs</code></td>
<td><code>isort [3,1]</code></td>
<td><code>[3,1]</code></td>
</tr>
</tbody>
</table>

As not all terms evaluate to the same value, they should no longer be in the same equivalence class. We thus split the terms into two new equivalence classes: terms 1 and 2 evaluate to the same value and remain together, while term 3 is separated. After this, no subsequent tests further split these equivalence classes, and we can read off the equation: `isort(isort xs) = isort xs`.

### 3 Conditional Lemmas and Recursion Induction

We now demonstrate how to employ Hipster interactively for theory exploration of conditional lemmas in the development of a theory. We first explain how conditional conjectures are generated in QuickSpec. We then explain our new automated induction tactic for recursion induction, and finally show how Hipster combines these in a case study proving the correctness of insertion sort.

#### 3.1 Generating Conditional Conjectures

The support in QuickSpec for generating conditional conjectures (implications) is still rather basic. In this case, QuickSpec will in addition to the other input require the user to specify a predicate to use as the premise of an implication. Term generation proceeds as described above, but testing takes the given predicate into account. Here, we are only interested in tests with values that make the premise true, otherwise we may split the equivalence classes when they should not be split. QuickCheck uses special functions called *generators* to produce random values of a given type. If using QuickSpec directly in Haskell, the user can program special purpose generators that could be made to only produce values satisfying a given predicate. In Hipster, however, these generator functions are simpler as they have to be automatically derived together with the Haskell code. Tests not satisfying the premise are simply discarded during conditional exploration, which means that we typically must generate more tests than for equational conjectures. Also, the risk of some non-theorem slipping through is slightly higher, but as Hipster then attempts to prove all conjectures, such a statement would be caught in the proving phase. Automatically generating customised generator functions is further work.

**Example 2.** In Example 1, we showed how QuickSpec generated equational conjectures about the insertion sort function `isort`. We are furthermore interested in the case with the condition
that the predicate sorted holds (for one variable). In this case, QuickSpec first performs one pass looking for plain equations, as in Example 1, then a second where it considers the condition sorted xs. In this second phase, QuickSpec performs a new exploration, this time requiring the predicate sorted xs to hold for all test values. Suppose we test with the sorted list: xs → [1,2] (other non-sorted values for xs would be discarded).

<table>
<thead>
<tr>
<th>Term</th>
<th>Ground Instance</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 isort xs</td>
<td>isort([1,2])</td>
<td>[1,2]</td>
</tr>
<tr>
<td>2 isort (isort xs)</td>
<td>isort (isort [1,2])</td>
<td>[1,2]</td>
</tr>
<tr>
<td>3 xs</td>
<td>[1,2]</td>
<td>[1,2]</td>
</tr>
</tbody>
</table>

This time, all terms evaluate to the same value on all tests where the list is sorted, so all three terms remain in the same equivalence class. QuickSpec realises that there is no point producing the conjecture sorted xs ⇒ isort (isort xs) = xs, as this is subsumed by the non-conditional equation discovered in the first phase. It will however produce the additional conjecture sorted xs ⇒ isort xs = xs, which clearly only holds if the list is already sorted.

### 3.2 Automating Recursion Induction

A recursion induction scheme is derived from a function’s recursive definition. Unlike structural induction, the recursion induction scheme corresponds to the originating definition, and hence, the cases considered in its simplification rules.

When defining a recursive function over an inductive datatype one might traverse arguments following a different pattern to the strictly structural one (the one arising from a datatype’s definition). This pattern could be more specific, or even less so, than following the datatype.

For instance, take the functions on lists sorted and last:

```haskell
fun sorted :: "Nat List ⇒ Bool" where
  "sorted [] = True"
| "sorted ([x]) = True"
| "sorted (x1 # (x2 # xs)) = (x1 ≤ x2 & sorted (x2 # xs))"

fun last :: "a List ⇒ a" where
  "last ([x]) = x"
| "last (x # xs) = last xs"
```

From these definitions’ structures one can derive a new induction principle. Structural induction on lists considers the base-case [] (the empty list) and step-case x # xs (a list with an element inserted at the front). In the case of sorted, cases are broken down into more detailed ones by including an additional base-case [x] (the singleton list) and restricting the step-case to lists with at least two elements x1 # x2 # xs. Meanwhile last is not defined for the case [] and hence partially defined, meaning the induction scheme it gives rise to is to be completed with such a case. This, in fact, results in the same recursion induction scheme derived from sorted:

\[
\begin{align*}
P ([]) & \quad \forall x \ P ([x]) \\
\forall x, y, xs \ (P (x # xs) \implies P (y # (x # xs)))
\end{align*}
\]

\[\forall x \ P (x)\]
Conditional Lemma Discovery and Recursion Induction in Hipster

\[
\begin{align*}
\forall x & \; P([x]) \\
\forall x, y, xs & \; (P(x \# xs) \implies P(y \# xs)) \quad P([])
\end{align*}
\]

Induction determined by these schemata is called recursion induction or computation induction. They can isolate sub-units not represented in a datatype’s structure as being atomic, such as lists with at least two elements in the scheme for \texttt{sorted}. Recursion induction hence provides an immediate and more specific structure for reasoning about other recursion patterns where a simple structural induction might fail to set appropriate base and step-cases for the induction to succeed.

Within Isabelle/HOL these schemata are automatically derived and proved as theorems from recursive functions’ termination order, and hence guaranteed to be sound [KST+11].

Example 3: Recursion Induction in a Proof. To exemplify the potential difference between recursion and structural induction, let us take the already introduced conditional lemma \texttt{sorted xs \implies sorted (insert x xs)}. Applying structural induction on the list \(xs\) would produce the subgoals:

1. \texttt{sorted [] \implies sorted (insert x [])}
2. \(\land y. \texttt{sorted ys \implies sorted (insert x ys)} \implies \texttt{sorted (y \# ys)}\)

Whilst \texttt{sorted}’s recursion induction scheme would yield:

1. \texttt{sorted [] \implies sorted (insert x [])}
2. \(\land y. \texttt{sorted [y] \implies sorted (insert x [y])}\)
3. \(\land y1 y2. \texttt{sorted (y2 \# ys) \implies sorted (insert x (y2 \# ys))} \implies \texttt{sorted (y1 \# y2 \# ys)}\)

The latter set of subgoals leads to an immediate proof of the main lemma thanks to its steps mirroring the actual predicate definition, hence having a correspondence with its simplification rules. In contrast, the former, even though it intuitively looks immediate to prove, is not sufficiently generalised nor does it specify any intermediate result on inserting an element on a concrete non-empty list (in our case, the singleton list) which would enable to prove the second subgoal for any arbitrary list. Structural induction is in some way a weaker scheme and additional case-splits or lemmas would be required to close the proof employing it in our example.

A New Induction Tactic for Hipster

We have implemented a new automated tactic, called \texttt{hipster_induct_schemes}, for induction in Isabelle. This tactic searches not only for proofs by structural induction, but may also employ recursion induction when appropriate. It is designed for Hipster to use as its “difficult reasoning” component, but human users may of course also employ this tactic in interactive proofs.

The tactic first tries structural induction using the induction scheme associated with the datatype(s) of variables in the problem. If this fails, the tactic then tries recursion induction, using the induction schemes associated with the functions occurring in the problem. When instantiating recursion induction schemes with variables of the problem, more complete instantiations are considered first. This leaves less specific partial instantiations to be tried later. For each attempted
Figure 2: Overview of Hipster's new tactic.

induction, the tactic will apply Isabelle’s simplifier followed by (if necessary) first-order reasoning using Isabelle’s built in first-order prover Metis. Figure 2 shows an overview of the tactic.

The user can configure the tactic to specify how to select facts to be passed to the simplifier and to Metis. The default is the simplification rules from the relevant function definitions, the datatype case distinction rules which are automatically derived by Isabelle, and the lemmas discovered by theory exploration so far. However, if we pass too many facts to Metis, it becomes slower. Therefore, the user can configure Hipster to include fewer of the discovered lemmas if needed. Hipster also impose a timeout on simplification and first-order reasoning, which can be set by the user. The default timeout is 1 second for each proof attempt. As further work, we plan to experiment with using Sledgehammer instead \cite{PB10}, which calls powerful external first-order provers and reports back exactly which facts were needed in the proof. Metis can then reconstruct the proof quickly inside Isabelle’s trusted kernel.

**Example 4: Simultaneous Induction.** A notable gain of the new tactic with recursion induction is that of having the capability of performing simultaneous induction, whereas previously only structural inductions on a single variable were performed by Hipster. Simultaneous induction schemata are those inducting over more than one variable at a time, whether those variables are of the same type or not. Such is the case for the list function zip’s recursion induction scheme, which corresponds to parallel induction on two lists:

fun zip :: "'a list ⇒ 'b list ⇒ ('a × 'b) list" where
  "zip [] y = []"
| "zip (x # xs) [] = []"
| "zip (x # xs) (y # ys) = (x, y) # (zip xs ys)"

\(ZI{\Pi}ND\)
\[
\forall y x \ P ([], y) \quad \forall x, x' \ P (x' \# x, []) \quad \forall x, y, x', y' \ P (x, y) \implies P (x' \# x, y' \# y)
\]
\[
\forall x, y \ P (x, y)
\]

This scheme, along with some initial theory exploration, allows theorems like the following to be proven automatically:

\(\text{zip} \ (xs @ ys) \ zs = (\text{zip} \ xs \ (\text{take} \ (\text{len} \ xs) \ zs)) @ (\text{zip} \ ys \ (\text{drop} \ (\text{len} \ xs) \ zs))\)

Or even the alternative related conditional lemma to be proven without prior exploration:

\(\text{len} \ xs = \text{len} \ ys \implies (\text{zip} \ xs \ ys) @ (\text{zip} \ zs \ ws) = \text{zip} \ (xs @ zs) \ (ys @ ws)\)
Neither of these lemmas were provable before, even having done exploration for all the occurring functions in them. Hipster’s prior structural induction approach could not capture in a scheme the relation between two variables. In these two cases, zip traverses its arguments taking steps on both at the same time, a pattern we can only capture with some form of simultaneous induction. Instead of synthesising a series of possible simultaneous structural induction schemata, recursion induction gives us an immediate choice which is also closer to the problem at hand.

3.3 Interactive Case Study: Insertion Sort

We here showcase Hipster’s handling of conditional lemmas via the proof of correctness for the theorem \( \text{sorted } (\text{isort } ts) \). For it, we assume the less-or-equal operator \( \leq \) for naturals (and no prior, additional lemmas), and the function definitions:

\[
\begin{align*}
\text{fun sorted :: } & \text{"Nat List } \Rightarrow \text{ Bool" where} \\
& \text{"sorted } [\cdot] \text{ = True"} \\
& | \text{"sorted } ([x]) \text{ = True"} \\
& | \text{"sorted } (x1 \# (x2 \# xs)) \text{ = } (x1 \leq x2 & \text{sorted } (x2 \# xs)) \\
\end{align*}
\]

\[
\begin{align*}
\text{fun insert :: } & \text{"Nat } \Rightarrow \text{ Nat List } \Rightarrow \text{ Nat List" where} \\
& \text{"insert } x \text{ } [\cdot] \text{ = } [x]" \\
& | \text{"insert } x1 \text{ } (x2 \# xs) \text{ = } \begin{cases} 
\text{if } (x1 \leq x2) \text{ then } x1 \# (x2 \# xs) \\
\text{else } x2 \# (\text{insert } x1 \text{ } xs)
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
\text{fun isort :: } & \text{"Nat List } \Rightarrow \text{ Nat List" where} \\
& \text{"isort } [\cdot] \text{ = } [\cdot]" \\
& | \text{"isort } (x \# xs) \text{ = } \text{insert } x \text{ } (\text{isort } xs)" \\
\end{align*}
\]

Running exploration from the simpler components is the first step, considering both equational and conditional lemmas, since we have two predicates involved in the definiens of functions in the final theorem. The following command invokes conditional exploration for \( \leq \):

\[
\text{hipster}\text{.cond } \leq
\]

which, along with conditional exploration for its negation, results in 10 discovered and proven lemmas, 6 of which are conditionals (we present the vital lemmas towards the final proof) and all require recursion induction:

\[
\begin{align*}
\\text{lemma lemma_ac [thy_expl]: } & "x \leq y \implies x \leq (S \ y) = \text{ True}" \\
& \text{by } (\text{hipster_induct_schemes } \leq .\text{simps Nat.exhaust}) \\
\end{align*}
\]

\[
\begin{align*}
\\text{lemma lemma_ad [thy_expl]: } & "y \leq x \implies (S \ x) \leq y = \text{ False}" \\
& \text{by } (\text{hipster_induct_schemes } \leq .\text{simps Nat.exhaust}) \\
& (...) \\
\end{align*}
\]

\[
\begin{align*}
\\text{lemma lemma_ai [thy_expl]: } & "(\neg (x \leq y)) \implies x \leq Z = \text{ False}" \\
& \text{by } (\text{hipster_induct_schemes } \leq .\text{simps Nat.exhaust}) \\
& (...) \\
\end{align*}
\]
Hipster automatically generates this output. For each case, the lemma command makes the statement to be proven and is followed by a tactic application via the by command, here using Hipster’s recursion induction tactic `hipster_induct_schemes`, which employs recursion induction where necessary. To enable the completion of the proof, exploration provides it with the automatically generated Isabelle rules for simplification of function definitions, such as `≤.simps`, and datatype case distinction rules, such as `Nat.exhaust`.

With a new exploration considering the functions about sorting itself and (potentially) taking sorted as a side-condition for which to discover lemmas, Hipster discovers and proves both the conditional auxiliary lemma required and the goal theorem. Note that the exploration command takes as its first argument the predicate with which to construct side-conditions:

```isar
hipster_cond sorted isort insert
(...)
lemma isortInvariant [thy_expl]: "sorted ys ⇒ sorted (insert x ys) = True"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)
(...)
theorem isortSorts [thy_expl]: "sorted (isort x) = True"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)
```

During this last exploration, other interesting lemmas are discovered, all of which can be now proven automatically by using the sub-lemma about insert’s invariant `isortInvariant`:

```isar
lemma isortFixes [thy_expl]: "sorted x ⇒ isort x = x"
by (hipster_induct_schemes sorted.simps isort.simps insert.simps)

lemma insertComm [thy_expl]: "insert x (insert y z) = insert y (insert x z)"
by (hipster_induct_schemes insert.simps)
```

Invoking the recursion induction tactic `hipster_induct_schemes` once proves all of the statements above, simplifying the interaction with the proof assistant. Particularly, the crucial lemma `isortInvariant` is proven applying sorted’s associated recursion induction scheme, highlighting once again the need for support of conditional lemmas in automated inductive proving and the possibilities recursion induction brings towards proof automation.

### 4 Evaluation

In this section we present an evaluation of Hipster’s automated tactics, an analysis which had not been performed for Hipster to the same extent priorly.

Keeping in mind evaluation of automated tools for interactive theorem proving necessarily has to consider some degree of interaction, two forms of evaluation have been carried out¹:

- case studies on algebraic data types and operations on them; in particular focusing on inductive theories for natural numbers and lists

¹ Source code for Hipster, examples presented and benchmarks are available online: [https://github.com/moajohansson/IsaHipster](https://github.com/moajohansson/IsaHipster)
- evaluation on problems from TIP (Tons of Inductive Problems) [CJRS15], a set of benchmarks and challenge problems for inductive theorem provers.

From TIP, we evaluate Hipster over two sets of problems employed in previous works on inductive theorem proving: Johansson, Dixon and Bundy’s work on case-analysis for rippling [JDB10] (we denote it case-analysis\(^2\)), and prior work by Ireland and Bundy on employing proof failure to guide lemma discovery and patch inductive proofs [IB96] (we denote it prod-failure\(^3\)). We now present these results and compare them with other tools’ reported results.

### 4.1 Method

To evaluate performance on TIP, each problem is analysed individually, in isolation from others, to assess how far Hipster can go from bare definitions. Theory explorations were only run whenever the problem was not provable by the induction tactic directly, i.e. when the problem was missing helping lemmas. Explorations were first performed on the individual functions appearing in the problem definition, jointly with their auxiliary functions. These were followed by explorations on groups of said functions if required, leaving conditional exploration as the last exploration to be run before defining the problem as non-provable by Hipster.

As already specified, conditional lemma discovery is limited to explore a single predicate at a time to define side-conditions. For the present evaluation this has sufficed.

Additionally, to test Hipster’s capacity when working on strictly newly defined theories, no assumptions nor properties from theories in Isabelle/HOL were considered during proof search. As an example, natural numbers are not Isabelle/HOL’s, but redefined. Hence, predefined notions of orderings and other properties do not play a part in proofs obscuring the results of Hipster’s actual work. In this way, we only consider as the base starting point a set of definitional statements, aligning with the purpose of proving based on structure and construction of programs.

### 4.2 Results

The following set of tables summarises statistics on the two sets of the benchmarks, with respect to the number of problems solved. Columns \(EQ\) and \(COND\) do so for problems defined by an equational and a conditional theorem respectively.

<table>
<thead>
<tr>
<th></th>
<th>Case-analysis</th>
<th>Prod-failure</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(EQ)</td>
<td>(COND)</td>
<td>(EQ)</td>
</tr>
<tr>
<td>Total number of benchmarks</td>
<td>71</td>
<td>14</td>
<td>38</td>
</tr>
<tr>
<td>Number of problems solved</td>
<td>71</td>
<td>13</td>
<td>35</td>
</tr>
</tbody>
</table>

Table 1: Total number of problems solved.

\(^2\) Case-analysis problems: https://github.com/tip-org/benchmarks/tree/master/benchmarks/isaplanner

\(^3\) Prod-failure problems: https://github.com/tip-org/benchmarks/tree/master/benchmarks/prod
### Automation

Table 2 shows the number of problems with automated solutions out of those which were solved. Full automation is understood as solving a problem only with discovered lemmas about the function symbols involved in the target theorem and Hipster’s automated recursion induction. Partially automated problems are those for which additional related functions of a datatype’s theory were provided to exploration for completion.

Overall, the rate of fully automated provability on the benchmark set is 90% ; considering partially automated problems as well, the overall rate is 97%.

A number of theorems (problems 52, 53, 72, 74 from case-analysis; and 2, 4, 5, 20, 22, 23 from prod-failure) required one of the following two similar lemmas:

\[
\begin{align*}
\text{len} \ (x \ @ \ y) &= \text{len} \ (y \ @ \ x) \\
\text{count} \ z \ (x \ @ \ y) &= \text{count} \ z \ (y \ @ \ x)
\end{align*}
\]

These two lemmas are not automatically proven in a first instance (neither by structural nor recursion induction). Each of them in turn needs an auxiliary lemma which is not discovered.

Nonetheless, their proof can be partially automated. In both cases, one can observe that the outermost function applied, \(\text{len}\) and \(\text{count}\) respectively, acts as a relator function between two datatypes. Furthermore, these will in fact act as relator functions between list concatenation \(\@\) and addition for natural numbers \(\text{plus}\). Since \(\text{plus}\) does not occur in the problems to be proven, it is not added to the exploration directly. Adding \(\text{plus}\) interactively, Hipster discovers and proves automatically the lemmas:

\[
\begin{align*}
\text{len} \ (x \ @ \ y) &= \text{plus} \ (\text{len} \ x) \ (\text{len} \ y) \\
\text{count} \ z \ (x \ @ \ y) &= \text{plus} \ (\text{count} \ z \ x) \ (\text{count} \ z \ y)
\end{align*}
\]

Along with the commutative law for \(\text{plus}\), also discovered and proven automatically, they enable the automation of the two pending proofs without further intervention. And so, the corresponding TIP problems are solved as well.

These two cases seem to indicate that recursion induction may not suffice when a non-commutative operation nested within another has commuting arguments on both sides of an equality. At least not in the absence of smaller related lemmas corresponding to subgoals. This seems reasonable: the structure of the terms at each side of the equality will differ upon induction.

### Theory exploration

Just over half of the problems required prior lemma discovery, showcasing the benefit of theory exploration. In Table 3 we show the number of solved problems which required prior theory exploration and specify how many required further conditional lemmas.

A smaller subset of problems were provable with the aid of conditional exploration, namely those involving functions defined in terms of some predicate.
Recursion induction. Whereas recursion induction was not necessary as often as theory exploration (whether for the main theorem or auxiliary lemmas), its impact is still notable. Some problems would not be provable employing only Hipster’s prior structural induction approach. In Table 4, problems solved by structural induction are those for which both the main theorem and any required auxiliary lemma only needed structural induction. Those solved by recursion induction required it for the main theorem’s proof or any of its helping lemmas.

<table>
<thead>
<tr>
<th>Case-analysis</th>
<th>Prod-failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>COND</td>
</tr>
<tr>
<td>No additional lemmas</td>
<td>38</td>
</tr>
<tr>
<td>Only equational lemmas</td>
<td>27</td>
</tr>
<tr>
<td>Equational and conditional lemmas</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 3: Number of problems requiring discovery of auxiliary lemmas.

<table>
<thead>
<tr>
<th>Case-analysis</th>
<th>Prod-failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>EQ</td>
<td>COND</td>
</tr>
<tr>
<td>Structural induction</td>
<td>38</td>
</tr>
<tr>
<td>Recursion induction</td>
<td>33</td>
</tr>
</tbody>
</table>

Table 4: Number of problems solved with both kinds of induction.

Overall, there seems to be a trade-off between using weaker induction schemes (structural induction) and reducing the number and complexity of needed auxiliary lemmas. Structural induction was always attempted first by the tactic, meaning theorems solved via recursion induction (around a third of the benchmarks) would have not been solved otherwise, at least not with the degree of exploration carried out.

The results suggest recursion induction can save on exploration time. It provides appropriate induction patterns that avoid the need for sub-lemmas about specific constructor combinations.

### 4.3 Comparison

Other inductive provers have also been evaluated on these test suites, serving as a good point of comparison. The following table collects the number of problems solved by some of them in comparison with Hipster; note that we compare on problems for which other provers have available data. Plain figures correspond to fully automated solutions and those in parentheses (x) indicate number of successful proofs after some adaptation of settings. In total, *case-analysis* has 85 problems whilst *prod-failure* has 50.

<table>
<thead>
<tr>
<th></th>
<th>Hipster</th>
<th>HipSpec</th>
<th>Zeno</th>
<th>IsaPlanner</th>
<th>CVC4</th>
<th>Pirate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case-analysis</td>
<td>80 (84)</td>
<td>80</td>
<td>82</td>
<td>47</td>
<td>80</td>
<td>85</td>
</tr>
<tr>
<td>Prod-failure</td>
<td>41 (47)</td>
<td>44 (47)</td>
<td>21</td>
<td>40</td>
<td>-</td>
<td>(47)</td>
</tr>
</tbody>
</table>

The already mentioned HipSpec uses theory exploration, structural induction and external first-order provers to prove properties about functional programs [CJRS13]. Zeno is a tool for proving
equational inductive properties of Haskell programs [SDE12]. CVC4’s approach to inductive proving is built on SMT solving whilst Pirate is built on first-order prover SPASS, both with a top-down approach in conjecture generation [RK15, WW]. IsaPlanner is a proof planning tool for Isabelle based on rippling [DJ07, JDB10].

In comparison to other (automated) inductive provers, the new Hipster is the only one (to the best of our knowledge) to employ recursion induction. As results show, its performance is on par to other state-of-the-art tools’. Additionally, unlike these tools, Hipster produces formal, certified proofs.

To be noted is that the failing problems for Hipster in the benchmark set prod-failure (problems 33-35) differ from those HipSpec and Pirate fail at (with the exception of 33 in Pirate’s case). These three problems involve definitions for multiplication, factorial and exponentiation operations for Peano numerals with accumulator arguments. Particularly, HipSpec employed adjusted settings for lemma discovery in these three cases: the generators for random values of datatypes are manually defined. As already pointed out in §3.1, Hipster derives generators automatically, which means the simplicity of these could lead to inefficiencies when it comes to generating values of larger sizes. Hipster has not been evaluated with adjusted settings at the HipSpec/QuickSpec level and hence the exploration phase was not feasible to perform for these problems due to memory usage during testing in QuickSpec. With similar settings to HipSpec’s, problems 33-35 are likely to be solvable in Hipster too.

5 Related Work

The work on lemma discovery for inductive proofs has mainly focused on equational lemmas, for instance in the theory exploration systems IsaScheme and IsaCoSy [MMDB12, JDB11], which also work on Isabelle/HOL theories. IsaScheme requires the user to provide term schemas, which are then automatically filled in with available symbols. IsaCoSy only generates irreducible terms, and uses an internal constraint language to avoid generating anything that could be reduced by a known equation. These systems focused more on automation, while Hipster is designed to be usable in an interactive theory development. Hipster is faster, and now also supports conditional theory exploration where the user specifies an interesting condition. Conditional lemma discovery has also been missing from the IsaPlanner system, which uses proof critics to deduce lemmas from failed proof attempts [DJ07, JDB10].

Theory exploration systems rely on having an automated prover at hand to prove generated conjectures. In the context of inductive theories, most other automated provers supporting induction such as IsaPlanner, Zeno, HipSpec, Dafny and CVC4 [DJ07, SDE12, CJK13, Lei12, RK15] only support structural induction. Hipster now also provides an automated tactic for recursion induction by exploiting Isabelle’s automated derivation of such induction schemata. It can both be used in theory exploration and as a stand-alone automated tactic.

The use of recursion induction and the fact that Hipster produces LCF-style re-checkable proofs is also the main difference between Hipster and its sister system HipSpec [CJK13], with which Hipster shares its conjecture generation component. HipSpec does instead rely on external first-order provers to solve the proof obligations arising in the step- and base-cases for inductive proofs, and does not produce checkable proofs.
6 Conclusion and Further Work

Generation of conditional lemmas in theory exploration is a challenging problem, not least as it is difficult for a tool to automatically assess which side conditions are interesting. Hipster is an interactive theory exploration system, and gets around this obstacle by relying on the user to decide which predicates are deemed interesting as conditions. In this paper we have also presented a new automated tactic for recursion induction, which improves the level of proof automation of discovered conjectures in Hipster. It can also be used as a powerful stand-alone induction tactic in Isabelle. Further work on the proving side includes experimenting with different heuristics for choosing which function’s recursion induction scheme is most likely to produce a proof, as well as extending Hipster with tactics that can handle mutual- and co-induction automatically.

Hipster has various configuration options for adjusting which of the discovered lemmas are passed to its tactics in subsequent proofs. For example, in larger theories, with many explorations, we may not want to pass all discovered lemmas to Isabelle’s Metis tactic, as too many lemmas might slow down the proof process. We plan to experiment with combining Hipster’s tactics with the relevance filtering ideas used in Sledgehammer [KBKU13]. Another item of further work is to extend Hipster to produce structured proofs in Isabelle’s Isar language, instead of just a one-line application of Hipster’s custom tactics. This will be easier to read for a human user, and can be more streamlined, not needing to repeat the search done in the automatic proof found by Hipster’s powerful tactics.

Bibliography


Computing Bounds for Counter Automata

Maximilien Colange¹, Dimitri Racordon¹, Didier Buchs¹

¹ first.last@unige.ch
Centre Universitaire d’Informatique
University of Geneva, Switzerland

Abstract: Qualitative formal verification, that seeks Boolean answers about the behavior of a system, is often insufficient for practical purposes. Observing quantitative information is of greater interest, e.g. for the calibration of a battery or a real-time scheduler. Historically, the focus has been on quantities in continuous domain, but recent years showed a renewed interest for discrete quantitative domains.

Counter Automata (CA) is a quantitative extension of classical \( \omega \)-automata. Recently a nice theory has been developed for them that extends the qualitative setting, with counterparts in terms of logics, automata and algebraic structure. We propose an adaptation, with plenty of practical applications, of this formalism to express properties over discrete quantitative domains. The behavior of a Counter Automaton defines a function from infinite words to integers. Finding the bounds of such a function over a given set of words can be seen as an extension of qualitative universal and existential model-checking. Although the problem of determining whether such bounds are finite have already been addressed, efficient algorithms to compute their exact values still lack.

We propose an non-naive method for the computation of the exact values of these bounds. It relies on a generalization of the emptiness problem of \( \omega \)-automata. To solve this generalized emptiness problem, we propose an algorithm that extends emptiness check algorithms based on SCC enumeration.

Keywords: model checking, non-stochastic quantitative properties, counter automata, \( \omega \)-automata, emptiness check

1 Introduction

Qualitative verification, asking questions with Boolean answers about a system may be too strict for various applications. Calibrating a battery, timing a scheduler, measuring quality of service are practical problems of systems designers for which formal verification can offer a guarantee. Many works focus on the case of continuous quantitative domains (stochastic systems, real-time systems . . . ), and the case of discrete domains have long been overlooked.

The ability to count events is an important feature, e.g. to evaluate logical time (number of actions done by a robot, number of context switches done by a scheduler . . . ). Such measurements are of primary interest to evaluate the behavior of a system at early stages of development. Logical time can also serve as a first approximation of real-time, when events have a known bounded
duration. We seek in this paper to use a model of automata able to count events in a system with infinite behaviors, with a focus on applicability.

Among numerous quantitative extensions to finite automata, we focus on the one defined by Colcombet and Bojańczyk [BC06]. Finite automata are extended with a finite set of counters that can be incremented and reset. A special operation observe allows to store the current value of a counter to further determine the value of a run, as the infimum or the supremum of such stored values. They are part of a vast theory that nicely extends the finite automata theory, with their logical and algebraic counterparts, closure properties, over finite and infinite words and finite trees. Such automata define functions from words to integers, termed cost functions. Due to the undecidability of comparing two cost functions, many interesting features of this theory rely on the consideration of cost functions up to an equivalence relation that erases the exact values and only retains their boundedness of functions.

Regarding infinite words, on which we focus in this paper, the theory of cost functions has links with other extensions of automata or logics, by bounding a discrete quantity: bounding the maximal time between two returns to an accepting state in \(\omega\)-automata [AHK10], or bounding the wait time for the finally operator in LTL [KPV07]. Considering exact values to count events is nevertheless a great tool for verification. Think for example to the maximal number of energy units consumed by a robot between two returns to its charging base, to calibrate a battery. Or the maximal number of simultaneous threads in a parallel computation, to tune an appropriate scheduler. Or the number of false steps permitted to a human operator before a safeguard restriction occurs. For such properties, determining whether the bound is finite or not is of little help. We thus propose to use the tools and methods developed towards cost function theory (over infinite words) to practical model-checking.

Our contribution is threefold:

• we propose a method to compute the bound of a counter automaton, through the construction of a so-called capped configuration automaton (Section 3);

• we define a generalized version of the emptiness check problem for \(\omega\)-automata, to which our bound problem is reduced (Section 3);

• we show how to adapt existing emptiness check algorithms to solve the latter problem (Section 4).

The paper is organized as follows: Section 2 recalls essential definitions and concepts of \(\omega\)-automata, then introduces Counter Automata and their semantics through their configuration automata. Section 3 defines the bound value problem, and its reduction to a problem on a finite automaton, labeled the capped configuration automaton. Section 4 discusses algorithmic solutions for the latter problem, by adapting existing emptiness check algorithms for \(\omega\)-automata. Section 5 presents related work. Finally Section 6 concludes the paper and presents perspectives for future work.

**Notations**  \(\mathbb{N}\) denotes the set of nonnegative integers, and \(\leq\) denotes the usual order on integers. We note \(\mathbb{N}_\infty = \mathbb{N} \cup \{\omega\}\), and we extend \(\leq\) to \(\mathbb{N}_\infty\) such that \(n \leq \omega\) for every \(n \in \mathbb{N}\).

Throughout the paper, \(\Sigma\) denotes a finite alphabet. \(\Sigma^n\) denotes the set of words of length \(n\) for \(n \in \mathbb{N}\), \(\Sigma^*\) denotes the set of finite words over \(\Sigma\) and the empty word is noted \(\epsilon\). \(\Sigma^\omega\) denotes the set of infinite words, or \(\omega\)-words over \(\Sigma\). The length of a word \(u \in \Sigma^* \cup \Sigma^\omega\) is noted \(|u|\). For
Given a set $A$ and $\equiv$ an equivalence relation on $A$, the equivalence class of $a \in A$ is noted $[a]_\equiv$.

For $M \in \mathbb{N}$, we define the cap relation $\sim_M$ on $\mathbb{N}_\infty$ as follows: $n_1 \sim_M n_2$ iff $n_1 = n_2$ and $n_1 < M$, or both $n_1$ and $n_2$ are greater than $M$. Thus, $\sim_M$ is an equivalence relation that has $M + 1$ equivalence classes $[0]_{\sim_M}$, $[1]_{\sim_M}$, $\ldots$, $[M]_{\sim_M}$, among which the $M$ first are singletons. $\sim_M$ is compatible with $\leq$ on $\mathbb{N}_\infty$: for every $n < p$, for every $n' \sim_M n$ and $p' \sim_M p$, we have $n' < p'$ or $n' \sim_M p'$.

When a variable denotes a vector, we add an arrow above it to distinguish it from scalar values.

For instance, $\vec{x} \in \mathbb{N}^A$ denotes a vector $(x_a)_{a \in A}$ of nonnegative integers indexed by set $A$.

## 2 Counter $\omega$-Automata

This section recalls the definition of $\omega$-automata, and extends them to Counter $\omega$-Automata.

### 2.1 $\omega$-Automata

We start by recalling the concept of $\omega$-automaton, a finite automaton running on infinite words. An $\omega$-automaton defines (or recognizes) a regular $\omega$-language. They are largely used to represent sets of infinite behaviors, such as the actual behaviors (or abstraction thereof) of an actual discrete event system, or the desired behaviors of a specification. Depending on the chosen representation, each letter of a word represents a state or an action of said system or specification. A common practice uses sets of system observables as letters. We will use $\omega$-automata to represent the infinite behaviors of systems to be verified. $\omega$-automata satisfy numerous closure properties (especially union, intersection, complementation) of great importance for verification algorithms design.

We use in this paper transition-based generalized Büchi conditions, for several reasons:

- $\omega$-automata are often used to check whether a LTL formula holds on every execution of a system. In this approach, the (negation of the) LTL formula is translated into an equivalent $\omega$-automaton. For such a translation, using generalized transition-based Büchi condition is quite natural, and almost all translation algorithms produce automata with generalized transition-based Büchi conditions [DG12]. Similarly, Counter Automata can be produced from a quantitative variant of LTL, named Cost LTL [KB12]. The translation algorithms are little affected by the addition of quantitative information, and also produce Counter Automata with generalized transition-based Büchi conditions.

- generalized conditions are more concise than degeneralized ones, i.e. they allow smaller automata for a given regular $\omega$-language. Similarly, transition-based conditions are usually more concise than state-based ones. The complexity of emptiness check algorithms depends on the size of the input automaton, so conciseness is primordial. We will see later that conciseness is also particularly important when it comes to Counter Automata.

**Definition 1** An $\omega$-automaton is a tuple $\mathcal{A} = \langle Q, \Sigma, \Delta, Q_0, F \rangle$ such that:

- $Q$ is a finite set of states;
- $\Sigma$ is a finite alphabet;
Computing Bounds for Counter Automata

A step is a triple \( q_1 \xrightarrow{a} q_2 \) where \( q_1 \) and \( q_2 \) are states, \( a \in \Sigma \) and \((q_1, a, f, q_2) \in \Delta \) for some \( f \in 2^\Gamma \). Let \( u \in \Sigma^* \cup \Sigma^0 \) be a word. A path on \( u \) is a sequence of transitions \((t_i)_{1 \leq i \leq |u|}\) such that the source of \( t_{i+1} \) is the target of \( t_i \) for every \( i + 1 < |u| \). An execution on \( u \) is a path \((t_i)_{1 \leq i \leq |u|}\) on \( u \) together with a state \( q \), from which the path is to be executed. A run on \( u \) is an execution \( \langle \sigma, q_0 \rangle \) on \( u \) such that \( q_0 \in Q_0 \). A run is accepting if it sees all acceptance marks of \( F \) infinitely often.

Note that several kinds of acceptance conditions for infinite words have been proposed over the years: Büchi, Rabin, Streett, Müller, parity . . . They happen to be equivalent in expressivity (on non-deterministic automata), although some allow more compact automata than others. As explained above, our choice of transition-based generalized Büchi conditions is not arbitrary. But our constructions and results actually do not depend on the type of acceptance condition, and therefore naturally extend to any acceptance flavor. Note however that Büchi are positive: they describe what transitions are to be seen infinitely often. On the other hand, all other types of acceptance conditions describe both required and forbidden sets: transitions to be seen infinitely often, and transitions to be seen finitely often. The positive formulation allows simple enumeration algorithms, as needed in Section 4. Furthermore, due to its practical interest for LTL verification, Büchi acceptance conditions have received almost all the attention for the design of efficient verification algorithms. This provides a handful of algorithms to choose from in Section 4.

2.2 Counter Automata

We now equip \( \omega \)-automata with a finite set of integer-valued counters. They are initially set to 0. Transitions in the automaton act on counters, by incrementing, resetting or observing them. Note that the counter values do not constraint the behavior of the automaton (there are no guards on transitions): counters act as observers of the behavior. Observation keeps track of the values encountered during a run to associate it a value. Specifically, the value of a run is the smallest value observed during the run (or \( \infty \) if no observation has been undertaken).

Formally, \( C = \{ \downarrow, \uparrow, r, x, \tau \} \) denotes the set of counter actions: \( \downarrow \) increments a counter, \( r \) resets a counter to 0 and \( x \) observes the value of the counter and resets it to 0. \( \tau \) is a no-op. If a counter has value \( n \), its value after an action \( \zeta \in C \) is noted \( n.\zeta \), and we have \( n.\tau = n ; n.\downarrow = n + 1 ; n.r = n.r = 0 \). This operation naturally extends to sequences of counter actions. It is also compatible with \( \sim_M \) for \( M \in \mathbb{N} \): for \( \zeta \in C \) and \( n_1, n_2 \in \mathbb{N} \), if \( n_1 \sim_{m_1} n_2 \) then \( n_1.\zeta \sim_{m_1} n_2.\zeta \). Thus, the operation of \( C \) on the quotient \( \mathbb{N}/\sim_m \) is well-defined.

Definition 2 A counter automaton is a tuple \( \mathcal{A} = \langle Q, \Sigma, \Delta, \Gamma, Q_0, F \rangle \) where:

- \( Q \) is a finite set of states;
- \( \Sigma \) is a finite alphabet;
- \( \Gamma \) is a finite set of counters;
• \( F \) is a finite set of acceptance marks;
• \( \Delta \subseteq Q \times \Sigma \times C \times 2^F \times Q \) is the transition relation;
• \( Q_0 \subseteq Q \) is a set of initial states.

A counter automaton is not interpreted as an automaton over the alphabet \( \Sigma \times C \), but as a cost function that associates integer values to words. The semantics of a counter automaton is given through its configuration automaton, which we describe below. A configuration of \( \mathcal{A} \) is a triple \( \eta = (q_1, m_1, \bar{v}_1) \) where \( q_1 \in Q \), \( m_1 \in \mathbb{N}_\infty \) and \( \bar{v}_1 \in \mathbb{N}_\Gamma \). \( q_1 \) is the state of the configuration \( \eta \), \( m_1 \) its value and \( \bar{v}_1 \) its counter values. For a configuration \( \eta = (q_1, m_1, \bar{v}_1) \) and a transition \( t = (q_1, a, \bar{\zeta}, f, q_2) \in \Delta \), we note \( \eta.t = (q_2, m_2, \bar{v}_2) \) the configuration such that \( \bar{v}_2 = \bar{v}_1, \bar{\zeta} \) and \( m_2 \) is the infimum of \( m_1 \) and the values of the counters observed by \( \bar{\zeta} \): \( m_2 = \inf(\{m_1\} \cup \{\bar{v}_1(\gamma) \mid \gamma \in \Gamma\}) \). \( \langle (q_1, m_1, \bar{v}_1), a, \bar{\zeta}, f, (q_2, m_2, \bar{v}_2) \rangle \) is a transition in the configuration automaton if, and only if, \( t = (q_1, a, \bar{\zeta}, f, q_2) \in \Delta \) for some \( f \in 2^F \) and \( (q_1, m_1, \bar{v}_1).t = (q_2, m_2, \bar{v}_2) \). The initial states in the configuration automaton are the configurations \( \eta = \langle q_0, \omega, \bar{0} \rangle \) with \( q_0 \in Q_0 \). Note that the configuration automaton has the same set of acceptance marks \( F \) as its initial counter automaton.

Let \( \rho = ((t_i)_{i \in \mathbb{N}}, \eta_0) \) be an infinite run, note \( \eta_{i+1} = \eta_i.t_i \) and \( \eta_i = (q_i, m_i, \bar{v}_i) \) for every \( i \in \mathbb{N} \). The sequence \( (m_i)_{i \in \mathbb{N}} \) is nonnegative and decreasing. Thus, it eventually stabilizes to a value noted \( \text{Val}(\rho) \), the value of the run \( \rho \). \( \mathcal{A} \) thus defines a cost function \( \|\mathcal{A}\| \) that associates to each word \( \omega \in \Sigma_\omega \) the supremum value of all accepting runs in \( \mathcal{A} \). We also note it \( \sup\|\mathcal{A}\| \) to ease the notation.

**Definition 3** The sup-bound value problem asks, for an input automaton \( \mathcal{A} \), the value in \( \mathbb{N}_\infty \) of \( \sup\|\mathcal{A}\| \).

Strictly speaking, information on the counters are all pushed in the states of the configuration automaton, so that there is no need to keep counter actions on its transition (although keeping them helps understanding the link with its initial counter automaton). Thus, a configuration automaton can rather be seen as a plain automaton, together with a value function that associates integers to states. We have seen that all states in an SCC have the same value, so that the value function may be defined on maximal SCCs instead of states. Our bound search problem can be thus restated as searching for the highest value among accepting SCCs of a (plain) automaton. Note however that at this point, there is no guarantee that the configuration automaton is finite.

**Example 1** Figure 1a presents an example of Counter Automaton. Non-deterministic, it features a single counter, and associates to a word the maximal distance between a \( a \) and the next \( b \). \( \bar{b} \) stands for any letter but \( b \). The automaton has a single acceptance mark, represented by a black bullet on transitions. An accepting run necessarily leaves the initial state \( q_0 \) on a letter \( a \). After leaving \( q_0 \), a \( b \) may be read only in state \( q_1 \). Thus, the automaton guesses the positions of the \( a \) and its next \( b \) that are the most distant in the input word. The maximum over accepting runs, in the definition of \( \|\mathcal{A}\| \), eliminates wrong guesses. Note for example that \( \|\mathcal{A}\|( (a^kb)\omega) = k - 1 \).
for every $k \in \mathbb{N}$, and $\llbracket \alpha \rrbracket (aba^2ba^3b\ldots a^nba^{n+1}b\ldots) = \infty$, so that $\sup \llbracket \alpha \rrbracket$ is not finite.

Such an automaton can be used to observe the behavior of another one. For instance, when studying an autonomous robot, say event $a$ represents the consumption of one unit of energy, and $b$ represents a full recharge of the battery. A synchronized product between the automaton representing the behaviors of the robot and our example automaton produces a new counter automaton $\mathcal{A}'$, and $\sup \llbracket \mathcal{A}' \rrbracket$ represents the maximal quantity of energy consumed between two recharges of the battery. Computing this bound can thus be used to properly calibrate the battery.

We also show on Figure 1b the first states of the corresponding (infinite) configuration automaton. A configuration is a triple whose first component is a state of the original counter automaton, second component is the current value of the run, and third component is the current value of the counter. Note how counter actions act on the counter value and on the run value.

3 From the bound value problem to generalized emptiness check

The answer to the sup-bound value problem in is $\mathbb{N}_\infty$, which raises the question of whether the sought value is finite or infinite. The latter question is known as the boundedness problem. It has already been investigated and shown decidable [Col09]. It seems quite natural that $\sup \llbracket \alpha \rrbracket$ can be obtained from the configuration automaton, which is possibly infinite. We present in this section a reduction of the configuration automaton to a finite one, based on an analysis of the
boundedness problem.

3.1 Boundedness

The main problem towards finding sup\[A\] is whether this supremum is finite or infinite, which we call the boundedness problem, known to be decidable [Cai09].

For \( M \in \mathbb{N} \), we extend the equivalence relation \( \sim_{M} \) to configurations: \( \langle q_{1}, m_{1}, \vec{v}_{1} \rangle \sim_{M} \langle q_{2}, m_{2}, \vec{v}_{2} \rangle \) iff \( q_{1} = q_{2}, m_{1} \sim_{M} m_{2} \) and \( \vec{v}_{1} \sim_{M} \vec{v}_{2} \), where \( \sim_{M} \) is extended component-wise to vectors of integers.

**Lemma 1** Let \( \sigma \) be a finite path, and \( M \in \mathbb{N} \). Let \( \eta_{1} \) and \( \eta_{2} \) be two configurations such that \( \eta_{1} \sim_{M} \eta_{2} \). Then \( \eta_{1}.\sigma \sim_{M} \eta_{2}.\sigma \).

**Proof.** This lemma is the consequence of the compatibility of both \(<\) and the action of \( C \) with \( \sim_{M} \): for every transition \( t \) and configurations \( \eta_{1}, \eta_{2} \), if \( \eta_{1} \sim_{M} \eta_{2} \), then \( \eta_{1}.t \sim_{M} \eta_{2}.t \). We then conclude by induction on the size of \( \sigma \).

We restate a result first established (although in a slightly different setting) by [Kup14]. We also provide a direct proof.

**Proposition 1** We note \( \Theta \) the length of the longest run (i.e. starting from an initial state) without loop in \( \mathcal{A} \). The following three are equivalent:

(a) \( \sup[\mathcal{A}] = \omega \).

(b) \( \sup[\mathcal{A}] > \Theta \).

(c) \( \mathcal{A} \) has an accepting ultimately periodic run in which, for every \( \gamma \in \Gamma \), every occurrence of \( \alpha \sigma \) for \( \gamma \) is preceded by a cycle that increments \( \gamma \) at least once and does not reset \( \gamma \).

**Proof.** Obviously, (a) implies (b), and (c) implies (a). We prove that (b) implies (c). Assume that \( \sup_{u \in \Sigma^{\mathcal{A}}}\|\mathcal{A}\|_{u} \geq \Theta \), and consider an accepting run \( \rho \) such that \( \text{Val}(\rho) \geq \Theta \). We first build an accepting lasso from \( \rho \), and we will then show that it satisfies (c).

Consider the sequence \( (\eta_{i} = \langle q_{i}, m_{i}, \vec{v}_{i} \rangle)_{i \in \mathbb{N}} \) of configurations visited by \( \rho \), and note \( \rho_{i} \sim_{\alpha} = (\langle q_{i}, [m_{i}]_{\sim_{\alpha}}, [\vec{v}_{i}]_{\sim_{\alpha}} \rangle)_{i \in \mathbb{N}} \). By definition of \( \rho \), for every \( i \in \mathbb{N}, m_{i} \geq \Theta \), so that \( [m_{i}]_{\sim_{\alpha}} = [\Theta]_{\sim_{\alpha}} \). The \( [\vec{v}_{i}]_{\sim_{\alpha}} \)'s have a finite number of possible values, and \( Q \) is finite, so there is some \( i_{0} \in \mathbb{N} \) such that \( \langle q_{i_{0}}, [\Theta]_{\sim_{\alpha}}, [\vec{v}_{i_{0}}]_{\sim_{\alpha}} \rangle \) occurs infinitely often in \( \rho_{i} \sim_{\alpha} \). Consider \( \pi = \rho(0) \ldots \rho(i_{0} - 1) \) and \( \sigma = \rho(i_{0}) \ldots \rho(j) \) for some \( j > i_{0} \) such that \( \langle q_{j}, m_{j}, \vec{v}_{j} \rangle \sim_{\Theta} \langle q_{i_{0}}, m_{i_{0}}, \vec{v}_{i_{0}} \rangle \) and all acceptance marks in \( F \) appear between positions \( i_{0} \) and \( j \). Such a \( \sigma \) exists because \( \rho \) is accepting. Note \( \rho' \) the run \( \pi \sigma^{\omega} \). By definition of \( \pi \) and \( \sigma \), \( \rho' \) is an accepting run of \( \mathcal{A} \). Furthermore, \( \eta_{i_{0}}.\sigma \sim_{\Theta} \eta_{i_{0}} \) by definition of \( \sigma \). By Lemma 1, a simple induction shows that \( \eta_{i_{0}}.\sigma^{k} \sim_{\Theta} \eta_{i_{0}} \) for every \( k \in \mathbb{N} \). Therefore \( \text{Val}(\rho') \sim_{\Theta} \text{Val}(\rho) \), and \( \text{Val}(\rho') \geq \Theta \).

Let \( \gamma \in \Gamma \) and consider an occurrence of \( \alpha \sigma \) for \( \gamma \) in \( \rho' \), say at position \( i \). \( \text{Val}(\rho') \geq \Theta \) implies that the value of \( \gamma \) is greater than \( \Theta \) when observed, so that this occurrence of \( \alpha \sigma \) for \( \gamma \) is preceded by at least \( \Theta \) increments of \( \gamma \). A transition carries at most one action for \( \gamma \), so there are \( \Theta \) positions \( j_{0} < \cdots < j_{\Theta - 1} \) before \( i \) in \( \rho' \) that increment \( \gamma \). Obviously, \( \gamma \) is never reset between positions
By definition of $\Theta$, among the source states of the transitions $(\rho'(j_k))_{k < \Theta}$, there is one that occurs twice, say at positions $j_p$ and $j_q$ ($p < q$). Thus, $\rho'(j_p) \ldots \rho'(j_q - 1)$ is a cycle in $\mathcal{A}$ that occurs before position $i$, increments $\gamma$ at least once and never resets $\gamma$. Such a cycle exists whatever the occurrence of $\omega \gamma$ for $\gamma$ in $\rho'$, and whatever $\gamma$. Therefore, the accepting ultimately periodic run $\rho'$ satisfies condition (c).

\section*{Example 2}
We illustrate Proposition 1 on the automata of Figure 1. The longest run without loop in the automaton of Figure 1a is $q_0 \to q_1 \to q_2$, so that $\Theta = 3$. A run $\rho$ of value greater than 3 necessarily goes through state $q_2$, passes at least once through the loop $q_2 \xrightarrow{\delta_1} q_2$, and goes to state $q_1$ through transition $q_2 \xrightarrow{\delta_2} q_1$. By repeating $n$ times the loop $q_2 \xrightarrow{\delta_1} q_2$, one builds a new accepting run $\rho'$ of value $\text{Val}(\rho') = \text{Val}(\rho) + n$. Thus $\mathcal{A}$ has accepting runs of arbitrarily high values, which shows that $\sup_{\mathcal{A}} \text{Val}(\rho)$ is infinite. The key to this construction is the third item of Proposition 1: there must be an incrementing cycle to be repeated before every operation $\omega$.

In this example, Proposition 1 also holds for $\Theta = 2$. However, runs of value 0 or 1 are not guaranteed to go through the incrementing loop $q_2 \xrightarrow{\delta_1} q_2$, and can be the base of the above construction of accepting of arbitrarily high values.

\section*{3.2 Reduction to a finite automaton}

Proposition 1 simplifies our problem: it suffices to search $[\sup_{\mathcal{A}} \text{Val}(\rho)]_{\sim_\Theta}$. Recall that the quotient $\mathbb{N}_\omega/\sim_\Theta$ is totally ordered, and that $[m]_{\sim_\Theta} < [n]_{\sim_\Theta}$ iff $m < n$ and $m < \Theta$. Thus, $[\sup_{u \in \Sigma^\omega} \text{Val}(\rho(u))]_{\sim_\Theta} = \sup_{u \in \Sigma^\omega} [\text{Val}(\rho(u))]_{\sim_\Theta}$. Similarly, for $u \in \Sigma^\omega$, $[\text{Val}(\rho(u))]_{\sim_\Theta} = \sup_{\rho \in \text{Acc}_{\mathcal{A}}(u)} [\text{Val}(\rho)]_{\sim_\Theta}$.

We define the configuration automaton capped at $\Theta$ of $\mathcal{A}$ as follows:

- $(Q \times \mathbb{N}_\omega \times \mathbb{N}^\mathbb{N})/\sim_\Theta$ is the set of states;
- $(Q_0 \times \{\omega\} \times \{\hat{0}\})/\sim_\Theta$ is the set of initial states;
- $\eta \xrightarrow{t} [\eta, t]_{\sim_\Theta}$ for every configuration $\eta \in \eta$ and every transition $t$. Lemma 1 ensures that this transition relation is well-defined, as $[\eta, t]_{\sim_\Theta}$ in fact does not depend on the choice of $\eta$ in the class $\tilde{\eta}$. $t$ in the capped configuration automaton keeps its same acceptance marks.

Note that the constructed automaton is finite as long as $\Theta$ is finite; if $\Theta = \omega$, the capped configuration automaton is the original configuration automaton.

We define the value of a run as in the capped configuration automaton. Let $\rho = ((\bar{t}_i)_{i \in \mathbb{N}}, \bar{q}_0)$ be an infinite run, note $\tilde{\eta}_{i+1} = \tilde{\eta}_i t_i$ and $\tilde{\eta}_i = (\bar{q}_i, \bar{m}_i, \bar{v}_i)$ for every $i \in \mathbb{N}$. The sequence $(\bar{m}_i)_{i \in \mathbb{N}}$ is decreasing in the finite set $\mathbb{N}_\omega/\sim_\Theta$. Thus, it eventually stabilizes to a value noted $\text{Val}_{\sim_\Theta}(\rho) \in \mathbb{N}_\omega/\sim_\Theta$, the value of the run $\rho$ in the capped configuration automaton.

\section*{Proposition 2}
For every accepting run $\rho$ of $\mathcal{A}$, there is an accepting run $\rho'$ in the capped configuration automaton such that $\text{Val}_{\sim_\Theta}(\rho') = [\text{Val}(\rho)]_{\sim_\Theta}$. Conversely, for every run $\rho'$ in the capped configuration automaton, there is a run $\rho$ of $\mathcal{A}$ such that $\text{Val}_{\sim_\Theta}(\rho') = [\text{Val}(\rho)]_{\sim_\Theta}$.

\textbf{Proof.} ($\Rightarrow$) Let $\rho = ((\bar{t}_i)_{i \in \mathbb{N}}, \bar{q}_0)$ be an accepting run of $\mathcal{A}$. We note $\eta_{i+1} = \eta_i t_i$ and $\eta_i = (\bar{q}_i, \bar{m}_i, \bar{v}_i)$ for every $i \in \mathbb{N}$. By definition, $[\eta_0]_{\sim_\Theta}$ is an initial state of the capped configuration automaton, and $[\eta_i]_{\sim_\Theta} \xrightarrow{\delta_i} [\eta_{i+1}]_{\sim_\Theta}$ is a step of the capped configuration automaton for every
\[ i \in \mathbb{N}. \text{ Thus, the } \rho' = (\langle t_i \rangle_{i \in \mathbb{N}}, [\eta_0]_{\sim \Theta}) \text{ is a run in the capped configuration automaton. It is not hard to see that it is accepting iff } \rho \text{ is accepting. Recall that the sequence } (m_i)_{i \in \mathbb{N}} \text{ converges to the value } \text{Val}(\rho), \text{ so that the sequence } ([m_i]_{\sim \Theta})_{i \in \mathbb{N}} \text{ converges to } [\text{Val}(\rho)]_{\sim \Theta}. \text{ But the limit of this sequence is also } \text{Val}_{\sim \Theta}(\rho') \text{ by definition, so that } \text{Val}_{\sim \Theta}(\rho') = [\text{Val}(\rho)]_{\sim \Theta}.

(\Leftarrow) \text{ Let } \rho' = (\langle t_i \rangle_{i \in \mathbb{N}}, [\bar{\eta}_0]) \text{ be an accepting run of the capped configuration automaton. We note } q_0 \text{ the state of the configuration } \bar{\eta}_0, \eta_0 = \langle q_0, \infty, 0 \ldots 0 \rangle \text{ and define } \rho = (\langle t_i \rangle_{i \in \mathbb{N}}, q_0). \text{ The definition of the transition relation of the capped configuration automaton guarantees that } \rho \text{ is a run of } \mathcal{A} (\text{all steps are well-defined, and } q_0 \text{ is an initial state of } \mathcal{A}). \text{ It is also immediate that } \rho \text{ is accepting if } \rho' \text{ is accepting. We note } \eta_{i+1} = \eta_i t_i \text{ and } \bar{\eta}_{i+1} = \bar{\eta}_i t_i \text{ for every } i \in \mathbb{N}. \text{ Obviously } \eta_0 \in \bar{\eta}_0, \text{ and Lemma 1 then guarantees that } \eta_i \in \bar{\eta}_i \text{ for every } i \in \mathbb{N}. \text{ This last property ensures that the limit } \text{Val}(\rho) \text{ belongs to the equivalence class } \text{Val}_{\sim \Theta}(\rho'): \text{Val}_{\sim \Theta}(\rho') = [\text{Val}(\rho)]_{\sim \Theta}. \]

\[\begin{align*}
q_0,\infty,0 & \xrightarrow{a/\tau} q_1,0,0 \\
q_2,\infty,0 & \xrightarrow{b/\tau} q_1,0,0 \\
q_2,\infty,1 & \xrightarrow{b/\tau} q_1,1,0 \\
q_2,\infty,2 & \xrightarrow{b/\tau} q_1,2,0 \\
q_2,\infty,3 & \xrightarrow{b/\tau} q_1,3,0 \\
q_2,\infty,n & \xrightarrow{b/\tau} q_1,n,0 \\
q_2,\infty,n+1 & \xrightarrow{b/\tau} q_1,n+1,0 \\
\cdots
\end{align*}\]

(a) Example of capped configuration automaton 

\[\begin{align*}
q_0,\infty,0 & \xrightarrow{a/\tau} q_1,0,0 \\
q_2,\infty,0 & \xrightarrow{b/\tau} q_1,0,0 \\
q_2,\infty,1 & \xrightarrow{b/\tau} q_1,1,0 \\
q_2,\infty,2 & \xrightarrow{b/\tau} q_1,2,0 \\
q_2,\infty,3 & \xrightarrow{b/\tau} q_1,3,0 \\
q_2,\infty,n & \xrightarrow{b/\tau} q_1,n,0 \\
q_2,\infty,n+1 & \xrightarrow{b/\tau} q_1,n+1,0 \\
\cdots
\end{align*}\]

(b) The pattern in the configuration automaton abstracted away by the capped configuration automaton

**Figure 2:** Illustration of the capped configuration automaton construction

**Example 3** Figure 2a shows the capped configuration automaton of the automaton of Figure 1a. Its configuration automaton indeed exhibits the pattern of Figure 2b, a pattern abstracted two nodes of the capped configuration automaton. Following Proposition 1 (and its illustration in the previous example), the capped configuration automaton does not keep track of counter values beyond } \Theta = 3 \text{ as this value suffices to build runs with arbitrarily high values. The exact behaviors of the original automaton must be kept for values below } \Theta, \text{ as they do not exhibit a pattern to be abstracted away.}
4 SCC enumeration to find the bound

We now reformulate the sup-bound value problem, in order to provide algorithms to solve it. Suppose we are given a $\omega$-automaton, along with a function that associate integer values to its maximal Strongly Connected Components (SCC). Our problem is to find the largest values among accepting maximal SCCs, which we call the largest accepting value problem. We will consider only maximal SCC, abusively called SCC in the remaining of the paper.

This problem generalizes the classical emptiness problem for Büchi automata. In the latter context, all SCCs have the same value $\top$, and $\bot$ denotes the neutral element for the operation sup, so that $\sup \emptyset = \bot$. The largest value among accepting SCCs is thus $\top$ if the language of the automaton is not empty, and $\bot$ otherwise.

To solve the largest accepting value problem, we naturally propose to take inspiration from classical Büchi emptiness check algorithms. Two families of emptiness check algorithms can be distinguished: those based on Nested Depth-First Search (NDFS), and those based on SCC enumeration. We focus on the latter category for two reasons. First, SCC enumeration matches exactly the definition of the largest accepting value problem, and thus seems more natural. Second, SCC enumeration-based algorithms are more suited to generalized Büchi conditions. While the nesting depth of DFS walks in NDFS-based algorithms depends on the number of acceptance conditions, SCC enumeration-based algorithms deal with generalized acceptance conditions in a single walk of the input automaton, which is a great advantage towards efficiency.

SCC-based emptiness checks enumerates the SCC of the input automaton, keeping track of the accepting conditions to detect accepting ones. As shown in algorithm 1, they typically return as soon as an accepting SCC is detected, but are in fact able to enumerate all accepting SCCs. The adaptation of such algorithms to the largest accepting value problem is straightforward, and is shown in algorithm 2. The algorithm keeps track at every moment of the largest accepting value encountered so far, and updates it whenever a new accepting SCC is found. At the end of the enumeration, the largest accepting value is known.

**Algorithm 1:** Typical SCC enumeration-based emptiness check algorithm

```plaintext
// Based on a SCC enumeration keeping track of acceptance conditions
1 foreach maximal SCC s do
2   if s is accepting then
3     return "not empty"
4 return "empty"
```

**Algorithm 2:** SCC enumeration-based largest accepting value algorithm

```plaintext
// v is the SCC value function
1 value ← $-\infty$
2 foreach maximal SCC s do
3   if s is accepting then
4     value ← max(value, v(s))
5 return value
```

4.1 A smaller automaton

Informally, Proposition 1 indicates that there is no need to keep track of the exact run values above $\Theta$ to compute $\sup{[A]}$. Similarly, as the value of a single run $\rho$ is the smallest one among the counter values observed in $\rho$, there is no need to keep track of the exact values of counters.
beyond the current run value. Once a value \( n \) has been observed, all counter values larger than \( n \) can be dismissed as they have no impact on the value of the run. We thus propose a slight improvement of our capped configuration automaton.

Formally, we first define a new equivalence relation over configurations: for \( M \in \mathbb{N} \), \( \langle q_1, m_1, \vec{v}_1 \rangle \approx_M \langle q_2, m_2, \vec{v}_2 \rangle \) iff \( q_1 = q_2 \), \( m_1 \sim_M m_2 \) and \( \vec{v}_1 \sim_{\min(m_1, M)} \vec{v}_2 \). Note that the action of \( C \) on configurations is compatible with \( \approx_\Theta \), so that we have an analogous of Lemma 1.

**Lemma 2**  Let \( \sigma \) be a finite path, and \( M \in \mathbb{N} \). Let \( \eta_1 \) and \( \eta_2 \) be two configurations such that \( \eta_1 \approx_M \eta_2 \). Then \( \eta_1.\sigma \approx_M \eta_2.\sigma \).

We define the double capped configuration automaton of \( \mathcal{A} \) as:
- \((Q \times \mathbb{N}_\infty \times \mathbb{N}^\Gamma)_{/\approx_\Theta} \) is the set of states;
- \((Q_0 \times \{ \omega \} \times \{ \emptyset \})_{/\approx_\Theta} \) is the set of initial states;
- \([\eta]_{\approx_\Theta} \xrightarrow{\sigma} [\eta']_{\approx_\Theta} \) whenever \( \eta \xrightarrow{\sigma} \eta' \) is a step for \( \mathcal{A} \).

**Lemma 2** ensures that this transition relation is well-defined. Once again, current values of runs are positive are decreasing, and thus eventually stabilizes. This stabilization value defines the value \( Val_{\approx_\Theta}(\rho) \) of a run in the double capped configuration automaton.

**Proposition 3**  For every accepting run \( \rho \) of \( \mathcal{A} \), there is an accepting run \( \rho' \) in the double capped configuration automaton such that \( Val_{\approx_\Theta}(\rho') = [Val(\rho)]_{\approx_\Theta} \). Conversely, for every run \( \rho' \) in the capped configuration automaton, there is a run \( \rho \) of \( \mathcal{A} \) such that \( Val_{\approx_\Theta}(\rho') = [Val(\rho)]_{\approx_\Theta} \).

We do not detail the proof of Proposition 3 as it does not really differ from the proof of Proposition 2.

### 4.2 A note on complexity

All in all, \( |\mathcal{A}|_S \) can be computed by enumerating the accepting SCC of either the capped configuration automaton or the double capped configuration automaton. Many variants have been proposed, sequential or parallel, based either on Dijkstra’s or Tarjan’s SCC enumeration algorithms, to handle simple or generalized, state-based or transition-based Büchi acceptance conditions. It is not the scope of this paper to detail all existing algorithms and their variants, and we refer the reader to [RDKP13] for a survey. SCC enumeration algorithms have time and space complexity related (mostly linear) to the size of the input automaton (in terms of number of transitions and/or number of states). For the sake of generality, we estimate the size of the both capped configuration automata, rather than the complexity of a specific algorithm.

We denote by \( |\mathcal{A}|_S \) the number of states of \( \mathcal{A} \), and by \( |\mathcal{A}|_T \) its number of transitions. In the capped configuration automaton, each state and each transition of \( \mathcal{A} \) is duplicated at most \((\Theta + 1)^{|T|+1}\) times. The capped configuration automaton is thus at most \((\Theta + 1)^{|T|+1}\) larger than \( \mathcal{A} \). We recall that \( \Theta \) depends on the graph structure of \( \mathcal{A} \) only, not on the transition labels (letters and counter actions).

The ratio is slightly better for the double capped configuration automaton, as each state and each transition of the initial counter automaton is duplicated at most \( \sum_{i=1}^{\Theta+1} i^{|T|} \). The latter sum is
a polynomial in Θ of degree |Γ| + 1, whose coefficient of highest degree is \( \frac{1}{|Γ|+1} \). For a fixed |Γ|, the double capped configuration automaton is asymptotically smaller by a factor |Γ| + 1 than the capped configuration automaton. This is a mild compensation of the size exponential in |Γ|.

It is easy to see that \( Θ ≤ |A|_S \), which gives in the worst-case a capped configuration automaton with |A| |Γ| + 2 states. Yet, expressing the size ratio in terms of Θ is much finer. Observe that the capped and double capped configuration automaton construction are still correct when Θ is over-approximated: unnecessary values may be tracked, which only impacts efficiency. Knowing the exact value of Θ is thus mandatory. Yet, Θ can be computed by enumerating the SCC of A, i.e. in time linear in |A|, a simple pre-computation dominated by the complexity of the subsequent search for the bound.

### 4.3 Improvement and optimizations

The (double) capped configuration automaton has a very specific structure that hints for several possible optimizations. The first feature to exploit is the fact that the current run value decreases along a run. A change of the current run value guarantees that a new SCC is entered. Such information, usually not accessible in the general setting of SCC enumeration, can be taken advantage of. When a new SCC is entered, say at state s, the exploration of the sub-automaton reachable from s does not need any information about the path from the root. This enables a recursive enumeration of the sub-automaton, with s as its initial state. Such recursive enumeration is particularly interesting for parallelism.

The second feature to exploit is the fact that a SCC value (current run value) is computable from a single state, and does not necessitate the whole maximal SCC to be known. Recall that the algorithm searches for the largest value among the accepting SCCs, by enumerating the SCC and maintaining the largest SCC value, say \( α \), encountered so far. Those SCC whose values are smaller than the current \( α \) need not be explored, and can be pruned from the enumeration. This can considerably reduce the size of the automaton to be explored. A simple heuristic may further be used: when such a choice occurs, the algorithm should explore the SCC with the larger value first. If it happens to be a accepting SCC, it yields a larger \( α \) and maximized the previous pruning optimization.

### 5 Related Work

A famous problem in language theory is the star-height problem: given a language \( L \) (of finite words) and an integer \( k \), is there a regular expression for \( L \) with at most \( k \) nested Kleene stars? Proposed in 1963 [Egg63], it was proven decidable in 1988 [Has88] by exhibiting an algorithm with non-elementary complexity, and a much more efficient algorithm was then proposed in 2005 [Kir05]. Both algorithms translate the problem to the existence of a bound for a function mapping words to integers, represented in both cases by an automaton equipped with counters (distance automata for the former, nested distance desert automata for the latter). This problem of the existence of a bound is then shown decidable. This is the first of many problems that reduce to the existence of a bound for such automata.

This motivated an in-depth study of automata with counters (as we use it) as a general frame-
work, that came up with a theory extending the one of regular languages, with logical and algebraic counter-parts [Col09]. On infinite words, the logical counter-part motivated the introduction and study of Cost Linear Temporal Logics [KB12], an extension of LTL able to count discrete events. This theory also encompasses promptness properties, a variant of liveness where a bound on the wait time of a recurring event must exist [KPV07, AHK10]. But all these works, motivated by the boundedness problem, overlook the exact values of the functions. On one hand, this relaxation enables nice closure properties (such as the equivalent expressiveness for inf-automata and sup-automata). On the other hand, it only allows to reason about the existence of a bound, not to compute values.

In verification, not all questions have a boolean answer, so that various quantitative extensions of automata have been considered, such as weighted automata (see [DG07] for a survey). Despite their various domains of application, they have limited expressivity, as the domain of weights is required to be a semi-ring. An extension to arbitrary operations on weights have been recently proposed [ADD+13]. It encompasses various extensions of weighted automata, such as Discounted Sum Automata [AHM03] and Counter $\omega$-Automata as considered in this paper. All these formalisms can be characterized by the absence of guards on register values. These extensions sometimes have equivalent logics (such as discounted linear temporal logics [ABK14] or counting LTL [LMP10]). We have already studied the problem of computing bounds for the aforementioned Cost Linear Temporal Logics [CRB15]. The present work on Counter Automaton is an extension of our previous work on CLTL, as Counter Automata are more expressive than CLTL.

Most of the works cited above only focus on expressivity, decidability and complexity problems, with little consideration to the practical use of such quantitative extensions of automata. This situation contrasts with older formalisms: $\omega$-automata have already received great focus towards practical applications, illustrated by numerous emptiness checks algorithms (see [SE05] or [RDKP13] for an overview) and many implementations, principally oriented towards LTL model-checking (see [RV10] for a survey). Some quantitative extensions of automata possess a similar maturity towards practical applications, especially timed automata [BDL+06] and weighted automata [KNP11].

6 Conclusion

We have presented in this paper a method to compute the sup-bound value problem on Counter $\omega$-automata. It is based on a reduction of the configuration automaton to a finite automaton, thanks to previous results on the boundedness of such automata. On this finite automaton, the sup-bound value problem is translated into a generalization of the emptiness check problem, as a single-player game with (generalized) Büchi objectives. This problem is itself solved by enumerating accepting SCCs of the finite automaton. Such enumeration algorithms being used for instance for $\omega$-automata emptiness checks, our approach thus builds upon methods already tried and tested for qualitative verification. To our knowledge, it is the first time that such a method for Counter Automata is proposed. It opens perspectives for the practical use of Counter $\omega$-automata towards system verification.

Future works include two axis. First, the method we have presented should now be imple-
mented, in order to perform experimental evaluation. Second, we believe that there is still rooms for optimization of the algorithms, and for heuristics that would increase performance. The setting of the intermediate problem of a single-player game with Büchi objectives would be an appropriate setting for such a study.

Bibliography


http://arxiv.org/abs/1506.05728


Transforming Event-B Models to Dafny Contracts

Mohammadsadegh Dalvandi¹, Michael Butler², Abdolbaghi Rezazadeh³

School of Electronic & Computer Science, University of Southampton

¹ ² ³ md5g11.mjb.ra3@ecs.soton.ac.uk

Abstract: Our work aims to build a bridge between constructive (top-down) and analytical (bottom-up) approaches to software verification. This paper presents a tool-supported method for linking two existing verification methods: Event-B (constructive) and Dafny (analytical). This method combines Event-B abstraction and refinement with the code-level verification features of Dafny. The link transforms Event-B models to Dafny contracts by providing a framework in which Event-B models can be implemented correctly. The paper presents a method for transformation of Event-B models of abstract data types to Dafny contracts. Also a prototype tool implementing the transformation method is outlined. The paper also defines and proves a formal link between property verification in Event-B and Dafny. Our approach is illustrated with a small case study.

Keywords: Formal Methods, Hoare Logic, Program Verification, Event-B, Dafny

1 Introduction

Various formal methods communities [CW96, HMLS09, LAB⁺06] have suggested that no single formal method can cover all aspects of a verification problem therefore engineering bridges between complementary verification tools to enable their effective interoperability may increase the verification capabilities of verification tools. We distinguish two major approaches to software correctness based on their target phases in the development cycle: the constructive approach and the analytical approach. The constructive approach focuses on the early stages of the development and aims at formal modelling of the intended behaviour and structure of a system in different levels of abstraction and verifying properties of models. The analytical approach focuses on the code level and its target is to verify properties of the final program code. A wide range of verification tools exist to support both the approaches provided by formal methods communities worldwide. A high level look at these two approaches suggests that the constructive and analytical approaches should complement each other well. Nevertheless, our understanding and experience of how these approaches can be combined at a large scale is very limited. This represents a wasted opportunity, as these approaches are not benefiting from each other effectively.

We have chosen Event-B [Abr10] and Dafny [Lei10] as examples of constructive and analytical approaches respectively. Event-B is a formal approach for modelling and verifying software systems. An Event-B model is built through a number of successive refinement steps starting from an abstract representation of the system and proceed towards a concrete level. Event-B is supported by an open platform called Rodin [ABH⁺10]. Dafny is a programming language and verifier. Given a program code and its formal specification, the Dafny tool [LW14] (which is an
SMT-based verifier) can verify the program against its contract. In Dafny, contracts are annotations within the code. Event-B in its initial form does not have any support for the final phase of the development (implementation phase). On the other hand, Dafny has very little support for abstraction and refinement. In this work we present a tool-supported development method by linking two verification technologies: Rodin and Dafny. Our combined methodology is beneficial for both Event-B and Dafny users. It makes the abstraction and refinement of Event-B available for generating Dafny specifications which are correct with respect to a higher level of abstract specification in Event-B and provides a framework in which Event-B models can be implemented and verified in a programming language.

We provide a method for transforming Event-B models to Dafny code contracts (method pre- and post-conditions). In this paper we use the terms code contracts and annotations interchangeably. Transformation of Event-B formal models to annotated Dafny method declarations is achieved by defining a set of transformation rules. Using this set of transformation rules, one can generate code contracts from Event-B models but not implementations. The generated code contracts must be seen as an interface that can be implemented. The implementation can be verified later against the generated annotations using an automatic verifier to prove the correctness of the implementation with regards to the high level Event-B specification. We also developed a tool that is based on these translation rules for generating Dafny code contracts from Event-B models. The transformation rules are validated by being applied to a number of case studies including a map, a queue, and a stack abstract data types. This paper extends our previous short paper [DBR14] by providing full details of our transformation method and its proof of correctness.

The organisation of the rest of the paper is as follows: in Section 2, background information on Event-B and Dafny is given. Section 3 and Section 4 explain the methodology, transformation rules, and the formal basis of our work. A small example of transformation of Event-B models to Dafny contracts is presented in Section 5. Section 6 provides an overview to the tool support for our method. In Section 7 related and future works are discussed and finally Section 8 contains the conclusions.

## 2 Background

### 2.1 Event-B

Event-B is a formal modelling language for system level modelling based on set theory and predicate logic for specifying, modelling and reasoning about systems, introduced by Abrial [Abr10]. Modelling in Event-B is facilitated by an extensible platform called Rodin [ABH+10]. A model in Event-B consists of two main parts: contexts and machines. The static part(types and constants) of a model is placed in a context and is specified using carrier sets, constants and axioms. The dynamic part (variables and events) is specified in a machine by means of variables, invariants and events. An event models the state change in the system. Each event may have a number of assignments called actions. Each event may also have a number of guards. Guards are predicates that describe the necessary conditions which should be true before an event can occur. An event may have a number of parameters. Event parameters are considered to be local to the event. Figure 1 illustrates machine $m0$ with two events $Add$ and $Remove$. KEYS and VALUES
are carrier sets in context \( c0 \) which is not shown here.

**Figure 1: Machine \( m0 \): the Most Abstract Level of Map ADT Model**

Modelling a complex system in Event-B can largely benefit from refinement. Refinement is a stepwise process of building a large system starting from an abstract level and proceeds towards a more concrete level by a series of successive steps in which new details of functionality are added to the model in each step [But13]. The abstract level represents key features and the main purpose of the system. It is essential to prove the correctness of refinement steps in Event-B. Refinement of a model may consist of refining existing events, adding new events, and adding new variables and invariants. The new events must not diverge. This means that they should not run for ever. Each refinement may involve introducing new variables to the model. This usually results in extending abstract events or adding new events to the model. It is also possible to replace abstract variables by newly defined concrete variables. Concrete variables are connected to abstract variables through gluing invariants. A gluing invariant associates the state of the concrete machine with that of its abstraction. All invariants of a concrete model including gluing invariants should be preserved by all events. All abstract events may be refined by one or more concrete events.

The built-in mathematical language of the Rodin platform is limited to basic types and constructs like integers, boolean, relations and so on. The Theory Plug-in [BM13] has been developed to make the core language extension possible. A theory, which is a new kind of Event-B component, can be defined independently from a particular model and it is the means by which the mathematical language and mechanical provers may be extended.

### 2.2 Dafny

Dafny [Lei10] is an imperative sequential programming language. A program in Dafny usually contains two parts, namely implementation and specification. Dafny supports generic classes with some basic features such as method definitions, dynamic allocation, and inductive types for implementation. A method in Dafny is a piece of imperative, executable code. The verification power of Dafny originates from its specification constructs. A program behaviour can be specified in Dafny using constructs such as methods’ pre- and post-conditions, framing constructs and termination metrics. The specification language also offers updatable ghost variables, recursive functions, sets, sequences and some other features. The Dafny verifier which is based on an SMT-solver called Z3 [DB08] uses specification to verify the implementation.

As mentioned above Dafny specification supports functions. A function has a very similar
concept to mathematical functions and cannot write to memory and is defined by an expression. Functions are required to have only one unnamed return value. A special form of function which returns a boolean value is called predicate. Dafny uses the \textit{ensures} keyword for post-condition declarations. A post-condition is always a boolean expression. Each method can have more than one post-condition which can be either joined with boolean \textit{and} (\&\&) operator or defined separately using the \textit{ensures} keyword. To declare a pre-condition the \textit{requires} keyword is used. Like post-conditions, multiple pre-conditions are allowed in the same style. Pre- and post-conditions are placed after method declarations and before method bodies. Dafny does not have any specific construct for specifying \textit{class invariants}. A work around is to place all class-level invariants in a predicate and then include this predicate in the pre- and post-conditions of all the class methods. In this we enforce the verifier to check if each method preserves all invariants.

3 Transferring Event-B Machines to Dafny Classes

Event-B supports a richer mathematical language than Dafny. For this reason, before an Event-B model is transformed to Dafny contracts, it must be refined to a level where the data types and operators have a Dafny counterpart. For example, relations do not have any Dafny counterpart so they are refined to a more concrete data structure (e.g. sequences) before transformation takes place. This is essential for reducing the syntactic gap between Event-B and Dafny. When the aforementioned point in the refinement process is reached then a machine and its elements (e.g. variables, invariants,...) are translated to a Dafny class. The translation of variables, generic types, and invariants is almost one-to-one. Transformation of events to method contracts is discussed in the next section. Generic types in an Event-B model, as mentioned earlier, are defined using carrier sets in a context. In Dafny, generics are declared in angle brackets after the name of a class. The following example shows how generic types are defined in Dafny:

\begin{verbatim}
class class_name<T_1, T_2,...,T_n>{ ...class body... }
\end{verbatim}

When a machine is translated to a Dafny class, all carrier sets which are defined in the context that is seen by that machine are translated as Dafny generics. Note that it is assumed at the moment that the context of the model only contains carrier sets. In an Event-B model variables are declared in the variables part of a machine and their types are specified using \textit{typing invariants} which are defined separately in the invariants part of the machine. All machine’s variables are translated as class variables in Dafny. Event-B invariants can be categorised as follows:

- \textbf{Typing invariants} that declare the types of a variable.
- \textbf{Model invariants} that express the properties of a model.
- \textbf{Gluing invariants} that relate the concrete variables to abstract variables.

As explained earlier, typing invariants are used for variable declarations. Preservation of typing invariants are checked implicitly by the Dafny type system. Gluing invariants are not translated to Dafny because at the moment we assume that the machine that is being translated is a data refinement of the abstract machine and none of the abstract variables are present in the refined
machine. Preservation of gluing invariants must be proved in Event-B. Only model invariants are translated to Dafny. The conjunction of all model invariants are placed in a Dafny predicate called \texttt{Invariants()}. It is explained later how the predicate \texttt{Invariants()} is used in pre-condition generation.

4 Transforming Events to Annotated Method Declarations

In the previous section we discussed how the declaration elements of an Event-B machine should be translated to Dafny class members. In this section we present the way in which machine events are transformed to annotated Dafny method declarations.

4.1 Constructor Statement

Machine events are translated to Dafny methods. In Event-B, each event has three main parts: \textit{parameters}, \textit{guards}, and \textit{actions}. Parameters may have different implicit roles (e.g. input or output) in an event. During the translation process, the role of each parameter must be made explicit by the modeller. There might be cases where the modeller decides to merge several events to form a single Dafny method. This is because the Event-B style is to represent different cases of some conceptual operation as separate events. Events that are going to be merged together and the target Dafny method should also be made explicit by the modeller. To cater for this we have have extended the underlying representation of Event-B machines with a new element called \textit{constructor statement}. Constructor statements are used to make the parameters’ roles and the merging events explicit. A constructor statement has the following form:

\begin{verbatim}
method mtd \_name\((\text{in}_1, \text{in}_2, \ldots)\) returns \((\text{out}_1, \text{out}_2, \ldots)\) \{\text{Evt}_1, \text{Evt}_2, \ldots\}
\end{verbatim}

Each constructor statement has four parts: name of the target method (\textit{mtd\_name}), a comma separated list of the method’s input arguments (\textit{in}_1, \textit{in}_2, \ldots), a comma separated list of the method’s output arguments (\textit{out}_1, \textit{out}_2, \ldots) and a comma separated list of events placed between braces (\textit{Evt}_1, \textit{Evt}_2, \ldots). A constructor statement may or may not have input and/or output arguments.

4.2 Method Contract Generation

As mentioned before, to transform a group of events to a single method contract, constructor statements are used. Because Dafny uses Hoare logic [Hoa69] as the basis for verification, each defined constructor statement gives rise to generation of a Hoare triple. Pre- and post-conditions of each Hoare-triple are generated from listed Event-B events in the constructor statement. Each generated Hoare triple then is translated to Dafny method contracts. Any implementation that satisfies the generated contracts would be considered as a correct implementation of the Event-B model.

Assume there is a model with \(n\) events, a set of variables \(v\), invariants \(I(v)\) and a constructor statement as follows:

\begin{verbatim}
\text{Evt}_1 \triangleq \text{any } x_1 \text{ where } P_1(x_1, v) \text{ then } v := E_1(x_1, v) \text{ end}
\end{verbatim}
As mentioned earlier, one of the purposes of having a constructor statement is to assign a role to each event parameter. For the purpose of generating contracts, event parameters can be categorised with regards to the constructor statement that the event is listed in as follows:

- **Input parameter** (x): the parameter has input behaviour (receives a value from the environment of the machine) and is listed as an input parameter in the constructor statement
- **Output parameter** (y): the parameter has output behaviour (returns a value to the environment of the machine) and is listed as an output parameter in the constructor statement
- **Internal parameter** (z): the parameter is a local variable to the event and is not listed as input/output parameter in the constructor statement

All input and output parameters that are listed in a constructor statement must exist in all listed events. They are used as a method’s input or return arguments. Parameters that are not listed in the constructor statement are treated as internal parameters. Internal parameters are local variables to events. It is explained later how internal parameters are dealt with. Events $Evt_1...Evt_n$ can be represented based on constructor statement 1 as follows:

$$Evt_1 \triangleq \text{any } x,y, z_1 \text{ where } P_1(x,y, z_1,v) \text{ then } v := E_1(x,y, z_1,v) \text{ end}$$

$$\vdots$$

$$Evt_n \triangleq \text{any } x,y, z_n \text{ where } P_n(x,y, z_n,v) \text{ then } v := E_n(x,y, z_n,v) \text{ end}$$

In the above events, the union of $x, y, \text{ and } z$ is equal to the set of all parameters of the respective event.

A number of pre-conditions may be defined for each method in Dafny to specify the conditions which must be true before a method is called. Pre-conditions are generated from invariants and some of the event guards. As was mentioned previously, conjunction of all model invariants are translated to a Dafny predicate. This predicate should be a pre-condition for all generated Dafny methods. The reason for this is that from the Event-B model, it is expected that invariants are true before execution of each event therefore it can be expected that invariants are true before execution of each method as well.

Event guards are used in both pre- and post-conditions depending on the role that they play in the event. Guards of each listed event in a constructor statement can be categorised as follows:

- **Typing guard** (GT): a guard that declares the type of an event’s input or output parameters
- **Method guard** (GP): a guard that is being shared between all listed events in a constructor statement and only refers to input parameter and variables and not to output or internal parameters and is not a typing guard
- **Output guard** (GO): a guard that determines the value of an output parameter
• **Internal guard (GI):** a guard that refers to internal parameters and is not an output guard

• **Case guards (GC):** a guard that makes the enabling condition of its respective listed event distinct from other listed events and only refers to input parameters and machine variables

Each guard can only fall into one of the above categories. For generating pre-conditions from event guards we only consider method guards. If there is only one listed event then there would not be any case guards. With regards to the above categorisation, listed events in constructor statement 1 can be represented as follows:

\[
\text{Evt}_1 \triangleq \text{any } x, y, z_1 \\
\text{where} \\
\text{GT}(x) \\
\text{GT}(y) \\
\text{GP}(x, v) \\
\text{GC}_1(x, y) \\
\text{GI}_1(x, z_1, v) \\
\text{GO}_1(x, y, z_1, v) \\
\text{then} \\
v := \text{E}_1(x, y, z_1, v) \\
\text{end}
\]

\[
\text{Evt}_n \triangleq \text{any } x, y, z_n \\
\text{where} \\
\text{GT}(x) \\
\text{GT}(y) \\
\text{GP}(x, v) \\
\text{GC}_n(x, y) \\
\text{GI}_n(x, z_n, v) \\
\text{GO}_n(x, y, z_n, v) \\
\text{then} \\
v := \text{E}_n(x, y, z_n, v) \\
\text{end}
\]

To form method pre-conditions based on a constructor statement, model invariants \((I)\), method guards \((GP)\), and typing guards \((GT)\) of input parameters are used. From Event-B model we expect that all invariants are true before execution of each event hence invariants are pre-conditions of the method. Method guards are conditions that are shared by all listed events and they must hold before execution of each of the listed events thus method guards are also pre-conditions of the method. Typing guards of input parameters are also a pre-condition of the method to guarantee that the input value is a valid one. Based on this, the following predicate is the pre-condition for the method that is generated based on the given constructor statement (1):

\[I(v) \land GT(x) \land GP(x, v)\]  

(2)

For generating post-conditions from events we use case guards, internal guards, output guards and before-after predicates of event actions. A before-after predicate denotes the relation that exists between the value of a variable just before and just after the execution of an action. As mentioned before, a case guard makes the enabling condition of its respective listed event distinct from the other listed events. Case guards are used to determine which case is enabled at each time and therefore what is the expected outcome of the method. Internal parameters are used to determine the outcome of an event. An output parameter is treated as a free variable whose value is determined by the body of the method in Dafny. The value of an output parameter in Event-B is determined by output guards. Before-after predicate of actions specify the value of variables after the execution of an event. Due to this, internal guards, output guards and before-after predicates of each listed event are used to form post-conditions of the method. Therefore,
each listed event in constructor statement 1 gives rise to generation of a predicate as follows where $i \in 1..n$:

$$GC_i(x,v) \implies GT(y) \land (\exists z_i. GI_i(x,z_i,v) \land GO_i(x,y,z_i,v) \land v' = E_i(x,y,z_i,v))$$  \hspace{1cm} (3)

By convention, all primed variables appearing in a before-after predicate refer to the value of the variables after execution of an event and all unprimed variables refer to the value of the variable before the execution. It was discussed before that input and output parameters are treated as constants and free variables, respectively. To determine the value of internal parameters we existentially quantify over them. As mentioned before, if there exists only one listed event in a constructor statement then there is no case guard. In this case, only one post-condition will be generated in the following form:

$$GT_{out} \land (\exists z \cdot GI \land GO \land v' = E)$$  \hspace{1cm} (4)

Given the constructor statement 1, the following Hoare triple is generated:

$$\{ I \land GT_{in} \land GP \} \implies \{ GT_{out} \land (GC_1 \implies (\exists z_1. GI_1 \land GO_1 \land v = E_1)) \land \cdots \land (GC_n \implies (\exists z_n. GI_n \land GO_n \land v = E_n)) \}$$  \hspace{1cm} (5)

where $impl$ is a placeholder for the (yet to be constructed) correct implementation of the method. The above Hoare triple can be translated to annotations of a Dafny method which is generated based on constructor statement 1:

```drafny
method Evt (x : T) returns (y : R)
requires Invariants()
requires GP
ensures GC_1 == > \exists z_1 :: GI_1 && GO_1 && v == E_1
: ensurses GC_n == > \exists z_n :: GI_n && GO_n && v == E_n
```

Note that all non-typing and non-gluing invariants are translated to $Invariants$ predicate in Dafny. Preservation of typing invariants and guards are checked by the Dafny type system, hence, they are implicitly part of the translated annotations. $T$ and $R$ in the above method declaration are the type of input and output parameters and are determined by typing guards.

### 4.3 New Proof Obligations

To ensure that the translation is sound and the generated code contracts are implementable a number of proof obligations should be discharged. These proof obligations are discussed in this subsection.
4.3.1 Internal and Output Parameter Feasibility

We should make sure that the specification of an internal or an output parameter is feasible by showing that there exists a value that satisfies internal and output guards:

\[ I, GT, GP, GC_i \vdash \exists y, z_i. GI_i \land GO \]

We split the above sequent to have two separate proof obligations for internal parameter feasibility and output feasibility. For internal parameter feasibility, if there are \( n \) listed events in a constructor statement and \( I \) is the conjunction of typing and model invariants and \( GC_i \) and \( GI_i \) where \( i \in 1..n \) are the conjunction of case guards and the conjunction of internal guards of \( i \)th listed event respectively, then we can generate \( n \) proof obligations with the following form:

\[ I, GT, GP, GC_i \vdash \exists z_i. GI_i \]

Similar to internal parameter feasibility proof obligation, if there are \( n \) listed events in a constructor statement and \( I \) is the conjunction of typing and model invariants and \( GC_i \), \( GI_i \) and \( GO_i \) where \( i \in 1..n \) are the conjunction of the case guards, the conjunction of the internal guards and the conjunction of the output guards of \( i \)th listed event respectively, then we can generate \( n \) proof obligations with the following form:

\[ I, GT, GP, GC_i, GI_i \vdash \exists y. GO_i \]

4.3.2 Disjointness

As explained in previous sections on generating post-conditions, if there is more than one listed event in a constructor statement then for each listed event (case) a predicate that specifies the behaviour of that event is generated and the conjunction of all the generated predicates would form the post-condition of the generated method. If there are situations where more than one of the cases are available then the generated Dafny specification would not be implementable. To avoid this, the specifier must make sure that case guards of all listed events are disjoint.

In principle, we could deal with non-disjoint events. But that would mean that the post-conditions corresponding to the separate events would need to be combined through disjunction. For pragmatic reasons we prefer to generate a separate Dafny post-condition for each listed event in a constructor statement. Since separate post-conditions are implicitly conjoined, the case guards need to be disjoint. This means that we remove any non-determinism arising from overlapping event guards prior to translation to Dafny contracts.

To prove the disjointness of the case guards a number of proof obligations must be discharged. If there are \( n \) events listed in the constructor statement and \( I \) is the conjunction of typing and model invariants and \( GC_i \) where \( i \in 1..n \) is the conjunction of all case guards of \( i \)th event then \( n \) sequent can be generated with the following form:

\[ I, GT, GP, GC_i \vdash \neg GC_1 \land ... \land \neg GC_{i-1} \land \neg GC_{i+1} \land ... \land \neg GC_n \]

The number of proof obligations can be reduced by simplifying the above sequent:

\[ I, GT, GP, GC_i \vdash \neg GC_{i+1} \land ... \land \neg GC_n \]
4.3.3 Completeness

As explained in previous sections, when there is more than one event listed in a constructor statement, case guards are one of the forming components of each generated post-condition. There might be situations in which the generated post-conditions do not specify the intended behaviour of the method for all possible values specified by the method’s pre-conditions. This problem can be referred to as an incompleteness issue i.e. the specification is not complete. If there are \( n \) events listed in the constructor statement and \( I \) is the conjunction of typing and model invariants and \( GC_i \) where \( i \in 1..n \) is the conjunction of all case guards of \( i \)th event and \( GP \) is the conjunction of method guards, to avoid incompleteness issue, the following sequent should be proved:

\[
I, GT, GP \vdash GC_1 \lor \ldots \lor GC_n
\]

Completeness is a desirable but not a must-have property and can be ignored by the modeller. If a method is executed in a state that satisfies the pre-condition but is not covered by any of the cases (i.e., all post-conditions are trivially satisfied), then any outcome for the method is allowed.

4.4 Invariant Preservation Proof

The validity of our transformation scheme is based on the fact that invariants of the Event-B model are also invariants of any Dafny implementation that satisfies the generated contract. Here we outline the proof of this for the case where a method contract is defined by one event. The proof easily generalises to multiple events as the cases are separate.

Assume we have a model \( M \) with variable \( v \) and invariant \( I(v) \) and event \( Evt \):

\[
Evt \triangleq \text{any } x \text{ where } P(x,v) \text{ then } v := E(x,v) \text{ end}
\]

Consider a method specified from an Event-B machine as follows:

\[
\text{method } Evt(x) \text{ returns } () \{ Evt \}
\]

The following Hoare triple characterises the correctness of the implementation of the contract generated from this method specification:

\[
\{ I(v) \land P(x,v) \} \implies \{ v = E(x,old(v)) \}
\]

(6)

where \( old(v) \) refers to the value of variable \( v \) before execution of \( impl \). We have the following rule with regards to the operator \( old \) [PM99]:

\[
\frac{\{ P(v) \} C \{ \top \}}{\{ P(v) \} C \{ P(old(v)) \}}
\]

(7)

This rule says that if the pre-conditions of the triple hold for the value of the variable \( v \) before the execution, then they still hold for the old value of the variable \( v \) after the execution. By applying these new rules to Hoare triple 6, it can be rewritten as follows:

\[
\{ I(v) \land P(x,v) \} \implies \{ I(old(v)) \land P(x,old(v)) \land v = E(x,old(v)) \}
\]

(8)
The invariant preservation proof within Event-B guarantees the following:

\[ I(\text{old}(v)) \land P(x,\text{old}(v)) \land v = E(x,\text{old}(v)) \implies I(v) \quad (9) \]

Finally, based on 8 and 9, we have that \textit{impl} preserves the invariants:

\[ \{I(v) \land P(x,v)\} \implies \{I(v)\} \quad (10) \]

5 Example: Map Abstract Data Type

Our method and tool for transforming Event-B models to Dafny code contracts have been validated through a number of case studies including a map, a stack, and a queue abstract data type. Due to space limitation, we only present the map ADT case study in this paper.

A map (also called associative array) is an abstract data type which associates a collection of unique keys to a collection of values. The abstract level of the map which is shown Figure 1 is modelled as a partial function which links a key to a value. Types KEYS and VALUES are defined in the context as sets. The variable \textit{map} is the only variable in this level and initialised with empty.

The abstract model (machine \textit{m0}) is refined by machine \textit{m1} (Figure 2). Event \textit{Add} is refined by two events \textit{Add1} and \textit{Add2} to deal with two different cases. Event \textit{Add1} will prepend a new key \( k \) to the sequence \textit{keys} and value \( v \) to the sequence \textit{values} . Event \textit{Add2} modifies the value associated with an existing key \( k \), in sequence of values. Event \textit{Remove} is refined in this level to be able to remove an existing key \( k \) and its associated value from both the sequences. Now, machine \textit{m1} has only those Event-B constructs that have a Dafny counterpart, hence it can be transformed to Dafny contracts. To do this, two constructor statements are provided:

\begin{verbatim}
method Add(k : KEYS, v: VALUES) returns()
requires Invariants()
ensures k !in keys ==> keys == [k] + old(keys) && values == [v] + old(values)
requires k in keys ==> \exists i :: i in (set k0 | 0\leq k0 \&\& k0\leq old(keys) - 1)
  && old(keys)[i] == k
  && values == old(values)[i:=v] && keys == old(keys)
endmethod

method Remove(k : KEYS) returns()
requires Invariants()
requires k in keys
ensures \exists i :: i in (set k0 | 0\leq k0 \&\& k0\leq old(keys) - 1)
  && old(keys)[i] == k && keys == old(keys)[..i] + old(keys)[i + 1..]
  && values == old(values)[..i] + old(values)[i + 1..]
endmethod
\end{verbatim}
We have constructed Dafny implementations of the methods by hand and used the Dafny verifier to verify these against the generated contracts.

When the abstract model of the map is refined, the correctness of the refined event `Add2` can be proved without guard `grd3`. After the introduction of the method `Add` constructor statement, a number of proof obligations are generated. As discussed in 4.3, one of the proof obligations that should be discharged before the transformation takes place is the internal parameter feasibility PO. Internal parameter feasibility for event `Add2` without `grd3` has the following form:

\[ I(v), k \in \text{KEYS} \land v \in \text{VALUES} \vdash \exists i. i \in 0..\text{seqSize(keys)}-1 \land \text{seqElemAccess(keys,i)} = k \]

\[ I(v) \] denotes the model invariants. This proof obligation is not provable. A counter-example for this PO is a key that is not already in the sequence of the keys. To be able to prove this PO, `grd3` should be introduced. The new PO is as follows and can be discharged trivially:

\[ I(v), k \in \text{KEYS} \land v \in \text{VALUES}, k \in \text{ran(keys)} \vdash \exists i. i \in 0..\text{seqSize(keys)}-1 \land \text{seqElemAccess(keys,i)} = k \]
6 Tool Support

A Rodin plug-in has been developed to facilitate the automatic transformation of Event-B models to annotated Dafny method declarations. The plug-in extends Event-B with a new machine element for storing constructor statements. Whenever a new constructor statement is added to a machine, the plug-in will generate a number of new proof obligations based on the constructor statement (See 4.3). After discharging all PO’s in Rodin the user can invoke the contract generator to generate a Dafny class including variable declarations, predicate *Invariants* and annotated Dafny method declarations. The tool has been validated by being applied to a number of small case studies.

7 Related and Future Work

As far as we are aware no attempt has been made to generate annotated Dafny programs from Event-B models and there is little research on linking the constructive approach to the analytical approach in the literature.

EventB2Dafny [CLR12] is a Rodin plug-in for translating Event-B proof obligations to Dafny code to use Dafny verifier as an external theorem prover for proving Event-B proof obligations. A Rodin plug-in called EventB2JML [RC14] has been developed to translate Event-B models to Java JML-specified code. EventB2JML implement Event-B models by producing a Java thread implementation for each event and does not impose any control flow on events. Also at the code level invariants need to be verified again using an static verifier. Mery and Monahan in [But09] proposed a transformation technique from an Event-B specification to an executable algorithm. In their approach the specification of the algorithm is provided at the start of the development in form of Spec# [BLS05] pre- and post-conditions and the algorithm is modelled in Event-B with regards to those code contracts. At the end the generated code from the Event-B model is verified against the code contracts in Spec#. Tasking Event-B [EB11] is a code generator that generates code from Event-B models to a target language but it does not support verification of the generated code.

Our current transformation rules allow us only to transform Event-B models of abstract data types to Dafny contracts. In the future we want to be able to transform Event-B model of more complex algorithms to Dafny contracts. One possible way is to use Event Refinement Structure (ERS) [FBR14]. By using ERS we will be able to impose algorithmic structures at Event-B level. This will ease the contract generation for more complex structures like loops.

8 Conclusion

We have presented a tool supported method for transforming Event-B models to Dafny code contracts. Using this method, Dafny users will enjoy the abstraction and refinement power of Event-B for building specifications that are correct with regards to an abstract specification. This approach provides a framework in which Event-B models can be implemented correctly in a sequential programming language. Our method also provides a way for merging Event-B events in order to generate contracts for a single method which implements different cases. We
have also proved that if generated contracts are satisfied by an implementation in Dafny then it also satisfies the invariants of the abstract model and there is no need to reprove invariant preservation in the Dafny level. A tool in the form of a Rodin plug-in has been developed in order to implement the link. Given a machine and a number of constructor statements, the tool automatically generates relevant code contracts. A number of extra proof obligations (discussed in 4.3) should be discharged in order to guarantee the soundness of the generated contracts.

Acknowledgements: This work was funded in part by a Microsoft Research 2014 Software Engineering Innovation Foundation Award.

Bibliography


http://eprints.soton.ac.uk/349769/

http://eprints.soton.ac.uk/373440/
<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Journal/Publication</th>
<th>Page Numbers</th>
</tr>
</thead>
</table>
Loop Patterns in C Programs
Thomas Pani, Helmut Veith, Florian Zuleger

Vienna University of Technology

Abstract: In this work, we conduct a systematic study of loops in C programs. We describe static analyses capable of efficiently identifying definite iteration in C code. Our experiments show that over one third of loops in our benchmarks take this form. To cover further loops, we systematically weaken our definition of definite iteration and derive a family of loop classes that are heuristics for definite iteration. We then measure the occurrence of these classes on real-world C code and investigate which statements are used to express them. Finally, we empirically show that our classification is meaningful – (a) it describes the majority of loops in our benchmarks, (b) the classes are good heuristics for termination, and (c) they can be used as software metrics to characterize benchmarks for software verification.

Keywords: Loops, loop patterns, structured programming, definite iteration, software metrics, program features.

1 Introduction

Historically, some programming languages provide restricted loop statements, such as the Fortran do statement or the Ada for statement. Such statements express definite iteration, i.e., structured iteration over the elements of a finite set, such as an integer sequence or the elements of a data structure [Sta95]. Other languages – like C – do not provide such constructs, and all language-provided loop statements have the full expressive power inherent to Turing-complete languages. Lately, object-oriented languages implement definite iteration via the iterator pattern, often supported by syntactic sugar (e.g., foreach-like statements in Java, C#, C++11, Python). However, we are not aware of a recent study determining if and how often this form of restricted iteration occurs in practice. This is the gap we intend to close.

In this paper, we study “typical” iteration patterns in real-world C code. Such a study is of interest because it allows us – for example – to answer the following questions: How do loops in source code usually “look like” (what are their properties)? How often do different kinds of loops appear in program source code? How difficult are these different kinds of loops to analyze for an automated procedure? In spirit of Dijkstra’s famous Go-to statement considered harmful [Dij68]: Can practical examples of iteration be expressed using a more well-behaved, structured construct? As we target the C language, which does put any restriction on looping constructs, does it make sense to introduce restricted loop statements like in Fortran or Ada?

To our knowledge, no up-to-date systematic study of definite iteration in real-world code exists. [Sta93] does a manual, ad-hoc classification of iteration over high-level, abstract structures.
such as sets, sequences or finite mappings. Our work takes an algorithmic approach, yielding a concise definition and automated classification. [RAPG14] presents a simple, powerful heuristic similar to our work for computing loop trip counts. In contrast, we base our study on a sound analysis, and present extensive experimental results.

We thus aim to study the occurrence of definite iteration in real-world C code. However, there is no dedicated, well-defined construct supporting definite iteration in the C language. We thus describe a pattern-based, lightweight static analysis to identify such loops (Section 2), which we call **FOR loops** \( L_{\text{FOR}} \): We define three easily verifiable restrictions on loops which capture the structured nature of definite iteration. As our experiments (Section 5) show, this allows us to identify about one third of the loops in our benchmarks as FOR loops – a major portion, but not the majority of loops in our benchmarks. We conjecture that programmers enjoy a bit more flexibility than what is provided by our definition of FOR loops.

In an attempt to understand further loop patterns, we extend our analysis to describe loops similar to, but not identified as FOR loops. To do so, we derive the family of **generalized FOR loops** \( L_{\text{FOR}}(...) \) from the definition of \( L_{\text{FOR}} \) by systematically weakening the three restrictions (Section 3). This lets us describe up to 82% of the loops in our benchmarks (Section 5). Next, it is natural to look at statement usage: The C language provides four statements of equal expressive power capable of implementing iteration (while, do-while, for, goto). We thus measure their use for each of the generalized FOR loop classes (Section 5.1).

Finally, we conduct two experiments to show that the defined loop classes are meaningful: We compare the loops matching our loop classes with the state-of-the-art bound analysis tool LOPUS (Section 6.1). LOPUS tries to statically determine a symbolic bound on the number of times a loop is executed. As the number of iterations of a FOR loops is predetermined, loop classes should align with results from bound analysis. We also sketch how to derive software metrics from our loop classification, and that they describe properties interesting for program analysis by applying them in a machine-learning portfolio for software verification (Section 6.2).

Summarizing, our work conducts a systematic study of loops in C programs, making the following contributions:

1. We give a definition of definite iteration, **FOR loops**, for the C programming language, which does not have dedicated support for this concept (Section 2).

2. We define the family of **generalized FOR loops**, which capture some aspects of definite iteration and allow us to describe a majority of loops in our benchmarks (Section 3).

3. We study the occurrence of these loop classes on benchmarks taken from different problem domains (Section 5) by measuring the occurrence of FOR loops and generalized FOR loops, and study which C statements are used to express them (Section 5.1).

4. We give empirical evidence for the usefulness of our loop classes (Section 6): (a) We show that the generalized FOR loop classes capture the difficulty of automated program analysis (Section 6.1). (b) We sketch how to use the loop classes as software metrics, and how to apply them in a machine-learning portfolio for software verification (Section 6.2, [DPVZ15]).
2 FOR Loops

In this section, we introduce a first loop pattern, \textit{FOR loops} $L^{\text{FOR}}$, expressing definite iteration. One central implication of FOR loops is that once started executing, they always terminate. We thus characterize FOR loops $L^{\text{FOR}}$ by giving an efficient, syntactic pattern-based termination proof for its members. This termination procedure exploits that in many cases, local reasoning is powerful enough to decide termination of loops expressing definite iteration.

2.1 Motivation

Our syntactic termination proof is based on the observation that FOR loops usually alter a limited set of variables during each iteration. These variables are then compared against a fixed (loop-invariant) bound to decide termination.

\textbf{Example.} Consider the program on the right: We can show the loop to terminate in a straight-forward manner: the value of $i$ is changed by the loop, while the value of $N$ is fixed. The loop’s condition induces a predicate $P(i) : i < N$. If $P(i)$ evaluates to false, the loop terminates. We show that executing the loop, $P(i)$ eventually evaluates to false: The domain of $P$ can be partitioned into two intervals $[-\infty, N)$ and $[N, \infty]$, for which $P(i)$ evaluates to true or false, respectively ($q$ in Figure 1). As $i$ is (in total) incremented during each iteration, we eventually have $i \in [N, \infty]$, and thus $\neg P(i)$ and the loop terminates.

2.2 Program Model

We base our analysis on the program’s \textit{labeled transition system}:

\textbf{Definition 1} A labeled transition system (LTS) is a digraph $T = (\text{Loc}, \text{Labels}, \text{Edges}, l_0)$ constructed from the program’s source code, where Loc is the finite set of program locations, Edges $\subseteq$ Loc $\times$ Labels $\times$ Loc is the transition relation, and $l_0 \in$ Loc is the singleton initial state. Labels is the set of edge labels, consisting of the program’s statements, and expressions $\text{assume}(a, b)$ corresponding to branches. If a node has a singleton successor, the edge is labeled with the corresponding program statement. Branching is modeled by two successors, where the edge labels $P : \text{assume}(a, b)$ and $\neg P : \text{assume}(a, c)$ are the predicates guarding control flow from program location $a$ to $b$ and $c$, respectively.

2.3 Determining membership in $L^{\text{FOR}}$

To compute the syntactic termination proof for a given loop $L$, we proceed in the three steps described below: First, we consider predicates $\text{assume}(a, b)$ on edges $a \in L, b \notin L$ leaving the loop. We try to show that the predicate eventually (during program execution) becomes true, meaning the edge becomes executable (Section 2.3.1). Second, we check that other variables $v \neq i$ occurring in the predicate are loop-invariant (Section 2.3.2). Finally, we impose a control flow constraint to make our termination proof sound (Section 2.3.3).
Loop Patterns in C Programs

2.3.1 Terminating Predicates

To find predicates guarding loop termination, we consider control flow edges leaving the loop:

Definition 2 (Exit node, exit predicate.) Let $T = (\text{Loc}, \text{Labels}, \text{Edges}, l_0)$ be a loop’s LTS, $(a, b)$ such that $a \in \text{Loc}, b \notin \text{Loc}$ an edge leaving the loop, and $P : \text{assume}(a, b)$ the edge’s label. We call $a$ an exit node and $P$ an exit predicate.

We introduce a predicate’s representing function as a means to define necessary characteristics for our termination proof:

Definition 3 (Representing function, monotonicity, eventually true predicates.) The representing function $f_P$ of a predicate $P$ with the same domain takes, for each domain value, the value 0 if the predicate holds, and 1 if the predicate evaluates to false, i.e. $P(X) \iff f_P(X) = 0$. A predicate $P$ is monotonically increasing (decreasing) if its representing function $f_P$ is monotonically increasing (decreasing). A predicate $P$ is eventually true if its representing function $f_P$ is eventually 0, i.e. if there exists an $x$ s.t. $f_P(x) = 0$. These definitions are inspired by [KW11].

Depending on the predicate’s syntactic form, we describe four strategies for showing that the predicate is eventually true. These strategies were chosen to represent cases that in our experience frequently occur in practice. We illustrate the strategies in Figure 1 and describe them in Table 1. Based on the strategies from Table 1, we call predicates whose representing function takes such a form well-formed:

<table>
<thead>
<tr>
<th>Name</th>
<th>Informal motivation</th>
<th>Formal definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Escape</td>
<td>$i$ “escapes” $\text{Exp}$, i.e. $P(i)$ iff $i \neq \text{Exp}$. Example: $i \neq N$</td>
<td>$f_P$ has exactly one non-root, i.e. there exists exactly one $x$ s.t. $f_P(x) = 1$.</td>
</tr>
<tr>
<td>Search</td>
<td>$i$ “searches” and eventually “finds” $\text{Exp}$, i.e. $P(i)$ iff $i = \text{Exp}$. Example: $i = N$</td>
<td>$f_P$ has a single root, i.e. there exists exactly one $x$ s.t. $f_P(x) = 0$.</td>
</tr>
<tr>
<td>Increase</td>
<td>$i$ increases enough to enter interval $[\text{Exp}, \infty]$, i.e. $P(i)$ iff $i \in [\text{Exp}, \infty]$. Example $i \geq N$</td>
<td>$f_P$ is monotonically decreasing and eventually 0.</td>
</tr>
<tr>
<td>Decrease</td>
<td>$i$ decreases enough to enter interval $[-\infty, \text{Exp}]$, i.e. $P(i)$ iff $i \in [-\infty, \text{Exp}]$. Example $i \leq N$</td>
<td>$f_P$ is monotonically increasing and eventually 0 ($f_P$ grows from 0).</td>
</tr>
</tbody>
</table>

Table 1: Strategies for proving termination based on the representing function $f_P$ of an exit predicate $P$. We assume expression $\text{Exp}$ to be loop-invariant.

Figure 1: Monotonic ($p, q$) and eventually true ($p, q, r, s$) predicates.
Definition 4 (Well-formed exit predicate.) Let $L$ be a loop, $(a, b)$ an edge leaving $L$, and $P : assume(a, b)$ the edge’s label. $P$ is a well-formed exit predicate if and only if its representing function $f_P$ matches one of the forms in Table 1.

Given a predicate’s representing function $f_P(i)$, we first determine the appropriate strategy in Table 1. We then consider updates on $i$ in loop $L$ to decide if the strategy’s condition holds: For each variable $i$ occurring in an exit predicate, we compute the set of possible updates to $i$ in a single iteration of the loop and obtain the accumulated increment $AccIncs_L(i)$. We do so by folding constant increments/decrements of $i$ along a path of the loop into a single, constant value. At nodes where branches join, we build the union of the accumulated increment along each path. In case of non-constant updates, or a nested loop that updates $i$, we cannot precisely determine the set and $AccIncs_L(i) = \mathbb{Z}$. We use the accumulated increment $AccIncs_L(i)$ to give sufficient conditions for an exit predicate $P(i)$’s eventual truth for a given loop $L$ and variable $i$ in Table 2. Intuitively, the conditions all describe scenarios under which the representing function $f_P$ eventually takes the value 0:

- For escape, either $f_P(i)$ is already 0, or any non-zero increment makes it 0 in the next iteration. The condition ensures such a non-zero increment exists along all paths of the loop.

- For search, either $f_P(i)$ already 0, or $i$ eventually takes all values in its type’s range. The condition ensures all increments are integer successor functions in a single direction. Assuming (cf. Section 4.1) two’s complement integer wrap-around on overflow, $i$ steps through all values in its type’s range.

- For increase (decrease), either $f_P(i)$ already 0, or $i$ is incremented (decremented) on all paths. The condition ensures all updates to $i$ along all paths are non-zero increments (decrements).

Definition 5 (Terminating node, terminating predicate.) Let $P : assume(a, b)$ be an exit predicate for which the condition given by Tables 1 and 2 holds. We call $a$ a terminating node and $P$ a terminating predicate.

Example. For our example in Listing 1, we have $AccIncs_L(i) = \{2 - 1, 3 - 1\} = \{1, 2\}$ and identify an exit predicate $P(i) : i \geq N$. Matching $P(i)$ against Tables 1 to 2 we can see that we need to apply strategy “Increase”, i.e. check whether all elements of $AccIncs_L(i)$ are positive. Clearly this is the case, and we proceed to check two further constraints.

2.3.2 Invariant Constraint

Our syntactic termination proof only considers predicates $P(i)$ in a single variable $i$. Other subexpressions are checked to be loop-invariant by verifying that none of the referenced variables are updated inside the loop.
Example. In our example (Listing 1), \( N \) is never updated inside the loop. The constraint is satisfied and we proceed to check the last constraint.

2.3.3 Control Flow Constraint

Finally, we need to make sure that a terminating predicate \( P : \text{assume}(a, b) \) is actually evaluated (i.e. node \( a \) is reached) when it evaluates to true. Listing 2 illustrates that due to our local reasoning this is not always the case: while there is a terminating predicate \( P : i > N \), it may never be evaluated (e.g. if \( \text{decide}(i) \) always returns false).

As determining feasibility of reaching \( a \) is in itself a hard problem, we restrict our analysis to a case in which we can ensure soundness: \( a \) has to lie on each path through the loop.

Example. In our example (Listing 1), the loop condition (and hence the exit predicate \( P(i) \)) is evaluated in each iteration of the loop. Thus we classify the loop as FOR loop \( L \in L_{\text{FOR}} \).

Definition 6 (FOR loop.) Given a loop \( L \) and an exit predicate \( P(i) : \text{assume}(a, b) \), we call \( L \) a \( \text{FOR loop} \) \( L \in L_{\text{FOR}} \) if and only if the following conditions hold:

- \( S_1 \): Predicate constraint. \( P(i) \) is a terminating predicate, i.e. \( \text{AccIncs}_L(i) \) implies eventual truth of \( P(i) \) (Section 2.3.1).
- \( S_2 \): Invariant constraint. We only consider predicates \( P(i) \) in a single variable \( i \). All other subexpressions are loop-invariant (Section 2.3.2).
- \( S_3 \): Control flow constraint. \( a \) lies on each path through the loop (Section 2.3.3).

2.4 Strengthening Syntactic Termination

So far, we have only considered an isolated notion of loop termination: If execution starts from \( l_0 \), any path of execution leaves the loop. A stronger notion considers a loop to be bounded if and only if the number of executions of the loop’s paths is bounded. The example below shows where the two notions differ:

```c
while (1) {
    for (unsigned i = 0; i < 42; i++) {} }
```

While the nested loop itself terminates whenever executed, the number of executions of the nested loop is infinite. We can strengthen the notion of syntactic termination to accommodate this property:

Definition 7 (Syntactically bounded loop.) Given a loop \( L \in L_{\text{FOR}} \), we call \( L \) syntactically bounded \( L \in L_{\text{SB}} \) if and only if \( L \) itself and all of its nesting (outer) loops are in \( L_{\text{FOR}} \).

3 Generalized FOR Loops

In this section, we aim for a classification of the remaining loops, i.e. those loops not identified as FOR loops. We develop heuristic loop classes based on \( L_{\text{FOR}} \), by systematically weakening \( L_{\text{FOR}} \) criteria along three dimensions corresponding to the three FOR loop constraints \( S_1-S_3 \).
The heuristics are designed to leave enough leeway to match a considerable amount of loops, but still capture some of the termination-related properties of FOR loops. We call this family of loop classes generalized FOR loops \( \mathcal{L}_{\text{FOR}} \).

### 3.1 Dimensions of Generalized FOR Loops

#### 3.1.1 Predicate Constraint

Depending on the chosen abstract domain and abstract semantics, the computed set \( \text{AccInc}_L(i) \) (cf. Section 2.3.1) may be too coarse to show termination. Thus we aim to cover cases where symbolic updates take sensible values, but our local, path-insensitive analysis cannot establish them. Our heuristic decouples the termination property from predicates by not requiring the accumulated increment to imply eventual truth of the predicate: We alter the constraint \( S_1 \) (Definition 6), which only considers terminating predicates, to heuristic \( W_1 \), which is content with well-formed predicates (Definition 4).

**Example.** In Listing 2, we cannot determine which values \( \text{update}(i) \) returns. Thus we do not classify the loop under constraint \( S_1 \). For constraint \( W_1 \), we only consider the syntactic form of \( P(i) : i > N \) and do not check the constraint from Table 2. In our example, pattern matching against (Table 1) succeeds, and \( P(i) \) fulfills \( W_1 \). **Intuition:** We assume \( \text{update}(i) \) returns sensible values.

#### 3.1.2 Invariant Constraint

Our syntactic termination proof only considers predicates \( P(i) \) in a single variable \( i \). Other subexpressions are checked to be loop-invariant (constraint \( S_2 \), Definition 6). For our heuristic \( W_2 \), we omit this check. Intuitively, we allow predicates in multiple variables and assume that all variables \( v \neq i \) are updated in a way that does not interfere with termination.

**Example.** In Listing 2, \( N \) is not loop-invariant, i.e. we have a predicate in two variables \( P(i,N) \). As \( N \) is updated in the loop, we do not classify the loop under constraint \( S_2 \). Heuristic \( W_2 \) omits the loop-invariance check, and thus the loop fulfills heuristic \( W_2 \). **Intuition:** We assume updates to \( N \) take sensible values (in our example they do, as \( N \) is only decremented).

#### 3.1.3 Control Flow Constraint

Instead of requiring the terminating node \( a \) associated with a terminating predicate \( P : \text{assume}(a,b) \) to lie on each path through the loop (constraint \( S_3 \), Definition 6), heuristic \( W_3 \) only requires existence of some terminating node in the loop.
Example. Consider the code in Listing 2: Predicate $P(i) : i > N$ is nested inside of a conditional branch. As we cannot determine if $\text{decide}(i)$ ever evaluates to true when $i > N$, we do not classify the loop under constraint $S_3$. For heuristic $W_3$ we only require that there is a terminating node somewhere in the loop. Intuition: We assume the node is reached when its associated predicate $P(i)$ evaluates to true.

Definition 8 (Generalized FOR loop constraint, generalized FOR loop.) A generalized FOR loop constraint is a tuple $(P, I, C)$, where $P \in \{S_1, W_1\}$ is a predicate constraint, $I \in \{S_2, W_2\}$ is an invariance constraint, and $C \in \{S_3, W_3\}$ is a control flow constraint. A loop $L$ satisfies constraint $(P, I, C)$ if and only if all of $P, I, C$ are satisfied. $L$ is a generalized FOR loop $L \in \mathcal{L}^{\text{FOR}(P,I,C)}$ if and only if $L$ satisfies $(P, I, C)$.

For a visual interpretation, consider Figure 2a showing the three categories above as independent dimensions. Moving away from the center along dimensions $P, I, C$, we cover additional loops at the expense of losing soundness. Note that the strongest generalized FOR loop class $\mathcal{L}^{\text{FOR}(S_1S_2S_3)}$ is the class of FOR loops introduced in Section 3, i.e. $\mathcal{L}^{\text{FOR}(S_1S_2S_3)} = \mathcal{L}^{\text{FOR}}$. Also note that loop constraints are partially ordered, i.e. $C_1 \leq C_2$ if and only if $\mathcal{L}^{\text{FOR}(C_1)} \subseteq \mathcal{L}^{\text{FOR}(C_2)}$. The obtained partially ordered set is shown in Figure 2b.

4 Implementation

We implemented the analyses described above in our tool SLOOPY\(^1\). It is built on top of Clang, a C language frontend for the LLVM compiler infrastructure. The analysis proceeds as follows:

\(^1\) Available at [http://forsyte.tuwien.ac.at/~pani/sloopy/](http://forsyte.tuwien.ac.at/~pani/sloopy/).
1. We use Clang’s representation of the control flow graph to identify so-called natural loops. These are subgraphs whose edges form a cycle with exactly one entry point. As an implementation detail, we choose this definition of loops to simplify the definition and implementation of data-flow analysis. In general, irreducible flow graphs may contain loops with multiple entry points. However, a recent study [SW12] “found 5 irreducible functions in a total of 10427, giving a total average irreducibility for this set of current programs of 0.048%”. The authors conclude that irreducibility “is an extremely rare occurrence, and it will probably be even rarer in the future”.

2. Next, our tool attempts to find terminating predicates (Section 2.3.1). It computes the accumulated increment using data-flow analysis and the constant propagation framework [WZ85, ASU86] for all variables occurring in exit predicates. We rewrite each exit predicate into a normal form and perform syntactic pattern matching on it. This selects a strategy according to Table 1, which we check against the accumulated increment according to Table 2. At the moment, our analysis considers linear inequalities \( P(i) \) in a single variable \( i \), which allows us to handle condition expressions with common comparison operators of the C programming language (==, ! =, <, >, <=, >=) as top-level connective.

3. Finally, we enforce the invariance constraint (Section 2.3.2) – by checking for statements that update variables – and the control flow constraint (Section 2.3.3) – by assigning to each basic block the number of open (un-joined) branches using data-flow analysis. Any LTS node \( a \) of natural loop \( L \) with zero open branches lies on all paths through \( L \).

4.1 Restrictions and Assumptions of Our Implementation

Our analysis makes a number of assumptions to determine eventual truth of predicates:

1. Due to the locality of our analysis, we assume the absence of aliasing and impure functions.

2. The C standard leaves pointer arithmetic undefined if operands and/or the result don’t point into or just beyond the same array object [ISO, 6.5.6]. We assume the result of pointer arithmetic is always defined.

3. The C standard does not define overflow on signed integer types [ISO, 6.5.6]. The search strategy relies on covering the whole value range, thus we assume two’s complement wrap-around behavior on overflow for search predicates over signed integer types.

   We may also prove termination of increase/decrease predicates over unsigned integers under this assumption, e.g. in the loop for (unsigned \( i = 42; i < N; i-- \) ); If we prove increase/decrease predicates over signed integers using this assumption, we likely discovered a bug [DLRA12]. We actually encountered one such bug, which had gone undetected for sixteen years in GPL Ghostscript, during experimental evaluation.

4. For any strategy other than escape, we assume that \( i \) in an exit predicate \( P(i) : E(i) \circ N \), where \( E(i) \) is an expression in \( i \) and \( \circ \in \{<, \leq, \geq, >\} \), is not prevented by its (finite integer) type to enter the required interval to make \( P(i) \) evaluate to true. As both \( E(i) \) and \( N \) are expressions, we cannot determine their ranges by merely syntactic means.
Loop Patterns in C Programs

| Name                  | Description                                                                 | \( | L | \) |
|-----------------------|-----------------------------------------------------------------------------|-----|
| cBench [cBe]          | Open-source sequential programs used in program and compiler optimization.   | 4157|
| coreutils [cor]       | GNU Core Utilities, a collection of basic userland utilities.               | 1002|
| SPEC CPU2006 [Hen06]  | Compute-intensive benchmarks composed from real life applications code. We only consider benchmarks written in C. | 15043|
| Mälardalen WCET [GBEL10] | Used in Worst-Case Execution Time (WCET) analysis.                  | 262 |

Table 3: Benchmarks.

<table>
<thead>
<tr>
<th></th>
<th>( L_{SB} )</th>
<th>( L_{FOR}(S_1S_2S_3) )</th>
<th>( L_{FOR}(W_1W_2W_3) )</th>
<th>( L^{FOR} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>cBench</td>
<td>23.8%</td>
<td>37.0%</td>
<td>63.6%</td>
<td>37.2%</td>
</tr>
<tr>
<td>coreutils</td>
<td>24.3%</td>
<td>33.7%</td>
<td>48.5%</td>
<td>52.4%</td>
</tr>
<tr>
<td>SPEC</td>
<td>21.2%</td>
<td>39.3%</td>
<td>65.5%</td>
<td>35.7%</td>
</tr>
<tr>
<td>WCET</td>
<td>44.5%</td>
<td>55.0%</td>
<td>82.3%</td>
<td>18.9%</td>
</tr>
</tbody>
</table>

Figure 3: Percentage of loops covered by four selected loop classes.

5. For strict inequalities \( P(i) : i < N \) \((i > N)\), we assume \( N \) evaluates to less (more) than its type’s minimum (maximum) value. Otherwise, \( P \) never evaluates to true for any \( i \).

5 Experiments: Occurrence of Loop Patterns

In this section, we measure the occurrence of the various loop classes introduced above on four widely used benchmark suits from different domains. Table 3 summarizes these benchmarks.

Figure 3 and Table 4 show the percentage of loops in each of the listed classes for the respective benchmark: To keep the comparison working, in charts we only consider syntactically bounded loops \( L_{SB} \), the strongest generalized FOR loop class \( L_{FOR}(S_1S_2S_3) \) = \( L_{FOR} \), the weakest generalized FOR loop class \( L_{FOR}(W_1W_2W_3) \), all loops of the benchmark \( L \), and all loops not in any simple loop class \( L^{FOR} = L \setminus L_{FOR}(W_1W_2W_3) \). These loops are especially interesting, because those in \( L_{SB} \) can directly be compared with bound analysis tools, \( L_{FOR}(S_1S_2S_3) \) and \( L_{FOR}(W_1W_2W_3) \) represent the range of simple loops, and \( L^{FOR} \) represents loops most different from our description of definite iteration.

<table>
<thead>
<tr>
<th></th>
<th>SB</th>
<th>S_1S_2S_3</th>
<th>S_1S_2W_3</th>
<th>S_1W_2S_3</th>
<th>S_1W_2W_3</th>
<th>W_1S_2S_3</th>
<th>W_1S_2W_3</th>
<th>W_1W_2S_3</th>
<th>W_1W_2W_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>cBench</td>
<td>22.6</td>
<td>37.0</td>
<td>46.6</td>
<td>37.2</td>
<td>47.0</td>
<td>48.0</td>
<td>62.7</td>
<td>48.3</td>
<td>63.1</td>
</tr>
<tr>
<td>coreutils</td>
<td>23.6</td>
<td>33.7</td>
<td>40.7</td>
<td>33.7</td>
<td>40.8</td>
<td>37.5</td>
<td>48.3</td>
<td>37.6</td>
<td>48.5</td>
</tr>
<tr>
<td>SPEC</td>
<td>21.5</td>
<td>39.2</td>
<td>45.9</td>
<td>39.3</td>
<td>46.0</td>
<td>56.0</td>
<td>64.5</td>
<td>56.2</td>
<td>64.7</td>
</tr>
<tr>
<td>WCET</td>
<td>43.5</td>
<td>54.6</td>
<td>66.8</td>
<td>55.0</td>
<td>67.6</td>
<td>67.2</td>
<td>80.5</td>
<td>67.6</td>
<td>81.7</td>
</tr>
</tbody>
</table>

Table 4: Percentage of loops covered by \( L_{SB} \) and the generalized FOR loop classes \( L_{FOR}(\ldots) \).
Discussion. The percentage of loops in $\mathcal{L}^{SB}$ is around 23%, except for WCET where it is at 44%. Some 9–18% more loops are only in $\mathcal{L}^{FOR}$, which amounts to about 36% of all loops, except for WCET (55%). The weakest generalized FOR loop class $\mathcal{L}^{\text{FOR}(W_1W_2W_3)}$ additionally contains 26–27% more loops on all benchmarks except coreutils, comprising about 64% of all loops in cBench and SPEC, and 82% in WCET. For coreutils, the ratio of $\mathcal{L}^{\text{FOR}(W_1W_2W_3)}$ is at 48% of all loops.

The percentage of loops in $\mathcal{L}^{SB}$ and $\mathcal{L}^{FOR}$ is stable across all benchmarks except WCET. This can be explained by the fact that WCET stems from a narrow domain and only contains single-path programs, which naturally fulfill the strict control flow constraint of FOR loops. We have included it to showcase the difference to more general-purpose benchmarks.

The small difference between $\mathcal{L}^{\text{FOR}}$ and $\mathcal{L}^{\text{FOR}(W_1W_2W_3)}$ in coreutils can be explained by a significant amount of loops containing system calls in the loop conditions. This is consistent with the low-level nature of the benchmark, but such system calls are not captured by our definition of generalized FOR loops.

5.1 Statement Usage

An important aspect of programming language design is which constructs and patterns are actually used by programmers to formulate algorithms. In our case, we analyze which statements are used to express loops from various classes. This is especially interesting as all C statements capable of implementing iteration have the same expressive power.

Discussion. Figure 4 shows the occurrence of statements in our selection of loop classes: Overall (bar (4) in Figure 4), the for statement is used a lot more than other statements, even though the number varies greatly between more than 80% of all loops for SPEC and WCET and less than 50% for coreutils. The next-most used statement is while, with an equally wide range from 10% on WCET to 47% on coreutils, where for and while statements occur about equally often. do and goto statements make up a minor share of less than 10% and 2%, respectively.

When we compare the ratio of statements used between different loop classes of the same benchmark, an interesting observation can be made: Regardless of the overall occurrence of for in $\mathcal{L}$, the stricter the loop class, the higher the percentage of loops expressed using a for statement. At the same time, the less restrictive a loop class is, the higher the percentage of while, do, goto statements. This correlation is especially strong for do and goto statements, where virtually no such loops are in $\mathcal{L}^{\text{FOR}}$.

6 Experiments: Usefulness of Our Definitions

6.1 Comparison with Loopus

In this section, we show evidence for a relation between our loop classes and the difficulty of automated program analysis. To this end, we describe how well LOOPUS [ZGSV11] – a state-of-the-art termination and bound analysis tool – performs on our selection of loop classes: Figures 5a to 8a show the percentage of loops contained in each of these classes. Bars (1)–(4) are classes with increasingly less restrictive constraints, i.e. classes we conjecture to grow harder to
Figure 4: C statements used to express loops from loop classes (1) $\mathcal{L}^\text{SB}$, (2) $\mathcal{L}^\text{FOR} = \mathcal{L}^\text{FOR}(S_1S_2S_3)$, (3) $\mathcal{L}^\text{FOR}(W_1W_2W_3)$, (4) $\mathcal{L}$, and (5) $\mathcal{L}^\text{¬FOR}$.

(a) Percentage of loop classes (1) $\mathcal{L}^\text{SB}$, (2) $\mathcal{L}^\text{FOR}$, (3) $\mathcal{L}^\text{FOR}(W_1W_2W_3)$, (4) $\mathcal{L}$, and (5) $\mathcal{L}^\text{¬FOR}$.

(b) Percentage of loops bounded (white) and not bounded (dark) by LOOPUS.

Figure 5: Comparison of various loop classes with LOOPUS results on cBench.
Figure 6: Comparison of various loop classes with LOOPUS results on coreutils.

Figure 7: Comparison of various loop classes with LOOPUS results on SPEC.

Figure 8: Comparison of various loop classes with LOOPUS results on WCET.
analyze in that order. Bars (5) below the dashed line refer to loops in $L^{\neg \text{FOR}}$, i.e. the complement of loops shown under (3). Figures 5b to 8b show the percentage of loops in the respective class for which LOOPUS succeeds and fails to compute a symbolic bound.

**Discussion.** We see that the more restrictive the investigated loop class, the better LOOPUS performs. This supports our conjecture that given two loop constraints $C_1 \leq C_2$, the stronger constraint $C_1$ not only describes a subset of loops, but also that this subset is less challenging for automated analysis. Additionally, as expected, for any termination proof we obtain in $L^{SB}$ on benchmarks that LOOPUS was optimized for (cBench and WCET), it also bounds the loop, i.e. LOOPUS agrees on each member of $L^{SB}$ identified by SLOOPY. Manual examination of loops in $L^{SB}$ unbounded by LOOPUS on coreutils and SPEC suggests that this is due to unmodeled system calls in LOOPUS. As a positive result for LOOPUS, while it performs worst on non-FOR loops $L^{\neg \text{FOR}}$, it is still successful on about two thirds of these loops, except for coreutils which seems a much harder benchmark.

### 6.2 A Portfolio Solver for Software Verification

In [DPVZ15] we present our tool VERIFOLIO, a machine learning based portfolio solver for software verification. Here, a portfolio solver is a software verification tool which uses heuristic preprocessing to select one of several existing tools [GS01].

To represent the source code of the unit under verification to the machine learning procedure, we introduce novel program metrics based on the loop classes presented in this work, and variable roles introduced in [DVZ13]. In particular, we compute class membership using SLOOPY and measure the relative occurrence $m_C$ of four loop classes:

$$m_C = \frac{|L_C|}{|Loops|} \quad C \in \{\text{SB, FOR, FOR(W}_1W_2W_3), \neg \text{FOR}\} \quad (1)$$

We tested our portfolio on benchmarks from the annual International Competition on Software Verification (SV-COMP) in its 2014 and 2015 editions [Bey14, Bey]. In both cases, our tool VERIFOLIO is the overall winner and outperforms all other tools.

Additional experiments show that removing loop classes from the feature set yields worse results. We thus infer that they are indeed contributing to the overall performance of the portfolio. As the portfolio incurs an overhead for feature extraction compared to standalone tools, it is indispensable to base it on efficiently extractable program features. Our experiments show that the median time for computing loop class memberships per verification task in SV-COMP’14 and ’15 is $\bar{x} \approx 0.2$ seconds (obtained on a single core of a 2 GHz Intel Core i7 processor with 2 GB 1333 MHz DDR3 RAM).

**Bibliography**


Studying Verification Conditions for Imperative Programs

Cláudio Belo Lourenço\(^2\) and Si-Mohamed Lamraoui\(^1\) and Shin Nakajima\(^1\) and Jorge Sousa Pinto\(^2\)

National Institute of Informatics, Japan\(^1\)
HASLab/INESC TEC & Universidade do Minho, Portugal\(^2\)

Abstract:
Program verification tools use verification condition generators to produce logical formulas whose validity implies that the program is correct with respect to its specification. Different tools produce different conditions, and the underlying algorithms have not been properly exposed or explored. In this paper we consider a simple imperative programming language, extended with assume and assert statements, to present different ways of generating verification conditions. We study the approaches with experimental results originated by verification conditions generated from the intermediate representation of LLVM.

Keywords: software verification, LLVM, bounded verification, single-assignment

1 Introduction

Formal verification of programs is an approach to achieve required reliability levels of software-intensive systems. Given a correctness criterion, expressed as a set of assertions, we generate a formula in a suitable logic that encodes all potential execution paths and the assertions (cf. [AFPS11]). Such a formula, known as a verification condition (VC), is then proved. If the proof succeeds, then we can conclude that the program under analysis satisfies the given correctness criterion. If it does not succeed, we extract as much information as possible to debug the program and/or the specification.

In the early days, the reasoning method relied on manual proof (cf. [LSS87]). Then, introducing some automation became one of the major topics in the subject matter [FLL+02]. With the advancement of Boolean satisfiability (SAT) methods [MMZ+01] and the successful introduction of bounded model checking (BMC) [CKY03], program verification methods also adopted the SAT [CKL04] and the satisfiability modulo theories (SMT) [DM06, MB08] approaches to automated proof. If the VC refers to a formula in a decidable sub-fragment of a first-order theory, it is automatically discharged.

Whether with manual or with automatic proofs, the VC encoding is crucial for the efficiency and precision of the program verification method. Precision here refers to how much detailed information is obtained for debugging if the VC is not satisfied. Indeed, many different methods for generating VCs are used by existing program verification tools [CKL04, AMP08, MFS12, BCD+06, FP13]. In spite of this large diversity, systematic analyses of such VCs are lacking so far. Users may experience that a certain tool, adapting a certain encoding of VCs, is efficient for particular classes of programs, but not so well suited to other types of programs.
We developed a tool infrastructure for systematic analyses of various VCs. Although the algorithm employed to generate VCs is different for each method, many components in the tool are common, such as the parser, the intermediate representation, and the interface to the backend solver. The algorithm to generate different VCs can be plugged into the framework. Since we can focus on a specific generation algorithm, comparing VCs can be transparent. We used the LLVM compiler infrastructure [LA04] as host, and developed our framework on top of it.

The first contribution of this paper is the presentation of different algorithms that generate VCs using different methods. Although the algorithms are standard in software verification, as far as we know, they have never been systematically presented – they are normally formalized using different notations and formalisms, or they are not formalized at all. With a common formalization we can highlight the differences on the generated VCs and reason about them.

This paper also reports empirical results from a study we have conducted with several VC generation algorithms implemented in our framework. The study reinforces the impression, from our use of various program verification tools mentioned above, that the choice of a method is not irrelevant. Furthermore, our study shows that choosing the most efficient VCs is not an easy task – no algorithm is better than the others in most situations – and that using proved assert statements as lemmas can improve solving time significantly.

This paper is organized as follows: Section 2 reviews the background. Section 3 describes several algorithms to generate VCs. In Section 4 after presenting the VCs comparison framework, we present our empirical results. Section 5 concludes the paper.

2 Background

Program verification tools are generally divided into two main categories: deductive verification tools [BCD+06, FP13] and model checking tools [JM09] (a third important category of tools is that of abstract interpretation-based program analyzers, which fall outside the scope of our paper since they are not based on VC generation). While the former allows for a rich and expressive system of annotations through the use of first-order logic, the latter only allows simple quantifier-free properties to be used. In deductive verification, specifications are normally given as contracts (such as preconditions and postconditions of subprograms), as well as loop invariants, which are in general very difficult to generate automatically in a way that allows correctness to be proved. Although loop invariants are mandatory for the verification to proceed, in general user intervention is required in order to find them. In model checking, the verification process can proceed in a fully automatic way by automatically inserting properties (namely safety checks, such as overflow, division by zero, array out-of-bounds checks, etc.), or else, by inserting simple properties through the use of assume and assert statements. An assume statement is used to indicate that a property should be considered to hold at a given point of the program, and assert statements are used to check if a property is true at a given location.

The price to pay for automation in model checking is the lack of scalability due to state-space explosion [JM09]. To address this problem, two different approaches have been proposed along the years, which increase scalability either by sacrificing completeness (possibly reporting false alarms) in the case of abstraction techniques [JM09], or else by sacrificing soundness (considering only bounded executions) in the case of bounded model checking of pro-
assume (n >= 0);
int pp = 0, p = 1, r, i = 2;
if (n == 0) r = pp;
else r = p;
while (i <= n) {
assert (pp <= p);
  r = p + pp;
  pp = p;
  p = r;
  i = i + 1;
}

assume (n0 >= 0);
int pp0 = 0, p0 = 1, r0, i0 = 2;
if (n0 == 0) r0 = pp0;
else r0 = p0;
r1 = (n0 == 0)? r0 : r0;
if (i0 <= n0) {
  assert (pp0 <= p0);
  r0 = p0 + pp0;
  pp0 = p0;
  p0 = r0;
  i0 = i0 + 1;
  assume (! (i0 <= n0)));
} r1 = (i0 <= n0) ? r0 : r1;
pp0 = (i0 <= n0) ? pp0 : pp0;
p0 = (i0 <= n0) ? p0 : p0;
i0 = (i0 <= n0) ? i0 : i0;

Figure 1: Fibonacci function (left) and its conversion to SSA after unwinding loop once (right)

grams [CKL04, AMP08, MFS12]. The latter technique generates potentially very large VCs, since it is based on loop unfolding, and for this reason the choice of a VC generation method may be particularly relevant.

In spite of the fundamental differences between deductive verification and BMC, the techniques also share many similarities. In both cases, state-of-the-art tools rely on the conversion of standard programs to an intermediate single-assignment form, from which a specific algorithm, called a verification condition generator (VCGen), generates VCs in the form of logical formulas. These VCs must then be discharged, that is, they must be sent to a solver for validity checking. If they are valid, the program is correct with respect to its specification.

In this paper, we will focus on single-assignment programs with no loops. Note, however, that we are not limiting our approach – programs with loops can be transformed into programs free of loops that can be verified instead of the original. In BMC tools (e.g. CBMC [CKL04]), loops are unwound $k$ times (where $k$ is given by the user or fixed/inferred by the tool) and an assume statement is inserted to ignore executions requiring further iterations. An alternative that generates sound VCs as opposed to BMC, is described in [BL05] and implemented in Boogie [BCD +06]. Loops are replaced by a series of statements that simulates an arbitrary iteration of the loop. Here, loop invariants are required to restrict the verification to feasible iterations. In this paper we will focus exclusively on the BMC approach, because it generates larger VCs, resulting in more interesting formulas for our evaluation.

The basic principle of single-assignment (SA) forms is that once a variable has been used, it cannot be assigned. The most popular SA form, Static Single-Assignment (SSA) [CFR +91], has been part of compiler pipelines for decades, and more recently it has been used in software verification tools. In this form each variable in a program can be assigned at most once. To synchronize variables assigned in branching blocks, Phi-functions (often replaced by conditional expressions) are used to capture the correct values at the merging point. An alternative approach, known as Dynamic Single-Assignment (DSA) form (e.g. [BL05, CFP12]), allows variables to be assigned more than once, as long as it is in different execution paths.

The example shown in Figure 1 (left) calculates the $n$-th Fibonacci number. On the right we show the result of unwinding the loop once and converting it to SSA form, following the BMC approach. The first assume statement is used as a precondition to specify that the verification
procedure should only consider executions where \( n \) is greater than or equal to 0\(^1\). The assert statement is used to check a basic property that in the recurrence relation of the Fibonacci sequence corresponds to \( F_{n-2} \leq F_{n-1} \). The second assume statement is introduced when the loop is unwound, and its role is for the executions requiring more iterations to be ignored.

3 Verification Condition Generators

This section presents different VCGens for SA programs whose loops and function calls have been previously removed as explained in the previous section. The presented VCGens are sound and can be used interchangeably. We will be considering an elementary SA language of branching programs with integer type expressions, whose commands are as follows:

\[
\text{Comm} \ni C ::= \text{skip} \mid \text{assume } \theta \mid C ; C \mid x := e \mid \text{if } b \text{ then } C \text{ else } C \mid \text{assert } \theta
\]

with, \( x \) ranging over a set of SA variables, \( e \) over integer expressions and \( b \) over Boolean expressions. We do not fix the language of integer and Boolean expressions, but we do restrict it to be a language that can be encoded in the chosen backend solver. The algorithms will take as input a command \( C \in \text{Comm} \), and will return a set of VCs in the form of logical formulas whose validity implies that the program is correct. Program in Figure 1 will be used as a running example to show differences in the generated VCs. For more details about theoretical aspects, such as soundness results, refer to [BFS15].

3.1 Symbolic Execution

Symbolic execution (e.g. [JMNS12]) is the simplest way of generating VCs. The approach consists of generating a VC for every single execution path that reaches an assert statement, such that the validity of each VC ensures that executions going through the encoded path satisfy the assert statement. To prove that a program is fully correct, a VC for each possible execution path and each assert statement must be generated and then discharged.

\(^1\) to perform modular verification, the same property must be checked with an assert statement at each calling point.
Definition 1 (SE VCGen) Given an SA program $C$, its Symbolic Execution VCs are given by the set $V$ where $(E, V) = \text{VC}^\text{SE}(\emptyset, C)$, and $\text{VC}^\text{SE}$ is the function given in Figure 2 (left).

Function $\text{VC}^\text{SE}$ in Figure 2 receives an extra parameter in addition to the input program and also returns an extra formula in addition to the set of VCs. The extra parameter corresponds to the encoding of part of the program that precedes the current statement. The extra formula is composed by the conjunction of the received formulas, with the encoding of the current statement. When an assert statement is encountered, the extra parameter contains the encoding of all possible execution paths that reach this assert statement. A VC will be generated for each of these paths. The generated SE VCs for our running example are as follows:

$$
\begin{align*}
VC_1 &: n_0 \geq 0 \land pp_0 = 0 \land p_0 = 1 \land l_0 = 2 \land n_0 = 0 \land r_1 = pp_0 \land r_3 = ((n_0 = 0) \land r_1) \land l_0 \leq n_0 \rightarrow pp_0 \leq p_0 \\
VC_2 &: n_0 \geq 0 \land pp_0 = 0 \land p_0 = 1 \land l_0 = 2 \land (n_0 = 0) \land r_2 = p_0 \land r_3 = ((n_0 = 0) \land r_1) \land l_0 \leq n_0 \rightarrow pp_0 \leq p_0
\end{align*}
$$

The first VC corresponds to executions going through the then branch in the first if statement, and the other to executions going through the else branch. Even though, the generated VCs are only two and they are relatively small in this case, it is important to note that the number of paths is, in the worst case, exponential with respect to the size of the program and so is the number of generated VCs [CFP12].

3.2 Efficient Strongest Postcondition

The weakest precondition (WP) and strongest postcondition (SP) predicate transformers introduced by Dijkstra [Dij76] for his guarded commands language may also be used to generate VCs. These techniques predate symbolic execution, and in their original form do not require the program to be transformed into an SA form. For non-SA programs the technique produces VCs whose size is, in the worse case, exponential with respect to the size of the program. Flanagan and Saxe showed that when the technique was applied to SA programs, the size of the generated VCs was, in the worst case, quadratic [FS01]. In this paper we focus exclusively on SP VCs for SA programs. Due to lack of space we omit the discussion of WP which generates similar VCs.

Definition 2 (SP VCGen) Given an SA program $C$, its Strongest Postcondition VCs are given by $V$ where $(E, V) = \text{VC}^\text{SP}(\top, C)$, and $\text{VC}^\text{SP}$ is the function given in Figure 2 (right).

Definition 2 differs slightly from the standard definition of SP, because the asserted properties are not being introduced into the context – we leave this discussion for Section 3.5. As in the SE auxiliary function (Section 3.1), the function $\text{VC}^\text{SP}$ receives an auxiliary parameter and returns an extra formula. Note however that the returned formula consists exclusively of the encoding of the present statement. The part of the program that has already been encoded is now propagated by the if and sequence rules and a disjunction is introduced to encode each branching statement. The generated SP VC for our running example is as follows:

$$
n_0 \geq 0 \land pp_0 = 0 \land p_0 = 1 \land l_0 = 2 \land (n_0 = 0 \land r_1 = pp_0) \lor (\neg (n_0 = 0) \land r_2 = p_0) \land r_3 = ((n_0 = 0) \land r_1) \land l_0 \leq n_0 \rightarrow pp_0 \leq p_0
$$

3.3 Conditional Normal Form

In the context of BMC, after unwinding loops, inlining function calls and converting the code to an SA form, VCs are generated by performing a series of transformations on the code. In
particular the following rules are applied to conditional branches:

1. if $b$ then $C_1$ else $C_2$ \implies if $b$ then $C_1$; if $\neg b$ then $C_2$
2. if $b$ then \{ $C_1$; $C_2$ \} \implies if $b$ then $C_1$; if $b$ then $C_2$
3. if $b_1$ then if $b_2$ then $C$ \implies if $b_1 \land b_2$ then $C$

Note that these transformations would not be sound in a standard imperative program (a program not in SA form). For instance, in the case 1 above, the execution of $C_1$ could modify some variable used in $b$ causing both branches of the conditional branch to be executed. Since in the SA form, once a variable has been used cannot be assigned, we have a sound transformation.

After the above transformations the program consists of a sequence of conditional statements of the form if $b$ then $C$, with $C$ an atomic (assignment, assume, or assert) statement. This is the so-called conditional normal form (CondNF) of an SA program. From this form, it is easy to extract a VC. In this paper, instead of applying each transformation individually, we present an algorithm that internally captures the above transformations with a single pass through the program and generates the corresponding VC.

**Definition 3** (CondNF VCGen) Given an SA program $C$, its Conditional Normal Form $VC$ is the formula $\bigwedge E \rightarrow \bigwedge P$, where $(E, P) = VC_{cnf} (\top, \emptyset, C)$, and $VC_{cnf}$ is the function given in Figure 3 (left).

In $VC_{cnf}$ function, two extra parameters are received. The first contains the path condition for the present statement to be reached. The second, which is not being used in this definition, contains the encoding of the program up to the present statement. The set $E$ contains the operational encoding of the program (assignment statements) and assume statements, and the set $P$ contains the asserted properties. Regardless of the number of execution paths and the number of asserted properties, only one VC, whose size is in the worse case quadratic, is generated [CFP12]. Even though, such a VC might be hard to debug, it has an advantage of being the unique one that has to be sent to the solver for validity checking.

The use of a single global context implies that the semantics of assume statements is different from the previous techniques: whereas before an assume statement dispensed executions from
having to pass subsequent assert statements, in the CondNF the use of a global context means
that an assume statement now dispenses executions from passing any of the assert statements
they meet. An alternative approach consists of generating multiple VCs with partial contexts –
each assert property is only implied by the relevant part of the program. This can be done by
generating a VC each time an assert statement is encountered – recall that the second parameter
in the auxiliary function VC_{cnf} contains the encoding of the program up to the present statement.

**Definition 4** (PCondNF VCGen)  Given an SA program C, its Partial context Conditional Normal Form VCs are given by V, where \((E, V) = VC_{pcnf}(\top, \emptyset, C)\), and VC_{pcnf} is the function in which \(VC_{pcnf}(\pi, C, \text{assert } \theta) = (\emptyset, \land C \rightarrow \pi \rightarrow \theta)\) and \(VC_{pcnf}(\pi, C, S) = VC_{cnf}(\pi, C, S)\), for \(S \in \text{Comm} \setminus \text{assert}\) and VC_{cnf} given in Figure 3 (left).

For our running example, VC_{cnf} produces the following sets of formulas which are then used by
CondNF VCGen (Definition 3) to generate a VC:

\[
E = \{ \top \rightarrow n_0 = 0, \top \rightarrow pp_0 = 0, \top \rightarrow p_0 = 1, \top \rightarrow i_0 = 2, n_0 = 0 \rightarrow r_1 = pp_0, \neg(n_0 = 0) \rightarrow r_2 = p_0, \top \rightarrow r_3 = ((n_0 = 0) \land r_1 : r_2), \\
\land i_0 \leq m_0 \rightarrow r_4 = pp_0, pp_0, i_0 \leq n_0 \rightarrow p_0 = p_0, i_0 \leq n_0 \rightarrow p_1 = r_4, i_0 \leq n_0 \rightarrow i_1 = i_0 + 1, i_0 \leq n_0 \rightarrow \neg(i_1 \leq n_0), \\
\top \rightarrow r_5 = \ell_0, \top \rightarrow p_2 = ((\ell_0 \leq n_0)?pp_1 : pp_0, \top \rightarrow p_2 = ((\ell_0 \leq n_0)?p_1 : p_0, \top \rightarrow r_2 = ((\ell_0 \leq n_0)?r_1 : r_0) \\
P = \{ i_0 \leq m_0 \rightarrow pp_0 \leq p_0 \}
\]

The VC generated by PCondNF VCGen (Definition 4) is as follows:

\[
((\top \rightarrow n_0 = 0) \land (\top \rightarrow pp_0 = 0) \land (\top \rightarrow p_0 = 1) \land (\top \rightarrow i_0 = 2) \land (n_0 = 0 \rightarrow r_1 = pp_0) \land (\neg(n_0 = 0) \rightarrow r_2 = p_0) \\
\land (\top \rightarrow r_3 = ((n_0 = 0)?r_1 : r_2)) \rightarrow (i_0 \leq m_0 \rightarrow pp_0 \leq p_0)
\]

In the case of CondNF, as opposed to PCondNF, the whole program is used as a context even
though only part of it is required. In both cases, every statement is encoded with its path condition,
resulting in larger formulas than those coming from the predicate transformers VCGens.

### 3.4 Simplified Conditional Normal Form

The previous VCGens generate sound VCs from programs in either SSA or DSA form. There
exists however an additional point of interest in the use of SSA form. Since a new variable is
introduced after each branch statement to synchronize both branches, we can encode assign-
ment statements without considering their path conditions. This approach omits part of the
control flow, but this is not a problem because the relevant information is propagated through
Phi-functions. Although the transformation is not operationally equivalent, it captures the nec-
essary information to generate sound VCs. The next definition enhances the CondNF VCGen
(Definition 3) in this way.

**Definition 5** (SCondNF VCGen)  Given an SSA program C, its Simplified Conditional Normal Form VC is the formula \(\land E \rightarrow \land P\), where \((E, P) = VC_{scnf}(\top, \emptyset, C)\), and VC_{scnf} is the function given in Figure 3 (right).

Note that, in the auxiliary function VC_{cnf}, only the rule that encodes assignments is differ-
ent from the VC_{cnf} function. As for CondNF, the previous definition can be adapted to gen-
erate multiple VCs with partial contexts, resulting in a similar algorithm as the one used by
CBMC [CKL04].
Definition 6 (PSCondNF VCGen)  Given an SSA program $C$, its Partial context Simplified Conditional Normal Form VCs are given by $V$, where $(E,V) = \text{VC}\text{pscnf}(\top, \emptyset, C)$, and $\text{VC}\text{pscnf}$ is the function in which $\text{VC}\text{pscnf}(\pi, C, \text{assert } \theta) = (\emptyset \land C \rightarrow \pi \rightarrow \theta)$ and $\text{VC}\text{pscnf}(\pi, C, S) = \text{VC}\text{scnf}(\pi, C, S)$, for $S \in \text{Comm} \setminus \text{assert}$ and $\text{VC}\text{scnf}$ given in Figure 3 (right).

Function $\text{VC}\text{scnf}$ produces the set of formulas as follows to be used by SCondNF (Definition 5)

$$E = (n_0 \geq 0 \land pp_0 = 0 \land p_0 = 0, \ i_0 = 2, r_1 = pp_0, r_2 = pp_1, r_3 = (n_0 = 0) \land r_2), r_4 = pp_0 \land pp_1 = p_0, p_1 = r_4, i_1 = i_0 + 1, i_0 \leq n_0 \rightarrow (i_1 \leq n_0), r_c = (i_0 \leq n_0) \land r_4 \land (r_2 = pp_1) \land (r_3 = (n_0 = 0) \land r_1), 0 \leq p_1 \land p_2 = (i_0 \leq n_0) \land p_1 \land p_2 = (i_0 \leq n_0) \land p_1 \land i_2 = (i_0 \leq n_0) \land i_1 = i_0, p = \{i_0 \leq n_0 \rightarrow pp_0 \leq p_0\}$$

and the PSCondNF VCGen (Definition 6) produces the following VC:

$$((n_0 \geq 0) \land (pp_0 = 0) \land (p_0 = 1) \land (i_0 = 2) \land (r_1 = pp_0) \land (r_2 = p_0) \land (r_3 = (n_0 = 0) \land r_1)) \rightarrow (i_0 \leq n_0 \rightarrow pp_0 \leq p_0)$$

The observations we made about the number and size of the VCs generated by CondNF and PCondNF are also applied to SCondNF and PSCondNF. Note however that the size of the formulas is slightly smaller, because the path conditions are not used for assignment statements.

3.5 Assert Statements as Lemmas

We have presented different ways of generating verification conditions, but we have not actually discussed how to solve them. It is important to note that we are assuming that one VC is checked at a time. Whenever a VC is false, one can stop the verification process because a violation was found. If all VCs are valid, one can say that the program is correct.

It is interesting to point out that assert statements, once proved, can be used as lemmas to prove the subsequent properties. One way of doing it is by inserting the proved assert statements’ properties in the context of the subsequent assert statements. In this case the VCs must be generated and solved in the same order as they appear in the code, so that all lemmas are proved before they are referred to. The previous VCGens algorithms can be modified to reproduce this behavior as in the following definition.

Definition 7 (VCGens with assert statements as lemmas)  Given a $\text{VCGen} \in \{SE, SP, PCondNF, PSCondNF\}$ and its corresponding auxiliary function $\text{VC} \in \{\text{VC}^\text{se}, \text{VC}^\text{sp}, \text{VC}^\text{pcnf}, \text{VC}^\text{pscnf}\}$, and $\text{VCGen}_l$ (resp. $\text{VC}_i$) to indicate that assert statements are used as lemmas. In this case, $\text{VC}_i(S) = \text{VC}(\ldots, S)$ for $S \in \text{Comm} \setminus \text{assert}$. $\text{VC}_i(\ldots, \text{assert } \theta)$ is given as follows:

1. $\text{VC}^\text{se}_i(\Phi, \text{assert } \theta) = (\Phi \land \theta, \Phi \rightarrow \theta)$
2. $\text{VC}^\text{sp}_i(\phi, \text{assert } \theta) = (\theta, \{\theta \rightarrow \theta\})$
3. $\text{VC}^\text{pcnf}_i(\pi, C, \text{assert } \theta) = (\pi \rightarrow \theta, \land C \rightarrow \pi \rightarrow \theta)$
4. $\text{VC}^\text{pscnf}_i(\pi, C, \text{assert } \theta) = (\pi \rightarrow \theta, \land C \rightarrow \pi \rightarrow \theta)$

These lemmas cannot be used by $\text{VC}\text{scnf}$ and $\text{VC}\text{scnf}$ because only one formula is generated with the whole encoding of the program and the assert statements – if we added the asserted properties to the global context, the generated VC would be trivially (and wrongly) discharged.

For our running example this modification would not produce any effect, because only one assert statement is present. Note however that if the loop statement was unwound twice, two assert statements would exist, one from each iteration. The assert statement from the first iteration could be used as lemma for the second iteration.
3.6 A Glance Over the Differences

Both the efficient predicate transformers and CondNF VCGens are able to curb the exponential growth of the global VCs’ size that is characteristic of SE and of the original predicate transformer methods. Although both algorithms produce VCs that cannot grow faster than quadratically in the size $n$ of the program, one cannot be said to be better than the other: think for instance of the program shown in Figure 4 (left), consisting only of a sequence of assignment statements and a single assert statement at the beginning – the CondNF VC has linear size because it contains a global context, and the SP VC has constant size; now, imagine a similar program as shown in Figure 4 (right), where an assert statement follows each assignment – CondNF VC still has linear size, but the SP VCs have quadratic size, since there are $n$ conditions, of size 1 to $\Theta(n)$. In these examples PCondNF VCGen, would generate similar VCs to those generated by SPVCGen. For different VCs to be generated, examples with conditional branches (as was the case of our running example) must be considered.

The CondNF and PCondNF VCs’ size can be reduced by taking advantage of the SSA form. This is achieved by the SCondNF and PSCondNF VCGens, where the flow of the program is captured by the Phi-functions only. Note however, that in this case, the solver might have to evaluate all assignments because they are not guarded by the path condition.

Lastly we have shown that VCs generated with partial contexts can be easily modified to make use of valid assert statements as lemmas for subsequent properties. Note however that it is not guaranteed that the lemmas will simplify the proofs. Therefore if this approach is applied blindly it might produce bigger encodings without adding useful information to the context.

4 Evaluation

Let us now present the results of an empirical study to compare the VCGens in terms of solving time. We focus exclusively on the solving time, disregarding other factors, such as, the time for parsing, applying the transformations referred in Section 2, generating VCs or even encoding the formulas in the backend solver, because it is clearly the solving process that dominates the growth of the verification time. Moreover, in the present paper we base our study on VCs generated by unwinding loops (VCs generated in a deductive verification setting may well be more complex). In what follows, when we write that a VCGen is more efficient than another, we mean that the VCs generated by the first are faster to solve than those generated by the second.

We have created a VCGen comparison framework based on components from the SNIPER tool set [LN14]. We focused on the ease of adding new VCGens, and adapted all the previous definitions for the LLVM intermediate representation (IR) [LA04]. We have chosen LLVM IR
firstly because it is already in SSA form, which allows us to compare all presented VCGens. Moreover, different languages, such as C, Ada or Objective-C, can be compiled into LLVM IR, enabling a single VCGen comparison framework for multiple input languages (this paper focused exclusively on a subset of C programs). Yices [DM06], version 1.2, is used as backend SMT solver, and following [AMP08] we used linear integer arithmetic to check the validity of the generated VCs. The user can choose if asserts are added to context with a command line flag.

The first part of this section shows the evaluation of the VCGens with the Fibonacci function. We compare how the solving time grows as the unwinding bound is increased. Even though the example is very simple, it already illustrates some properties of the VCGens. We then use more complex programs taken from different benchmarks previously used to validate verification tools (e.g. [GCNR08, GR09]). First we expose the difficulties of selecting the most efficient VCGen, then we motivate the use of assert statements as lemmas. Due to the inefficiency of the SE VCGen we only consider it for the Fibonacci function, where the exponential growth is not seen. All tests were performed on a 1.7GHz MacBook Air with 4GB of RAM and OS X 10.10.

### 4.1 The Fibonacci Running Example

Let us start by analysing our running example as the unwinding bound increases. The solving time for each VCGen is shown in Figure 5. For this particular example, adding assert statements as lemmas does not produce significant differences in the solving time (neither positively nor negatively). In the log-scale chart on the left we have merged SP, PCondNF, and PSCondNF together, since their solving time is almost the same. From the chart on the right we can see that SP and PSCondNF are slightly faster than PCondNF (on average 13% faster). The use of path conditions in the PCondNF to guard the assignment statements does not improve the solving time – instead it just increases the formula size and the solving time.

It can be observed from the results that the SE solving time is not growing exponentially. This is justified by the fact that the number of paths that reach each assert statement is always two: only one path results from the code obtained from unwinding the loop; this code consists of nested if statements, where the assert statement is always in the path in which all precedent if conditions are true. Therefore, the most relevant difference when comparing with SP, PCondNF and PSCondNF is that two VCs are generated for each assert statement instead of one. Another interesting lesson we can draw from this chart is that irrelevant code, which is present in CondNF and SCondNF due to the fact that whole program is encoded (inclusively the code after the assert statements), may heavily degrade the solving time, even if only one formula has to be solved.

![Figure 5: Time to solve VCs generated from Fibonacci running example](image-url)
4.2 Electing the Most Efficient VCs

The first question that arises when evaluating different algorithms with a common goal is: “which one is the most efficient?”. Unfortunately the answer to such a question is not straightforward. To answer this question by means of an empirical study, one should run the different algorithms with different case studies, and select the method that produces the most efficient VCs. In order to show how difficult it is to choose a particular VCGen, let us consider a benchmark\(^2\) whose case studies have been previously used to test and validate a wide variety of software verification tools (e.g. [GCNR08, JMNS12]). In this benchmark, assume statements are exclusively used as preconditions (placed at the beginning of the program) and assert statements are exclusively used as postconditions (placed at the end of the program). Therefore, it only makes sense to compare SP, CondNF and SCondNF: all others (apart from SE) will generate equivalent VCs. All case studies have loops, which allows us to adjust the complexity of the input program.

Different test target programs were selected to illustrate how the most efficient VCs can change from one target benchmark program to another. Results are shown in Figure 6: the x-axis shows different programs and the number of times they were unwound; the y-axis shows the solving time (in logarithm scale) for the generated VCs. The most consistent pattern is that CondNF is never the best option, but it is also seldom the worst. All the others vary from case to case.

Let us now focus on a particular benchmark target program, to show how subtle modifications in the code or specification can change drastically the efficiency of a particular VCGen. Figure 7 shows two similar variants (1) and (2) of a particular case study, and the required time to solve the generated VCs, as the loop unwinding bound is increased. While SCondNF is the most efficient VCGen for (1) and SP is the less efficient, for (2) SP is the most efficient and SCondNF is the less efficient. Note also that the solving time is longer in (2). The main difference between both variants is that (1) iterates exactly 100 times, while (2) iterates between 0 and 100 times depending on the value of the variable x. In (1), 100 assignments are always reached, but in (2) the number of assignments that are reached depends on the value of a variable whose value

\(^2\) Available from http://map.uniroma2.it/smc/simp/
is greater than or equal to zero. In both cases, only dead code results from unwinding the loop more than 100 times. We imagine that the inefficiency of PCondNF in (2) is justified by the amount of unguarded expressions. Surely the VCs generated from (1) also have unguarded expressions, but they can be easily solved if the SMT solver applies some simplification strategy, such as constant propagation. In this case, having a condition guarding assignment statements only increases the size of the expressions that have to be simplified. In the chart from (2) note also how the SCondNF solving time increases drastically when loops are unwound more than 100 times: it is precisely at this point that some dead code is added.

It can be argued that it is not useful to unwind the loop more than 100 times, and that this is an artificial problem. Therefore we considered a variant where we replaced the value 100 in the loop condition and assert statement, by an undetermined variable. Naturally the solving time for both sets of VCs increased. However, PSCondNF is no longer the most efficient solver for (1). Instead, the solving time is now very similar for the different sets of VCs. In the variant (2), SP still produces the most efficient VCs and PSCondNF the most inefficient.

After looking at this case study one could intuitively think that SP is the most efficient VCGen, since it performs better in the presence of indeterminately-valued variables, but we remark that this is not always true. For instance, the first target program shown in the chart has a structure that depends on indeterminate variables, but SCondNF still generates the most efficient VCs.

### 4.3 Assert Statements as Lemmas

The previous examples are already complex enough, but they follow a specific pattern for using assume and assert statements as preconditions and postconditions respectively, which does not allow us to compare the algorithms of Section 3.5. An additional benchmark\(^3\) we consider in our evaluation has been used to test Invgen [GR09], an invariant generator tool. It contains iterative programs with relatively complex data flow, and some of the examples contain multiple assert statements spread throughout the program. We present here a representative selection of the results obtained with this benchmark. We do not consider the CondNF and SCondNF algorithms, which in general perform worse than PCondNF and PSCondNF.

The chart in Figure 8 shows the required solving time for each set of VCs generated from a particular program using different VCGens. Due to space constraints we are not able to discuss

---

\(^3\) Available from [http://www.tcs.tifr.res.in/~agupta/invgen/](http://www.tcs.tifr.res.in/~agupta/invgen/)
each case study separately. Instead, let us categorize the differences in terms of the usage of assert statements. The first three contain multiple assert statements throughout the program. The next three contain assert statements inside the loop(s), resulting in multiple assert statements as the loops are unwound. The last three contain assert statements at the end of the program only.

It can be observed from the chart that in many cases the time required for solving the generated VCs decreases when assert statements are used as lemmas. In other cases, the lemmas do not improve the solving time, but they also do not degrade it considerably. The last two case studies have a single assert statement at the end, therefore the VC that is produced is the same, whether assert statements are used as lemmas or not.

5 Conclusion and Future Work

As far as we know, this is the first time that the VCGens are presented in a systematic way – they are normally formalized using different languages and concepts, or they are not formalized at all. We used a simple imperative programming language to expose the algorithms in a very concise and clear way which allowed us to highlight the differences between them. Note however, that a crude comparison of VCGen algorithms in isolation, as we have carried out, may be unfair to some of them, since their performance may depend on the combination with other techniques.

The SE VCGen is impractical due to its inefficiency, but it can be seen as the basis of tools like TRACER [JMNS12], which employs an interpolation technique, based on weakest preconditions and unsatisfiable cores, for detecting infeasible paths and avoiding exponential path enumeration. Although we opted to use forward propagation for the sake of uniformity, the SP VCGen produces VCs similar to those produced by the Boogie tool [BCD+06], which is based on efficient weakest preconditions. CondNF VCs are described for instance in [CKL04, BCD+06, AMP08], and the PSCondNF VCGen is our interpretation of the method used by the CBMC tool, which includes partial contexts and the SSA-specific optimizations described in Section 3.4.

Even though the VCGens were presented for a simple imperative language, we based our implementation on an intermediate representation used by real world applications. We chose the LLVM IR due to its native SSA form, which allows us to compare all the presented algorithms.

The empirical study reveals that it is not possible to select a single most efficient VCGen – it varies from case to case. In general, we have seen that solving a large VC, generated by CondNF and SCondNF, is slower than solving multiple smaller VCs generated by PCondNF and PSCondNF respectively. However, in exceptional cases, solving a single VC is slightly faster than solving multiple smaller ones. Moreover, it is sometimes useful to guard the assignments, but not always: sometimes the guards create larger formulas, increasing the solving time.

What seems to contribute decisively to decrease the solving time, is to use proved assert statements as lemmas. This never seems to increase the solving time considerably, and in some cases it decreases considerably. Therefore, adopting these variants is recommended in general in order to generate efficient VCs.

We also used faulty programs for our evaluation, but the results mostly followed what had been mentioned before. Of course, if an assert statement at the beginning of the program is violated, the VCs that contain only partial contexts will rapidly fail – on the other hand, those with a global context will be extremely inefficient. If there is a violated assert in the final part of
the program, VCs with partial contexts sometimes perform worse than VCs with global contexts.

Considering our evaluation results, which shows that the chosen VCGen algorithm is not irrelevant, we suggest that verification tools are desirable to provide different VCGens and leave to the end user the decision of which one to use (in the same way that some tools allow using of various solvers). This way, a default VCGen might be used, but if it does not scale as desired, an alternative VCGen might be chosen. Note, however, that this may not always be possible or easily achievable, since the VCGen algorithm may be deeply integrated into the tool in a way that makes it difficult to decouple.

As future work we believe that it would be interesting to categorize which kinds of programs perform better with each VCGen. In particular, it would be interesting to have a wrapper algorithm or heuristic to choose the best VCGen for each situation. We also believe that it would be interesting to make a similar study using DSA instead of SSA – this is in fact a limitation of our framework, because LLVM does not allow a DSA representation. We did not consider, in this paper, multiple backend solvers. It would be equally interesting to observe the efficiency of different VCs in different solvers. Moreover, considering the simplification of VCs before they are sent to the backend solver – applying constant propagation, or expressions rewriting (as done by CBMC) – can produce a new insight for a future evaluation.

Acknowledgements: This work is partially financed by the FCT – Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project UID/EEA/50014/2013. This work is partially supported by the NII International Internship Program.

Bibliography


Automated Verification of Asynchronous Communicating Systems with TLA$^+$

Florent Chevrou, Aurélie Hurault, Philippe Quéinnec

firstname.lastname@enseeiht.fr
IRIT – Université de Toulouse
2 rue Camichel
F-31000 Toulouse, France
http://www.irit.fr

Abstract:

Verifying the compatibility of communicating peers is a crucial issue in critical distributed systems. Unlike the synchronous world, the asynchronous world covers a wide range of message ordering paradigms (e.g. FIFO or causal) that are instrumental to the compatibility of peer compositions. We propose a framework that takes into account the variety of asynchronous communication models and compatibility properties. The notions of peer, communication model, system and compatibility criteria are formalized in TLA$^+$ to benefit from its verification tools. We present an implemented toolchain that generates TLA$^+$ specifications from the behavioral descriptions of peers and checks compatibility of the composition with respect to given communication models and compatibility criteria.

Keywords: asynchronous communication, peer composition, compatibility checking, TLA$^+$

1 Introduction

Building systems through assembling and coordinating off-the-shelf components is a thriving software production principle. The formal verification of the correctness of the composition of a set of peers is crucial to this approach when it comes to critical systems. In this setting, the interaction model can directly impact the properties of the global system. In distributed algorithms research, it has long been known that the properties of the communication, and especially the order of message delivery, is essential to the correctness of the algorithms. For instance, the Chandy-Lamport snapshot algorithm [CL85] requires that the communication between two processes is FIFO, and Misra’s termination detection algorithm [Mis83] works with a ring containing each node once if the communication ensures causal delivery, but requires a cycle visiting all network edges if communication is only FIFO.

Although the question of characterizing the properties of a set of combined services has been extensively studied for quite a long time (e.g. [BZ83, LW11]), existing works are restricted, to the best of our knowledge, to a specific interaction model (either synchronous or asynchronous, or coupling via bounded buffers), to which their formalization and verification framework are dedicated. Moreover the diversity of asynchronous communication models is generally ignored.
We present a framework and a ready-to-use automated toolchain, based on TLA\(^{+}\), that enables to check LTL properties on distributed systems. A system is the conjunction of peers and communications models (TLA\(^{+}\) modules) that interact through channels. Unlike many existing approaches, explicit senders and receivers are not required which allows for a greater variety of specifications. Although several communication models and compatibility LTL properties are supplied, the framework can be extended at will with additional ones thanks to its modular structure and transition system base. These additions can be generic or fulfill system-specific needs (case-by-case adaptation). Finally, it integrates well with other tools: peer specification helpers, not part of the core framework, are provided in the presented toolchain.

The outline of this paper is the following. Section 2 introduces our views on asynchronous communication and the choices made to model interaction. Several classic communication models are then presented. They highlight the diversity of asynchronous communication. Section 3 presents the core framework and an automated toolchain where peers are specified with transition systems derived from CCS terms. Section 4 presents a use case example and a performance benchmark. Section 5 provides an overview of the conceptual background of this work and, eventually, the conclusion draws perspectives after summing up this work.

2 Asynchronous communication

2.1 Intuition

Communication consists in exchanging messages whose content is not relevant outside the scope of peers’ internal behavior. Messages are sent on channels. Channels do not have explicit sender and receiver and are not limited to one sender/one receiver. They are nevertheless a point-to-point communication abstraction: a given message has exactly one sender and is received only once. Loose channels allow for richer and more elegant system specifications, where the reception of a message can occur on a peer that depends on the communication medium’s state.

From the traditional distributed systems viewpoint, the communication medium controls the messages deliveries. Peers cannot impose a delivery order. However, a peer specifies which channels it listens to in order to prevent the delivery of a message it is not and never will be concerned about. For instance, a peer that only expects to receive a message from channel \(a\) followed by a message from channel \(b\) cannot impose the reception order (the communication model will have to if this is essential to compatibility), but it can specify it only listens to channels \(a\) and \(b\), thus preventing the communication medium from imposing a message from another channel.

2.2 Communication Models

We describe seven asynchronous communication models in table 1 and provide instance implementations, often based on queues, to illustrate them. They enforce an order of message receptions in relation with the order of their emissions. For instance, in \(M_{n-1}\), each peer has a unique mailbox which is FIFO-ordered. Messages are delivered to this peer in their absolute send ordering, whatever the sending peers are. This contrasts with \(M_{1-1}\) where a queue is present between every couple, and no order is imposed between two messages coming from different peers.
Table 1: Communication Models Description

<table>
<thead>
<tr>
<th>Model</th>
<th>Specification</th>
<th>Intuitive Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_{async}$</td>
<td>Fully asynchronous. No order on message delivery is imposed.</td>
<td>A bag from which messages are non-deterministically retrieved.</td>
</tr>
<tr>
<td>$M_{n-n}$</td>
<td>Global ordering. Messages are delivered in their send order.</td>
<td>A unique FIFO queue where all messages are put in and retrieved from.</td>
</tr>
<tr>
<td>$M_{1-n}$</td>
<td>Messages from the same peer are delivered in their send order.</td>
<td>An output queue for each peer (outbox) from which messages are instantly retrieved.</td>
</tr>
<tr>
<td>$M_{n-1}$</td>
<td>On a given peer, messages are received in their send order.</td>
<td>An input queue for each peer (mailboxes) where messages are instantly deposited.</td>
</tr>
<tr>
<td>$M_{1-1}$</td>
<td>Messages between two designated peers are delivered in their send order.</td>
<td>A FIFO queue between each couple of peers.</td>
</tr>
<tr>
<td>$M_{causal}$</td>
<td>Messages are delivered according to the causality of their emission [Lam78].</td>
<td>Using causal histories [SM94] or logical matrix clocks [RST91].</td>
</tr>
<tr>
<td>$M_{RSC}$</td>
<td>Messages are immediately delivered after their send [CMT96].</td>
<td>A 1-slot unique buffer shared by all peers.</td>
</tr>
</tbody>
</table>

**Bounded Implementations** Some variations also include the possibility to count and/or limit the number of in transit messages, locally (peer) or globally. A counter of in transit messages is updated at send and receive events. It is used in a threefold manner. First, an enforced limit models a bounded network where the emission of a message is not always enabled. Secondly, the state space is reduced, thus making its exploration quicker. Thirdly, for a correct finite system, the maximal value of the counter is the highest number of in transit messages.

**Composite Models** We also consider composite communication models made up of several other models. Each one of them manages communication on its own subset of channels. This makes it possible to ensure different ordering properties on these channel groups.

3 A Framework for the Verification of Asynchronously Communicating Peers

This section presents a framework aimed at checking compatibility properties over a composition of a set of peers and a communication model (possibly composite). Peers and communication models are both specified using transition systems. Interactions between them are represented by
a synchronous product. These notions are translated into TLA\(^+\) specifications where this product appears as a conjunction of actions. The choice of TLA\(^+\) as the specification language arises from the high-level structures (such as sets, tuples and functions) it offers. This paves the way for evolved communication models implementations that for instance rely on nested message histories. The "actions as predicates" approach also eases the synchronous product operation.

Figure 1 provides an overview of the different implemented steps to perform the automatic verification of a composition. They are detailed in the following sections.

### 3.1 TLA\(^+\) Specification Language

TLA\(^+\) [Lam03] is a formal specification language based on untyped Zermelo-Fraenkel set theory for specifying data structures, and on the temporal logic of actions (TLA) for specifying dynamic behaviors. Expressions rely on standard first-order logic, set operators, and several arithmetic modules. System properties are specified using TLA which is a variant of linear temporal logic (LTL).

The dynamic behavior of a system is expressed as a transition system whose specification is usually written as \(\text{Init} \land [\Box \text{Next}]_{\text{vars}} \land \mathcal{F}\), where \(\text{Init}\) is a predicate specifying the initial states, \(\text{Next}\) is the transition relation, usually expressed as a disjunction of \(\text{actions}\), and \(\mathcal{F}\) expresses fairness conditions. Weak fairness \(\text{WF}_v(A)\) means that either infinitely many \(A\) steps occur or \(A\) is infinitely often disabled. An action formula describes the changes of state variables after a transition. In an action formula, \(x\) denotes the value of a variable \(x\) in the origin state, and \(x'\) denotes its value in the destination state. UNCHANGED \(x\) means that \(x' = x\).

Functions are primitive objects in TLA\(^+\). The application of function \(f\) to an expression \(e\) is written as \(f[e]\). \([x \in X \mapsto e]\) denotes the function that maps any \(x \in X\) to \(e\), and \([f \ \text{EXCEPT} \ ![e] = e_2]\) is a function which is equal to \(f\) except at point \(e_1\), where its value is replaced with \(e_2\). Tuples (a.k.a. sequences) are functions with domain \(1..n\). Tuples are written \((a_1, a_2, a_3)\). \(\langle\rangle\) is the empty sequence. Modules are used to structure complex specifications. They can extend other modules, importing all their declarations and definitions, or be an instantiation of another module. \(MI = \text{INSTANCE } M\) with \(q_1 \leftarrow e_1, q_2 \leftarrow e_2 \ldots\) is an instantiation of \(M\), where each symbols \(q_i\) is replaced by the expression \(e_i\). \(MI!x\) references \(x\) in the instantiated module.

### 3.2 System Model

A system is composed of a set of indexed peers \(P_1, \ldots, P_n\) and a communication model \(M\) (possibly composite). They are specified using transition systems labelled by communication events.
Communication occurs when matching transitions in $M$ and one of the $P_i$ are synchronized and the $P_j$ ($j \neq i$) stutter. Internal actions $\tau$ can occur without synchronization. Since delivery is stable in our models described in table 1, a minimal progress property (weak fairness in TLA$^+$) prevents infinite stuttering.

Channels are used instead of explicit sender and receiver. Thus, peer transitions are only characterized by the nature of the communication (send "$!" or receive ">") and the concerned channel (e.g. $e^i \rightarrow$). Peer states are characterized by program counters in TLA$^+$ modules (e.g. figure 2). The state of a composition $P_1, \ldots, P_n$ is an array of program counters which carries $P_i$'s state at index $i$. As for $M$, a send (resp. receive) transition accounts for a peer that has sent (resp. received) a message. However, unlike peers, $M$ also requires information about the identity of the peer concerned by the communication operation to guarantee interesting ordering properties. For instance, $3,c^i \rightarrow$ in $M$ is to be synchronized with a $c^i \rightarrow$ transition in $P_3$. A TLA$^+$ module corresponding to a communication model is specified using state variables and transition predicates that implement the rules of message ordering. The notion of listened channels mentioned in 2.1 is also taken into account as additional information in the case of receive transitions, both in the peers and communication models. For instance, in $M$, $2,a^i,a,b,c \rightarrow$ accounts for the reception of a message on channel $a$ by $P_2$ in a context where $P_2$ listens to channels $a$, $b$, and $c$. It is to be synchronized with a $a^i,a,b,c \rightarrow$ transition in $P_2$. Additional fields could easily be added on the transitions to model data passing. However, this would not be relevant when it comes to checking the compatibility of the peers composition with no concern for their inner functioning.

Figure 2 shows an example of a system composed of two peers. The module instantiates the causal communication model to get the send and receive actions. The two peers are respectively initialized in states 11 and 14. Two transitions departs from state 14, depending on the reception channel.

### 3.3 Causal Communication Model ($M_{causal}$) Implementation

As an example, figure 3 shows the TLA$^+$ module corresponding to an implementation of the causal communication model $M_{causal}$. The causal order [Lam78] is the weakest partial order which contains both the peer local order and the send-receive order. Message histories are used to keep track of this order. The state variables are $net$ the set of in transit messages, and $H$ the array composed of a history (message set) for each peer. A message consists of a channel, the sender id, and a snapshot of the sender’s history at send. The reception predicate requires that no in transit message (whose channel is listened to) appears in the history snapshot of the to-be-received message. Thus it guarantees that messages are not received in an order that would violate the causality of their emission. The local history of each peer is updated in a way that describes the causal ordering: at send and receive, the message is added in the peer local history (see "∪{message}" at $\alpha$ and "∪{($c1,p1,h1$)}" at $\beta$); the link between send and receive is performed when merging the message’s history to the receiver’s local history (see "∪h1" at $\beta$).

---

1 Messages are unique because a peer history is strictly increasing.
EXTENDS Naturals, peermanagement
CONSTANTS N
VARIABLES net
Vars := ⟨peers, net⟩
Com := INSTANCE causal WITH CHANNEL ← {"a", "b"}

Init \(=\) Com \& Init \& peers = \{11, 14\}  Initial states: First peer: state\(_{11}\) Second peer: state\(_{14}\)

First peer: a\(_{1}\).b\(_{1}\)
\(t1(\text{peer}) =\) trans(\text{peer}, 11, 12) \& Com ! receive(\text{peer}, "a", \{"b", "a"\}) \hspace{1cm} \text{First peer: state}_{11} \rightarrow \text{state}_{12}
\(t2(\text{peer}) =\) trans(\text{peer}, 12, 13) \& Com ! receive(\text{peer}, "b") \hspace{1cm} \text{First peer: state}_{12} \rightarrow \text{state}_{13}

Second peer: a\(_{2}\).b\(_{2}\)
\(t3(\text{peer}) =\) trans(\text{peer}, 14, 15) \& Com ! receive(\text{peer}, "a", \{"b", "a"\}) \hspace{1cm} \text{Second peer: state}_{14} \rightarrow \text{state}_{15}
\(t4(\text{peer}) =\) trans(\text{peer}, 15, 16) \& Com ! receive(\text{peer}, "b", \{"b"\}) \hspace{1cm} \text{Second peer: state}_{15} \rightarrow \text{state}_{16}
\(t5(\text{peer}) =\) trans(\text{peer}, 14, 17) \& Com ! receive(\text{peer}, "b", \{"b", "a"\}) \hspace{1cm} \text{Second peer: state}_{14} \rightarrow \text{state}_{17}

Fairness \(=\) \(\forall i \in 1 \ldots N : (WF\_{\text{Vars}}(t1(i)) \& WF\_{\text{Vars}}(t2(i)) \& WF\_{\text{Vars}}(t3(i)) \& WF\_{\text{Vars}}(t4(i)) \& WF\_{\text{Vars}}(t5(i))) \) ∧ WF\_{\text{Vars}}(Com \& internal \& UNCHANGED peers)
Next \(=\) \(\exists i \in 1 \ldots N : (t1(i) \lor t2(i) \lor t3(i) \lor t4(i) \lor t5(i)) \lor (\text{Com} \& \text{internal} \& \text{UNCHANGED peers})\)
Spec \(=\) Init \& □[Next] \& Fairness

Figure 2: Generated TLA\(^{+}\) Module: \(\rightarrow a_{1} \rightarrow b_{1}\) Composed with \(\rightarrow a_{2} \rightarrow b_{2}\)

3.4 Compatibility

We define two universal peer states: 0 the terminal state, and \(\perp\) the faulty state. 0 characterizes a peer that has reached a point where the tasks it was supposed to perform are done. \(\perp\) is reached after an unexpected reception (that is to say a reception, imposed by the communication model, that is not correctly handled by a peer). Whether a transition leads to 0, \(\perp\), or another state is part of the peer specification. A compatibility property is given as an LTL formula. We denote \(s_{i}\) the state of peer \(i\) and the following predicates are defined:

\[ 0_{\forall} \triangleq \forall i \in 1..n : s_{i} = 0 \quad \text{peers are all in the terminal state} \]
\[ 0_{i} \triangleq s_{i} = 0 \quad \text{termination of peer} \ i \]
\[ \perp_{\exists} \triangleq \exists i \in 1..n : s_{i} = \perp \quad \text{an unexpected message has been delivered} \]

The following compatibility properties are defined:

**System Termination** The system always reaches a terminal state. \(System \models \Box 0_{\forall}\)

**Peer Termination** Peer \(i\) always reaches a terminal state. \(System \models \Box 0_{i}\)

**No faulty receptions** No unexpected reception ever occurs. \(System \models \Box \neg \perp_{\exists}\)

**No forever blocking communication** At any time, at least one communication event is possible (except if terminated or after a faulty reception).

\(System \models \Box (0_{\forall} \lor \perp_{\exists} \lor ENABLED(R))\) where \(R\) is the system transition relation.

In the figure 2 example: if we replace states 13 and 16 by 0, and 17 by \(\perp\), \(M_{\text{causal}}\) makes it impossible to reach \(\perp\) and the four mentioned compatibility properties hold.
EXTENDS Naturals, FiniteSets

CONSTANTS CHANNEL, N

VARIABLES net, H

Init := ∨ net = {} ∧ H = { i ∈ 1..N → { }}

TransitingMessages := net ≠ {}

send(peer, chan) := Emission from peer of a message on channel chan

let message := ⟨ chan, peer, H[peer] ⟩ IN

∧ net' = net ∪ { message } \ The message is added to the network
∧ H' = [ H EXCEPT ![peer] = H[peer] ∪ { message } ] α : This send is made part of the peer history

receive(peer, chan, listened) := Reception, on peer, of a message on chan

∃ ⟨ c1, p1, h1 ⟩ ∈ net : There is an in transit message such that
∧ c1 = chan \ The channels match
∧ ¬ ∃ ⟨ c2, p2, h2 ⟩ ∈ net : c2 ∈ listened ∧ ⟨ c2, p2, h2 ⟩ ∈ h1 \ No in transit message of interest is in conflict
∧ net' = net \ ⟨ c1, p1, h1 ⟩ \ It is retrieved from the in transit messages
∧ H' = [ H EXCEPT ![peer] = H[peer] ∪ { message } ] β : The peer history is updated

Figure 3: TLA+ Module Associated to the Causal Communication Model

3.5 User Friendliness

Explicitly defining even quite simple peer transition systems can be cumbersome. One may want to step back and provide more abstract specifications. Besides, compatibility checking requires information about terminal and faulty states. The proposed framework provides alternate ways to assist peers specification.

3.5.1 Peer Alternative Specification

A peer can alternatively be defined by a process specified with a CCS term. The peer transition system is derived from the CCS term using the standard CCS rules [Mil99, p.39] and excluding the synchronous communication rule. The translation from a CCS term to a transition system is achieved through the Edinburgh Concurrency Workbench [CPS93].

3.5.2 Faulty Reception Completion

The faulty receptions completion (FRC) consists in revealing the unexpected receptions in a peer and mark them as faulty by adding a corresponding transition toward ⊥. It makes peers fit the intuitive viewpoint where the communication medium impose messages: if a peer is not interested in a channel at a given time, it will never be later.

For each state s where a receive transition exists, the future channels of s is the set of channels corresponding to possible future receptions. For each channel c in the future channels that is not already specified as an alternative choice in this state, such a choice is provided by a transition towards ⊥ and labeled by c? : s ⊢ c⊥. These are called faulty receptions.

For instance, let us consider the peer represented on the left in figure 4. The future channels are indicated next to each state. When there is no departing reception of a future channel, a faulty
transition is added which results in the peer represented on the right.

When composed with a peer $a! \cdot b! \cdot c! \cdot 0$ and $M_{1-1}$, the faulty receptions are impossible (because the send order must be respected) and the peer always ends up in the far-right state (which may be of interest; e.g. 0 the terminal state). This cannot be guaranteed with $M_{async}$ for example.

### 4 Experiments and Results

This section presents a concrete example which illustrates the interest of a diversity of asynchronous models, and some benchmark results which show the usability of the framework for larger systems.

#### 4.1 Practical Example

Let us consider an examination management system composed of a student, a supervisor, a secretary, and a teacher. When the supervisor notices that a student has failed and can resit, he
Figure 6: Supervisor-Secretary-Student-Teacher Specification

\[ \begin{align*}
\text{Supervisor} & \triangleq \text{studentname}! \cdot \text{studentname}! \cdot \text{resit}! \cdot (\text{ok}? \cdot 0 + \text{ko}? \cdot \text{cancel}! \cdot \text{mark}! \cdot 0) \\
\text{Secretary} & \triangleq \text{studentname}? \cdot \text{mark}? \cdot 0 \\
\text{Student} & \triangleq \text{resit}? \cdot (\tau \cdot \text{ko}? \cdot 0 + \tau \cdot \text{StudentOK}) \\
\text{StudentOK} & \triangleq \text{ok}! \cdot \text{examreq}! \cdot \text{materials}? \cdot \text{exam}? \cdot \text{answers}! \cdot 0 \\
\text{Teacher} & \triangleq \text{studentname}? \cdot (\text{cancel}? \cdot 0 + \text{examreq}? \cdot \text{TeacherExam}) \\
\text{TeacherExam} & \triangleq \text{materials}! \cdot \text{exam}! \cdot \text{answers}? \cdot \text{mark}! \cdot 0
\end{align*} \]

Figure 7 presents the results. It confirms that causality is needed to ensure compatibility of the composition but not required over the whole set of channels. The considered composite model is restrictive enough. In this example, with that composite communication model, model checking generates 135 distinct states.

 sends the name of the student to the teacher and the secretary, and the resit information to the student. If the student chooses to resit, he answers ok and asks the teacher for the exam. The teacher then sends the needed materials and then the exam, after which the student sends back his answers, then the teacher sends a mark to the secretary. If the student declines to resit, he informs the supervisor who sends a cancel message to the teacher and the former mark to the secretary. Sample executions are depicted in figure 5 and the system is specified in figure 6.

We consider the models defined in table 1 and the composite model \( M_{\text{comp}} \):

\[
M_{\text{comp}} = \begin{cases} 
M_{\text{causal}} & \{ \text{studentname, resit, examreq, cancel, mark} \\
M_{1-1} & \{ \text{materials, exam} \\
M_{\text{async}} & \{ \text{ok, ko, answers} \} \text{ (no constraint)}
\end{cases}
\]

In this example, \textit{studentname} is a channel over which two messages are sent and from which they are received by different services (teacher and secretary). In addition, \textit{mark} is a channel over which only one message is to transit, but it may be emitted by different services (supervisor and teacher). Therefore, compatibility, especially termination of the secretary service, is not trivial. Consequently, in addition to the generic compatibility properties defined in 3.4, we also consider the termination of the secretary and we check if all messages have been received upon full termination.

Consider the properties needed to make this work as intended. There is a causal dependency between the \textit{studentname} message and the \textit{examreq} message (the request for the exam must not arrive before the student name). This causal dependency comes from the \textit{resit} message, which follows the \textit{studentname} message and is the cause of the \textit{examreq} message. Causal communication is thus required. Moreover, if a \textit{cancel} message is sent, it should be received after the student’s name by the teacher. Therefore, \textit{cancel} is part of this causal group. The same holds for the \textit{mark} channel, since the secretary first expects a \textit{studentname}. Finally, the \textit{materials} and the \textit{exam} are sent in two separate messages and are not expected to be received in the reverse order by the student.

Figure 7 presents the results. It confirms that causality is needed to ensure compatibility of the composition but not required over the whole set of channels. The considered composite model is restrictive enough. In this example, with that composite communication model, model checking generates 135 distinct states.
4.2 Benchmarking

**Studied system** We study the composition of two peers derived and completed from the following CCS terms: $[(a_1!\ldots\cdot a_N!\cdot b?)^M$ and $((a_1?\ldots\cdot a_N?)\cdot b!)^M$. It consists in transmitting $M$ series of $N$ messages (emitted in the same order and that can be received in any order) separated by a synchronization message. We check for termination and observe the number of generated states and runtime. This benchmark is relevant to study looping systems consisting in sections where ordering may be crucial, explicitly sequenced by synchronization points. Depending on the communication model, results are expected to vary. Indeed, without constraint ($M_{async}$), all the reception interleavings are possible, while other models like $M_{1-1}$ impose a single path that corresponds to the send order. These are two extreme cases. We rely on a bag-based implementation for $M_{async}$ and a sequence-based implementation for $M_{1-1}$. The tests ran on a machine with $2 \times 4$ cores Intel Xeon CPU E5-2690 v3 at 2.60GHz and 23GiB of RAM.

**Results** The results are presented in figures 8 and 9. They show that the number of states and runtime increase linearly with $M$ the number of critical sequences. They increase exponentially when it comes to $N$ because it accounts for the maximum number of in transit messages at a given time and all the possible receptions that have to be tried during model-checking. $M$ corresponds to the number of repetitions of the scenario, thus the linear profile. These results show that in practice, systems should scale up well because high degrees of chaos, for instance when more
than 20 messages are to transit on a communication medium at the same time (\(N \gtrsim 20\)), are seldom met.

5 Related Work

5.1 Compatibility Checking

Compatibility of services / software components has largely been studied, with two main goals: Can services communicate and provide more complex services? And can one service be replaced by another one (substitutability)?

These two notions of compatibility are different. In the first case, the services must be complementary, whereas in the second case they should provide the same functionality. Classically, either the notion of simulation (as in [ADF08]) or the notion of trace inclusion (as in [CLB08]) is used to express this sameness. In this taxonomy, we can also include different models of failure traces [GGH+10], where refusal sets may be used to model (preservation of) process receiving capabilities and therefore absence of forever pending messages. We are mainly interested in the first problem. Many approaches exist to verify behavioral compatibility of web services or software components.

Different formalisms are used to represent the services: finite-state machines [DOS12, CLB08, BCT04, FUMK04], process algebra [DWZ+06, BCPV04, CPT01], Petri nets [LFS+11, TFZ09, Mar03]. Different criteria are used to represent compatibility: deadlock freedom [DOS12, FUMK04], unspecified receptions [BZ83, DOS12], at least one execution leads to a terminal state [DOS12, BCT04, DWZ+06, LFS+11], all the executions lead to a terminal state [BCT04, BCPV04], no starvation [FUMK04], divergence [BCPV04]. Domain application conditions are also used [CLB08, CPT01]. The communication models used are synchronous [DOS12, BCT04, FUMK04, DWZ+06, BCPV04, CPT01] or FIFO n-1 [BBO12, OSB13].

On the specific point of faulty reception completion (section 3.5.2), this is reminiscent of Brand and Zafiropulo’s unspecified reception approach [BZ83]. In their work, if a state can
receive a given message, then a successor state (accessible via send events) must also accept this message. In other words, for a system to be correct w.r.t. unspecified reception (and thus for compatibility), if a message can be received at a given state, its reception must also be specified at later states. In our work, we reverse the proposition: if a message can be received at a given state, the communication model may deliver it earlier and the system must expect this situation. The faulty reception completion ensures that fault transitions are introduced to get this property.

To sum up, although some works use several compatibility criteria, all of them are dedicated to one communication model, mostly the synchronous model. None of them proposes a verification parameterized by both the compatibility criteria and multiple communication models. Moreover, only a few approaches also provide a tool to automatically check the proposed composition. Compared to these works, we propose a unified formalization of several communication models and compatibility criteria, and a framework which makes it possible to check the correctness of a composition in a unified manner, using any combination of the communication models. Lastly, the prototype tool returns an invalid execution counterexample when a compatibility criterion is not met.

5.2 System Description

5.2.1 IO Automata

Input/output automata [Lyn96] provide a generic way to describe components that interact with each other thanks to input and output actions. Those actions are partitioned into tasks over which fairness properties can be defined in the same way fairness properties can be set over TLA+ actions. Components can either describe processes or communication channels. They can also be composed and some output actions can be made internal (hiding) in order to specify complex systems. IO automata can model asynchronous systems in a broad sense. IO automata provide a powerful framework to describe distributed systems, but are less practical to verify properties about them. Furthermore, few tools have been developed to make use of IO automata and perform modeling and property checking.

5.2.2 Process Calculi

One of the interest of process calculi is their algebraic representation which is simple, concise and powerful. The processes are described by a term under an algebra. They are constructed from other processes thanks to composition operators (parallel composition, sequence, alternative. . . ). The basic processes represent elementary actions, which are most often communication operations (send or receive).

CCS [Mi182] is an early and seminal calculus that we chose for its simplicity. Its main disadvantage for our work is that communications are synchronous, so we had to adapt its semantics. Milner also defined the π-calculus [Mi199]. The main difference is the introduction of parameters: channels can be communicated through channels themselves. This can describe systems with dynamic configurations. Still, the π-calculus is also synchronous.

Richer process calculi exist, such as the Join-calculus [FG96] (and its extension to mobility [FGL+96]) based on the reflexive CHAM (CHEmical Abstract Machine) [BB92] and also the Ambient calculus [CG98]. They can describe separated membranes/domains, where pro-
cesses interact with each other within a domain or perform explicit actions to move into or out of domains. These calculi are mainly used to model mobility, distribution, firewalls and security properties. But they are not fitted to our concerns for two reasons. Firstly, modelling distribution is not straightforward (usually a mix of local communications and moves between domains) whereas we want to keep it as simple as possible, as distribution is at the core of our concerns. Secondly, they are not parameterized over communication models and directly encoding them would also be cumbersome.

6 Conclusion and Perspectives

This paper presents a framework to check the compatibility of asynchronously communicating peers. It provides a general approach on the diversity of asynchronous communication and takes part in our current study and comparison of the asynchronous communication models. Their differences indeed play a major role in the compatibility of peer compositions as highlighted by the studied use case. The framework enables one to check concrete examples that hint at similarity between different models or reveal their differences.

In the considered approach, point-to-point asynchronous communication occurs between peers through channels without explicit sender and receiver. A communication model manages the communication events and induces properties on the transmission. Being able to associate channels of a peer composition to different communication models makes it possible to study which setup and which implementations, for given peers and compatibility properties, offer the lowest overhead or the fewest constraints. Formalizing and studying these notions is part of an ongoing work which also aims at automating the process. Extending the asynchronous models by introducing broadcast (analogous to a message consumed by more than one peer) and communication failures (mainly message loss) is planned too.

Finally, thanks to its modular conception and the reliance on transition systems, the framework is easily extensible and adaptable. Alternative ways to ease peer specification using CCS terms and a completion step on the derived transition system have been integrated to the automated toolchain. It accounts for the flexibility and adaptability of the developed tool. Benchmarking results also shows that the tool scales up well.

Bibliography


KriQL: a query language for the diagnosis of transition systems

Khaoula Es-Salhi¹, Siham Rim Boudaoud², Ciprian Teodorov², Zoé Drey² and Vincent Ribaud¹

¹Lab-STICC CNRS UMR 3128, Université de Bretagne Occidentale
20, avenue Le Gorgeu, 29200, Brest, France
Email: essalhi.khaoula.mii@gmail.com, ribaud@univ-brest.fr
²Lab-STICC CNRS UMR 3128, ENSTA-Bretagne
2 rue François Verny, 29200, Brest, France
Email: boudaoud.siham.rim@gmail.com, ciprian.teodorov,zoe.drey@ensta-bretagne.fr

Abstract: The formal verification of a concurrent system with a model-checker provides the user with a counterexample trace when a property is violated; however the problem diagnosis still remain a complex issue. Diagnosis is made difficult for several reasons: the trace conforms to a structure that is internal to the verification tool and hence hard to exploit, the trace yields low-level information, the trace size can be large. Traces cannot be understood without the Labelled Transition System (LTS) underlying the model-checker exhaustive exploration; we designed KriQL: a query language over the LTS graph featuring a blend of set filters and graph-based operations and we implemented some feasibility prototypes. Benchmark results indicate that a hybrid management system using a graph-based database and dedicated data structures should achieve sufficient performances.

Keywords: trace, transition system, query language

1 Introduction

System verification aims to establish that the system under study (SUS) possesses certain properties, such as fairness or reachability. Properties are generally issued from the system’s specification. A defect is found once the system violates one or several of the specified properties. Model-checking is a verification technique that relies on the systematic analysis of all the states of the system to check if the system model (formalized using process algebra or concurrent automata) satisfies the specification (typically expressed using temporal logics). If a state is encountered that violates the specification, the model checker produces a counterexample - an execution path that leads to the undesired state - also called a witness trace.

In this context, the typical diagnosis process is simple: a human or machine troubleshooter provides a description of the problem that occurred, a kind of analysis is performed and the root cause identified in order to provide the user with a remedy to the problem. Yuan et al. resume the paradigm of problem diagnosis : problem description → root cause → solution [YLW⁺06]. When the problem is represented with a witness trace, problem description needs to interpret the trace and this interpretation is challenging for several reasons: the trace conforms to a structure that might be or might not be available with the trace, the interpretation has to deal with the
different levels of details from which the traces are built, in practical settings, the size of the trace can be very large.

Because the states space explored by a model-checker is a graph, graph querying that yields restricted sub-graphs or aggregated results is an helpful companion for problem understanding. The work presented in this paper relies on the research hypothesis that the capability to efficiently query the potential behaviors of the system will ease problem interpretation and will enable the development of diagnosis environments, providing specialized visualizations and analysis tools.

This paper presents KriQL a language for query-based diagnosis. This language focuses primarily on the manipulation of the labelled transition system (LTS) constructed during a model-checking process. KriQL enables the identification, extraction, composition, and inquiry of system states, execution traces and sub-clusters of the LTS (representing symbolically encoded sets of execution traces). At a high-level, our approach merges the graph-theoretic view over the system execution, obtained through model-checking, with trace and state-oriented analysis tools (used traditionally during test-debug approaches [PTP07]). The uniqueness of our approach stems from the reification of the systems behaviors enabling the homogeneous manipulations of execution traces through a dedicated query-language. From a practical point of view, KriQL should be seen as a kernel language offering the high-level facilities needed for understanding and diagnosing complex concurrent systems. The kernel can serve as basis for implementing automated fault-localization strategies, such as the work presented in [GSB07].

To validate our approach, we have integrated the KriQL language into the OBP (Observer-Based Prover) verification environment [DBRL12]. To expose the labeled transition system generated during model-checking in KriQL, there is a need for a storage back-end enabling fast query execution, requirement which is not needed during the model-checking process. As such, the LTS storage infrastructure used by OBP was not adapted to our needs. Hence, for our prototype implementation we have evaluated two alternative solutions using relational and graph databases, respectively. The benchmark results indicate however, the need of a more specialized LTS management system that achieves a better performance trade-off between the runtime of the “state-oriented” and the “transition-oriented” queries in KriQL. Nevertheless, the capabilities offered by KriQL facilitate the understanding of systems exhibiting large number of potential behaviors enabling bridging the gap between automated system verification and diagnosis.

The rest of this paper is organized as follows. Section 2 overviews the motivation behind our approach through a simple example. Section 4 presents the KriQL language semantics and pragmatics. Different implementation architectures are evaluated in Section 5. Section 3 situates our approach with respect to the state-of-the art. This study concludes (Section 6) indicating some future research directions.

## 2 A motivating example

We borrow our example from an invited talk given by Leslie Lamport’s [Lam84] about two neighbours Alice and Bob sharing a yard in an exclusive manner because Alice and Bob pets cannot be together in the yard. Lamport’s solution uses two threads sharing only two boolean variables (two flags in the story), each of which can be written by one thread and read by the other. The mutual exclusion algorithm proposed by Lamport is given in figure 1. Incidentally,
it’s a chivalrous algorithm; Alice has priority over Bob.

Alice:

\[
\text{while (true) }
\begin{align*}
\text{flagAlice} &= \text{up}; \\
\text{while (flagBob == up) skip; } \\
\text{catInYard;} \\
\text{flagAlice} &= \text{down}; 
\end{align*}
\]

Bob:

\[
\text{while (true) }
\begin{align*}
\text{flagBob} &= \text{up}; \\
\text{while (flagAlice == up) }
\begin{align*}
\text{flagBob} &= \text{down}; \\
\text{while (flagAlice == up) skip; } \\
\text{flagBob} &= \text{up}; 
\end{align*}
\end{align*}
\]
\[
\text{dogInYard;} \\
\text{flagBob} &= \text{down}; 
\]

Figure 1: Lamport’s mutual exclusion algorithm

2.1 A typical model-checking approach

For verification purposes, the system based on the mutual exclusion algorithm should be translated into a model specification in a concurrent automata-like language. Alice and Bob automata are given in figure 2. Transitions between states bear Event-Condition-Action expression; for instance the expression \{rain\} \{catInYard == true / AliceCatGoesHome\} means that when the event rain occurs and if the condition catInYard == true is satisfied then the action AliceCatGoesHome is performed. Once a system model is specified, the property to be checked are formalized using a property specification language; for instance the mutual exclusion property may be represented with the predicate not (catInYard and dogInYard). Finally the model checker is run to check the validity of the property in the model specification. In our example, the model checker would check that the property holds in all configurations (there is no state violating the assertion).

According to [BK08], whenever a property is falsified, the negative result may have different causes. There may be a modelling error, this implies a correction of the model. It might be a design error or a property error. In case of a design error, the verification is concluded with a negative result, and the design (together with its model) has to be improved. It may be the case that upon studying the exposed error it is discovered that the property does not reflect the informal requirement that had to be validated. This implies a modification of the property, and a new verification of the model has to be carried out.
2.2 Diagnosis of a modelling error

Suppose that, in the state `checkAlice`, we mistyped the guard `flagA == down` in an assignment `flagA = down` and that we did not assert that Bob process cannot assign the shared variable `flagA`. Running the model-checker will violate the mutual exclusion property because there exists a path in the transition system where, once Bob has leaved the waiting state because Alice lowered her flag, Bob does not check Alice’s flag (because of the mistyped instruction) before to unleash his dog, and if Alice has changed her mind at the same time, she might have raised her flag again, and unleashes her cat before Bob has raised his flag. Hence the model-checker will detect the violation and provides the user with a counterexample.

Suppose that we run a model-checker and got a labelled transition system (LTS), a graph resulting of the exploration of Alice and Bob system with a modelling error and the predicate `mutualExclusion`. Suppose that this cyclic graph is made of 14 configurations and 37 transitions. In order to distinguish LTS or process states, we call `configuration` an LTS state. A LTS configuration holds all information about processes including their state. The model-checker indicates that the mutual exclusion property has been violated for the first time in configuration...
12. The counterexample trace is 0 → 1 → 3 → 6 → 9 → 12.
A textual and verbose detailed description of the counterexample is generally available where we find the value of each component of the automata: state, variable values ... However, the diagnostician might want to see only the value of variables flagA and flagB in the violation state 9 and its predecessor 6. In a pseudo-SQL language, we might write the query as

```
select LTS.confID, Alice.state, Alice.flagA, Bob.state, Bob.flagB
from myTrace where LTS.confID = 9 or LTS.confID.successor = 9,
```

and the results might be

```
ID = 6; Alice : state = catInYard, flagA = true; Bob : state = checkFlagAlice, flagB = true
ID = 9; Alice : state = catInYard, flagA = false; Bob : state = dogInYard, flagB = true
```

Such a result will help the diagnostician to notice that something happened to flagA (that went to down when the process Bob was in state checkFlagAlice) and probably will lead her to the modelling error.

### 2.3 Diagnosis of a design error

Mutual exclusion is only one of several properties of interest. Herlihy and Shavit introduce their book with the same example and list three other properties: deadlock-freedom, starvation-freedom and waiting [HS12]. A possible approach to implement properties in a model-checking approach is the use of observers. An observer is an automaton that monitors the model behaviour in order to verify faults. An observer is composed with the model through a synchronization product, i.e. the observer automaton is added to other automata and all possible states explored in a brute-force manner. The observer state is changing during the exploration and can reach special states, called reject states that denote a violation of the property monitored by the observer.

Alice and Bob are dead-locked if each of them has raised their flag and is waiting until the other lowers its flag. It does not happen in the Lamport’s algorithm from figure 1 because as afore mentioned the algorithm is chivalrous. If Alice and Bob each raise their flags, Bob eventually notices that Alice’s flag is raised, and defers to her by lowering his flag, allowing her cat into the yard [HS12]. The deadlock-free property can be implemented by a small automaton, presented in Figure 3, that changes state when Alice or Bob has raised their flag and enter in a reject state if Bob or Alice raises also their flag\(^1\).

![Figure 3: An observer automaton for checking Alice and Bob deadlock](image)

\(^1\) Actually, the observer rejects when Alice and Bob flags are both up and it might not be a problem if the algorithm can deal with this situation; we assume that the observer detects deadlock for simplicity sake.
Suppose that Bob adopts the same algorithm as Alice: a) he raises his flag; b) when Alice’s flag is lowered, he unleashes his dog; c) when his dog returns, he lowers his flag.

With this design, there are several paths in the transition system where Alice and Bob both raised their flag and are dead-locked. The model-checker will reach the reject state of the observer and produces a counterexample to this reject state.

Suppose that we run again a model-checker and got a LTS graph resulting from the exploration of Alice and Bob system with a design error and the observer deadLock. Suppose that this cyclic graph is made of 13 configurations and 32 transitions. The model-checker indicates that the deadlock observer reached reject for the first time in configuration 4. The counterexample trace is straightforward $0 \rightarrow 2 \rightarrow 4$, that is one of the shortest paths to achieve a deadlock: Alice raises her flag, Bob raises his flag, and they are deadlocked.

In a large trace, an useful query for the diagnostician will highlight the configurations where the observer did change its state (a pseudo-predicate called changed), and a way to figure out some information about the intermediate paths between these highlighted configurations, for instance the size of intermediate paths. In a pseudo-SQL language, we might write the query as select LTS.confID, count(*) from myTrace where deadlock.changed and the results might be

$LTS.confID = 0$
$LTS.confID = 2, count(*) = 1$
$LTS.confID = 4, count(*) = 1$

Such a result helps the diagnostician to focus on configuration 2 and she might query again to get Alice or Bob processes values in configuration 2, its predecessors or successors. The diagnostician proceeds with a mixture of navigation queries changed, successors; selection queries select LTS.configuration and aggregation queries count(*) until she is able to localize and fix the error.

3 Related work

3.1 Context-aware verification

Several model checkers such as SPIN [Hol97], Uppaal [LPY97], TINA [BRV04], have been developed to help the verification of concurrent asynchronous systems. In most if not all model-checking approaches, environmental conditions applying to the system execution (that we call contexts) are included in the system model. Our approach (that we called context-aware verification) chooses to explicit contexts separately from the model. Context-aware verification focuses on the explicit modelling of the environment as one or more contexts, which are then iteratively composed with the System Under Study (SUS). Requirements are expressed either with predicates or with observer automata, as introduced in section 2.3. Requirements are verified within the contexts that correspond to the environmental conditions under which they should be satisfied, each context verification is orchestrated in a fully automatic divide-and-conquer algorithm. The interleaving of these contexts generates a labelled-transition system representing all behaviours of the environment, which can be fed as input to traditional model-checkers. The verification is performed by the tool OBP (Observer-Based Prover) [DBRL12]. As other model-checking research groups, a part of our research effort is dedicated to push further the state
explosion limit but we develop also others tools such as step-by-step simulation, trace search engine or graph visualization of the underlying labelled transition system. All these developments are implemented in a tool kit OBP Observation Engine and are freely available².

Fig. 4 shows a global overview of the OBP approach. The SUS is described using the formal language Fiacre [BBF⁺08], which enables the specification of interacting behaviours and timing constraints through timed-automata. The surrounding environment and properties are specified using the CDL formalism. Different model-checkers can process verification.

### 3.2 Trace query languages

Querying traces can be applied in any domain where execution traces can be recorded. There are two types of query-based debugging, those that operate a posteriori on traces and those where the query is weaved with the source program and parameterize the trace recording.

In [MLL05], authors propose PQL (Program Query Language), a language intended to query sequences of events associated with a set of related objects. They developed both static and dynamic techniques to find solutions to PQL queries. The static analyzer finds all potential matches conservatively using a context-sensitive, flow-insensitive, inclusion-based pointer alias analysis. Static results are also used to reduce the scope of dynamic analysis. The dynamic analyzer instruments the source program to catch all violations precisely as the program runs and to optionally perform user-specified actions. A PQL query is a pattern to be matched on the execution trace and actions to be performed upon the match. Subqueries allow users to specify recursive event sequences or recursive object relations. Such constructions are interesting and might be integrated in the KriQL language. However PQL does not yield aggregate queries as we need.

In [GOA05], authors propose Program Trace Query Language (PTQL), a language based on relational queries over program traces. Produced traces result from an automatically instrumentation by a tool PARTIQLE that monitors particular properties. Given a PTQL query and a Java program, PARTIQLE instruments the program to execute the query on-line. PTQL is a subset of SQL. It does not work on the LTS but on linear traces that play the role of relational tables (for

---

² OBP Language and Tools set website: http://www.obpcdl.org
instance, a trace can be joined with another trace). PTQL does not offer navigation queries as typical graph-based queries that we need.

Classical property specification languages, such as PSL[11], can also be considered as inputs to KriQL features. In this case the verification tools using these languages can be seen as the query execution runtime, and the witnesses produced would be considered as the query result.

4 KriQL: a graph-based query language

Generally, diagnosis encompasses any activity that provides information about the SUS, including analysis, observation, proofs, testing, etc. Merriam-Webster on-line dictionary defines diagnosis as an “investigation or analysis of the cause or nature of a condition, situation, or problem”. However, for this study we use a narrower vision of diagnosis, which is restricted to the analysis of concurrent systems captured through Labelled Transition Systems, and considers that symptoms are materialized by traces. This setup corresponds to a model-checking verification approach. In this context, we consider that the semantics of the system was captured explicitly in a LTS, and that the symptoms are exhibited through counter-example traces. In his Turing Award lecture [CES09], Edmund Clarke emphasized that “interpreting long counterexamples” is still an open problem hindering the wide usage of model-checking (especially in industrial settings).

To address this problem, in this paper we propose the definition of a specialized query language that enables the uniform manipulation of the state-space and the counter-example as a set of traces. This approach complements the model-checking toolkit with the means for exploratory analysis (manual or automated) of model-checking results to facilitate the understanding, the localization and the isolation of the defects witnessed through the counter-examples.

In this section, we present KriQL, a query language operating on traces and the underlying transition system. KriQL is a front-end tool working on traces issued from our exploration engine called OBP, presented in the section 3.1. An overview of KriQL main concepts is presented in the section 4.1 as a meta-model, and its denotational semantics is sketched in the section 4.2. There are several candidate architectures to implement KriQL features that are presented in the section 5.1. KriQL implementation performances and architecture are discussed in sections 5.2 and 5.3.

4.1 KriQL overview

To ease the understanding of traces, we defined KriQL (for Kripke Query Language), a language that is aimed at expressing queries to extract relevant information from the traces produced by the exploration of a SUS. A trace (or path) is a part of the LTS – a Kripke structure [Var07] – representing the exploration graph of the SUS. Queries include the search of a path between two configurations, such as the shortest or longest path between two configurations according to criteria such as the change of variable values or the calculation of the range values of a variable in a given trace. The information resulting from the queries consists of either a subset of states (i.e., configurations) or a subset of transitions representing an excerpt of the traces. This information can be further filtered to show only the data of interest (such as process identifiers, or the evolution of a specific variable in a given trace).
KriQL meta-model

To enable the extraction and the representation of such information, we have defined a meta-model of KriQL, partly illustrated in Fig. 5. The meta-model provides the data structures (classes) that are necessary for an Application Programming Interface that will implement KriQL queries and answers. A ConfigurationSet is a set of configurations, each of which contains the status of a set of behavioral elements representing a subpart of the SUS. Since we work on data produced by the OBP explorer tool, the behavioral elements correspond to Fiacre elements, whether components (as the parallel combination of a set of processes) or independent processes. A TransitionSet is a set of transitions between configurations. A PathSet is a set of paths (i.e., traces), where a path corresponds to an ordered set of transitions. Finally, a Trail is a folded representation of a path. Specifically, it explicits only parts of a path according to filtering criteria given in a query.

![Figure 5: KriQL meta-model](image)

4.2 The KriQL language

The denotational definition of a language consists of three parts [Sch97]: the abstract syntax definition of the language, the semantic domains, and a collection of valuation functions. The valuation functions provide the meaning of the language, by mapping the abstract syntax to the semantic domains.

4.2.1 KriQL abstract syntax

An excerpt of the abstract syntax of KriQL is given below. A query can be defined on any element of an exploration graph, such as configurations, processes or variables. It is composed
KriQL: a query language for the diagnosis of transition systems of the kind of element (Key) to be extracted from the graph, and on a set of conditions (Cond) on this element. Conditions can either be test equalities, or test on whether some variable value has changed (CondCh) or some process state has been visited (CondV) or not.

\[
\text{Query ::= } \text{Get Key where Cond} \\
\quad \mid \text{Query union Query} \mid \text{Query inter Query}
\]

\[
\text{Key ::= Configuration } \mid \text{BE } \mid \text{Process } \mid \text{Component}
\]

\[
\text{Cond ::= ExprBE = ExprBE} \\
\quad \mid ( \text{Cond or Cond} ) \\
\quad \mid ( \text{Cond and Cond} ) \\
\quad \mid \text{not Cond} \\
\quad \mid \text{CondV} \\
\quad \mid \text{CondCh}
\]

\[
\text{CondV ::= ExprP.state[Identifier].Visited} \\
\text{CondCh ::= ExprP.state[Identifier].Changed} \\
\quad \mid \text{ExprBE.Variable[Identifier].Value Changed}
\]

\[
\text{ExprP ::= Process[Identifier]} \\
\text{ExprBE ::= ExprP } \mid \text{Component[Identifier]}
\]

### 4.2.2 KriQL semantic domains

The meta-model of KriQL as defined in Sec. 4.1 provides a specification for the semantic domains of KriQL. We express an excerpt of them in a conventional notation as follows, where a set (e.g., ConfigurationSet) is expressed as a function, mapping an id (e.g., Configuration identifier) to a data structure (e.g., Configuration). Hence, the access to an element identified by \( i \) in a set/function \( s \), is done with the call \( s(i) \).

\[
\text{Domain } v \in \text{EnvVariable} = \text{Id} \rightarrow \text{Value} \\
\text{Domain } p \in \text{Process} = (\text{CurrentState} \times \text{EnvVariable}) \\
\text{Domain } cp \in \text{Component} = \text{EnvVariable} \\
\text{Domain } c \in \text{Configuration} = (\text{Process} + \text{Component}) \\
\text{Domain } envC \in \text{EnvConf} = \text{Id} \rightarrow \text{Configuration} \\
\text{Domain } t \in \text{Transition} = (\text{Id}_{conf\ Source} \times \text{Id}_{conf\ Target}) \\
\text{Domain } envT \in \text{EnvTransition} = \text{Id} \rightarrow \text{Transition}
\]

Additionally, we defined the Result domain to represent the result of queries, which depend on the Key provided by the developer in its query.

\[
\text{Domain } res \in \text{Result} = (\text{EnvConf} + \text{EnvComponent} + \text{EnvProcess})
\]

### 4.2.3 KriQL valuation functions

The definition of valuation functions is provided in [ES15]. For simplicity, we provide here only a sketch of it. Given the abstract syntax of a query \( Q = \text{Get Key where C} \), an example of a valid query is:

\[
\text{Query } Q = \text{Get configuration where} \\
\quad \text{proc[’Alice’].state=catInYard and proc[’Bob’].variable[’flagB’]=up}
\]
We define three valuation functions $Q$ (query), $C$ (predicate), and $E$ (expressions inside a predicate, applying on process states or variable values). When applied on a configuration, the valuation function $Q$ iterates over a set of configurations ($e_c$), selecting those satisfying the predicate $C$: each configuration with identifier $i$ in the set $e_c$ is stored if $C$ applied on it is true, otherwise an undefined value (the $\perp$ symbol) is returned.

\[
Q \cdot \langle e_c \rangle = \text{inEnvConf}(\lambda i. C \cdot (e(i)) \rightarrow e(i) \mid \perp)
\]

\[
E \cdot \langle \text{proc[Id].state} \rangle = \text{getProcessByld}(\langle \text{Id} \rangle, c) \downarrow_1 \quad (1 \text{ selects the first element of the process})
\]

\[
E \cdot \langle \text{proc[Id].Id.var} \rangle = \text{let} p_v = \text{getProcessByld}(\langle \text{Id} \rangle, c) \downarrow_2 \text{ in } p_v(\langle \text{Id.var} \rangle)
\]

where $\text{getProcessByld} : (\text{Id} \times \text{Configuration}) \rightarrow \text{Process}$

\[
C \cdot \langle C1 \text{ and C2} \rangle = (\langle C1 \rangle \text{ and } \langle C2 \rangle) \quad (\text{equals and or are the standard boolean operators})
\]

5 Implementation issues

5.1 Implementing a LTS in a database

Each system to be verified is represented by a design and the LTS structure is related with the design structure. The design structure allows the user to understand traces because it provides the user with structural information about the design: process names, variables names, constants and so on. Because we want traces to be stored in a database, the design structure yields the database structure. In the next subsections, we discuss the possible implementations of the design structure.

5.1.1 Relational DBMS

A relational DBMS stores data as table rows conforming to a database schema. Binding the design structure to a database schema can be accomplished in a generic or a specialized way.

A generic binding is applied to any design structure in the same manner; one set of tables is suitable for hosting data independently of the design structure. Database structure is issued from the structure of model-checker languages (Fiacre and CDL in our case, see 3.1). For instance, a single table Process will host any instance of any process of the SUS. Reference to any particular design construct such a process or a variable name, are provided via parameters as input of any operation.
A specialized binding produces a different set of tables upon a specific design structure. Hence the set of operations is dedicated to the design structure instead of being passed via parameters. A specialized binding needs to be generated each time the design of the SUS is updated. For instance, a specialized binding for the motivating example of section 2 uses two tables AliceProcess and BobProcess, each table hosting all instances of the specified process.

The generic binding will hold all the data in a few tables, while the specific binding will use many smaller tables. Auto-joins and recursive queries will suffer of a bias related to the table size. Hence, despite its conceptual advantage, the generic binding was discarded.

5.1.2 Graph-based database

A graph database stores data as vertices and edges. Because we used the Neo4j system, we will use the Neo4j vocabulary, nodes for vertices and relationships for edges. A graph database does not have a schema as a relational database has. Neo4j language for querying graph database is called Cypher. Cypher queries find data that matches a specific pattern. A Cypher query anchors one or more parts of a pattern to specific locations in a graph using predicates and then flexes the unanchored parts around to find local matches [RWE13]. Cypher queries are small graphs made from real nodes and relationships. Hence domain modelling in a graph database is isomorphic to graph modelling. According to [RWE13], “in a graph database what you sketch on the whiteboard is typically what you store in the database.”

5.1.3 Querying data

Classical graph queries such as shortest path or node reachability depend only on the graph structure. However, recent applications using graph databases require novel queries such as graph pattern matching, keyword search or graph aggregation. In [KWY12], authors state that novel queries raise several challenges: queries integrate both the structure and the attributes of the network; when graphs become complex and large, scalability becomes an issue; due to the lack of fixed schema, it might be infeasible to use conventional SQL or SPARQL framework to answer these queries. Three categories are proposed for novel graph queries: mining queries, matching queries, selection queries [KWY12]. However implementing a LTS in a graph database leads to a special case: the LTS respects an underlying schema that stems from the system and properties models. Hence we divided queries in two categories, those using essentially graph attributes and those using essentially the graph structure.

5.2 Node queries

We call node queries the KriQL operations that process essentially nodes information but might in some cases use transitions information. We classified node queries in three categories:

side-effect free restriction such an operation applies a predicate to a ConfigurationSet source and returns nodes where the predicate is true.

side-effect restriction such an operation monitors an element over a ConfigurationSet source and needs neighbourhood information to process results.
set construction such an operation gathers all values reached by a single or a list of a configuration element.

Relational queries use joins to combine data and dedicated structures such as indexes to improve join performances. Restrictions are typically very efficient in a relational implementation. Set construction operations require auto-join or recursive queries over the same table; it might suffer of poorer performances due to the number of auto-join or related to the tables size.

The concept of query in a graph database is graph traversal. A traversal is the operation of visiting a set of nodes by moving between nodes connected with relationships. The traversal stops when rules stop apply such as a depth size. Traversals are well-adapted for navigation along a path. When the whole graph needs to be traversed because the operator needs to process all nodes from a certain type (such as needed by a restriction operator), we can expect poorer performances of a graph database vs. a relational one.

5.3 Edge queries

We call edge queries the KriQL operations that process essentially edges information but might in some cases use nodes information. We classified edge queries in three categories:

partition such an operation splits a Path in a a sequence of Paths.

path existence such a graph traversal operation searches a path between two nodes over a TransitionSet.

path computation such a graph traversal operation searches exhaustively all paths between two nodes over a TransitionSet.

Edge queries are path walks and will require a massive use of joins in a relational implementation because edges are essentially couples of node identifiers (the source and the destination of the edge) stored in a single Transition table. We can expect that the longer the path walk is, the poorer the performance will be because each step along the walk requires a join.

The strength of a graph database is its ability to move between nodes connected with relationships without performance loss whatever the graph size. Thus we can expect excellent performances of a graph database vs. a relational one.

5.4 Benchmark results

We performed benchmark measurements about node and edge queries with 3 typical queries in each case. We tested 2 different implementations: a relational database (Postgres) and a graph database (Neo4J). For the benchmarks we used several LTS ranging from 100 000 to 1 million states. Table 1 synthesizes the results emphasizing the slowest (dark red cells) and the fastest (light green cells) execution times. The time explosion cells represents queries that exceeded the allocated execution time. In [BERT15] we detailed these results, focusing on a realistic case-study from the automotive domain.

Unfortunately, no implementation were successful for all test cases and we concluded for the necessity of a blended implementation: a specific implementation for node queries and a graph-based implementation for edge queries.
KriQL: a query language for the diagnosis of transition systems

Table 1: Query performances

<table>
<thead>
<tr>
<th></th>
<th>Relational DB</th>
<th>Graph DB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node queries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>side-effect free restriction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>side-effect restriction</td>
<td></td>
<td>time explosion</td>
</tr>
<tr>
<td>set construction</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Edge queries</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>partition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>path existence</td>
<td></td>
<td></td>
</tr>
<tr>
<td>path computation</td>
<td></td>
<td>time explosion</td>
</tr>
</tbody>
</table>

6 Conclusion

A major advantage of model-checking is the production of a counterexample, a trace that provides a detailed witness of how the model violates the property. However, without diagnosis tools, the task is hard to relate the counterexample to its roots cause and progress toward a solution to fix the problem. We presented KriQL a query language over a transition system, and the purpose of this work relies on the research hypothesis that efficient traces query will support better visualization and ease problem interpretation. We need now to perform several usability studies to figure out if and how KriQL features are achieving our objectives.

Bibliography


Approximate Active Learning of Nondeterministic
Input Output Transition Systems
Michele Volpato\textsuperscript{1}\textsuperscript{*} and Jan Tretmans\textsuperscript{2}

\textsuperscript{1} m.volpato@cs.ru.nl, \textsuperscript{2} tretmans@cs.ru.nl

Radboud Universiteit, Nijmegen, The Netherlands

\textsuperscript{2} TNO - ESI, Eindhoven, The Netherlands

Abstract: Constructing a model of a system for model-based testing, simulation, or model checking can be cumbersome for existing, third party, or legacy components. Active automata learning, a form of black-box reverse engineering, and in particular Angluin’s L\textsuperscript{*} algorithm, support the automatic inference of a model from a System Under Learning (SUL), through observations and tests. Most of the algorithms based on L\textsuperscript{*}, however, deal with complete learning of deterministic models, thus being unable to cope with nondeterministic SULs, and always learning a complete and correct model as they are based on equivalence between the SUL and the model. We present an adaptation of Angluin’s algorithm for active learning of nondeterministic, input-enabled, input-output transition systems. It enables dealing with nondeterministic SULs, and it allows to construct partial, or approximate models, by expressing the relation between the SUL and the learned model as a refinement relation, not necessarily an equivalence. Thus, we can reason about learned models being more, or less precise than others. Approximate learning has benefits in model-based regression testing: we need not to wait until a complete model has been learned; with an approximate model \textit{ioco}-based regression testing can start.

Keywords: Approximate Active Learning, Nondeterminism, \textit{ioco}

1 Introduction

Model-based testing, model-driven design, model simulation, model checking: once you have a model of the behaviour of a software component, all kinds of analyses can be performed contributing to the construction of better software in less time. A key problem, however, is the initial construction of a model, in particular for existing, third party, or legacy components, for which no or only limited documentation is available.

Active automata learning helps with automatically inferring a state-based model from the behaviour of a System Under Learning (SUL) by observing and testing that behaviour, i.e., through black-box reverse engineering. Automata learning started with the L\textsuperscript{*} algorithm by Angluin [Ang87], after which several variations and improvements were made, among others, for Mealy machines [RS93, Nie03] for learning models of reactive systems such as controllers and network

\textsuperscript{*} This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO), and which is partly funded by the Ministry of Economic Affairs.
Approximate Active Learning of Nondeterministic IOTS protocols. Most of them, however, concern learning of deterministic systems, which means that (i) they cannot deal with nondeterministically behaving systems, and, (ii) the correctness of a learned model is based on an isomorphism with the SUL. In particular the latter implies that a learned model is either complete and correct, or not correct at all.

We present an adaptation of L* for active learning of nondeterministic, input-enabled, input-output transition systems [Tre96]. Nondeterminism allows to reason about partially correct models, using relations that are not equivalences, e.g., pre-orders. This enables reasoning of learned models being better than another learned model, and thus to approximate the model of the SUL. In particular we want to (i) avoid the equivalence checking step used in L*, which is impossible to implement in practice, given non-exhaustiveness of testing, and, (ii) define a relation between a partially learned model and the SUL, as an invariant through the learning process.

The notion of refinement, for the approximate models, is based on the ioco-theory for model-based testing [Tre96], which defines a framework comprehending a precise notion of conformance in terms of implementation relations that can deal with nondeterminism. The learning approach is based on L*, with corresponding observation table. At any moment during the learning process an under- and an over-approximation model is constructed for which we prove a refinement-relationship to the SUL. After any step the algorithm can be stopped with a correct, but perhaps not precise enough model. Continuation will lead to gradually better models, until the model is considered precise enough. The learning algorithm uses properties of input-output labelled transition systems for optimization. Moreover, it must be assumed that, due to nondeterminism, there exists an oracle that knows when all nondeterministic outputs have been observed. Without this assumption the over-approximation can only be the most imprecise, yet correct model, i.e., chaos. An implementation of such an oracle depends on the system under learning: does observing an output x exclude the possibility to observe the output y after the same input? Does repeating the same sequence of inputs k times produce all observable outputs after that sequence?

Approximate learning can have big advantages in regression testing, for example when testing the complete model is beyond feasibility: because already with a partial model, model-based ioco-regression testing can start.

Related Work This paper improves the results presented in our previous work [VT14] by weakening two assumptions on which the learning of nondeterministic systems was based.

First, we assumed in [VT14] that we can somehow obtain all nondeterministic outputs in a particular state at once (i.e., the set out(SUL after σ); cf. Section 2). In this paper, more realistically, we observe all outputs individually, but we assume the presence of an oracle that knows when all nondeterministic outputs have been observed (e.g., by repeating observations often enough). The difference with [VT14] is that we are now able to deal with incomplete observations, i.e., we can still construct a model even if not all outputs have been observed.

Secondly, we assumed in [VT14] the ability to exhaustively check the equivalence of a learned model w.r.t. the SUL (called equivalence exhaustiveness, or equivalence oracle in [Ang87]). This assumption is completely dropped, implying that we can never be sure anymore to have learned the final, complete model. The assumption is replaced by the property that at any moment during the learning process we can always construct a correct approximation of the complete model, in
the sense of having an ioco-like relation between the currently learned model and the (unknown) complete model.

Active learning of observable nondeterministic finite state machines (ONFSMs) [Ea10, Pa13], attempts to learn a deterministic model of systems behaving as a nondeterministic Mealy machine. Such a behaviour is comparable to a labelled transition system with alternation between inputs and outputs. The work in [MS11] applies learning-based testing to reactive systems by combining incremental learning algorithms with formal requirements specified in temporal logic. The idea of incremental learning is similar to the approximate learning and it is a good candidate to be considered for future work.

2 Preliminaries

Labelled transition systems and ioco relation [Tre96] A labelled transition system is a 5-tuple \((Q, L_I, U, \rightarrow, q_0)\), where \(Q\) is a set of states, \(L_I\) and \(U\) are two disjoint sets of inputs and outputs (labels), respectively, \(\rightarrow\) is the transition relation and \(q_0 \in Q\) is the initial state. We may refer at a labelled transition system by using its initial state. We use \(q \xrightarrow{\lambda} q'\) for \((q, \lambda, q') \in \rightarrow\) and we say that \(q\) enables \(\lambda\). We shorten \(L_I \cup U\) by \(L\). A special label \(\tau\) is used for internal, unobservable transitions. Let \(q, q'\) be states and \(\epsilon\) be the empty sequence, we define \(q \xrightarrow{\epsilon} q' \iff q = q'\) or \(q \xrightarrow{\tau} \ldots \xrightarrow{\tau} q'\). Given a label \(\lambda\), we define \(q \xrightarrow{\lambda} q'\) as \(q \xrightarrow{\tau} p \xrightarrow{\lambda} p' \xrightarrow{\epsilon} q'\) and extend \(\Rightarrow\) for sequences of labels in the usual way. The set \(\text{traces}(q) = \{\sigma \in L^* \mid \exists q' : q \xrightarrow{\sigma} q'\}\) indicates the enabled traces from a state \(q\). We denote the set of states reachable from \(q\) via a trace \(\sigma\) as \((q \text{ after } \sigma) = \{q' \mid q \xrightarrow{\sigma} q'\}\). We say that such states are reached by running \(\sigma\) from \(q\). A system is deterministic iff \(|(q \text{ after } \sigma)| \leq 1\). We write \(\sigma_1 \cdot \sigma_2\) or just \(\sigma_1 \sigma_2\) to denote the concatenation of sequences \(\sigma_1\) and \(\sigma_2\). We extend this notation to sets of sequences in the usual way.

A state \(q\) is called quiescent if \(\forall \lambda \in L_I \cup \{\tau\} : q \xrightarrow{\lambda} q'\) for all \(q' \in Q\). Let \(\delta \notin L_I \cup U\), \(L_\delta\) is defined as \(L \cup \{\delta\}\) and \((Q, L_I, U, \delta \rightarrow, q_0)\) is the labelled transition system \((Q, L_I, U, \rightarrow, q_0)\) to which transitions we add \((q, \delta, q)\) for all quiescent states \(q\). We identify \(\delta\) as quiescence and we sometimes include it in the outputs. The set of suspension traces is \(\text{Straces}(q) = \{\sigma \in L_\delta^* \mid \exists q' : q \xrightarrow{\sigma} q'\}\). The set of outputs, including quiescence, that are enabled in a set of states \(P\) is \(\text{out}(P) = \{\lambda \in L_U \cup \{\delta\} \mid \exists q \in P, q' \in Q : q \xrightarrow{\lambda} q'\}\). In an input-enabled labelled transition system, also called input-output transition system, all inputs are enabled in every state. Given a set \(F \subseteq L_\delta^*\), an input-output transition system \(i\) and a labelled transition system \(s\), we define: \(i \text{ioco}_F s \iff \forall \sigma \in F : \text{out}(i \text{ after } \sigma) \subseteq \text{out}(s \text{ after } \sigma)\) and \(i \text{ioco} s \iff i \text{ioco}_{\text{Straces}(s)} s\). The ioco relation is used for testing the conformance of an implementation w.r.t. a given specification.

Active Learning of Regular Languages Angluin’s \(L^*\) [Ang87] is a well known, efficient algorithm that, given an alphabet \(L\), infers a Deterministic Finite Automaton (DFA) for a regular language over \(L\). First some membership queries, in the form “Is this word in the target language?”, are asked in order to construct an hypothesis DFA. Then an oracle replies to an equivalence query affirmatively if the hypothesis is equivalent to the system under learning (SUL), otherwise it provides a counterexample that can be used to improve the hypothesis, by asking more membership queries, and the process is repeated. The central structure of the \(L^*\) algorithm.
is the **observation table**. The observation table is a triple $\langle S, E, T \rangle$, where $S$ is a prefix-closed set of traces that represent **access sequences** to states of the learned DFA and $E$ is a suffix-closed set of traces that represent **distinguishing sequences** of the states in the learned DFA. The function $T$ maps traces in $((S \cup S \cdot \delta) \cdot E)$ to either **TRUE**, if that trace is accepted by the language, or **FALSE**, otherwise. In $T$ both access sequences and their one letter extensions are considered, because this information is needed in order to construct a valid DFA. If two traces in $(S \cup S \cdot \delta)$ are mapped to the same boolean value by $T$ for each suffix in $E$, then they represent the same state. $(S, E, T)$ is called observation table because one can depict the elements of $(S \cup S \cdot \delta)$ as rows and the elements of $E$ as columns. Each entry $s \cdot e$ in the table is given by the result of $T(s \cdot e)$. We do not give precise definitions for active learning of DFAs. However, in the next section, we will provide detailed definitions for active learning of input-output transition systems.

### 3 Manipulating the Observation Table

In this paper we aim to learn a nondeterministic input-output transition system by using the $L^*$ approach, with some differences: we do not alternate membership and equivalence queries, because we introduce the concept of preciseness that will guide to the termination of the learning process. Furthermore, we assume that the SUL behaves as a (**nondeterministic**) input-output transition system, and for this reason, in the observation table, we store sets of outputs instead of boolean values. We will present our notions of closedness and consistency, which are similar to the ones used in Angluin’s $L^*$. In the rest of the paper we use SUL for both the system under learning and the labelled transition system that represents it.

#### 3.1 Nondeterministic Observation Table

We define a **nondeterministic observation table**, from now on simply **observation table**, as a triple $\langle S, E, T \rangle$ where $S$ and $E$ are non-empty, finite sets of traces over $L_\delta$, prefix-closed and suffix-closed, respectively, and $T$ is a function that maps traces in $((S \cup S \cdot \delta) \cdot E)$ to a subset of $(L_\delta \cup \{\delta\})$. We often use $s$ for elements of $(S \cup S \cdot \delta)$ and $e$ for elements of $E$. A matrix view of the table has the prefixes, i.e., elements of $(S \cup S \cdot \delta)$, as row labels, and suffixes, i.e., elements of $E$ as column labels. Each entry contains the observed outputs after running the related prefix followed by the related suffix on the SUL. An image of $T$ is a set of outputs due to the nondeterministic behaviour of the SUL. Furthermore, at any point in the process of learning, there might be some entries in the observation table whose content is not completely known and others for which we are sure that all the enabled outputs, for that specific trace, have been observed. In the latter case we mark the entry as complete, more precisely, if $T(s \cdot e)$ is marked as complete, then $T(s \cdot e) = \text{out}(\text{SUL after } s \cdot e)$. If an entry $T(s \cdot e)$ is incomplete, i.e., it is not marked as complete, then $T(s \cdot e) \subseteq \text{out}(\text{SUL after } s \cdot e)$.

In the learning process, the table is modified repeatedly, adding rows or columns and changing the content of entries. New entries are not marked as complete and $T$ maps them to the empty set. Given an observation table $\langle S, E, T \rangle$ and a trace $s \in (S \cup S \cdot \delta)$, row$(s)$ denotes the function from $E$ to $2^{(L_\delta \cup \{\delta\})}$ defined by row$(s)(e) = T(s \cdot e)$. Given a trace $s \in S$, if an output $\lambda$ is not in $T(s \cdot \varepsilon)$, because it is either not enabled or not observed yet, then row$(s \cdot \lambda)$ is not defined.
In classic L*, for each entry of the table, it is necessary to ask a membership query in order to fill that entry. While learning an input-output transition system, some entries can be filled automatically, and some others are not needed at all, because of some properties of labelled transition systems. We will present such special cases in the next section.

3.2 Filling the Observation Table

In order to be able to infer a valid input-output transition system from an observation table, also called hypothesis, we need to fill the table. Asking an output query, the analogue of a membership query in Section 2, consists of obtaining an output that is enabled after running a trace $\sigma$ on the SUL: $\text{output}(\sigma)$ gives an output $x$ (including quiescence) such that $x \in \text{out}(\text{SUL after } \sigma)$.

For deterministic systems, replying to this kind of questions is easy. If a trace is enabled from the initial state, then the output obtained from the output query on that trace is the only possible output. An implementation of the output query for deterministic systems just needs to run the sequence, waiting for all the outputs during the process. For nondeterministic systems, on the contrary, given a trace $\sigma$, processing $\text{output}(\sigma)$ is not so trivial. We are not sure that we will observe precisely the outputs that are contained in $\sigma$ while running it. If we assume that there exists a maximum number $k$ such that, after running a trace $k$ times, we are sure of having observed all the possible outputs after that trace, then the output query can be implemented. It’s out of scope of this paper to give an implementation of the output query. We assume that such an implementation exists.

We assume also the existence of another type of queries: the completeness queries. Given a sequence of labels and a set of outputs (the observed outputs so far, after that sequence), if the completeness query replies affirmatively then the set of outputs defines exactly the outputs that are enabled after running that sequence on the SUL. As for the output query, we do not provide an implementation of the completeness query. An approach for implementing it, similarly to [Pa13], assuming that after a given number of output queries on the same sequence, we are sure to have observed all the possible outputs, is testing if that many output queries have already been asked. We couple output and completeness queries in order to fill the observation table.

When an output query is answered, there are more entries in the table that can be updated with the result of that query. The result of $\text{output}(\sigma_1 \cdot \delta \cdot \sigma_2)$ must also be a possible outcome of $\text{output}(\sigma_1 \cdot \sigma_2)$, given that, in a labelled transition system, quiescence is always modelled as a self loop. For this reason we define $\delta^*(\sigma)$ as the smallest set s.t. $\sigma \in \delta^*(\sigma)$ and $\sigma_1 \cdot \delta \cdot \sigma_2 \in \delta^*(\sigma) \Rightarrow \sigma_1 \cdot \sigma_2 \in \delta^*(\sigma)$. We use $\text{update}(S, E, T)$ for populating the observation table. In Algorithm 1, for each incomplete entry $T(s \cdot e)$ of the observation table, if $s \cdot e$ ends with $\delta$ only quiescence, or some inputs, can be enabled, thus we set $T(s \cdot e)$ to $\{\delta\}$ and mark it as complete. If $s$ ends with $\delta$ and the state reachable by the longest proper prefix of $s$ enables only $\delta$, then there will be a $\delta$ self loop in that state, thus $\text{row}(s \cdot \delta) = \text{row}(s)$, Lines 8 to 11. If none of the previous special cases is met, then we ask an output query for the trace $s \cdot e$ and update each entry identified by $s \cdot e$. Afterwards, a completeness query is asked, and the entries are updated accordingly.

Example 1 Figure 1b gives an observation table for the system under learning of Figure 1a. This is not the only possible observation table; to a different nondeterministic reply to an output or a completeness query corresponds a different table. We will show later that the learning
Algorithm 1 \texttt{update}(S, E, T)

1: for each $s \in (S \cup S \cdot L_\delta), e \in E \text{ s.t. row}(s)$ is defined do
2: \hspace{1em} if $T(s \cdot e)$ is marked as complete then
3: \hspace{2em} continue;
4: \hspace{1em} if $s \cdot e$ ends with $\delta$ then
5: \hspace{2em} $T(s \cdot e) \leftarrow \{\delta\}$
6: \hspace{1em} mark $T(s \cdot e)$ as complete
7: \hspace{1em} else
8: \hspace{2em} if $s = s' \cdot \delta \land T(s' \cdot e) = \{\delta\} \land T(s' \cdot e)$ is complete then
9: \hspace{3em} $T(s \cdot e) \leftarrow T(s' \cdot e)$
10: \hspace{1em} if $T(s' \cdot e)$ is complete then
11: \hspace{2em} mark $T(s \cdot e)$ as complete
12: \hspace{1em} else
13: \hspace{2em} output $\leftarrow$ output($s \cdot e$)
14: \hspace{1em} for each $s' \in (S \cup S \cdot L_\delta), e' \in E$ s.t. $s' \cdot e' \in \delta'(s \cdot e)$ do
15: \hspace{2em} $T(s' \cdot e') \leftarrow T(s' \cdot e') \cup \{\text{out}\}$
16: \hspace{1em} if isComplete($s \cdot e$) then
17: \hspace{2em} for each $s' \in (S \cup S \cdot L_\delta), e' \in E$ s.t. $s' \cdot e' = s \cdot e$ do
18: \hspace{3em} mark $T(s' \cdot e')$ as complete

The output queries will reply with the only possible output for each entry: output($\epsilon$) = $\delta$ and output($a$) = $\delta$. Let’s say that the completeness queries for $\epsilon$ and $a$ result in \texttt{FALSE}, then we do not mark those entries as complete. Because of Line 5 we know that $\delta$ will be added in the third entry and the entry will be marked as complete.

![Figure 1](image-url)

(a) An iots $q$, $L_I = \{a\}$ and $L_U = \{a, y\}$
(b) A possible first observation table for $q$. (c) The observation table after it has been stabilized.

If isComplete$(s \cdot e)$ does not reply affirmatively, then we do not know if we have observed all the outputs enabled after $s \cdot e$. This uncertainty can be expressed in the current hypothesis induced by the learning algorithm in either a restrictive way, considering the set as complete, or in a permissive way, considering all outputs, including quiescence, as a possible outcome for $s \cdot e$. These two different approaches lead us to construct two different hypotheses: the restrictive $\mathcal{H}^-$ and the permissive $\mathcal{H}^+$. The hypothesis $\mathcal{H}^-$ is constructed in a way similar to the hypothesis in [VT14], where this uncertainty was not considered. In order to build the other hypothesis, $\mathcal{H}^+$, we need to give a formal notion of what its states are, and how to handle the uncertain transitions. For this reason, we define a function row$^+$, similar to row: given a trace $s \in (S \cup S \cdot L_\delta)$, row$^+$ denotes the function from $E$ to $(2^{L_U \cup \{\delta\}}, \{\text{TRUE,FALSE}\})$ defined by row$^+(s)(e) = (T(s \cdot e), \text{isComplete}(s \cdot e))$. Given two prefixes $s_1, s_2 \in (S \cup S \cdot L_\delta)$ and two suffixes $e_1, e_2 \in E$ we define the equivalence over marked entries of the table as $T(s_1 \cdot e_1) =^+ T(s_2 \cdot e_2)$ if and only if $T(s_1 \cdot e_1) = T(s_2 \cdot e_2)$ and isComplete$(s_1 \cdot e_1) = \text{isComplete}(s_2 \cdot e_2)$. As for the function row, given $s \in S$, if an output $\lambda \notin T(s \cdot e)$, row$^+(s, \lambda)$ is not defined. If row$(s)$ is not defined, also row$^+(s)$ is not, and vice versa.

The two hypotheses $\mathcal{H}^-$ and $\mathcal{H}^+$ are suspension automata [Tre96], label-deterministic ver-
ions of some labelled transition systems, where quiescence has been made explicit. Ideally, we want to stop the learning process when either the set of traces of $\mathcal{H}^-$ or the set of traces of $\mathcal{H}^+$ is equivalent to the set of suspension traces of SUL. Given the nondeterministic nature of labelled transition systems, we are satisfied when we find a good approximation of SUL. We will elaborate more on this concept in Section 5.

### 3.3 Global closedness and consistency

In the classic $L^+$ algorithm, two properties of the observation table are necessary in order to construct an hypothesis: closedness and consistency. In this paper we derive two different hypotheses $\mathcal{H}^-$ and $\mathcal{H}^+$, and for this reason, we need to define similar properties for constructing both of them. **Global closedness**, Definition 1, extends the notion of closedness introduced in [VT14] to rows with cells that are not marked as complete. A globally closed observation table must be closed on both row and row$^+$ functions. Closedness on row$^+$ implies closedness on row, given the definition of row$, thus it is enough to check only the most specific one.

**Definition 1** An observation table $(S, E, T)$ is **globally closed** if

$$\forall s' \in (S \cdot L_\delta) \text{ s.t. row}^+(s') \text{ is defined: } \exists s \in S \text{ such that row}^+(s') = \text{row}^+(s)$$

Note that prefixes whose function row is not defined are not taken into consideration for global closedness. The same is true for global consistency. Two elements in $S$ that are mapped in the same way by row (resp. row$^+$), represent the same state in $\mathcal{H}^-$ (resp. $\mathcal{H}^+$). Thus their one label extensions, in $S \cdot L_\delta$, must also represent the same state.

**Definition 2** An observation table $(S, E, T)$ is **globally consistent** if

$$\forall s_1, s_2 \in S: \text{row}(s_1) = \text{row}(s_2) \Rightarrow \forall e \in L_\delta \cdot \text{row}(s_1 e) = \text{row}(s_2 e) \text{ AND}$$

$$\text{row}^+(s_1) = \text{row}^+(s_2) \Rightarrow \forall e \in L_\delta \cdot \text{row}^+(s_1 e) = \text{row}^+(s_2 e)$$

A globally closed and consistent observation table is called stable. Algorithm 2 stabilizes a given observation table. It checks for global closedness and global consistency. If any of

#### Algorithm 2 Stabilize observation table

1: while not globally closed or consistent do
2: if not globally closed then
3: pick an $s' \in (S \cdot L_\delta)$ such that $\forall s \in S, \text{row}^+(s') \neq \text{row}^+(s) \wedge \text{row}^+(s')$ is defined
4: $S \leftarrow S \cup \{s'\}$
5: $\text{update}(S, E, T)$
6: if not globally consistent then
7: pick $\lambda \in L_\delta$ and $e \in E$ such that the suffix $\lambda \cdot e$ is inconsistent with row or row$^+$
8: $E \leftarrow E \cup \{\lambda \cdot e\}$
9: $\text{update}(S, E, T)$
10: return the stable observation table $(S, E, T)$

the two properties is not valid, the algorithm takes the same actions that are taken for classic closedness and consistency, i.e., either adding some elements to $S$ from $S \cdot L_\delta$, or adding one or more elements to $E$. After each step the table is updated again, because new entries are created. Note that only prefixes whose row function is defined are added to $S$. 
Example 2  Let us consider the observation table of Example 1. It is not globally closed, because row\(^+\) (δ) does not have a representative in S. Thus we add δ to S. Once quiescence has been observed, no other output can be observed again, unless an input is provided. Thus row (δx) can never be defined. After having updated the table, a possible result, due to nondeterminism, is the table of Figure 1c. Now the observation table is both globally closed and globally consistent.

Quiescence Reducibility  The explicit representation of quiescence in suspension automata introduces some properties that must be satisfied by any valid suspension automaton to be suspension-trace equivalent to a labelled transition system. In [Wil07] four properties are identified and in [VT14] it is proven that three of them are always satisfied by the transition systems constructed from any observation table. Even though \(H^-\) and \(H^+\) are new constructions, the main points for the proofs in [VT14] are still valid. The fourth one, quiescence reducibility, needs to be checked on the table before the construction of the two hypotheses, after it has been stabilized. It results in adding some suffixes to \(E\) preserving the observations made so far. We refer the reader to [VT14] for an algorithm that ensures quiescence reducibility.

4 Construction of Hypotheses

If \(T\) maps a trace \(s\cdot e\) to an empty set, then we never observed any output after running \(s\cdot e\). This can happen because \(s\cdot e\) contains rare outputs. Rows with an empty set in the first column represent states from which the output behaviour is not known. Empty entries in the table are, by definition, incomplete. During the construction of hypothesis \(H^+\), they are handled as full entries, where the entire set of outputs, including quiescence, is enabled. For constructing \(H^-\), on the contrary, such entries remain empty and if we would use the same hypothesis construction algorithm of [VT14] we would obtain some states with no output transitions. A labelled transition system, and in particular a suspension automaton, must be non-blocking, i.e., it must be possible to observe an output, or quiescence, in every state. Thus a naive construction of \(H^-\) would produce a non valid suspension automaton. For this reason we consider this kind of unknown behaviour as quiescence, and we add a δ-transition to any state which would have no output transitions. If our guess is proven wrong in a future query, then that query will also provide a valid output to add to the empty entry, allowing us to proceed with a “more” correct model. Otherwise, either our guess was correct, or that part of the system was not easily reachable and it is not possible to derive a better model of it.

The construction of \(H^-\) is given by Algorithm 3. It is similar to the one used in [VT14], the only difference is a quiescent state that collects δ-transitions from states with unknown output behaviour. In order to construct \(H^-\), Algorithm 3 first creates a state for each row in the top part of the table and the quiescent state ∆. Then it adds, for each state and for each input label, a transition from that state to the state identified by the row function. Finally, for each output, a transition is added only if that output is enabled in that state. If the state enables no outputs, it adds a δ-transition from that state to ∆.

Chaotic behaviour  In \(H^+\), an incomplete entry \(T(s\cdot e)\) is treated as a sort of full entry, in which all the outputs that are not yet observed might be observed in future queries. However, the
Algorithm 3 Construct $\mathcal{H}^-$ from a stable and quiescent reducible $(S, E, T)$

\begin{algorithm}
\begin{algorithmic}
\ State $Q$ ← $\{\text{row}(s) \mid s \in S\}$
\ State $Q$ ← $Q \cup \{\Delta\}$
\ add $\Delta \xrightarrow{\delta} \Delta$
\ State $q_0$ ← $\text{row}(\varepsilon)$
\ for each $\text{row}(s) \in Q$ do
\ State $\text{for each } \lambda \in L_I$ do
\ 7: add $\text{row}(s) \xrightarrow{\lambda} \text{row}(s \cdot \lambda)$
\ 8: if $T(s \cdot \varepsilon) \neq \emptyset$ then
\ 9: for each $\lambda \in T(s \cdot \varepsilon)$ do
\ 10: add $\text{row}(s) \xrightarrow{\lambda} \text{row}(s \cdot \lambda)$
\ 11: else
\ 12: add $\text{row}(s) \xrightarrow{\lambda} \text{row}(\Delta)$
\end{algorithmic}
\end{algorithm}

Algorithm 4 Construct $\mathcal{H}^+$ from a stable and quiescent reducible $(S, E, T)$

\begin{algorithm}
\begin{algorithmic}
\ State $Q$ ← $\{\text{row}^+(s) \mid s \in S\}$
\ State $Q$ ← $Q \cup \{\chi, \chi_\delta\}$
\ add $\chi \xrightarrow{\delta} \chi_\delta$ and $\chi_\delta \xrightarrow{\delta} \chi_\delta$
\ for each $\text{row}^+(s) \in Q$ do
\ State $\text{for each } \lambda \in L_I$ do
\ add $\text{row}^+(s) \xrightarrow{\lambda} \text{row}^+(s \cdot \lambda)$
\ State $\text{for each } \lambda \in (L_I \cup \{\delta\})$ do
\ State $\text{if } \lambda \in T(s \cdot \varepsilon) \text{ then}$
\ State $\text{if } \lambda = \delta \text{ then}$
\ State $\text{else if } T(s \cdot \varepsilon) \text{ is not complete then}$
\ State $\text{else}$
\ State $\text{add } \text{row}^+(s) \xrightarrow{\lambda} \text{row}^+(s \cdot \lambda)$
\ State $\text{add } \text{row}^+(s) \xrightarrow{\delta} \text{row}(\Delta)$
\ State $\text{add } \text{row}^+(s) \xrightarrow{\lambda} \chi$
\ State $\text{add } \text{row}^+(s) \xrightarrow{\delta} \chi_\delta$
\ State $\text{add } \text{row}^+(s) \xrightarrow{\delta} \chi$
\end{algorithmic}
\end{algorithm}

Example 3 The table of Figure 1c is stable and quiescence reducible. We can construct $\mathcal{H}^-$ and $\mathcal{H}^+$. The construction of $\mathcal{H}^-$ is easy. There is only one state, with output $\delta$ ($\Delta$ is not observed, table is not able to specify any behaviour for such outputs from a state $\text{row}^+(s)$ if $T(s \cdot \varepsilon)$ is not complete, because $\text{row}^+(s \cdot \lambda)$, where $\lambda$ is an unobserved output, is not defined. For this reason, during the construction of $\mathcal{H}^+$, we keep a non-conservative approach and we allow any behaviour, i.e., we add $\text{row}^+(s, \lambda, p)$ to $\rightarrow$, where $\lambda$ is an output never observed after the query $s \in S$, and $p$ is a chaotic state [BRT04]. Such a chaotic state can be defined in many ways. Figure 2a shows a representation of a chaotic state. The construction of $\mathcal{H}^+$ starts with $\chi$ and $\chi_\delta$ as states. Then more states are added according to $\text{row}^+$. Output transitions whose behaviour is unknown, i.e., whose target state is unknown, will target $\chi$. Transitions labelled with $\delta$ whose behaviour is unknown will target $\chi_\delta$. Figure 2: Chaotic behaviour and the two hypotheses constructed from Figure 1c.

shown in the figure because it is not reachable. All transitions are self loops: Figure 2b. The construction of $\mathcal{H}^+$ is slightly more complex. First we add the chaotic state ($\chi_\delta$ is not drawn in Figure 2c). Then we create two more states, one for $\text{row}^+(\varepsilon)$ and the other for $\text{row}^+(\delta)$, and finally, we add the transitions to the relevant states. From the state identified by $\text{row}^+(\varepsilon)$, we
need to reach the chaotic state for each output that is not in the entry \( T(\varepsilon \cdot \varepsilon) \). This action is not necessary for the state represented by \( \text{row}^+(\delta) \) because \( T(\delta \cdot \varepsilon) \) is complete.

**Theorem 1** Let \((S,E,T)\) be a globally closed, globally consistent and quiescent reducible observation table, and let \( \mathcal{H}^- \) and \( \mathcal{H}^+ \) be the hypotheses obtained with Algorithm 3 and Algorithm 4, respectively, then: \( \mathcal{H}^- \mathcal{ioco} \mathcal{H}^+ \).

### 4.1 Conformance Relation on Observed Behaviour

We provided the construction of two models, based on the observation table, that represent the current information we have been able to infer from observing the behaviour of the SUL. How can we formally describe how they are related to the SUL? We need to introduce a new conformance relation based on that behaviour.

The fact that \( \mathcal{H}^- \) and \( \mathcal{H}^+ \) are constructed from a limited amount of information, i.e., the replies to some output queries, offers the idea of reasoning in terms of relations restricted to that information only. For this reason we define \( \mathcal{ioco}_{(S,E,T)} \) in Definition 3. Informally, a labelled transition system \( i \) is \( \mathcal{ioco}_{(S,E,T)} \) conforming to another labelled transition system \( i' \) if and only if, for all the traces that can be constructed from the table \((S,E,T)\) by concatenating any prefix with any suffix, the set of outputs observable after executing those traces on \( i \) can also be observed after executing the same trace on \( i' \).

**Definition 3** Let \((S,E,T)\) be an observation table, \( i \) and \( i' \) two labelled transition systems, then: \( \mathcal{ioco}_{(S,E,T)} i' \iff \forall \sigma = s \cdot e \text{ such that } s \in (S \cup S \cdot L_\delta), e \in E \land \text{row}(s) \text{ is defined } \land T(s \cdot e) \neq \emptyset : \text{ out}(i \text{ after } \sigma) \subseteq \text{ out}(i' \text{ after } \sigma) \)

The system under learning and the two hypotheses are related accordingly to \( \mathcal{ioco}_{(S,E,T)} \).

**Theorem 2** Let \((S,E,T)\) be a globally closed and consistent observation table obtained from SUL, and \( \mathcal{H}^- \) and \( \mathcal{H}^+ \) be the suspension automata constructed using Algorithm 3 and Algorithm 4 respectively. Then \( \mathcal{H}^- \mathcal{ioco}_{(S,E,T)} \text{ SUL and SUL } \mathcal{ioco}_{(S,E,T)} \mathcal{H}^+ \).

**Proof idea.** First note that \( T(s \cdot e) \subseteq \text{ out}(\text{SUL after } s \cdot e) \). Then the theorem can be proven by showing that \( \mathcal{H}^- \) and \( \mathcal{H}^+ \) are consistent with the observation table.

Note that, fixing \((S,E,T)\), the relation \( \mathcal{ioco}_{(S,E,T)} \) is transitive, thus \( \mathcal{H}^- \mathcal{ioco}_{(S,E,T)} \mathcal{H}^+ \). Furthermore, SUL \( \mathcal{ioco}_{(S,E,T)} \mathcal{H}^- \) implies that each entry of the observation table must be complete and, thus, \( \text{Straces}(\mathcal{H}^-) = \text{Straces}(\mathcal{H}^+) \).

**Theorem 3** Let \((S,E,T)\) be a globally closed and consistent observation table obtained from SUL, and \( i \) be an input-output transition system, then \( i \mathcal{ioco} \text{ SUL } \Rightarrow i \mathcal{ioco}_{(S,E,T)} \text{ SUL} \).

**Proof idea.** It is easy to prove the contrapositive by showing that the concatenation of a prefix \( s \) and a suffix \( e \) for which \( \text{ out}(i \text{ after } s \cdot e) \nsubseteq \text{ out}(\text{SUL after } s \cdot e) \) is in \( \text{Straces}(\text{SUL}) \).

We gave the basis for maintaining an observation table for learning a nondeterministic system and we provided the definitions of the hypotheses that can be constructed from the observation
5 Learning Process

When learning deterministic systems, under the assumption that a correct and sound equivalence oracle exists, it is proven that the learning process ends when the correct model has been learned. Due to nondeterminism, one can never be sure of having observed all the possible outputs after a given trace, thus we do not know when to stop the learning process.

![Flow diagram of the learning process.](image)

In Figure 3 we give a flow diagram that includes the nondeterministic decision to either stop learning and keep the current hypotheses as final, even though they do not represent exactly the system under learning, or continue learning, trying to obtain a more precise hypothesis and postpone the decision of stopping to a later time. The learning process starts with the initialization of the observation table. Then the table is updated, stabilized and checked for quiescence reducibility. After these steps, it might be necessary to update and stabilize the table again, and, subsequently, checking again for quiescence reducibility. Only when the table is stable and quiescence reducible, its preciseness is addressed. If the table is precise enough, then the learning can stop. Otherwise we can try to improve its preciseness. Given the nondeterministic behaviour, it is not always possible to increase the preciseness.

A table that is the result of updating another table, after having used a proper technique, should always be more precise than the initial one. In this section we define a qualitative preciseness relation on tables and show that the update does not decrease the preciseness. We introduce some techniques that attempt to increase it, either directly or by triggering an update of a stable table.

5.1 Preciseness of the Table

The preciseness of the observation table depends on three factors: 1) the size of the observation table, in term of rows and columns, i.e., the cardinality of \((S \cup S \cdot L)\) and \(E\); 2) the cardinality of the set of outputs in each entry; and 3) the number of completed entries.

If two observation tables are comparable, i.e., they are constructed from the same SUL, and the first has a bigger observation table, bigger sets of output for some entries, and more completed entries than the second, then the first is more precise than the second.

**Definition 4** Let \((S, E, T)\) and \((S', E', T')\) be two observation tables. We say that \((S', E', T')\) is
more precise than \((S, E, T)\), written as \((S, E, T) \subseteq (S', E', T')\), if and only if:

\[
S \subseteq S' \text{ and } E \subseteq E' \text{ and } \\
\forall s \in (S \cup S \cdot L_\delta), e \in E : \begin{cases} 
T'(s \cdot e) = + T(s \cdot e) & \text{if } T(s \cdot e) \text{ is complete} \\
T'(s \cdot e) \supseteq T(s \cdot e) & \text{otherwise}
\end{cases}
\]

If an observation table is more precise than another one, then the relation it describes is stronger than the relation implied by the other observation table (Theorem 4). By increasing the preciseness of the table, we increase the strength of the relation.

**Theorem 4** \((S, E, T) \subseteq (S', E', T') \Rightarrow ioco (S, E, T) \supseteq ioco (S', E', T')\)

It is easy to verify that Algorithm 1 and Algorithm 2 do not decrease the preciseness of the table, neither does the quiescence reducibility check.

**Proposition 1** During learning, each table is more (or equally) precise than the previous one.

### 5.2 Techniques for Increasing the Preciseness

Classic L* stops with the exact model of the system under learning: if the current model is not the correct one, there must be a counterexample that will improve the model. In our approach, instead, we are more interested in learning an approximation of the SUL. If the approximation is not good enough we want to make it better. We can achieve this goal by acting on the observation table directly, adding elements to \(S\), \(E\) and \(T(s \cdot e)\) or by testing our current hypotheses.

**Extending \(S\) and \(E\)** The first method for increasing the preciseness of the table relies on directly enlarging the table, i.e., increment the number of rows and columns. The cardinality of \(S\) and \(E\) will therefore grow. Even though this is an easy modification to perform on the observation table, the choice of new elements for those sets is not trivial. It is important to keep \(S\) and \(E\) prefix and suffix closed, respectively. Thus good candidates for \(S\) are elements of \(S \cdot L_\delta\), and good candidates for \(E\) are elements of \(L_\delta \cdot E\). After having added elements to \(S\) and \(E\) the observation table must be updated and stabilized again.

**Updating the Table** Another method that acts on the table is to run Algorithm 1 with the goal of finding new outputs for existing entries, or making the table not globally closed or consistent. The former case increases the preciseness directly, while the latter will result in an increase of the preciseness due to the stabilizing step. This method is less direct than the previous one, because it might result in not increasing the preciseness at all. Consider a table where all entries are completed: updating the table will have no consequences.

**Testing the Current Hypotheses** The last method we mention is analogue to the classic learning process: once an hypothesis has been constructed, an equivalence oracle confirms that it is equivalent to the system under learning. If not, it provides a counterexample. In practice, this is done by testing. The hypothesis is tested for equivalence against the system under learning.

In learning nondeterministic systems, we can use an \(ioco\)-based model-based testing tool to test if the SUL is conforming to one of the hypotheses. We can search for a counterexample by
testing against $\mathcal{H}^-$ or $\mathcal{H}^+$, using either $\text{ioco}_{S,E,T}$ or classic $\text{ioco}$. Testing against $\mathcal{H}^+$ will find less counterexamples than testing against $\mathcal{H}^-$, because SUL $\text{ioco}_{S,E,T}$ $\mathcal{H}^+$, see Theorem 2.

The conformance relation also plays an important role in the conformance query. If $\text{ioco}_{S,E,T}$ is used, then the test is concentrated in finding new outputs for the entries in the table, while classic $\text{ioco}$ can explore the SUL more widely. A reason for using $\text{ioco}_{S,E,T}$ is that it is defined on a finite set of traces, making the testing process exhaustive.

By handling a counterexample, either we add an output in one or more already existing entries of the table, or we add some suffixes to $E$ and prefixes to $S$, increasing the preciseness.

### 5.3 The Learning Algorithm

Combining all the algorithms presented in previous sections results in Algorithm 5. We start by initializing the observation table with a set containing only the empty trace $\varepsilon$ for both prefixes and suffixes. At this point the table has one column, for the empty trace, and a row for each element of $L_\delta$, plus a column for the empty trace. Then we fill it using Algorithm 1 and Algorithm 2. Quiescence reducibility is checked for both row and row+ and, if that results in adding suffixes to the table, we update and stabilize it again and check again for quiescence reducibility. Once we obtain a table and quiescence reducible table we consider whether the table is precise. If it is, then we are done: we construct the two hypotheses $\mathcal{H}^-$ and $\mathcal{H}^+$ and return them. If the table is not precise, we need to use some techniques to increase its preciseness.

**Example 4** The first three examples cover Algorithm 5 until the decision point at Line 12. Let’s assume the preciseness is not high enough. We try to improve the preciseness by updating the table, and as a result we discover that $T(\varepsilon,\varepsilon)$ is complete. The table is not globally closed any more, accordingly to row+: we add a to $S$ and update the table. The table is still not globally closed because of row(aa), thus we add a to $S$ obtaining Figure 4a. The table is globally closed, but not globally consistent. For row we have that row($\varepsilon$) = row(a) but row($\varepsilon a$) = row(a) $\neq$ row(aa). We solve the inconsistency by adding a to $E$. This makes the table not globally closed, because of a$\delta$ and aux which are added to $S$ obtaining the table in Figure 4b. This table is globally closed and consistent. It also is quiescence reducible, thus we can construct $\mathcal{H}^-$ and $\mathcal{H}^+$. Figure 4c shows $\mathcal{H}^-$, while $\mathcal{H}^+$ is shown in Figure 5a. Compared with the previous hypotheses, these new ones contain more information. We want to increase the preciseness of the observation table by testing. We test the SUL against $\mathcal{H}^-$ for $\text{ioco}_{S,E,T}$ and we find that, out(SUL after aa) $\neq$ out($\mathcal{H}^-$ after aa) because we observed a y during our tests. Assume also

---

**Algorithm 5 LearnLTS**

1. $S \leftarrow \{\varepsilon\}$  
   \{Initialize $(S,E,T)$\}
2. $E \leftarrow \{\varepsilon\}$
3. loop
   4. repeat
   5. update$(S,E,T)$ using Algorithm 2
   6. Stabilize $(S,E,T)$ using Algorithm 2
   7. Check quiescence reduc. on row and row+ obtaining suffix-closed sets $E^-$ and $E^+$
   8. $E \leftarrow E \cup E^- \cup E^+$
   9. until $E^- \cup E^+ = \emptyset$
   10. Construct $\mathcal{H}^-$ using Algorithm 3
   11. Construct $\mathcal{H}^+$ using Algorithm 4
   12. if $(S,E,T)$ is not precise enough then
   13. Try to increase preciseness
   14. else
   15. return $\mathcal{H}^-, \mathcal{H}^+$
that we tested the system enough to obtain $\text{TRUE}$ in some entries. A possible observation table could be the one in Figure 5b ($S-L_\delta$ is not shown). If we construct $\mathcal{H}^-$ from Figure 5b, we obtain the SUL, $\mathcal{H}^+$ is shown in Figure 5c. We consider the table precise enough and we stop.

6 Conclusions

We presented an algorithm for learning nondeterministic input-output transition systems by using an active learning $L^*$-style approach. The observation table has been modified to handle nondeterminism and unknown behaviour. We defined two different hypotheses, that can be derived from the modified observation table, which are able to describe the unknown behaviour in two different ways. We also adapted the properties that the table must satisfy for successfully inferring such hypotheses. The hypotheses are an under and an over-approximation of the SUL according to a newly defined relation $\text{ioco}_{(S,E,T)}$. We uncoupled the membership and equivalence queries, used in classic $L^*$, by following a learning process based on preciseness of the observation table: the learning stops, always with an $\text{ioco}_{(S,E,T)}$ conforming model, when the table is considered precise enough, otherwise some actions can be taken to increase its preciseness. Thus, a conformance test, analogue of the equivalence query, is not used directly in the learning process, but only as a mechanism to increase the preciseness. Stopping without reaching an isomorphism of the system under learning, contrary to $L^*$, allows to obtain a valid, conforming, but approximate model which can be used to start (regression) testing.
Future Work  In this paper we focus on qualitative approximation for the learned model. As future work we want to quantitatively define the preciseness using some measures. A possible measure is given by a combination of the size of $S$ and $E$, the number of outputs in the table and the number of complete entries, avoiding redundancy (see Algorithm 1). Another measure we are considering is the discounted reachability of $\chi$ when $\mathcal{H}^+$ is considered as a Markov chain. After each modification of the table such measures are calculated and the learning process might end, because a certain value for the preciseness has been reached, or because the preciseness is not changing rapidly any more.

Moreover, we wish to investigate which hypothesis, $\mathcal{H}^-$ or $\mathcal{H}^+$ is better for regression testing, and for other testing and verification techniques where the use of a model is of help. Another topic concerns the techniques used in incremental learning-based testing [MS11] and their adaptation to active learning of nondeterministic systems. Finally, an experimental evaluation of our approach is necessary to assess its practical feasibility.

Bibliography


Vincent Rahli, David Guaspari, Mark Bickford, and Robert L. Constable†

Cornell University

Abstract:
Distributed programs are known to be extremely difficult to implement, test, verify, and maintain. This is due in part to the large number of possible unforeseen interactions among components, and to the difficulty of precisely specifying what the programs should accomplish in a formal language that is intuitively clear to the programmers. We discuss here a methodology that has proven itself in building a state of the art implementation of Multi-Paxos and other distributed protocols used in deployed database systems. This article focuses on the basic ideas of formal EventML programming illustrated by implementing a fault-tolerant consensus protocol and showing how we prove its safety properties with the Nuprl proof assistant.

Keywords: functional programming formal methods formal verification theorem proving distributed systems fault tolerance event logic event-based programming

1 Introduction
Protocol Specification, Verification, and Synthesis. There is good evidence that appropriate formal methods can substantially improve the reliability of distributed protocols and that such methods are especially valuable for this kind of programming because of its intrinsic complexity. We have invested in this line of work for several years, using constructive logic because it supports provably correct code synthesis from proofs and because aspects of distributed computing are essentially constructive: agents make decisions according to some local information, and a protocol specifies how that information is acquired. “Provably correct” here means that machine checked proofs guarantee that programs satisfy desired correctness properties.

One reason that distributed systems are especially difficult to code correctly and maintain is that there are many intricate failure scenarios to consider. Failure scenarios can be hard to generate and testing them all is not usually possible. Model checkers are often used to verify that distributed systems are correct [29, 1, 19]. However, only models of the actual code are verified correct, and such tools may not be able to exhaustively search the space of failure scenarios. Proof assistants, however, allow one to provide definitive arguments.

We use the EventML language to develop protocols. EventML works synergistically with the Nuprl proof assistant [12, 2] which is closely related to the Coq [4, 13] proof assistant. Nuprl is a programming/logical environment based on Constructive Type Theory (CTT) [12, 2], that allows one to both prove mathematical results, and program and prove properties about these programs.

EventML. EventML is a domain-specific ML-like functional programming language for distributed protocols based on asynchronous message passing. It allows programming distributed

† This work was partially supported by the DARPA CRASH project, award number FA8750-10-2-0238, by the SnT, and by the National Research Fund Luxembourg (FNR), through PEARL grant FNR/P14/8149128.
programs in an event-based style, hence the name “EventML”. The language provides *combinators* to implement what can be regarded as event recognizers and event handlers. EventML is based on two formal models of distributed computing implemented in Nuprl: (1) the Logic of Events (LoE) [5, 7] to specify and reason about the information flow of distributed program runs, (2) a General Process Model (GPM) [6] to implement these information flows. The semantic meaning of an EventML program is expressed both by a LoE formula and a GPM program. Because of this dualism we also refer to EventML programs as *constructive specifications*.

Currently, EventML *docks* with Nuprl, but in principle can connect to any prover that implements LoE and GPM. Because every EventML type is a Nuprl type, docking means that any Nuprl expression whose type is an EventML type can be imported into an EventML program.

The diagram below shows the interaction between EventML and Nuprl. Once we have extracted the semantic meaning of an EventML specification in terms of a LoE formula and a GPM program, we automatically prove that the program satisfies the formula. It remains to interactively prove that the LoE formula satisfies the desired correctness properties.

**Computation Model.** EventML’s computation model is based on GPM. A process that takes inputs of type $A$, and outputs elements of type $B$, is an element of the following co-recursive type (the definition of the Nuprl `corec` type is outside the scope of this paper): $\text{corec}(\lambda P. (A \to P \times \text{Bag}(B)) + \text{Unit})$. Unit is a singleton type and $+$ is the disjoint union type. Therefore, a process is one of two things: a function that given an input of type $A$, generates a (possibly empty) bag of outputs of type $B$, and becomes a possibly different process; or a special value, which we call `halt`, that is used to denote a terminated process. Because GPM is implemented in Nuprl, a process is a Nuprl program (i.e., an expression of Nuprl’s programming language, an untyped $\lambda$-calculus) that can be executed by interpreting it according to the rules of Nuprl’s computation system.

**The Logic of Events.** The Logic of Events (LoE) [5, 7], related to Lamport’s notion of causal order [21], was developed to reason about events occurring in the execution of a distributed system. LoE has been used among other things to verify consensus protocols [28] and cyber-physical systems [3]. In the context of this paper, an event is an abstract entity corresponding to the receipt of a message$^1$; the message is called the *primitive information* of the event. An event happens at a specific point in space/time. The space coordinate of an event is called its location, and the time coordinate is given by a well-founded causal ordering on events that totally orders all events at the same location. Using LoE one can describe systems in terms of the causal relations among events and (ultimately) their primitive information.

**Event orderings.** To reason about a protocol in LoE, we reason about its possible runs. An *event ordering* is an abstract representation of one run of a distributed system; it provides a formal definition of a *message sequence diagram* as used by systems designers. It is a structure consisting of: (1) a set of events; (2) a function $\text{loc}$ that associates a *location* with each event;

---

$^1$ Events are formally more general than that in the sense that they might correspond to something else than just the receipt of messages.
We express system properties as predicates on event orderings. A system satisfies such a property if every execution satisfies the predicate.

The message sequence diagram on the right depicts a simple event ordering. Event $e_1$ corresponds to the receipt of a message with header "echo" at location $L_1$. Upon receipt of that "echo" message, $L_1$ forwards it to $L_2$, which causes $e_2$. Upon receipt of that "forward" message, $L_2$ sends an acknowledgment to $L_1$, which causes $e_3$. Events $e_1$ and $e_3$ have same location, and $e_1$ happens causally before $e_2$, which happens causally before $e_3$. We write $e_1 < e_2 < e_3$, and $e_1 < loc(e_3) (e < loc(e')$ is defined as $e < e' \land loc(e) = loc(e'))$.

**Event observers.** In LoE, we specify systems by defining and combining event observers [5]. An event observer is a function that assigns to any event ordering $eo$ and event $e$ in that event ordering $eo$, an unordered bag of outputs observed (or produced) at $e$. For example, the following event observer of type $Obs(Loc)$, where $Loc$ is the type of locations, recognizes every event and observes its location: $\lambda eo.\lambda e.\{loc(e)\}$. Event observers can therefore be regarded as combinations of event recognizers and event handlers. They effectively partition events into those they "recognize" by associating values to those events, and those they do not. For example, the base observer denoted $vote'base$ recognizes the arrival of any message with header "vote" and handles that event by simply returning the content of the message. We may define another observer, call it $X$, which recognizes that, in the context of some protocol, certain "vote" messages signify that the protocol has completed and will assign to such an event a value that means "send the 'done' message to $Y'." $X$ will recognize some but not necessarily all "vote" events; and the values that it assigns to them differ from the values assigned by $vote'base$. We specify systems in LoE and EventML by defining and combining such event observers that appropriately classify system events.

We reason about event observers in terms of the event observer relation, which relates events, observers, and observations: we say that the event observer $X$ observes $v$ at event $e$ (in an event ordering $eo$), and write $v \in X(e)$, if $v$ is a member of the bag $(X eo e)$. For readability, our notation suppresses $X eo e$. $X$ recognizes $e$ when $(X eo e)$ is nonempty, in which case we also say that $e$ is an $X$-event.

An EventML specification describes event observers that produce and consume messages (among other values), and especially, it describes a main observer that specifies the entire information flow of a system. Main observers output directed messages represented by pairs location/message. Given a directed message $(l,m)$, the communication system attempts to deliver message $m$ to location $l$. This directed message can be seen as the instruction "send message $m$ to location $l". By default, in our framework (more precisely in LoE), we assume that messages are delivered reliably but asynchronously, and may be delivered more than once.

**Automation.** Formally verifying distributed protocols is not trivial and can be time consuming. However, because we are using a tactic-based proof assistant in the style of LCF [18], there is much room for automation. We have built two main automation tools to assist us in proving properties of distributed systems.

From an EventML specification we automatically generate an inductive logical form (ILF),
a first order formula that characterizes the observations made by the main observer in terms of
the event observer relation. It characterizes the response to any event \( e \) in terms of observations
made at some causally prior event \( e' < e \). ILFs are the heart of our verification method, providing
a powerful way to prove program properties by induction on causal order.

In addition, we have automated some patterns of reasoning on state machines, because typical
specifications are composed of several small state machines.

**Contributions.** This paper introduces basic ideas of EventML, which implements a program-
ing paradigm in which programmers can flexibly use proof assistants to develop verified dis-
tributed programs. We show how EventML can be used to (1) define a non-trivial fault-tolerant
consensus protocol in Sec. 2, (2) prove the safety properties of this protocol using automation
tools in Sec. 3, and (3) generate a verified implementation in Sec. 4. Even though we illustrate
our methodology on a simple consensus protocol, we have successfully used this methodol-
yogy to implement industrial strength fault-tolerant distributed protocols such as Multi-Paxos.

2 A Specification of 2/3 Consensus

Consider the following problem: A system has been replicated for fault tolerance. It responds to
commands identified by values in some type \( \text{Cmd} \), a parameter of the specification. Commands
are issued to any of the system replicas, which must come to consensus on the order in which
those commands are to be performed, so that all replicas process commands in the same order.
Replicas may fail.\(^3\) We assume that all failures are crash failures, that is, a failed replica ceases
all communication with its surroundings. The 2/3 consensus protocol \([8]\) tolerates up to \( F \) fail-
ures (another parameter of the specification) by using \( 3 F + 1 \) replicas. (An appealing feature of
the protocol is that with a small change, and using \( 5F + 1 \) replicas, it can tolerate Byzantine fail-
ures.) Input events communicate proposals, which consist of slot number/command pairs. The
slot numbers are modeled by integers: \( (n, c) \) proposes that command \( c \) be the \( n^{th} \) one performed.
The protocol is intended to decide which proposals to accept, and to broadcast those decisions to
clients (whose locations are also a parameter of the specification). Each copy of the replicated
system contains a module that carries out the consensus negotiations. To save space, this paper
describes only those modules (which we continue to call \textit{Replicas}). An account of how these
consensus decisions are used may be found in the description of the Paxos protocol \([30]\).

2.1 A Top-Down Look at the Protocol

This section shows how EventML can organize a top-down description of the protocol, decom-
posing it to a level at which our remaining task is to define a few event observers that act like state
machines. Sec. 2.2 describes one of those state machines, which performs the key computation
used to detect consensus. Sec. 2.3 shows how EventML defines an event observer to accomplish
that. Figures 1 and 2 provide the full EventML specification.

We begin by describing a structure common to many consensus protocols: Each slot \( n \) of an
array of commands gets filled whenever a quorum of agents reach consensus on which command
to place in \( n \). Decisions result from holding elections, and we spawn a separate process to conduct
each one. In this case, for each slot number \( n \), we hold an election to decide which proposals
of form \( (n, c) \) to accept. The tally from any particular ballot may be indecisive, so additional

\(^3\) It follows from the FLP impossibility result \([14]\) that consensus might never be reached.
Figure 1 2/3 consensus: Part 1

**Specification**

```plaintext
 esos ============ Parameters ============
parameter Cmd, cmdeq : Type + Cmd Deq (+ Command type with equality decider cmdeq +)
parameter F : Int (+ max number of failures +)
parameter reps : Loc Bag (+ locations of (3 * F + 1) replicas +)
parameter clients : Loc Bag (+ locations of the clients to be notified +)

(los ----------- Type definitions -----------)
type SlotNum = Int
type RoundNum = Int
type Proposal = SlotNum * Cmd
type VotingRound = SlotNum * RoundNum
type Ballot = VotingRound * Cmd

(los ----------- Interface -----------)
input propose : Proposal
output notify : Proposal

(los ----------- State machine -----------)
observer QuorumState (n, r) = Memory
(observer Quorum (n, r) = (when quorum (n, r)) o (vote base, QuorumState (n, r)) :)
```

rounds of balloting will be spawned as needed. The crucial decisions are when to begin a new round of balloting, what constraints participants must observe in their successive votes, and how to detect that consensus has been achieved (complicated by the fact that multiple rounds in the same election may be occurring simultaneously).

**Interface.** An input event to the protocol is the arrival of a message with header `propose` whose body is a proposal—i.e., a value of type:

```plaintext
type Proposal = Int + Cmd
```

The type of commands is a parameter of the specification:

```plaintext
parameter Cmd, cmdeq : Type + Cmd Deq
```

One subtlety: The protocol requires the ability to determine whether two values of Cmd are equal. So we require an additional parameter, an “equality decider”—here called cmdeq—able to perform that computation. The inputs to the protocol are messages with header `propose` and body of type Proposal:

```plaintext
input propose : Proposal
```

This declaration implicitly defines the base observer propose’base that detects these input events and observes their data.

Outputs of the protocol are directed messages with header `notify`. The data component of an output contains a Proposal value that has been accepted.

```plaintext
output notify : Proposal
```
This declaration does not introduce a base observer recognizing the arrival of "notify" messages, because those events occur outside our system. However, it implicitly declares the functions notify'\text{send}' and notify'\text{bcast}' for creating directed messages. If m is the "notify" message with body p, then the expression (notify\text{'send I p}) is the directed message (l,m) instructing that m be sent to \text{l}; and the expression (notify'\text{bcast \{l1,l2,...\} p}) is the bag \{\{l1,m\},\{l2,m\},...\} of such instructions.

Typically, the complete interface of a system is defined in terms of its input messages, its output messages, and its internal messages, i.e., messages that can only be produced and consumed by the participants of the system. The internal messages exchanged by the participants of the protocol presented in this section are as follows: "vote" messages, by which the replicas cast their votes; "decided" messages, which inform replicas that consensus has been detected on a particular proposal; and "retry" messages, which are described below.

\textbf{Replicas.} To characterize top-level agents in the protocol we will define the event observer \text{Replica}. The main program, \text{SC}, executes the protocol:

\begin{verbatim}
main SC where SC = Replica @ reps
\end{verbatim}
while Replica cannot because it responds to events at all possible locations.

For each \( n \), the protocol conducts a separate election to vote on proposals for the \( n^{th} \) command. Replica spawns subprocesses that cast votes in these elections and identify the winners. The spawning (delegation) operator \( \_ \_ >>= \_ \_ \) is an EventML primitive which is used by processes to start sub-processes:

\[
\text{observer \ Replica } \Rightarrow \text{NewVoters} \_ \_ >>= \text{Voter}
\]

The event observer NewVoters decides when to spawn a new voting process. Voter is a higher-order function; the values it returns are event observers that do the voting. When some NewVoters-event \( e \) occurs and \( v \in \text{NewVoters}(e) \), Replica spawns a local instance of the observer \( \text{Voter}(v) \). By local instance we mean this: each subprocess spawned at a NewVoters-event \( e \) at location \( \text{loc} \) acts only at \( \text{loc} \) and can only react to messages arriving at \( \text{loc} \) after \( e \). For any \( e \) there will be at most one \( v \) such that \( v \in \text{NewVoters}(e) \). So a NewVoters-event spawns only one subprocess. (Though it is not required, we typically apply delegation only to such “singled-valued” observers.)

A note on terminology: SC requires several higher-order functions, such as Voter, that return event observers. For convenience we will use “a Voter observer” or “a Voter” as a shorthand for “an event observer returned by Voter.”

State machines. Informally, we will call an observer a state machine if it defines a distinct state machine at each location. We will say that it reacts to an event if it recognizes the event or if the event can cause its internal state to change.

NewVoters is a state machine. It reacts to “proposal” (from outside the system) and “vote” messages (from inside), and it filters those events. At any location \( \text{loc} \), NewVoters recognizes the first time that \( \text{loc} \) has received a proposal or vote about the \( n^{th} \) command and, when it does, outputs (a singleton bag containing) its value. If the value of such an event is \((n,c)\), the effect of \( \text{NewVoters} \_ \_ >>= \text{Voter} \) is therefore to spawn a local instance of the event observer \( \text{Voter}(n,c) \) at location \( \text{loc} \). The initialization data \((n,c)\) instructs that \( \text{Voter} \) to vote for \((n,c)\) on the first round.

Voter. Voter observers cast votes and tally the votes they receive to determine whether some proposal has achieved consensus. A Voter will not announce a consensus for proposal \((n,c)\) unless it has received \(2*F+1\) votes for \((n,c)\) from \(2*F+1\) different replicas.

We cannot guarantee that any particular poll of the Voter observers will achieve such a result. Accordingly, for each \( n \) we allow arbitrarily many do-over polls: Successive polls for slot number \( n \) are assigned consecutive integers called round numbers. Voting rounds (or just rounds for short) are pairs of the form \((n,r)\)—(slot number/round number). Ballots are pairs of the form \(((n,r),c)\)—(voting round/command). Thus, a Voter casts votes for a particular proposal in a particular round. Votes are pairs of the form \(((n,r),c),\text{loc}\)”—(ballot/location). A voter includes its location in each vote. By arranging that replicas ignore duplicate votes, we guarantee that the protocol works even if messages get duplicated.

A Replica spawns Voter subprocesses to conduct separate elections for each slot number. A Rounds observer uses essentially the same idiom to spawn Round observers that handle individual balloting rounds within a single election. A Voter is, essentially, a Rounds process that runs until its election has been decided:

\footnote{We use the symbol \( >>= \) because the event observers have the structure of a monad having this combinator as its bind operation.}
where “| |” performs parallel composition. For any event observers $A$ and $B$, the observer $(A \text{ until } B)$ acts like $A$ until a $B$-event occurs, at which point it terminates. We use this to terminate any voting for $n$ once consensus has been reached on $n$. $\text{Rounds}$, $\text{Round}$, $\text{NewRounds}$, and $\text{Notify}$ are also functions that return event observers.

A local instance of $\text{Round}((n,r),c)$ conducts the voting for round $(n,r)$ at a particular location. By definition it will cast its vote in round $(n,r)$ for $(n,c)$. Therefore, the first component of $\text{Rounds}(n,c)$ ensures that $\text{Voter}(n,c)$ votes for proposal $(n,c)$ in round $(n,0)$; other instances of $\text{Round}$, spawned by the second component of $\text{Rounds}$, may cast votes for other proposals in later rounds. $\text{Round}$ (detailed in Sec. 2.2) inputs “vote” messages and outputs directed messages of various kinds: “vote”; “decided”; and “retry”, an internal message calling for a new round when a poll does not achieve consensus.

$\text{NewRounds}(n)$ recognizes events that call for new rounds of voting for the $n^{th}$ command. Thus $(\text{NewRounds } n >> \text{ Round})$ spawns instances of $\text{Round}$ as required.

$\text{Notify}(n)$ handles “decided” message with data $(n,c)$ indicating that consensus has been reached about the $n^{th}$ command, by sending notifications to the clients of the system indicating that slot $n$ has been filled with command $c$.

### 2.2 Detecting Consensus

$\text{Round}((n,r),c)$ has two components:

The first component multicasts a vote for $(n,c)$ in round $(n,r)$ to all locations in reps and then terminates. The second executes the consensus-detecting process, $\text{Quorum}(n,r)$, and terminates once it has either announced a consensus or called for a new round. $\text{Once}(A)$ is an observer that acts like $A$ but terminates after the first $A$-event. Because there is at most one $\text{Quorum}(n,r)$ event at any location the use of $\text{Once}$ is logically redundant; but effects an optimization that guarantees that a process is cleaned up once it has produced an output.

$\text{Quorum}(n,r)$ produces an output as soon as it has received votes in round $(n,r)$ from $2 * F + 1$ distinct locations. If all of them are votes for the same proposal, call it $(n,d)$, it decides that $(n,d)$ has achieved consensus and sends appropriate “decided” messages (which will be handled by $\text{Notify}$ observers which will send “notify” messages). If the received votes are not unanimous then it is possible that, however many more votes are tallied, no proposal will receive $2 * F + 1$ votes on this round. (Note that if $F$ failures have occurred, no more votes will arrive, so $\text{Quorum}$ cannot wait for more votes or it might become permanently stuck.) In that case it sends a “retry” message to call for round $(n,r+1)$. That “retry” message also tells the $\text{Voter}$ that spawned the $\text{Quorum}$ how to vote in the new round. If some command $d$ received a majority of the $2 * F + 1$ votes, the $\text{Voter}$ must vote for $(n,d)$. (If no command gets a majority, how it votes does not matter to the logical correctness of the protocol.)

It is possible that a round will occur in which a $\text{Quorum}(n,i)$ at one location detects a consensus and a $\text{Quorum}(n,j)$ at another location calls for a new round of voting. As a result, multiple notifications may be sent about $n$, in a single round or in different rounds. Sec. 3 shows that, for any $n$, all notifications about the $n^{th}$ command will agree on which command has been chosen.
2.3 Implementing Quorum

Quorum\((n,r)\) is a Mealy state machine: in response to inputs it may change state and produce outputs. Let us factor its definition. We first define QuorumState\((n,r)\), a Moore machine whose state is the collection of votes for round \((n,r)\) that the process has received thus far. Quorum\((n,r)\) observes QuorumState\((n,r)\) and issues directed messages as appropriate. EventML provides primitives for defining Moore machines. We use the primitive Memory to define QuorumState:

\[
\text{observer QuorumState}(n,r) = \text{Memory}(\text{loc}\{(\text{[]},\text{[]})\},\text{upd_quorum}(n,r),\text{vote’base})
\]

A QuorumState\((n,r)\) state is a pair of lists \((\text{cmds},\text{locs})\), where \text{cmds} is a list of commands and \text{locs} is a list of locations. The state \(([c_1;c_2;\ldots],[l_1;l_2;\ldots])\) means that, in round \((n,r)\), the state machine has thus far received a vote from \(l_1\) for \(c_1\), a vote from \(l_2\) for \(c_2\), etc. By maintaining that location list in addition to the command list, QuorumState can ignore duplicates; thus, as mentioned above, we need not assume that messages are delivered only once. In the definition of QuorumState, the arguments to Memory have the following meanings: (a) The expression \((\text{loc}\{(\text{[]},\text{[]})\})\) assigns the initial state to each location, i.e., a pair of empty lists. (b) The transition function \text{upd_quorum}(n,r)\) computes the next state from the location and value of the input event and the current state. If an input vote arrives for \(c\) from \(l\), and \(l\) is not listed in the current state, then \text{upd_quorum} adds \(c\) and \(l\) to its state, otherwise the current state stays unchanged. (c) \text{vote’base} recognizes input “vote” events and supplies their values.

Memory is defined so that QuorumState will recognize every “vote” event, update its internal state, and then return (a singleton bag containing) the value of the internal state before performing that update. Had it been more convenient that QuorumState return the value of the internal state after the update we would have used the primitive combinator State instead of Memory.

We define the observer Quorum from QuorumState using the primitive composition combinator \((f \circ (X_1,\ldots,X_n))\), which combines the function \(f\) with the event observers \(X_1,\ldots,X_n\). This combinator behaves as follows: for all \(i \in \{1,\ldots,n\}\), if \(X_i\) observes \(x_i\) at event \(e\) then the event observer \((f \circ (X_1,\ldots,X_n))\) observes each value of the bag \((f \text{loc}(e)\times x_1\ldots x_n)\) at event \(e\). Quorum is defined as follows:

\[
\text{observer Quorum}(n,r) = (\text{when_quorum}(n,r)) \circ (\text{vote’base},\text{QuorumState}(n,r))
\]

This computes the response of Quorum\((n,r)\) to event \(e\) by applying when_quorum\((n,r)\) to \text{loc}(e), and to the values observed at \(e\) by \text{vote’base} and QuorumState\((n,r)\). Note that Quorum\((n,r)\) observes only votes, but not all of them since when_quorum\((n,r)\) sometimes returns an empty bag. If an input vote arrives for \(c\) from \(l\), and \(l\) is listed in the current state, then when_quorum does not output anything. Otherwise, it calls roundout, which requires the most complex definition:

\[
\text{let roundout loc } (((n,r),c),\text{sender}) = \text{cmds,locs} =
\begin{align*}
\text{if length } \text{cmds} = 2 + F \text{ then let } (k,c') = \text{pass-maj cmdeq } (c,\text{cmds}) \text{ in } \\
\text{if } k = 2 + F + 1 \text{ then decided bcast reps } (n,c) \\
\text{else } \{ \text{retry’send loc } ((n,r+1),c) \} \\
\text{else } \{
\end{align*}
\]

The first argument \text{loc} is the location of the Quorum process calling roundout on receipt of a vote; the second argument \(((n,r),c),\text{sender})\) matches the data from the input vote; and the third argument \((\text{cmds},\text{locs})\) matches the state when the input arrives. Therefore \text{cmds},...
where the dot is the cons operation on lists, is the value of the command list that results from processing the input.

We can now understand the outer conditional: If its condition is false then, even after the input event, we have not seen $2 \times F + 1$ votes; so Quorum returns an empty bag, and the input event is not a Quorum $(n, r)$-event. Suppose now that the condition is true and consider the inner conditional.

The poss-maj function, imported from EventML’s library (a snapshot of Nuprl’s library), implements the Boyer-Moore majority vote algorithm. The pair $(k, c’)$ satisfies the following property: If there is a majority entry in the list $c'.cmds$, $c’$ is its value and $k$ is the number of times $c’$ occurs in that list. The condition $(k = 2 \times F + 1)$ therefore tests whether the vote is unanimous. If so, the function returns instructions that the choice of $c’$ be broadcast in appropriate “decided” messages; if not, it returns the instruction to send a “retry” message. Recall that the declaration of “retry” messages introduces the operation retry’send, for constructing directed messages. Therefore, retry’send loc $((n, r+1), c’)$ is the instruction to send to loc a “retry” message with body $((n, r+1), c’)$. So Quorum sends a message to its own location, which will be observed by NewRounds, which will spawn the round $(n, r+1)$. The message data directs the spawned instance of Round to vote for $c’$ in the new round.

3 The Safety Properties of 2/3 Consensus

From $\mathsf{SC}$, our EventML specification of the 2/3 consensus protocol, we generate a LoE specification and a GPM program that express $\mathsf{SC}$’s semantic meaning in our two models of distributed computing. We verify $\mathsf{SC}$’s correctness using the LoE specification, and we execute it using the GPM program. This section describes the formal verification, in the Nuprl proof assistant, of this protocol using the LoE specification, and Sec. 4 below describes the process of generating the GPM program and automatically verifying that it implements the LoE specification.

3.1 Agreement and Validity

The basic safety properties of any consensus protocol are agreement and validity. Both these properties have been formally proved by induction on the causal order of events in Nuprl for the 2/3 consensus protocol of Sec. 2. We state them in terms of notifications. Recall that system properties are predicates on event orderings; we must prove that the predicates are true of all possible runs of the system consistent with the $\mathsf{SC}$ specification\footnote{The formal statements of these properties contain a universally quantified variable that the notation suppresses: eo, denoting an event ordering.}. Agreement says that notifications never contradict one another:

$$\forall e_1, e_2 : \mathsf{E}. \forall l_1, l_2 : \mathsf{Loc}. \forall n : \mathbb{Z}. \forall c_1, c_2 : \mathsf{Cmd}. (\text{notify’send } l_1 (n,c_1)) \in \mathsf{SC}(e_1) \land (\text{notify’send } l_2 (n,c_2)) \in \mathsf{SC}(e_2) \Rightarrow e_1 = e_2$$

Validity says that any proposal decided on must be one that was proposed:

$$\forall e : \mathsf{E}. \forall l : \mathsf{Loc}. \forall v : \mathsf{Proposal}. (\text{notify’send } l v) \in \mathsf{SC}(e) \Rightarrow \exists e’ : \mathsf{E}. e’ < e \land v \in \text{propose’base}(e’)$$

One subtlety: The reader can think of $\downarrow \exists$ as a classical existential. The $\downarrow$ type operator, called “squash”, enforces proof erasure, which is necessary here because, generally, there is no constructive way to pinpoint the exact “propose” event that led to a notification being sent. For example, there might have been two such proposals sent, and once we receive them, we have no way to distinguish between them if the content of these messages is identical.
3.2 Assumptions

For every distributed system we assume that every internal or output message received must have been sent by one of the agents of the system. Formally, we make a separate assumption for each base observer that observes an internal or an output message. For example, if $v \in vote'base(e)$, and $e$ occurs at location $loc$, there must exist some $e' < e$ such that $(vote' send loc v) \in SC(e')$. This can be enforced, e.g., by physical means or by message encryption. We also assume that $reps$ is a bag of size $3 * F + 1$ without repetitions.

3.3 Automation

We have developed two automation tools that help us prove properties of distributed systems. One is a rewriting tool that consists in using the ILFs mentioned in Sec. 1 in order to prove properties by induction on causal order. The other one consists in the automation of standard patterns of reasoning on state machines.

Inductive Logical Form. ILFs are declarative logical statements that precisely answer questions such as: “What led the process at location $l_1$ to send a vote to the process at location $l_2$?” in terms of input messages’ content and state machines’ states. ILFs are automatically generated from main observers using logical simplifications, and characterizations of the LoE combinators. For example, one of the simplest but subtle such characterizations is the one for “$\_ | | \_”$: $v \in X | Y(e) \iff v \in X(e) \land v \in Y(e)$. This says that $v$ is produced by $X | Y$ iff it is produced by either of its components. The $\downarrow$, which enforces proof erasure, is needed because just by knowing that $X | Y$ produced $v$, we cannot in general know whether $v$ was produced by $X$ or $Y$. For example, if identical replicas run in parallel, and receive the same inputs, then there is no way to distinguish between their outputs if they do not label them with different tags.

Given a main observer $X$, we wrote a program that starts with a formula of the form $v \in X(e)$, and keeps on rewriting it using equivalences such as the one presented above, to finally generate a formula of the form $v \in X(e) \iff C$, where $C$ is a complete declarative characterization of $X$’s outputs. In addition, our program also applies various logical simplifications to $C$. Finally, we have built a proof tactic that automatically proves such double implications.

An ILF provides a characterization of all the messages sent by a system. Because it is often useful to get these characterizations for specific kinds of messages, we also generate ILF instances for all the kinds of messages that the system outputs.

Fig. 3 shows the ILF instance for “vote” messages as generated by Nuprl. The details of this formula are not critical for understanding our methodology. However, let us explain how it characterizes the sending of “vote” messages. This formula says that a vote of the form $\langle n', r', c', sender \rangle$ (Nuprl’s pair constructor) is sent by $SC$ at event $e$ to location $i$ (see box 1) iff: (box 2): $e$ happens at a replica location, which we call $R$; (box 3): $i$ is also a replica location; (box 4): there exists a proposal $\langle n', c' \rangle$ that was received by $R$ in a “propose” or “vote” message at a prior event $e'$; (box 5): $\langle n', e' \rangle$ is such that $n'$ has never been received by $R$ prior to $e'$ (there is no important distinction between ReplicaStateFun and ReplicaState, which maintains the list of proposed slot numbers); (box 6): $\langle n', e' \rangle$ is such that no decision has been made about $n'$ between $e'$ and $e$; finally, (box 7): either $\langle n, c \rangle$ is $\langle n', c' \rangle$ and is being voted for at the initial round $r=0$ in response to the “propose” or “vote” message mentioned above (see box 4) that led to a new Voting process being spawned; (box 8): or $\langle n, c \rangle$ comes from a “retry” or “vote” message, and $i$ is not the initial round, i.e., either some replica believed that consensus
could not have reached at round \( r-1 \) (in case of a "retry"), or \( R \) was still working on a smaller round number when it received \( \tau \) (in case of a "vote"), and is now voting at round \( r \).

Using such formulas we can trace back the outputs of a distributed system to the states of its state machines, and to its inputs. For example, to prove SC’s validity property we start from the characterization of "notify" messages and trace these messages back to "proposals" using the various ILF instances.

**State Machine Properties.** As mentioned in Sec. 2.3, one can define Moore machines in EventML using the `Memory` and `State` keywords. Reasoning about such state machines often turns out to be a large part of the verification effort of a distributed program’s correctness. Therefore, our system provides some automation to prove four kinds of local properties of `Memory` and `State` state machines, called: invariant, ordering, progress, and memory.

Informally, a state machine invariant is a unary property about all possible states of the state machine. A state machine ordering property is a binary property about all pairs of states ordered in time. A state machine progress property w.r.t. some predicate \( P \) is a binary property about all pairs of states ordered in time, such that \( P \) is true about at least one of the transitions made between the two states, i.e., such that some process characterized by \( P \) has been made between the two states. A state machine memory property is a ternary property between an input, the current state of the machine at the time it received this input, and a later state. Memory properties are used to specify that state machines keep track of some parts of their inputs in their states.

We have proved in Nuprl, by induction on causal order, that `Memory` and `State` state machines satisfy each of these properties if, among other things, they satisfy some transition property regarding consecutive states (in the case of invariants, a base case is also necessary). Therefore, to prove that a state machine satisfies an instance of one of these four properties, we simply have to instantiate the corresponding general lemma and prove the simpler transition property.

We have developed an annotation language to state such properties in EventML, as well as general Nuprl tactics that try to prove these properties automatically (and often succeed) using logical simplifications and simple reasoners on lists, integers, etc. We illustrate invariants using `QuorumState` (the other properties are described in a longer version of this paper [27]): an invariant of a `QuorumState` state of the form \( (\text{cmds}, \text{locs}) \) is that \( \text{locs} \) has no repeats and same
length as \texttt{cmds}. We call that invariant \texttt{quorum\_inv}, which we state in EventML as follows:

```
import norepeats length invariant quorum\_inv on (cmds,locs) in (QuorumState n1) ≡ norepeats :Loc locs /\ length(cmds) = length(locs):
```

The Nuprl tactic we have designed tries to automatically prove this statement by unfolding \texttt{QuorumState}'s definition to a \texttt{Memory} observer and by instantiating the corresponding general lemma. It (mainly) remains to prove that the base and induction properties are satisfied, which are trivial to prove in this case. Because we have already proved the general principle by induction on causal order, the tactic does not have to use induction on causal order to prove \texttt{quorum\_inv}.

### 3.4 Proof Effort

Thanks to our automation tools and to the rich library of definitions, facts, and proof tactics about LoE and GPM that we have developed over the years, we have specified 2/3 consensus and have proved its two safety properties in Nuprl in merely two days. Proving these two properties about LoE and GPM that we have developed over the years, we have specified 2/3 consensus and have proved its two safety properties in Nuprl in merely two days. Proving these two properties involved: automatically generating and proving 8 state machine properties; automatically generating and proving 1 ILF and 4 instances of that ILF; and interactively proving 8 other lemmas (3 of them being trivial, and therefore candidates for future automation).

### 4 Correct-by-Construction Program Generation

As mentioned in Sec. 1, the semantic meaning of an EventML program is both a LoE event observer and a GPM program. We carry out our correctness proofs on the LoE description of the main event observer. To trust the program we run, we prove that the GPM program implements that LoE description, i.e., that it outputs exactly the same observations. Given an EventML specification, proving that the corresponding GPM program satisfies the corresponding LoE specification is trivial: For each EventML combinator \( C \), there exists a corresponding LoE combinator \( LC \) and a corresponding GPM combinator \( PC \) which provably implements \( LC \).

For example, let \( X_1 \) and \( X_2 \) be event observers of type \( T \), implementable by \( pr_1 \) and \( pr_2 \), respectively. The LoE parallel combinator \( X_1 \parallel X_2 \) is defined as \( \lambda eo.\lambda e.(X_1\ eo\ e) \ast (X_2\ eo\ e) \), where \( \ast \) is the append operation on bags. The GPM parallel combinator \( pr_1 \parallel pr_2 \) is defined as follows (for simplicity we use the same symbol as for the LoE combinators):

\[
\lambda l.\fix \begin{cases}
\lambda R.\lambda s.\let p_1,p_2 = s \in \\
\text{if } \text{halted}(p_1) \land \text{halted}(p_2) \text{ then halt} \\
\text{else run } \begin{cases}
\lambda m.\let p'_1, out_1 = p_1(m) \in \\
\let p'_2, out_2 = p_2(m) \in \\
(R(p'_1,p'_2), out_1 \ast out_2)
\end{cases}
\end{cases}
\end{cases}
\]

This function takes a location \( l \) and returns a process that runs \( p_1 \) and \( p_2 \) in parallel at \( l \). This process maintains a state \( s \) composed of two processes: its two components. Its initial state is \( \langle pr_1\ l, pr_2\ l \rangle \). If the current state \( s \) of the process is a pair \( \langle p_1, p_2 \rangle \), then if both \( p_1 \) and \( p_2 \) have halted, i.e., they are the special halted process \texttt{halt}, then the process becomes \texttt{halt}. Otherwise, the process waits for an input message \( m \), and once it has received one, then (1) for \( i \in \{1,2\} \), it applies\(^6\) \( p_i \) to \( m \) to obtain a new process \( p'_i \) and a bag of output values \( out_i \); (2) it outputs \( out_1 \ast out_2 \) and recursively calls itself on the new state \( \langle p'_1, p'_2 \rangle \).

We proved that \( pr_1 \parallel pr_2 \) implements \( X_1 \parallel X_2 \). The same is true about the other combinators.

\(^6\) The application of a process \( p \) to a message \( m \) is defined as follows: if \texttt{halted}(p) then return (\texttt{halt},{}), otherwise \( p \) is of the form \texttt{run}(f), and therefore, return \( (f\ m) \).
5 Related Work
Much work has been done on specifying and reasoning about distributed systems [24, 17, 10, 11, 9, 19] (to only cite a few).

**IOA.** IOA [16, 15, 17] is a programming/specification language for describing asynchronous distributed systems as I/O automata (labeled state transition systems) and stating their properties. IOA can interact with a large range of tools such as type checkers, simulators, model checkers, theorem provers, and there is support for synthesis of Java code. Both I/O automata and event observers can specify I/O observations of distributed systems. While IOA is state-based, LoE is event-based (states are implicitly maintained by recursive combinators). Also, our methodology allows us to both prove protocol properties and generate code within Nuprl, and does not require any translation to another language.

**TLA.** TLA is a temporal logic, based on first-order logic and set theory, that “provides a mathematical foundation for describing systems” [23]. TLA+ [23, 11] is a language for specifying systems described in TLA. TLAPS “is a platform for the development and mechanical verification of TLA+ proofs” [11]. To validate proofs, TLAPS uses a collection of theorem provers, proof assistants, SMT solvers, and decision procedures. One can use a model checker to help catch errors before attempting any proof. At its current stage, TLAPS allows one to prove safety properties (the safety property of a variant of Paxos has been verified using TLAPS) but not liveness/non-blocking properties (we have not yet proved such properties either). TLA+ does not perform program synthesis.

**seL4.** Our approach is similar to the one taken by Klein et al. to verify the seL4 microkernel [20]. They use Haskell as their specification language, which roughly corresponds to the level of abstraction of EventML in our framework. Then, they translate this code to an Isabelle/HOL version. They prove that this executable specification refines an abstract one, which corresponds to LoE’s level. Finally, they generate by hand a C implementation of the specification, which they translate into Isabelle/HOL, in which they defined a model of C, and manually prove that this implementation refines their executable specification. This corresponds to GPM’s level. Among other things, our paper shows that a similar methodology can be used to design and implement correct fault-tolerant distributed systems.

**Verdi.** More recently, Wilcox et al. developed Verdi [31], which is a framework, similar to ours, to develop and reason about distributed systems using Coq. As in our framework they do not have gaps between the code they verify and the code they run: they run OCaml code that they extract from Coq. Verdi provides a compositional way of specifying distributed systems. This is done by applying verified system transformers. For example, Paxos transforms a distributed system into a crash-tolerant distributed system (they verified Raft [25] instead of Paxos). One difference between our respective methods is that they verify systems by reasoning about the evolution of the state of the world, while our approach relies on the notion of causal order.

6 Conclusion, Current and Future Work
Our methodology scales to more complicated and subtle distributed protocols. For example, we have specified the Multi-Paxos protocol [22, 30] in EventML and proved its safety properties to be correct in Nuprl. We have also built an ordered broadcast service that can switch between various consensus protocols [28]. To get efficient code, we have built in Nuprl a formal tool tuned to
automatically optimize GPM programs and prove that the optimized code and the non-optimized program are bisimilar [26]. We are also experimenting with compilers to Lisp and Scala. We are now building support in EventML and Nuprl to: (1) abstract away from implementation details such as specific data structures, (2) automatically prove simple properties such as validity properties, (3) replay large proofs in order to support modifications to specifications. This paper only discussed safety properties. We have started proving progress and non-blocking properties of the 2/3 consensus protocol. However, it turns out that these proofs are tedious. Next, we want to build automation tools to assist us in proving such properties.

Bibliography

First-order logic for safety verification of hedge rewriting systems

Alexei Lisitsa

1a.lisitsa@liverpool.ac.uk
Department of Computer Science, The University of Liverpool

Abstract: In this paper we deal with verification of safety properties of hedge rewriting systems and their generalizations. The verification problem is translated to a purely logical problem of finding a finite countermodel for a first-order formula, which is further tackled by a generic finite model finding procedure. We show that the proposed approach is at least as powerful as the methods using regular invariants. At the same time the finite countermodel method is shown to be efficient and applicable to the wide range of systems, including the protocols operating on unranked trees.

Keywords: hedge rewriting, safety verification, first-order logic

1 Introduction

Hedges, or arbitrary width trees over unranked alphabets, also known as unranked trees or forests provide with the abstractions useful in several verification contexts. Hedges and hedge transformations (rewritings) have been used for specification and verification of at least the following:

- Protocols working in tree-shaped networks of unbounded degree [27];
- XML transformations [30, 23, 27];
- Multithreaded recursive programs [8, 29].

The most challenging task of infinite state or parameterized verification appears when one would like to establish the correctness of a system in a widest possible context - either for all possible sizes of the system or for unbounded computations. In general such a problem is undecidable and the only way to address it is to focus on restricted classes of systems and properties.

In this paper we consider the problem of safety verification for hedge rewriting systems and explore the applicability of the very general method based on disproving the first-order formulae by finding a countermodel. The basic intuition behind the method is that if the evolution of the system of interest can be faithfully modelled by the derivations within first-order logic then the safety (non-reachability of bad states) can be naturally reduced to disproving the formulae (non-provability).

Such an approach to the safety verification has been proposed in the early work on the verification of cryptographic protocols [28, 25, 12] and later has been extended to various classes of parameterized and infinite state verification tasks [16, 18, 19, 20, 21].

Two attractive properties of the approach have emerged. On the one side it has turned out to be practically efficient in many applications. On the other side it has proved to be relatively com-
complete with respect to the methods using \textit{regular} invariants, in particular regular model checking, regular tree model checking and tree completion.

Safety verification for hedge rewriting systems has been already addressed e.g. in [14, 11, 27] with the \textit{regular} invariants playing the major role.

We show in this paper the Finite CounterModel method (FCM) provides a viable alternative to the methods using regular invariants for the verification of hedge rewriting systems. We show that theoretically the proposed approach is at least as powerful as the methods using regular invariants. At the same time the finite countermodel method is shown to be very flexible and applicable to the wide range of systems. The practical efficiency of the method is illustrated on a set of examples on verification of (1) parameterized protocols operating on arbitrary width tree topology – for most of them an automated verification is reported for the first time; and (2) dynamic multithreaded recursive programs.

The paper is organized as follows. The next section provides with necessary preliminaries. In Section 3 we formulate basic verification problem and its translation into a logical problem of first-order formulae disproving. In Section 4 the verification method is discussed and its relative completeness is demonstrated. Section 5 presents the experimental results and Section 6 concludes the paper.

2 Preliminaries

2.1 First-order Logic

A \textit{first-order vocabulary} is defined as a finite set \( \Sigma = \mathcal{F} \cup \mathcal{P} \) where \( \mathcal{F} \) and \( \mathcal{P} \) are the sets of functional and predicate symbols, respectively. Each symbol in \( \Sigma \) has an associated arity, and we have \( \mathcal{F} = \cup_{i \geq 0} \mathcal{F}_i \) and \( \mathcal{P} = \cup_{i \geq 1} \mathcal{P}_i \), where \( \mathcal{F}_i \) and \( \mathcal{P}_i \) consist of symbols of arity \( i \). The elements of \( \mathcal{F}_0 \) are also called \textit{constants}.

A \textit{first-order model} over vocabulary \( \Sigma \), or just a \textit{model} is a pair \( \mathcal{M} = (D, \llbracket \Sigma \rrbracket_D) \) where \( D \) is a non-empty set called \textit{domain} of \( \mathcal{M} \) and \( \llbracket \Sigma \rrbracket_D \) denotes the interpretations of all symbols from \( \Sigma \) in \( D \). For a domain \( D \) and a function symbol \( f \) of arity \( n \geq 1 \) an interpretation of \( f \) in \( D \) is a function \( \llbracket f \rrbracket_D : D^n \to D \). For a constant \( c \) its interpretation \( \llbracket c \rrbracket_D \) is an element of \( D \). For a domain \( D \) and a predicate symbol \( P \) of arity \( n \) an interpretation of \( P \) in \( D \) is a relation of arity \( n \) on \( D \), that is \( \llbracket P \rrbracket_D \subseteq D^n \). The model \( \mathcal{M} = (D, \llbracket \Sigma \rrbracket_D) \) is called \textit{finite} if \( D \) is a finite set.

We assume that the reader is familiar with the standard definitions of first-order formula, first-order sentence, satisfaction \( \mathcal{M} \models \phi \) of a formula \( \phi \) in a model \( \mathcal{M} \), deducibility (derivability) \( \Phi \vdash \phi \) of a formula \( \phi \) from a set of formulae \( \Phi \), first-order theory and equational theory.

2.2 Hedge Rewriting

We borrow the definitions from [14] with minor modifications. Let \( \Sigma \) be a finite alphabet and \( \mathcal{X} \) be a countable set of variables. Then the set of \textit{terms} \( \mathcal{T}(\Sigma, \mathcal{X}) \) and the set of hedges \( \mathcal{H}(\Sigma, \mathcal{X}) \), both over \( \Sigma \) and \( \mathcal{X} \), are defined inductively as the least sets satisfying

- \( \mathcal{T}(\Sigma, \mathcal{X}) = \mathcal{X} \cup \{ f(h) \mid f \in \Sigma, h \in \mathcal{H}(\Sigma, \mathcal{X}) \} \)
- \( \mathcal{H}(\Sigma, \mathcal{X}) = \bigcup_{n>0} \{ t_1, \ldots, t_n \mid t_j \in \mathcal{T}(\Sigma, \mathcal{X}), j = 1, \ldots, n \} \bigcup \{ \Lambda \} \)
Thus, a hedge is a finite (possibly empty) sequence of terms, whereas a term is obtained by applying an unranked functional symbol to a hedge. We denote by Λ the empty sequence of terms. We do not make a difference between a term and a hedge of length one, i.e. consider that \( T(\Sigma, \mathcal{X}) \subset \mathcal{H}(\Sigma, \mathcal{X}) \). We will also do not distinguish the hedges of the form \( f_i(\Lambda), \ldots, f_i(\Lambda) \) and words \( f_1, \ldots, f_i \in \Sigma^* \).

The sets of ground terms and ground hedges (i.e., terms and hedges without variables) are denoted \( \mathcal{T}(\Sigma) \) and \( \mathcal{H}(\Sigma) \). A variable \( x \in \mathcal{X} \) is called linear in a hedge \( h \in \mathcal{H}(\Sigma, \mathcal{X}) \) if it has exactly one occurrence in \( h \).

The set of variables occurring in a term \( t \in \mathcal{T}(\Sigma, \mathcal{X}) \) is denoted \( \text{var}(t) \). A substitution \( \sigma \) is a mapping from \( \mathcal{X} \) to \( \mathcal{H}(\Sigma, \mathcal{X}) \). The application of a substitution \( \sigma \) to a hedge \( h \), denoted \( h \sigma \), is defined inductively as follows. For \( t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{X}) \) we have \( (t_1, \ldots, t_n) \sigma = t_1 \sigma \ldots t_n \sigma \) and \( f(h) \sigma = f(h \sigma) \).

A context is a hedge \( h \in \mathcal{H}(\Sigma, \mathcal{X}) \) with a distinguished variable \( x \) linear in \( h \). We write \( C[x] \) to denote a context with a distinguished variable \( x \). The application of a context \( C[x] \) to a hedge \( h \) is defined by \( C[h] = C[x \mapsto h] \).

A hedge rewriting system (HRS) is a set of rewriting rules of the form \( l \rightarrow r \), where \( l, r \in \mathcal{H}(\Sigma, \mathcal{X}) \). The rewrite relation \( \rightarrow \) (resp. outer rewrite relation \( \rightarrow^\ast \)) of an HRS \( \mathcal{R} \) is defined as follows: \( h \rightarrow h' \) iff there is a context \( C[x] \) (resp. a trivial context \( C[x] = x \)), a rule \( l \rightarrow r \in \mathcal{R} \) and a substitution \( \sigma \) such that \( h = C[\sigma] \) and \( h' = C[r \sigma] \). The reflexive and transitive closure of \( \rightarrow \) (resp. \( \rightarrow^\ast \)) is denoted by \( \rightarrow^\ast \) (resp. \( \rightarrow^\ast \)).

Given a set of ground hedges \( L \subset \mathcal{H}(\Sigma) \) and an HRS \( \mathcal{R} \) we denote by \( \text{post}^\ast_{\mathcal{R}}(L) \) the set of all hedges reachable from \( L \), that is \( \{ h \in \mathcal{H}(\Sigma) \mid \exists g \in L, g \rightarrow^\ast_{\mathcal{R}} h \} \). Similarly \( \text{post}^\ast_{\mathcal{R}, \Sigma}(L) \) denotes \( \{ h \in \mathcal{H}(\Sigma) \mid \exists g \in L, g \rightarrow^\ast_{\mathcal{R}, \Sigma} h \} \).

### 2.3 Forest Automata and Regular Hedge Languages

The following definition is taken from [5].

**Definition 1** A forest automaton over an unranked alphabet \( \Sigma \) is a tuple \( \mathcal{A} = ((Q, e, \ast), \Sigma, \delta : (\Sigma \times Q \rightarrow Q), F \subset Q) \) where \( (Q, e, \ast) \) is a finite monoid, \( \delta \) is a transition function and \( F \) is a set of accepting states.

For every ground hedge \( h \) the automaton assigns a value \( h^\mathcal{A} \in Q \) which is defined by induction:

- \( \Lambda^\mathcal{A} = e \);
- \( f(h)^\mathcal{A} = \delta(f, h^\mathcal{A}) \)
- \( (t_1, \ldots, t_n)^\mathcal{A} = t_1^\mathcal{A} \ast \ldots \ast t_n^\mathcal{A} \)

A hedge \( h \) is accepted by the forest automaton \( \mathcal{A} \) iff \( h^\mathcal{A} \in F \). The hedge language \( L^\mathcal{A} \) of the forest automaton \( \mathcal{A} \) is defined as \( L^\mathcal{A} = \{ h \mid h^\mathcal{A} \in F \} \).

A hedge language \( L \) is called regular iff it is a hedge language \( L^\mathcal{A} \) of some forest automaton \( \mathcal{A} \). Alternative but equivalent definitions of regular hedge languages using hedge automata and unranked tree automata have been considered e.g. in [14, 27, 26, 15].
2.4 Finitely based sets of hedges

For a finite set of hedges $B \subseteq \mathcal{H}(\Sigma, X)$ the set of ground instances of all hedges from $B$ is denoted by $GI(B)$, that is $GI(B) = \{ h' \in B \land h = h'\theta; h'\theta \text{ is ground} \}$.

Example 1 For $\Sigma = \{ a, b, f \}$ and $B = \{ f(x, b, b, y) \}$ the language $GI(B)$ is the set of all ground hedges with the outer symbol $f$ and containing two consecutive symbols $b$ at leaves.

Notice that finitely based sets of hedges are not necessarily regular; on the other hand, it is easy to see that if every hedge in a set $B$ is linear then $GI(B)$ is regular.

3 Safety Verification: from hedge rewriting to FO logic

In this section we define the basic verification problem and its translation into a purely logical problem of disproving the first-order formula.

3.1 Basic verification problem

The general form of safety verification problems we address in this paper is as follows.

Given: An unranked vocabulary $\Sigma$, a hedge rewriting system $R$ over $\Sigma$, a language $I \subseteq \mathcal{H}(\Sigma)$ of initial ground hedges, a language $U \subseteq \mathcal{H}(\Sigma)$ of unsafe ground hedges.

Question: Is it true that $\forall h \in I \forall h' \in U \ h \not\rightarrow_R h'$?

We will also consider a variant of the basic verification problem for outer rewriting, where $\rightarrow^*$ above is replaced by $\leftrightarrow^*$.

Notice that in the definition of the basic verification problem the sets $I$ and $U$, in general, may be infinite. In that case we assume that the sets are defined by some finitary and constructive means, for example as regular languages given by forest automata, or as finitely based sets of hedges.

3.2 From hedge rewriting to first-order logic

For an unranked alphabet $\Sigma$ we denote by $\Sigma_{fo}$ a ranked vocabulary $\Sigma' \cup \{e, *\} \cup \{R(2)\}$ where $\Sigma' = \{ \tilde{f} \mid f \in \Sigma \}$, with all $\tilde{f}$ being unary functional symbols, $e$ is a constant (0-ary functional) symbol, $*$ is a binary functional symbol, which we will use in infix notation and $R$ is a binary predicate symbol. The constant $e$ will denote the empty hedge, i.e. the hedge of length 0, $*$ will denote the concatenation of hedges and the semantics of $R$ is going to capture reachability for hedge rewriting.

Then we define the translation $\tau : \mathcal{H}(\Sigma, X) \to \mathcal{T}(\Sigma_{fo}, X)$ from hedges to terms over the extended alphabet inductively as follows:

- $\tau(x) = x$ for $x \in X$
• $\tau(\Lambda) = e$
• $\tau(t_1, \ldots, t_n) = \prod_{i=1}^n \tau(t_i)$
• $\tau(f(h)) = \tilde{f}(\tau(h))$

Here $\prod_{i=1}^n \tau(t_i)$ denotes the $*$-product of $\tau(t_i)$ defined as

• $\prod_{i=1}^1 \tau(t_i) = \tau(t_1)$
• $\prod_{i=1}^{k+1} \tau(t_i) = (\prod_{i=1}^k \tau(t_i)) * \tau(t_{k+1})$

Notice that we will specify associativity of $*$, so exact association of brackets in the product will be immaterial.

For a hedge rewriting system $\mathcal{R}$ over alphabet $\Sigma$ and a set of variables $\mathcal{V}$ we define its translation $\Phi_{\mathcal{R}}$ as the set of the following universally closed first-order formulae:

1. $(x \ast y) \ast z = x \ast (y \ast z)$
2. $e \ast x = x$
3. $x \ast e = x$
4. $R(\tau(h_1), \tau(h_2))$ for all $(h_1 \rightarrow h_2) \in \mathcal{R}$
5. $R(x, y) \rightarrow R((z \ast x) \ast v, (z \ast y) \ast v)$
6. $R(x, y) \rightarrow R(\tilde{f}(x), \tilde{f}(y))$ for all unary $\tilde{f} \in \Sigma_{fo}$
7. $R(x, x)$
8. $R(x, y) \land R(y, z) \rightarrow R(x, z)$.

**Proposition 1** *(Adequacy of the first-order translation)* $h_1 \rightarrow^*_{\mathcal{R}} h_2 \iff \Phi_{\mathcal{R}} \vdash R(\tau(h_1), \tau(h_2))$ for all ground $h_1$ and $h_2$.

**Proof.**

$\Rightarrow$ Due to transitivity of $R$ specified in $\Phi_{\mathcal{R}}$ it is sufficient to show that if $h_1 \rightarrow^*_{\mathcal{R}} h_2$ then $\Phi_{\mathcal{R}} \vdash R(\tau(h_1), \tau(h_2))$. Assume $h_1 \rightarrow^*_{\mathcal{R}} h_2$ then $h_1 = C[l \sigma]$ and $h_2 = C[r \sigma]$ for some context $C[x]$, some substitution $\sigma$ and a rewriting rule $(l \rightarrow r) \in \mathcal{R}$. Now the argument proceeds by a simple induction on the context construction. Indeed, for the base case of the simplest context $C[x] \equiv x$ we have $h_1 \equiv l \sigma$ and $h_2 \equiv r \sigma$ and $R(\tau(h_1), \tau(h_2))$ is a ground instance of the formula $R(\tau(l), \tau(r)) \in \Phi_{\mathcal{R}}$ (item 4 in the translation). It follows $\Phi_{\mathcal{R}} \vdash R(\tau(h_1), \tau(h_2))$. There are two step cases. Assume $C[x] \equiv g_1 C'[x] g_2$ where $g_1$ and $g_2$ are ground hedges and $C'[x]$ is a context. Then we have $h_1 \equiv g_1 C'[l \sigma] g_2$, $h_2 \equiv g_1 C'[r \sigma] g_2$, and $C'[l \sigma] \rightarrow C'[r \sigma]$. By induction hypothesis $\Phi_{R} \vdash R(\tau(C[l \sigma]), \tau(C'[r \sigma]))$. That together with the congruence axiom (5) being in $\Phi_{\mathcal{R}}$ implies $\Phi_{\mathcal{R}} \vdash R(\tau(g_1) \ast \tau(C'[l \sigma]) \ast \tau(g_2), \tau(g_1) \ast \tau(C'[r \sigma]) \ast \tau(g_2))$, and therefore, by definition of $\tau$, $\Phi_{\mathcal{R}} \vdash R(\tau(h_1), \tau(h_2))$. The second step case with $C[x] \equiv \tilde{f}(C'[x])$ is dealt with in a similar way using the congruence axiom (6).
Consider the following first-order model $\mathcal{M}$ in the vocabulary $\Sigma_{fo}$. The domain of the model is a set $\mathcal{H}(\Sigma)$ of all ground hedges over $\Sigma$. The interpretations of functional symbols and constants are defined inductively as

- $[[e]] = \Lambda$;
- $[[x \cdot y]] = [[x]] [[y]]$;
- $[[f(x)]] = f([[x]])$ for all $f \in \Sigma$.

The interpretation of $R$ is given by $[[R]] = \{ (h, h') \mid h, h' \in \mathcal{H}(\Sigma) \land h \rightarrow^*_R h' \}$. By simple structural induction we have $[[\tau(h)]] = h$. That concludes the construction of $\mathcal{M}$.

A straightforward check shows now that for such defined $\mathcal{M}$ we have $\mathcal{M} \models \Phi_{A'}$. Assume now $\Phi_{A'} \vdash R(\tau(h_1), \tau(h_2))$. It follows that $\mathcal{M} \models R(\tau(h_1), \tau(h_2))$, that is $([[\tau(h_1)]], [[\tau(h_2)]]) \in [[R]]$, and therefore $(h_1, h_2) \in [[R]]$, which in turn means $h \rightarrow^*_R h'$ by definition of $[[R]]$. □

Taking a contraposition of Proposition 1 we get

$$\Phi_{A'} \vdash R(\tau(h_1), \tau(h_2)) \Rightarrow h_1 \not\rightarrow^*_R h_2$$

which expresses the essence of the proposed verification method: in order to prove non-reachability ($\approx$ safety) it is sufficient to disprove a first-order formula. In order to apply this observation to the solution of basic verification problems we need to provide also the first-order representations for the sets of initial and unsafe terms. We start with very general

**Definition 2** Given an unranked alphabet $\Sigma$ and a set of ground hedges $H \subseteq \mathcal{H}(\Sigma)$ a first-order formula $\varphi_H(x)$ with one free variable and in vocabulary $\Sigma_{fo}$, possibly extended by relational and functional symbols, is called first-order representation of $H$ if $h \in H$ implies $\vdash \varphi_H(\tau(h))$.

### 3.2.1 First-order representations of regular hedge languages

Let $L_{A'}$ be a regular hedge language given by a forest automaton $A' = ((Q, e, \cdot, A), \delta : (A \times Q \rightarrow Q), F \subseteq Q)$.

The vocabulary of the first-order representation of $A'$ consists of unary predicates $P_a$ for each $a \in A$ and unary functional symbols $\tilde{a}$ for each $a \in A$.

Define $\Phi_{A'}$ as the set of the following universally closed formulae:

1. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$;
2. $(x \cdot e) = x$;
3. $(e \cdot x) = x$;
4. $P_{\delta(a,e)}(\tilde{a}(e))$ for every $a \in A$;
5. $P_q(x) \rightarrow P_{\delta(a,q)}(\tilde{a}(x))$ for every $a \in A$ and $q \in Q$.
6. $P_{q_1}(x) \land P_{q_2}(y) \rightarrow P_{q_3}(x \cdot y)$ for all $q_1, q_2, q_3 \in Q$ such that $q_1 \cdot q_2 = q_3$ in $Q$.

Now for the forest automaton $A'$ we denote by $\Phi_{L_{A'}}$ the formula $\Phi_{A'} \rightarrow \bigvee_{q \in F} P_q(x)$.
Proposition 2  $\Phi_{L_{cf}}$ is a first-order representation of $L_{cf}$

Proof (Sketch) Simple induction on the hedge construction shows that $\Phi_{L_{cf}} \vdash P_{h_{cf}}(\tau(h))$. The proposition statement follows immediately. □

3.2.2 Finitely based sets of hedges

For a finite set of hedges $B \subseteq \mathcal{H}(\Sigma, \mathcal{X})$ the formula $\bigvee_{h \in B}(\exists \bar{y}(x = \tau(h)))$ is a first-order representation of $GI(B)$. Here $\bar{y}$ denotes all variables in $\tau(h)$ and $x$ is different from all variables in $\bar{y}$.

3.3 Outer rewriting and unary reachability encoding

In many specification and verification scenarios outer rewriting is sufficient to model all essential aspects of the evolution of the system of interest. In that case the first-order translations can be simplified as there is no need in the congruence axioms and the unary reachability predicate does suffice.

Let $\mathcal{R}$ be a hedge rewriting system over alphabet $\Sigma$. We define its unary translation $\Phi_{\mathcal{R}}^{U}$ as the set of the following universally closed first-order formulae:

1. $(x * y) * z = x * (y * z)$;
2. $(x * e) = x$;
3. $(e * x) = x$;
4. $R(\tau(h_1)) \rightarrow R(\tau(h_2))$ for all $(h_1 \rightarrow h_2) \in \mathcal{R}$.

Such a simplified translation captures reachability by the outer rewrite relation $\rightsquigarrow_{\mathcal{R}}$ as the following proposition states.

Proposition 3  (Adequacy of the unary first-order translation) $h_1 \rightsquigarrow_{\mathcal{R}}^{*} h_2 \Leftrightarrow \Phi_{\mathcal{R}}^{U} \land R(\tau(h_1)) \vdash R(\tau(h_2))$ for all ground $h_1$ and $h_2$.

Proof (Hint) The proof follows the arguments in the proof of Proposition 1 appropriately modified. To prove $\Rightarrow$ entailment an easy induction on the length of rewriting is used. To prove $\Leftarrow$ a semantical argument taking as the interpretation of unary predicate $R$ the set of the term encodings of all reachable from $h_1$ hedges, is applied. □

4 Back to safety verification

Now we have defined all necessary concepts and the basic verification problem defined in Sect. 3.1 gets its full specification.
Given: An unranked vocabulary $\Sigma$, a hedge rewriting system $R$ over $\Sigma$, a first-order representation $\varphi_I$ of the language $I \subseteq \mathcal{H}(\Sigma)$ of initial ground hedges, a first-order representation $\varphi_U$ of the language $U \subseteq \mathcal{H}(\Sigma)$ of unsafe ground hedges.

Question: Is it true that $\forall h \in I \forall h' \in U \ h \not\rightarrow^*_R h'$?

Proposition 4 If the verification problem above has a negative answer, that is $\exists t_1 \in I \exists t_2 \in U t_1 \rightarrow^*_R t_2$ then $\varphi_R \vdash \exists x \exists y (\varphi_I(x) \land \varphi_U(y) \land R(x,y))$. 

Proof. The proposition statement follows immediately from Proposition 1 and Definition 2.

4.1 Verification method

Taking contraposition of Proposition 4 we have $\varphi_R \not\vdash \exists x \exists y (\varphi_I(x) \land \varphi_U(y) \land R(x,y))$ implies $\forall h \in I \forall h' \in U \ h \not\rightarrow^*_R h'$, that is a positive answer to an instance of the basic verification problem.

Thus, the essence of the finite countermodels (FCM) verification method we advocate in this paper is:

To demonstrate a positive answer to an instance of a basic verification problem apply an automated generic finite model procedure to find a countermodel to $\varphi_R \rightarrow \exists x \exists y (\varphi_I(x) \land \varphi_U(y) \land R(x,y))$, or equivalently a model for $\varphi_R \land \neg \exists x \exists y (\varphi_I(x) \land \varphi_U(y) \land R(x,y))$.

The next theorem, which is a generalization of similar theorems in [17, 21] (the case of word languages) and [19] (the case of tree languages), shows the applicability of FCM method and its relative completeness with respect to the methods based on regular invariants [14, 27].

Theorem 1 Let $R$ be a hedge rewriting system over unranked alphabet $\Sigma$, $\varphi_I$ and $\varphi_U$ be first-order formulae representing the regular sets $I$ and $U$ of initial and unsafe hedges, respectively. Let $Inv$ be a regular set of hedges such that $post^*_R(I) \subseteq Inv$ and $Inv \cap U = \emptyset$, that is a regular invariant separating $I$ and $U$. Then there is a finite model $\mathcal{M}$ such that $\mathcal{M} \models \varphi_R \land \neg \exists x \exists y (R(x,y) \land \varphi_I(x) \land \varphi_U(y))$.

Proof. (Sketch) Assume the condition of the theorem holds. Let $\varphi = ((Q_I, e_I, *_I), A, \delta_I, F_I \subseteq Q_I)$, $\varphi_{Inv} = ((Q_{Inv}, e_I, *_{Inv}), A, \delta_{Inv}, F_{Inv} \subseteq Q_{Inv})$ and $\varphi_U = ((Q_U, e_I, *_U), A, \delta_U, F_U \subseteq Q_U)$ be forest automata recognizing regular hedge languages $Inv$ and $U$, respectively. Then the required finite model $\mathcal{M}$ is constructed as follows. The domain $D$ of the model is $Q_I \times Q_{Inv} \times Q_U$. The interpretation of $e$ is $\llbracket e \rrbracket = (e_I, e_{Inv}, e_U)$. The interpretation of $*$ is given by $(q_1, q_2, q_3)(q'_1, q'_2, q'_3) = (q_1 * q'_1, q_2 * q'_2, q_3 * q'_3)$. For all $a \in A$ the interpretations are given by $\llbracket a \rrbracket(q_1, q_2, q_3) = (\delta_I(a, q_1), \delta_{Inv}(a, q_2), \delta_U(a, q_3))$. Once we defined the interpretations of all functional symbols (including constants) any ground term $t$ gets its interpretation $\llbracket t \rrbracket \in D$ in a standard way. Define the interpretation of $R$ as $\llbracket R \rrbracket = \{ ([\tau(h)]_I, [\tau(h)]_U) | h \rightarrow^*_R h' \}$. For all predicates $P_q$ with $q \in Q_I$ used in $\varphi_I$ the interpretations are defined as $\llbracket P_q \rrbracket = \{ (q, x, y) | x \in Q_{Inv}, y \in Q_U \}$. Similarly, $\llbracket P_q \rrbracket = \{ (x, q, y) | x \in Q_I, y \in Q_{Inv} \}$ for $q \in Q_{Inv}$ and $\llbracket P_q \rrbracket = \{ (x, y, q) | x \in Q_I, y \in Q_{Inv} \}$. Now it is straightforward exercise to show that the such defined model is indeed as required.
5 Experiments

In this section we present the experimental results of applications of the FCM method for safety verification of hedge rewriting systems. In the experiments we used the finite model finder Mace4 [24] within the package Prover9-Mace4, version 05, December 2007\(^1\).

5.1 Tree arbiter protocol and other protocols on unranked trees

We consider here as a case study the verification of parameterized Tree-arbiter Protocol working on the trees of unbounded branching degree. We take the description of the protocol from [4] where its automated verification has been demonstrated for the case of binary trees.

The protocol supervises the access to a shared resource of a set of processes arranged in a tree topology. The processes competing for the resource are located in the leaves of the trees. The safety property to be verified is that of mutual exclusion - at no point, two or more processes at leaves of the tree can get an access to the resource (obtain a token). A process in the protocol can be in state idle (i), requesting (r), token (t) or below (b). All the processes are initially in state i. A node is in state b whenever there is a node below (descendant) in state t. When a leaf is in state r, the request is propagated upwards until it encounters a node in state t of b (aware of the presence of the token). A node state t can choose to pass it upwards or pass it downwards to a requesting node in state r.

We model the tree arbiter protocol as a hedge rewriting system \( R_{ta} \) consisting of the rules:

1. \( i(x r(y) z) \rightarrow r(xr(y)z) \) (request propagated upwards)
2. \( t(x r(y) z) \rightarrow b(xt(y)z) \) (token passed downwards)
3. \( b(x t(y) z) \rightarrow t(xi(y)z) \) (token passed upwards)
4. \( i(\Lambda) \rightarrow r(\Lambda) \) (idle leaf node becomes requesting)

The set of initial configurations is a hedge language consisting of all hedges in which all leaves are either \( r(\Lambda) \) or \( i(\Lambda) \), all intermediate nodes (neither leaves, nor the top) are \( i(\ldots) \) and the top node is \( t(\ldots) \).

The hedge language of unsafe configurations consists of all hedges with two or more leaves with \( t(\Lambda) \) (tokens). Now we represent the verification problem in the format required by FCM method.

The automata \( A_I \) and \( A_U \) reconginizing the sets of initial, respectively unsafe states and the first-order translation \( \Phi \) of the hedge rewriting system and the automata can be found in the Appendix A of [22].

After translation of the protocol specification and forest automata recognizing the initial and unsafe configurations into a first order formula \( \Phi \) and (un)safety condition into a formula \( \Psi \) the safety is established automatically by Mace4 finding a countermodel for \( \Phi \rightarrow \Psi \) of size 3 in 0.03\(s\).

\(^1\) the system configuration used in experiments: Intel(R) Core (TM)2 Duo CPU, T7100 1.8Ghz 1.79 Ghz, 1.00 GB of RAM.
We have applied FCM to the verification of other protocols operating on the unranked trees. The specification of the protocols are taken from [4, 6] where their automated verification has been demonstrated for the binary trees case. In [19] we have considered the application of FCM for the verification of binary tree case protocols. The table below lists the parameterized tree protocols and shows the time it took Mace4 to find a countermodel and verify a safety property for the case of unranked trees (first column). For comparison the times it took to verify the binary tree versions of the same protocols by FCM [19] and by regular tree model checking [6] are given in the second and third columns, respectively.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Token</td>
<td>0.04</td>
<td>0.02s</td>
<td>0.06s</td>
</tr>
<tr>
<td>Two-way Token</td>
<td>0.04</td>
<td>0.03s</td>
<td>0.09s</td>
</tr>
<tr>
<td>Percolate</td>
<td>0.06</td>
<td>0.09s</td>
<td>2.4s</td>
</tr>
<tr>
<td>Tree arbiter</td>
<td>0.03</td>
<td>0.03s</td>
<td>0.31s</td>
</tr>
</tbody>
</table>

* the system configuration used in [6] was *Intel Centrino 1.6GHZ with 768MB of RAM*

Further details of the experiments can be found in [16].

As far as we are aware the automated verification of the above parameterized protocols over trees of unbounded degree is reported here for the first time, with an exception being Two-way Token protocol verified in [27].

### 5.2 Verification of Synchronised PAD Systems

The verification techniques based on disproving in first-order logic are very flexible and can accommodate various constraints on the properties of the systems of interest. To illustrate this point we consider an automated verification of a program which involves dynamic creation of processes. We take as an example a program from [29] specified in terms of Synchronised PAD Systems [29]. Such systems can be seen as an extension of hedge rewriting systems to the case of two different associative constructors, one of which being commutative. Furthermore the rewriting is constrained to capture the effects of modelled synchronisation.

Let $\text{Sync} = \{a, b, c, \ldots\}$ be a set of actions such that every action $a \in \text{Sync}$ corresponds to a co-action $\bar{a} \in \text{Sync}$ s.t. $\bar{\bar{a}} = a$. Let $\text{Act} = \text{Sync} \cup \{\tau\}$ be the set of all the actions, where $\tau$ is a special action.

Let $\text{Var} = \{X, Y, \ldots\}$ be a set of process variables and $\mathcal{T}$ be the set of process terms $t$ over $\text{Var}$ defined by:

$$ t ::= 0 \mid X \mid t \cdot t \mid t||t $$

**Definition 3** A Synchronised PAD (SPAD) is a finite set of rules of the form $X \hookrightarrow^a t$ or $X \cdot Y \hookrightarrow^a t$, where $X, Y \in \text{Var}; t \in \mathcal{T}$ and $a \in \text{Act}$

The process terms are considered modulo the following equational theory $SE$ of structural equivalence:

**A1:** $t \cdot 0 = 0 \cdot t = t||0 = 0||t = t$
A SPAD $\mathcal{R}$ induces a transition relation $\rightarrow^a$ over $\mathcal{F}$ by the following inference rules:

\[
\begin{align*}
\theta_1 : & \frac{t_1 \rightarrow^a t_2 \in \mathcal{R}}{t_1 \rightarrow^a t_2} \\
\theta_2 : & \frac{t_1 \rightarrow^a t_2 \in \mathcal{R}}{t_1 \rightarrow^a t_2} \\
\theta_3 : & \frac{t_1 = 0, t_2 \rightarrow^a t_2}{t_1 \rightarrow^a t_2} \\
\theta_4 : & \frac{t_1 \rightarrow^a t_2 \in \mathcal{R}}{t_1 \rightarrow^a t_2} \\
\theta_5 : & \frac{t_1 \rightarrow^a t_2 \in \mathcal{R}}{t_1 \rightarrow^a t_2}
\end{align*}
\]

The equational theory $SE$ induces a transition relation $\rightarrow^a_{SE}$ over $\mathcal{F}$ defined by $\forall t, t' \in \mathcal{F}, t \rightarrow^a_{SE} t'$ iff $\exists u, u' \text{ s.t. } t =_{SE} u, u' \rightarrow^a u'$ and $u' =_{SE} t'$. For $w \in Act^*$ the transition relations $\rightarrow^w$ and $\rightarrow^w_{SE}$ are defined in a standard way.

5.2.1 Example

Consider the program represented in Fig 1 which involves dynamic creation of processes. The figure represents the flow graph of a program having two procedures $\pi_1$ and $\pi_2$ such that:

- $\pi_1$ calls itself in parallel with another procedure $\pi_2$.
- $\pi_2$ calls itself recursively.
- $\pi_1$ and $\pi_2$ communicate via the synchronizing actions $a, b$, and their corresponding co-actions $\tilde{a}$ and $\tilde{b}$, and the program starts at point $n_0$.

The safety property to be verified is “starting from $n_0$, the program never reaches a configuration where the control point $m_1$ is active.”

The program is modelled by the Synchronized PAD $\mathcal{R}$ [29] which includes the following rules:
The above verification task is formulated in terms of $\mathcal{R}$ as follows. Show that there are no $w \in \tau^*$ and $t \in \mathcal{T}$ such that $m_0$ is a subterm of $t$ and $n_0 \rightarrow_{SE}^w t$.

Notice that the transition relation $\rightarrow_{SE}^w$ encodes the reachability by a rewriting process, where the rules with $\rightarrow^\tau$ can be applied without any restrictions, whereas the rules with $\rightarrow^a$ with $a \in \text{Sync}$ can be applied only if a some rule $\rightarrow^\bar{a}$ with a co-action superscript is applied simultaneously "in parallel" subterm (cf. the rule $\theta_5$).

We claim now that one can specify the first-order formulae:

- $\Phi_\mathcal{R}$ describing the reachability by $\rightarrow_{SE}^w$, the initial configuration and the set of unsafe configurations, and
- $\Psi$ specifying unsafety condition

such that if $\Phi_\mathcal{R} \vdash \Psi$ then the program is unsafe. See the details in Appendix B of [22]. We apply Mace4 and a countermodel of the size 3 for $\Phi_\mathcal{R} \rightarrow \Psi$ is found in 2s.

6 Conclusion

We have shown in this paper that the simple encoding of hedge rewriting systems by first-order logic and using available finite model finders provides with an interesting and viable alternative to the existing methods for the verification of systems working on the trees of unbounded degree. On the one side it is relatively complete w.r.t. the methods using regular invariants. On the other side it is either practically efficient, as our automated verification of parameterized protocols working on the trees of unbounded degree illustrates well, or at least looks promising to explore the boundaries of its practical applicability, as our SPAD example shows. The advantages of the FCM method are (1) it is simple, (2) it is flexible, (3) it is modular and (4) it reuses existing tools. The FCM method has a natural limitation that it can be used only to establish the safety. If safety actually does not hold looking for the countermodels does not help. One can use then automated theorem provers to confirm that indeed the safety is violated by searching for the proofs of formulas of the form $\Phi \rightarrow \Psi$. One direction for the future work here is the development of the systematic procedures which would extract the unsafe traces from first-order proofs.

Bibliography


[8] A. Bouajjani and T. Touili. On computing reachability sets of process rewrite systems, in Proc. 16th Int. Confenerece on Rewriting Techniques And Applications (RTA’05), LNCS 3467, 2005


[23] S. Maneth, A. Berlea, T. Perst, and H. Seidl, Xml type checking with macro tree transducers, PODS, 2005


Distributed Verification of Rare Properties using Importance Splitting Observers

Cyrille Jegourel, Axel Legay, Sean Sedwards and Louis-Marie Traonouez

INRIA Rennes – Bretagne Atlantique

Abstract: Rare properties remain a challenge for statistical model checking (SMC) due to the quadratic scaling of variance with rarity. We address this with a variance reduction framework based on lightweight importance splitting observers. These expose the model-property automaton to allow the construction of score functions for high performance algorithms. The confidence intervals defined for importance splitting make it appealing for SMC, but optimising its performance in the standard way makes distribution inefficient. We show how it is possible to achieve equivalently good results in less time by distributing simpler algorithms. We first explore the challenges posed by importance splitting and present an algorithm optimised for distribution. We then define a specific bounded time logic that is compiled into memory-efficient observers to monitor executions. Finally, we demonstrate our framework on a number of challenging case studies.

Keywords: Rare events, statistical model checking, importance splitting, scalable verification, lightweight observers.

1 Introduction

Failure in critical systems is required to be very infrequent. Numerical model checking can quantify the probability of such failure with certainty, but is limited in its application to real systems because of the ‘state explosion problem’ [CES09]. This is addressed by statistical model checking (SMC) [YKNP06], which includes a number of approximative techniques based on Monte Carlo sampling [HH64]. Using SMC, only a subset of system states are generated on the fly during stochastic simulation, while results converge in a predictable way. Performance is typically independent of the size of the state space [Nie92] and simulations may be efficiently divided on parallel computation architectures. SMC has therefore been successfully applied to real systems in a critical context, such as to the cabin communication system of an aeroplane [BBB+10]. Rare properties (those with probability close to zero) nevertheless pose a problem because the standard and relative estimation errors scale quadratically with rarity [HH64, RT09]. For example, 4000 simulations would be sufficient to estimate a probability of 0.1 ± 10% with 95% confidence, whereas $4 \times 10^{13}$ simulations would be necessary to estimate a probability of $10^{-6} \pm 10\%$ with the same confidence. Desirable failure rates in critical systems may be orders of magnitude lower, so we seek to enhance SMC with variance reduction techniques, such as importance sampling and importance splitting [KH51, HH64, RT09], without sacrificing the easy distribution that SMC affords.

Importance sampling weights the executable model of a system so that the rare property occurs more frequently in simulations. The proportion of simulations that satisfy the property using
the weighted model overestimates the true probability, but the estimate may be exactly compensated by the weights. It is generally not feasible to implement a perfectly weighted executable model for importance sampling because (i) the perfect model may not actually exist as a re-parametrisation of the original model and (ii) a perfect re-parametrisation typically requires an iteration over all the transitions, defeating the benefits of sampling. Practical approaches tend to use a low dimensional vector of parameters to weight the model [JLS12b, JLS12a]. Given such a parametrisation, importance sampling can be implemented with minimal memory and may be distributed efficiently on parallel computational architectures. The principal limitation of importance sampling is that without a guarantee that the simulation model is perfect, it is difficult to formally bound the error of estimates. In contrast, useful confidence intervals have been defined for importance splitting [CG07, CDFG12].

Importance splitting divides a rare property into a set of less rare sub-properties that correspond to an increasing sequence of disjoint levels: the initial state corresponds to the lowest level, while states that satisfy the rare property corresponds to the final level. Importance splitting algorithms use a series of easy simulation experiments to estimate the conditional probabilities of going from one level to the next. Since relatively few simulations fail to satisfy the sub-properties, the overall simulation budget may be reduced. Each experiment comprises simulations initialised with the terminal states of previous simulations that reached the current level. The overall probability is the product of the estimates, with the best performance (lowest variance) achieved with many levels of equal conditional probability.

Importance splitting poses several challenges for optimisation and distribution. In the context of SMC, importance splitting algorithms repeatedly initialise simulations with states of the model-property product automaton. For arbitrary properties this may have size proportional to the length of a simulation trace. At the same time, increasing the number of levels to maximise performance reduces the number of simulation steps in each simulation experiment. The cost of sending the model-property state across slow communication channels may be significantly greater than the cost of short simulations. In addition, to specify levels with equal conditional probabilities it is necessary to define a ‘score function’ that maps the states of the product automaton to a value. This cannot easily be automated, so a syntactic description of the property automaton must be accessible for the user to construct a score function manually.

To address the above challenges we present an importance splitting framework for SMC, specifically considering the problems of distribution. We first discuss the problems of distributing importance splitting algorithms and present a fixed level algorithm optimised for distribution. We then define an expressive bounded time temporal logic and describe the system of efficient lightweight observers that implement it. These make the product automaton (i) accessible to the user, (ii) efficient to construct, (iii) efficient to distribute and (iv) efficient to execute. Finally, we demonstrate the performance and flexibility of our framework on a number of case studies that are intractable to numerical methods.

We believe the present work is the first to describe a practical importance splitting framework for SMC and is therefore the first to consider the problems of distributing importance splitting for SMC.
Related Work

There have been many ad hoc implementations of importance splitting based on the original ideas of [Kah50, KH51]. The algorithm of [VV91] is a relatively recent example that is often cited. The work of [CG07, CDFG12] is novel because the authors define efficient adaptive importance splitting algorithms that also include confidence intervals. To our knowledge, [JLS13] is the first work to explicitly link importance splitting to arbitrary logical properties.

SMC tools construct an automaton (a monitor) to accept traces that satisfy a temporal logic formula, typically based on a time bounded variant of temporal logic. The proportion of independent simulations of a stochastic model that satisfy the property is then used to estimate the probability of the property or to test hypotheses about the probability. There have been several works that construct runtime verification monitors from temporal logic (e.g., [Gei01, GH01, HR02, FS04, BLS06]). Such monitors typically comprise tableau-based automata [GPVW95] whose states represent the combinations of subformulas of the overall property. While some have considered timed properties (e.g., [BLS06]), the focus is predominantly unbounded LTL properties interpreted on finite paths [EFH03]. In contrast, SMC typically checks formulas with explicit time bounds (see, e.g., (1)), which are inherently defined on finite traces. To avoid the combinatorial explosion of subformulas caused by including time in this way, the monitors used by [JLS12b, BCLS13] and other high performance tools are compact “programs” that generate the states of an automaton on the fly and do not store them. Such programs incorporate notions of optimality that may be subtly different from those that apply in other contexts. Since states of the automaton are generated on the fly, it is not necessary for the automaton to have the minimum number of states. The actual requirements are that the automaton reaches a conclusion with the minimum number of input states and that its programmatic representation is as compact as possible. We adapt this “lightweight” approach to allow importance splitting for SMC to be efficiently distributed on high performance parallel computational architectures.

2 Technical Background

Our SMC tools (Plasma [JLS12b], Plasma-Lab [BCLS13]) implement a bounded linear temporal logic having the following syntactic form:

\[
\phi = X^k\phi | F^k\phi | G^k\phi | \phi U^k\phi | \neg\phi | \phi \lor \phi | \phi \land \phi | \phi \Rightarrow \phi | \alpha
\]  

(1)

This syntax allows arbitrary combinations and nesting of temporal and atomic properties (i.e., those which may be evaluated in a single state and denoted by \(\alpha\)). The time bound \(k\) may denote discrete steps or continuous time, but in this work we consider only discrete time semantics.

Given a finite trace \(\omega\), comprising sequence of states \(\omega_0\omega_1\omega_2\cdots\), \(\omega^{(i)}\) denotes the suffix \(\omega_i\omega_{i+1}\omega_{i+2}\cdots\). The semantics of the satisfaction relation \(\models\) is constructed inductively as fol-
Distributed Verification of Rare Properties using Importance Splitting Observers

\[ \omega(i) \models true \]
\[ \omega(i) \models \alpha \iff \omega(i) \models \neg \phi \]
\[ \omega(i) \models \varphi_1 \lor \varphi_2 \iff \omega(i) \models \varphi_1 \lor \omega(i) \models \varphi_2 \]
\[ \omega(i) \models X^k \varphi \iff \omega(k+i) \models \varphi \]
\[ \omega(i) \models \varphi_1 U^k \varphi_2 \iff \exists j \in \{i, \ldots, i+k\} : \omega(j) \models \varphi_2 \land (j = i \lor \forall l \in \{i, \ldots, j-1\} : \omega(l) \models \varphi_1) \]  \(2)\)

Other elements of the relation are constructed using the equivalences \(false \equiv true\), \(\varphi_1 \land \varphi_2 \equiv \neg(\neg \varphi_1 \lor \neg \varphi_2)\), \(F \varphi \equiv true U^k \varphi\), \(G^k \varphi \equiv \neg(true U^k \neg \varphi)\). Hence, given a property \(\varphi\) with syntax according to \((1)\), \(\omega \models \varphi\) is evaluated by \(\omega(0) \models \varphi\).

Importance Splitting and Score Functions

The neutron shield model of \([Kah50, KH51]\) is illustrative of how importance splitting works. The distance travelled by a neutron in a shield defines a monotonic sequence of levels \(0 = s_0 < s_1 < s_2 < \cdots < s_m = \text{shield thickness}\), such that reaching a given level implies having reached all the lower levels. While the overall probability \(\gamma\) of passing through the shield is small, the probability of passing from one level to another can be made arbitrarily close to 1 by reducing the distance between levels. Denoting the abstract level of a neutron as \(s\), the probability of a neutron reaching level \(s_i\) can be expressed as \(P(s \geq s_i) = P(s \geq s_i \mid s \geq s_{i-1})P(s \geq s_{i-1})\). Defining \(\gamma = P(s \geq s_m)\) and \(P(s \geq s_0) = 1\),

\[ \gamma = \prod_{i=1}^{m} P(s \geq s_i \mid s \geq s_{i-1}) \]  \(3)\)

Each term of \((3)\) is necessarily greater than or equal to \(\gamma\), making their estimation easier. By writing \(\gamma_i = P(s \geq s_i \mid s \geq s_{i-1})\) and denoting the estimates of \(\gamma\) and \(\gamma_i\) as respectively \(\hat{\gamma}\) and \(\hat{\gamma}_i\), \([JLS13]\) defines the unbiased confidence interval

\[ CI = \left[ \frac{\hat{\gamma}}{1 + \frac{z_\alpha \sigma}{\sqrt{n}}} \right], \quad \frac{\hat{\gamma}}{1 - \frac{z_\alpha \sigma}{\sqrt{n}}} \]  \(4)\)

Confidence is specified via \(z_\alpha\), the \(1 - \alpha/2\) quantile of the standard normal distribution, while \(n\) is the per-level simulation budget. We infer from \((4)\) that for a given \(\gamma\) the confidence is maximised by making both the number of levels \(m\) and the simulation budget large, with all \(\gamma_i\) equal.

The concept of levels can be generalised to arbitrary systems and properties in the context of SMC, treating \(s\) and \(s_i\) in \((3)\) as values of a score function over the model-property product automaton. Intuitively, a score function discriminates good paths from bad, assigning higher scores to paths that more nearly satisfy the overall property. Since the choice of levels is crucial to the effectiveness of importance splitting, various ways to construct score functions from a temporal logic property are proposed in \([JLS13]\). Formally, given a set of finite trace prefixes
\( \omega \in \Omega \), an ideal score function \( S : \Omega \to \mathbb{R} \) has the characteristics \( S(\omega) > S(\omega') \iff P(\models \varphi | \omega) > P(\models \varphi | \omega') \), where \( P(\models \varphi | \omega) \) is the probability of eventually satisfying \( \varphi \) given prefix \( \omega \).

Intuitively, \( \omega \) has a higher score than \( \omega' \) iff there is more chance of satisfying \( \varphi \) by continuing \( \omega \) than by continuing \( \omega' \). The minimum requirement of a score function is \( S(\omega) \geq s_\varphi \iff \omega \models \varphi \), where \( s_\varphi \) is an arbitrary value denoting that \( \varphi \) is satisfied. Any trace that satisfies \( \varphi \) must have a score of at least \( s_\varphi \) and any trace that does not satisfy \( \varphi \) must have a score less than \( s_\varphi \). In what follows we assume that (3) refers to scores.

3 Distributing Importance Splitting

Simple Monte Carlo SMC may be efficiently distributed because once initialised, simulations are executed independently and the result is communicated at the end with just a single bit of information (i.e., whether the property was satisfied or not). By contrast, the simulations of importance splitting are dependent because scores generated during the course of each simulation must be processed centrally. The amount of central processing can be minimised by reducing the number of levels, but this generally reduces the variance reduction performance.

Alternatively, entire instances of the importance splitting algorithm may be distributed and their estimates averaged, with each instance using a proportionally reduced simulation budget. We use this approach to generate some of the results in Section 6, but note that if the budget is reduced too far, the algorithm will fail to pass from one level to the next (because no trace achieves a high enough score) and no valid estimate will be produced.

Distribution of importance splitting is thus possible, but its efficiency is dependent on the particular problem. In this work we therefore provide the framework to explore different approaches.

In Section 3.1 we first describe the concept of an adaptive importance splitting algorithm and then explain why this otherwise optimised technique is unsuitable for distribution. In Section 3.2 we motivate the use of a fixed level algorithm for “lightweight” distribution and provide a suitable algorithm. The results we present in Section 6 demonstrate that this simpler approach can be highly effective.

3.1 The Adaptive Algorithm

The basic notion of importance splitting described in Section 2 can be directly implemented in a so-called fixed level algorithm, i.e., an algorithm in which the levels are pre-defined by the user. With no a priori information, such levels will typically be chosen to subdivide the maximum score equally. In general, however, this will not equally divide the conditional probabilities of the levels, as required by (4) to minimise variance. In the worst case, one or more of the conditional probabilities will be too low for the algorithm to pass between levels. Finding good or even reasonable levels by trial and error may be computationally expensive and has prompted the development of adaptive algorithms that discover optimal levels on the fly [CG07, JLS13, JLS14]. Instead of pre-defining levels, the user specifies the proportion of simulations to retain after each iteration. This proportion generally defines all but the final conditional probability in (3).

The adaptive importance splitting algorithm first performs a number of simulations until the
overall property is decided, storing the resulting traces of the model-property automaton. Each trace induces a sequence of scores and a corresponding maximum score. The algorithm finds a level that is less than or equal to the maximum score of the desired proportion of simulations to retain. The simulations whose maximum score is below this current level are discarded. New simulations to replace the discarded ones are initialised with states corresponding to the current level, chosen at random from the retained simulations. The new simulations are continued until the overall property is decided and the procedure is repeated until a sufficient proportion of simulations satisfy the overall property.

The principal advantage of the adaptive algorithm is that by simply rejecting the minimum number of simulations at each level it is possible to maximise confidence for a given score function. The principal disadvantage is that it stores simulation traces, severely limiting the size of model and simulation budget. The use of lightweight computational threads is effectively prohibited. Moreover, minimising the number of rejected simulations reduces the number of simulations performed between levels, thus reducing the possibility to perform computations in parallel. Minimising the rejected simulations also maximises the number of levels, which in turn minimises the number of simulation steps between each level. This further limits the feasibility of dividing the algorithm, since sending a model-property state over a slow communication channel may be orders of magnitude more costly than performing a short simulation locally.

### 3.2 A Fixed Level Algorithm for Distribution

In contrast to the adaptive algorithm, the fixed level importance splitting algorithm does not need to store traces, making it lightweight and suitable for distribution. Scores are calculated on the fly and only the states that achieve the desired level are retained for further consideration. While the choice of levels remains a problem, an effective strategy is to first use the adaptive algorithm with a relatively high rejection rate to find good fixed levels. An estimate with high confidence can then be generated efficiently by distributing the fixed level algorithm.

Algorithm 1 is our fixed level importance splitting algorithm optimised for distribution. We use the terms server and client to refer to the root and leaf nodes of a network of computational devices or to mean independent computational threads on the same machine. In essence, the server manages the job and the clients perform the simulations.

The server initially sends compact representations of the model and property to each client. Thereafter, only the state of the product automaton is communicated. In general, each client returns terminal states of simulations that reached the current level and the server distributes these as initial states for the next round of simulations. Algorithm 1 optimises this. The server requests and distributes only the number of states necessary to restart the simulations that failed to reach the current level, while maintaining the randomness of the selection. Despite this optimisation, however, the performance of this and other importance splitting algorithms will be confounded by the combination of large state size and properties having short time bounds. Under such circumstances it may be preferable to distribute entire instances of the algorithm, as described above.

The memory requirements of Algorithm 1 are minimal. Each client need only store the state of \(n\) simulations. As such, it is conceivable to distribute simulations on lightweight computational threads, such as those provided by GPGPU (general purpose computing on graphics processing...
units).

Algorithm 1: Distributed Fixed Level Importance Splitting

**input:** $s_1 < s_2 < \cdots < s_m$ is a sequence of scores, with $s_m = s_\varphi$ the score necessary to satisfy property $\varphi$

1. $\hat{\gamma} \leftarrow 1$ is the initial estimate of $\gamma = P(\omega \models \varphi)$
2. server sends compact description of model and observer to $k$ clients
3. each client initialises $n$ simulations
4. for $s \leftarrow s_1, \ldots, s_m$ do
5. each client continues its $n$ simulations from their current state simulations halt as soon as their scores reach $s$
6. $\forall$ clients, client $i$ sends server the number of traces $n_i$ that reached $s$
7. server calculates $\hat{\gamma} \leftarrow \hat{\gamma} n'/kn$, where $n' = \sum n_i$
8. for $j \leftarrow 1, \ldots, kn - n'$ do
9. server chooses client $i$ at random, with probability $n_i/n'$
10. client $i$ sends server a state chosen uniformly at random from those that reached $s$
11. server sends state to client corresponding to failed simulation $j$, as initial state of new simulation to replace simulation $j$

**output:** $\hat{\gamma}$

4 Linear Temporal Logic for Importance Splitting

High performance SMC tools, such as [JLS12b, BCLS13], avoid the complexity of standard model checking by compiling the property to a program of size proportional to the formula and memory proportional to the maximum sum of nested time bounds. This program implicitly encodes the model checking automaton, but is exponentially smaller. For example, the property $X^k \varphi$ can be implemented as a loop that generates $k$ simulation steps before returning the truth of $\varphi$ in the last state; the property $\vartheta U^k \varphi$ can be implemented as a loop that generates up to $k$ simulation steps while $\vartheta$ is true and $\varphi$ is not true, returning the value of $\varphi$ in the last state otherwise. If $\vartheta$ and $\varphi$ are atomic, the programs require just $O(\log k)$ bits of memory to hold a loop counter.

In contrast, the nested property $F^k(\vartheta \lor G^{i_2} \varphi)$ has an $O(k^2)$ memory requirement. If $\vartheta$ is not true on step $i < k1$ it may be necessary to simulate up to step $i+k2$ to decide subformula $G^{i_2} \varphi$. If $\vartheta \lor G^{i_2} \varphi$ turns out to be false on step $i$, it will then be necessary to consider the truth of $\vartheta$ on step $i+1$, noting that the last simulated step could be $i+k2$. To evaluate this formula it is effectively necessary to remember the truth of $\vartheta$ on $O(k^2)$ simulation steps. Similar requirements can arise when the until operator ($U$) is a subformula of a temporal operator. In all such cases the sequence of stored truth values become part of the state of the property automaton.

SMC using importance splitting requires that simulations are repeatedly and frequently initialised with the state of the model-property product automaton. If the size of this state is proportional to the time bounds of temporal operators, initialisation may have comparable complexity to simulation. This becomes especially problematic if the state is to be transmitted across rel-
atively slow communication channels for the purposes of distribution. We therefore define a subset of (1), the size of whose automata is not dependent on the bounds of temporal operators:

\[
\phi = X^k \phi | \psi U^k \psi | \neg \phi | \phi \lor \psi | \phi \land \psi | \phi \Rightarrow \psi | \psi
\]

\[
\psi = X^k \psi | F^k \psi | \alpha
\]

(5)

The semantics of (5) is the same as (1), but (5) restricts how temporal operators may be combined. In particular, \(U\) may not be the subformula of a temporal operator other than \(X\) and temporal operators that are subformulas of other temporal operators may not be combined with Boolean connectives. Temporal operators containing other temporal operators as subformulas may, however, be combined. This logic expresses many useful properties, including nested bounded temporal properties that are not implemented in the numerical model checker Prism\(^1\).

5 Lightweight Observers for Importance Splitting

To facilitate the construction of score functions we implement the logic given by (5) as a set of nested observers. Each observer corresponds to either a temporal operator, a Boolean operator acting on temporal operators, or as a predicate describing an atomic property. In our implementation observers are written in a syntax based on the commonly used reactive modules language [AH99], using the notion of ‘guarded commands’ [Dij75] with sequential semantics. The observers are easily implemented in other modelling languages.

An observer comprises a set of guarded commands, any number of which may be enabled and executed on a given simulation step. Updates are performed in syntactic order after all guards have been evaluated, hence the update of one command does not affect the guards of commands in the same observer. In general, the output of one observer is the input to another and observers are therefore executed in reverse order of their nesting.

Observers evaluate states as they are generated by the simulation. Since it may not be possible to decide a property before seeing a certain number of states, observers implement a three valued logic. In Figs. 1, 2 and 3 we use the symbols \(\?, \top\) and \(\bot\) to denote the three values undecided, true and false, respectively. The state of an observer changes only when at least one of its inputs is decided. An observer may reach a deadlock state (no commands enabled) once its output is decided and cannot be changed by further input. A simulation terminates when the output of the root observer is decided, i.e., the property is decided. Simulations may also be paused by the importance splitting algorithm if the score reaches a desired level.

Observers implementing the same temporal operator behave differently according to their level of nesting within a formula. We therefore distinguish outer and inner temporal observers. The temporal operators closest to the root of any branch of the syntax tree induced by a formula are implemented by outer observers. Their output proceeds from undecided to either true or false and then does not change. Inner observers encode temporal operators that are the subformulas of other temporal operators. Their output proceeds from undecided to a possibly alternating sequence of true, false and undecided values because their enclosing operator(s) cause them to evaluate a moving window of states in the execution trace. The inner and outer variants of \(X, F\)

\(^1\) www.prismmodelchecker.org
and \( G \) are closely related—outer observers are essentially simplified inner observers. When \( U \) is a subformula of \( X \), however, the \( X \) is implemented as a delay within the \( U \) observer.

In what follows we describe the important aspects of the various observers that implement (5). The accompanying figures include diagrammatic representations of how the observers work and sets of commands written in the form \( \text{predicate : update} \). Each observer has Boolean output variables \( o \) and \( d \) to indicate respectively the result and whether the property has been decided (observers for atomic formulas omit \( d \)). Observers for temporal operators take discrete time bound \( k \) as a parameter and use a counter variable \( w \) (\( U \) uses counter variables \( w' \) and \( w'' \)). Inner temporal operators make use of an additional counter, \( t \) (\( U \) uses \( t' \) and \( t'' \)). The inputs of observers are Boolean variables \( o' \) and \( o'' \), with corresponding decidedness \( d' \) and \( d'' \).

### Connective Observers

These observers implement Boolean connectives at syntactic level \( \phi \) in (5) and take advantage of the equivalences \( \text{false} \land ? = \text{false}, \text{true} \lor ? = \text{true}, \text{false} \Rightarrow ? = \text{false} \) and \( ? \Rightarrow \text{true} = \text{true} \), for any truth value of \( ? \). Figure 1a describes the observer for conjunction and Fig. 1b describes the observer for implication. The observer for disjunction may be derived from that of conjunction by negating all instances of \( o' \) and \( o'' \), and by exchanging \( o \leftarrow \text{true} \) and \( o \leftarrow \text{false} \). Negation is implemented by inverting the truth assignment of the observer to which it applies, i.e., by exchanging \( o \leftarrow \text{true} \) and \( o \leftarrow \text{false} \). The connectives may be combined with themselves and with outer temporal operators. Boolean connectives that apply only to atomic properties (i.e., syntactic level \( \alpha \)) are implemented directly in formulas within observers for atomic properties.

### Inner Temporal Observers

These observers act on a moving window of states created by an enclosing temporal operator. The output may pass from one decided value to the other and also become undecided.

---

![Figure 1: Connective observers. Initially \( d = \text{false} \).](image-url)
Distributed Verification of Rare Properties using Importance Splitting Observers

1. $-d \land d' \land w < k : w \leftarrow w + 1$
2. $d' \land w = k : d \leftarrow \text{true}, o \leftarrow o'$
3. $d \land -d' : d \leftarrow \text{false}$

(a) $o \leftarrow X^k o'$

1. $-d \land d' \land -o' \land w < k : w \leftarrow w + 1$
2. $d' \land o' : o \leftarrow \text{true}, d \leftarrow \text{true}, t \leftarrow w$
3. $d' \land -d' \land t = 0 \land w = k : d \leftarrow \text{true}, o \leftarrow \text{false}$
4. $d \land d' \land -o' \land t = 0 \land w < k : d \leftarrow \text{false}$
5. $d \land d' \land -o' \land t > 0 : t \leftarrow t - 1$
6. $d \land -d' : d \leftarrow \text{false}$

(b) $o \leftarrow F^k o'$

Figure 2: Observers for inner temporal operators. Initially $w = t = 0, d = \text{false}$.

Figure 2a describes the observer for $X^k$. Command 1 counts decided input states until bound $k$ is reached. Thereafter command 2 sets the output decided and equal to the value of the input.

Figure 2b describes the observer for $F^k$. While decided inputs are not true, command 1 increments $w$ from 0 to $k$. If at any time the input is true, command 2 sets the output to true and the “true-counter” $t$ is set to $w$. Command 5 decrements $t$ on subsequent false inputs. The output remains true while $t > 0$. If $w$ reaches $k$ while $t = 0$, command 3 sets the output to false.

The observer for $G^k$ may be derived from that of $F^k$ by negating all instances of $o'$ and $-o'$, and by exchanging $o \leftarrow \text{true}$ and $o \leftarrow \text{false}$.

**Outer Temporal Observers**

The outer observers for $X^k$ and $F^k$ are not illustrated but may be derived from their respective inner observers given in Fig. 2. For $X^k$, command 3 is removed and the guard of command 2 is strengthened with $-d$. For $F^k$, commands 4, 5 and 6, together with all references to counter $t$, are removed, while the guards of commands 2 and 3 are strengthened by $-d$. The outer observer for $G^k$ can be derived from that of $F^k$ in the same way as described for inner temporal observers.

Figure 3 describes the observer for properties of the form $X^k (\varphi)$, which can be simplified to implement properties of the form $\varphi$. Since $\varphi$ may be temporal formulas that are satisfied on different simulation steps in arbitrary order, the observer employs variables $w'$ and $w''$ to respectively count the sequences of $\neg \varphi$ and $\varphi$ (commands 3 and 5). Variable $t'$ then records the position of the first $\varphi$ (command 4), while $t''$ records the position of the last $\varphi$ (command 5). Using $t'$ and $t''$, commands 7 and 8 are able to determine if the property is satisfied or falsified, respectively. The $X^k$ part of the formula is implemented by initialising variables $w'$ and $w''$ to $-kX$, forcing the observer to ignore the first $kX$ decided values of $\varphi$. In the case of properties of the form $\varphi$, $w'$ and $w''$ are initialised to 0 and the automaton may be simplified by removing commands 1 and 2 and all instances of expressions $w' \geq 0$ and $w'' \geq 0$.  

222 Distributed Verification of Rare Properties using Importance Splitting Observers
6 Case Studies

We have implemented our importance splitting framework in PLASMA-LAB [BCLS13] and demonstrate its use on three case studies whose state space is intractable to numerical model checking. The following results do not seek to promote a particular methodology (adaptive or fixed level algorithm, distributed or single machine), but serve to illustrate the flexibility of our platform. The software, models and observers can be downloaded from our website\(^2\). The leader election and dining philosophers models are also illustrated on the PRISM case studies website\(^3\).

For each model we performed a number of experiments to compare the performance of the fixed and adaptive importance splitting algorithms with and without distribution, using different simulation budgets and levels. Our results are illustrated in the form of empirical cumulative probability distributions of 100 estimates, noting that a perfect (zero variance) estimator distribution would be represented by a single step. The results are also summarised in Table 1. The probabilities we estimate are all close to \(10^{-6}\) and are marked on the figures with a vertical line. Since we are not able to use numerical techniques to calculate the true probabilities, we use the average of 200 low variance estimates as our best overall estimate.

As a reference, we applied the adaptive algorithm to each model using a single computational thread. We chose parameters to maximise the number of levels and thus minimise the variance for a given score function and budget. The resulting distributions, sampled at every tenth percentile, are plotted with circular markers in the figures. Over these points we superimpose the results of applying a single instance of the fixed level algorithm with just a few levels. We also superimpose the average estimates of five parallel threads running the fixed level algorithm, using the same levels.

The figures confirm our expectation that the fixed level algorithm with few levels is outperformed by the adaptive algorithm. The figures also demonstrate that the average of parallel instances of the fixed level algorithm are very close to the performance of the adaptive algorithm. The timings given in Table 1 show that the distributed approach achieves these results in less time. For comparison we also include the estimated time of using a simple Monte Carlo

---

\(^2\) projects.inria.fr/plasma-lab/importance-splitting

\(^3\) www.prismmodelchecker.org/casestudies
Distributed Verification of Rare Properties using Importance Splitting Observers

Estimate × 10^7

Cumulative probability

1 3 10

0 0.2 0.4 0.6 0.8

adaptive
parallel fixed
single fixed

Figure 4: Leader election.

Estimate × 10^7

Cumulative probability

5 10 20 40

0 0.2 0.4 0.6 0.8

adaptive
parallel fixed
single fixed

Figure 5: Dining philosophers.

(MC) estimator to achieve the same standard deviation. Importance splitting gives more than three orders of magnitude improvement in all cases. All results were generated using an Intel Core i7-3740 CPU with 4 cores running at 2.7 GHz.

In the remainder of this section we briefly describe our models and their associated properties and score functions.

Leader Election

Our leader election case study is based on the PRISM model of the synchronous leader election protocol of [IR90]. With \( N = 20 \) processes and \( K = 6 \) probabilistic choices the model has approximately \( 1.2 \times 10^{18} \) states. We consider the probability of the property \( G^{420} \neg \text{elected} \), where \( \text{elected} \) denotes the state where a leader has been elected. Our chosen score function uses the time bound of the \( G \) operator to give nominal scores between 0 and 420. The model constrains these to only 20 actual levels (some scores are equivalent with respect to the model and property), but with evenly distributed probability. For the fixed level algorithm we use scores of 70, 140, 210, 280, 350 and 420.

Dining Philosophers

Our dining philosophers case study extends the PRISM model of the fair probabilistic protocol of [LR81]. With 150 philosophers our model contains approximately \( 2.3 \times 10^{144} \) states. We consider the probability of the property \( F^{30} \text{Phil eats} \), where \( \text{Phil} \) is the name of an arbitrary philosopher. The adaptive algorithm uses the heuristic score function described in [JLS14], which includes the five logical levels used by the fixed level algorithm. Between these levels the heuristic favours short paths, based on the assumption that as time runs out the property is less likely to be satisfied.
### Dependent Counters

Our dependent counters case study comprises ten counters, initially set to zero, that with some probability dependent on the values of the other counters are either incremented or reset to zero. This can be viewed as modelling an abstract computational process, a set of reservoirs of finite capacity, or as the failure and repair of ten different types of components in a system, etc. With a maximum count of 10, the model has approximately $2.6 \times 10^{10}$ states.

We consider the probability of the property $X^1(\neg \text{init} U^{1000} \text{complete})$, where init and complete denote the initial state and the state where all counters have reached their maximum value. Our score function ranges over values between 0 and 99, but the probabilities are not evenly distributed. With a budget of 500, uniformly distributed fixed scores fail to produce traces that satisfy the property until the difference between the last two levels is about 5. Note that our budget is limited to only 500 simulations due to the length of the traces that must be stored by the adaptive algorithm. We maintain this budget for the fixed level algorithm to simplify comparison. After a small amount of trial and error, we adopted fixed scores of 80, 90, 95 and 99.

### 7 Challenges and Prospects

Our results demonstrate the effectiveness and flexibility of our framework with discrete time properties applied to standard case studies. Future challenges include industrial scale examples and the implementation of continuous time properties. We also intend to provide proofs of the correctness of our observers and of our logic’s memory requirements.

Although the manual construction of score functions adds to the overall cost of using importance splitting, we believe that distribution relaxes the need for these to be highly optimised. We nevertheless expect that it will be possible to construct good score functions automatically using statistical learning techniques.
Bibliography


Model-based WCET Analysis with Invariants

Bojan Nokovic, Emil Sekerinski

{nokovib, emil}@mcmaster.ca, McMaster University, Department of Computing and Software, Canada

Abstract: The integration of worst case execution time (WCET) analysis in model-based designs allows timing problems to be discovered in the early phases of development, when they are less expensive to correct than in the later phases. In this paper, we show how model-based WCET analysis can improve timing calculations compared to program-based WCET analysis. The models are described by hierarchical state machines with concurrency, probabilistic transition, stochastic transitions, costs/rewards attached to states and transitions, and invariants attached to states. In these models, user-specified invariants serve to check the correctness of designs by restricting allowed state configurations. Our contribution is to use invariants additionally to determine transition combinations (paths) that can be eliminated from the WCET analysis, with the help of a decision procedure, thus making the analysis more precise. The assembly code of transitions for a specific target is generated and execution time for that code calculated. From the model, a probabilistic timed automaton (PTA) or Markov decision process (MDP) can be created. On that model, execution times of transitions are calculated as costs.

Keywords: Hierarchical state-machines, Compiler, Worst-case analysis

1 Introduction

The full integration of embedded systems into larger products leads to an increased reliance on their correct service: in modern embedded systems, not only functional requirements and liveness, but also strict timing constraints must be satisfied [LM11]. Properties of a program like a loop bound, exponential path space, path feasibility, and properties of the hardware like memory access time, determine program execution time. An established approach is to insert constraints as deadlines in basic blocks of the source code, and after compiling, the assembly code is used in a low-level analysis to determine execution time [LS14]. If the calculated times are less than or equal to the specified deadlines, the timing constraints are satisfied.

In this work we consider model-based WCET analysis. The models are expressed as pCharts, a formalism for reactive systems based on hierarchical state machines with invariants, probabilistic transitions, timed transitions, stochastic timing, and costs/rewards [NS13, NS14]. The pCharts formalism underlies pState, a tool for the specification, design, quantitative analysis, and implementation of embedded systems [Nok15]. With pState users can specify what a system does; then from the specified structure the tool generates executable code which determines how the system works, and finally by generating input code for a model checker it is possible to reason why the system does what it does (or fails at what is specified to do).
We show that state invariants, originally written by developers to capture design decisions, can be used to improve WCET analysis. The bound for the execution time can be specified directly on a transition and is automatically checked for feasibility. If the calculated WCET is greater than the bound, the design has to be alternated or the design target updated by selecting a different processor, increasing the clock of the processor, etc. This allows a design process in which low-level timing considerations can impact the model structure, making the model potentially “less high-level”. The advantage of this holistic modelling approach is the simplicity gained by having only a single model.

Statecharts [Har87], hierarchical state machines with concurrency and broadcasting are used as a graphical specification tool for reactive systems, but they are also executable and compilable [Har07]. Similarly to iState [SZ01], pState implements an event-centric semantics in which external events trigger immediate execution, unlike the state-centric semantics of UML and Statemate, in which events are data (in queues). These interpretations are also called requirements-oriented and implementation-oriented semantics in [EJW02]. For example, the TCM toolkit for conceptual modelling also follows the requirements-oriented semantics [DW97]. Our event-centric semantics simplifies the correctness analysis, as every transition is an atomic step, even in presence of broadcasting. The semantics is suitable for small embedded systems, e.g. with micro-controllers with limited memory, where events are processed quickly enough so that no queuing of events is necessary.

Following the trend that part of the modern software engineering process is the integration of WCET [KP05], we implement automatic early WCET calculation. In our holistic modelling approach, from the same model, executable code generation and quantitative analysis is possible. Timing-analysis is divided into tasks, (1) analyzing control-flow properties and (2) calculating execution time of instructions or basic blocks of instructions [LS14]. From a pCharts model, pState generates code and automatically calculates the execution time of instructions. To this end, bodies of transitions are normalized to nested if-then-else statements with multiple assignments and a satisfiability modulo theories (SMT) checker, Yices [Dut14], is used to analyze control-flow properties, with the use of invariants. The number of execution cycles is then calculated for each basic block. The calculation of the upper bound of execution time of programs is not possible in general (the halting problem), but currently we neither allow loops nor recursive calls, so in our models programs always terminate, and automatic calculation of the upper bound is possible.

State invariants are conditions that are attached to states in a state hierarchy and specify what has to hold in that state [Sek08]. Every incoming transition to a state must ensure that its state invariant holds and every outgoing transition can assume that the invariant holds. This gives a method for checking a chart against an invariant annotation. State invariants can express safety of an embedded system or consistency of a software system. Invariants used in this way are related to flow facts annotations [KP05]. The invariants are used before in finding unfeasible paths, e.g. see [HGB”08, EES01, GJK09] for overviews. Our contribution here is that we do not rely on the user suggesting or a tool automatically deriving invariants for the purpose of finding unfeasible paths, but we use model-based invariants which users state for checking the correctness of models. If a stated invariant is not true, it is ignored for unfeasible path analysis.

---

1 Examples of automatic calculation of WCET from pCharts models can be found at http://pstate.mcmaster.ca/
This extends to invariants in models with *concurrency* and *broadcasting*. The specific benefit of this approach is that invariants can be used in early WCET calculation, without need for data flow analysis. As a simple example, in Fig. 1 the invariant

\[ \text{in PortA.Off} \lor \text{in PortB.Off} \]

states that PortA is in Off, or PortB is in Off, or both are in Off. They can not be in Through simultaneously, therefore the code will never execute *tick* transitions in both PortA and PortB at the same run. This is not the strongest invariant; a stronger invariant would be

\[ \text{in PortA.Off} \oplus \text{in PortB.Off} \]

where \( \oplus \) is exclusive OR. For the execution time calculation of *tick* event, there is no difference if we use either one, as both invariants do exclude the possibility that PortA and PortB are in *Through* at the same time. The event *tick* is generated by a timer periodically, every 10ms. The time for executing both *tick* transitions should not contribute to the WCET of transition \( t \). Informally, in this case:

\[
\text{WCET}(t) = \max(\text{WCET}((10\text{ms/tick}) \parallel (\text{tick/readA})), \text{WCET}((10\text{ms/tick}) \parallel (\text{tick/readB}))).
\]

Without the invariant, WCET of the transition \( t \) would be calculated as

\[
\text{WCET}(t) = \text{WCET}((10\text{ms/tick}) \parallel (\text{tick/readA}) \parallel (\text{tick/readB})).
\]

Transition conditions under invariants are checked by the SMT solver Yices as the backend tool. When parallel composition is sequentialized, all combinations of executable paths are created. If some conditions are never satisfied, the associated paths should be excluded.
In Sec. 3.3 we show how conditions can be automatically verified and unreachable paths removed from the executable code. Besides simplifying the generated code, this makes the WCET calculation more accurate. We illustrate in Sec. 3.4 how user-specified state invariants improve WCET calculation. Accurate WCET analysis can be done only at assembly/object code level [KP05]; therefore our model of nested hierarchical states has to be translated into assembly code. A brief description of this process is given in Sec. 4.

2 Related Work

Our method of WCET calculation is static, or verification-based. The upper bound of task execution is estimated on the code itself taking the hardware architecture into account. Static analysis guarantees that execution time will not exceed the upper bound, but sometimes this estimation may be too pessimistic, which can be confirmed by measurement-based methods. Other techniques for WCET calculations are simulations and path analysis for the calculation of execution scenarios [KP05]. In general, methods of static program analysis include value analysis, CFA, processor behaviour analysis, and symbolic simulation. Some well-known WCET tools for static program analysis are aIT, Bound-T, OTAWA, and SWEET [WEE+08].

A method for analyzing real-time behaviour of reactive synchronous system with special focus on statecharts is described in [EA99]. In addition to WCET and schedulability analysis of statecharts models, worst case response time (WCRT) of synchronous models is introduced. WCRT includes possible interferences by other programs. The method for calculation of WCET/WCRT is implemented in StateMate [HN96].

The main difference between UML State Machine Diagrams [Fow03] and pCharts is related to the specification of internal state activities. UML allows two types of internal state activities, do-activities, and regular activities specified by entry and exit keywords. Regular activities occur “instantaneously” while do-activities can take finite time and can be interrupted. pCharts does not allow the explicit specification of internal state activities, but the entry part of the activity can be specified in the body of incoming transitions, and exit part can be specified in the body of outgoing transitions. The execution of a transition is instantaneous, like in UML, and cannot be interrupted by another event.

The UML Profile for Modelling QoS provides facilities for defining a wide variety of QoS requirements and properties [KJ10], divided into categories: performance, dependability, security, integrity, coherence, throughput, latency, efficiency, demand, reliability, and availability. In pCharts, a transition deadline can be specified directly on the model. The specified deadline is similar to the QoS latency property, since both refer to a time interval during which a response to an event must be completed. Other properties like performance, dependability, throughput are expressible in pCharts models.

The behavioural constructors used in the OMG system modelling language SysML consists of activity diagrams, sequence diagrams, state machine diagrams, and use case diagrams [OMG15a]. State machine diagrams in SysML are based on the standard UML state machine concept, so does not support state invariants.

The UML profile for real-time systems (UML-RT) extends the basic UML concepts to facilitate the design of complex real-time systems [KJ10]. UML-RT is industrial standard and can
be used to design event-driven real-time systems. The standard is primarily focused on architecture specification of real-time systems, but the behaviour can be modelled by state machine diagrams. UML-RT state machines do not support nested concurrent states and automatic WCET cacluation whereas pCharts do.

WCET by model checking is explored in [Met04], showing that model checking is adequate for low level WCET analysis dealing with caches and pipelines. Using model checking, but not dealing with caches and pipelines, \textit{pState} first predict the timing of program instruction for dedicated hardware, and then the time of the specified path is calculated by model checker using predicted timing of program instructions. For basic blocks, the number of processor ticks is introduced as the cost of a transition. The runtime is calculated as the cost on a specified path. A process of model-based design with static program analysis tools which guarantees safety requirements is described in [CBC08]. It is shown how tools for determining the precise bounds of resource usage like StackAnalyzer and aiT can be integrated with automatic code generators. It is known that timed-automata and model checker like UPPAAL are suited for WCET calculation [BC11]. METAMOC [DOT+10] reduces computing of WCET to finding the longest path of CFG represented as timed-automata. User-specified invariants are not part of any of these approaches.

The MARTE UML profile [OMG15b] for modelling and analysis of real-time and embedded systems is intended to replace the existing UML Profile for Schedulability, Performance and Time (SPT) [KJ10]. Behaviour scenarios are annotated with expressions of describing some non-functional properties (NFP), like \textit{NFP\_Duration = \text{wcet1}}. The annotated model indicates to the analysis tools what property has to be computed. While WCET calculation is part of the MARTE extension, we are not aware that any of tools that implements MARTE profile like Rational Software Architect, MagicDraw, or Papyrus UML takes into account invariants while calculate WCET of events.

It is shown that state invariants as functional properties can be used to improve estimation of WCET on synchronous models [RMPC13]. Some of the functional properties are hard to find, and sometimes even impossible by static analysis of programs specified by high-level models. Invariants can be used to exclude infeasible program paths, which can not be find by static analysis. In our approach invariants are part of pCharts syntax, and semantic information represented by state invariants are specified at design level.

The application area for which \textit{pState} was originally developed are embedded systems with 8 and 16 bit micro-controllers, e.g. as used in active RFID tags [Pau06]. Those applications have a small number of states, so the problem of scalability of model checkers is not a concern. Data processing elements known as prefetching and delayed branching and branch prediction, which can contribute to WCET on advanced microprocessors, are not present in those simple micro-controllers. We assume that the execution time of the instruction is independent of the instruction order.
3 Model-based WCET

3.1 Preliminaries

Every event is translated to an intermediate representation consisting of skip, assignments, if-then-else statements, and parallel compositions, which arise from broadcasting. Following rules are used to eliminate parallel composition. Let $b$ be boolean expressions, $Q, R, S$ statements, $x, y$ variables, $e, f$ expressions. Multiple assignment $x, y := e, f$ is translated to $x := e$; $y := f$; if $x$ not in $f$. The transformation rules are:

\[
\begin{align*}
x := e \parallel y := f &= x, y := e, f \quad (1) \\
(if \ b \ then \ Q \ else \ R) \parallel S &= if \ b \ then \ (Q \parallel S) \ else \ (R \parallel S) \quad (2) \\
(Q \parallel R) \parallel S &= Q \parallel R \parallel S \quad (3) \\
Q \parallel \text{skip} &= Q \quad (4) \\
if \ b \ then \ Q \ else \ R &= (b \rightarrow Q) \parallel (\neg b \rightarrow R) \quad (5)
\end{align*}
\]

Applying these rules transforms a statement to nested if-then-else statements with the innermost statements being multiple assignments. That form is used for WCET as follows. On several examples we show in detail how WCET is calculated by $pState$.

3.2 Transition with Specified Deadline

In Fig. 2, a transition from state $S$ to state $T$ on event $E$ is shown. Guard $g$ represents the condition which must be satisfied for the transition to happen, body $a$ is the action which is executed when the transition is taken, and $d$ is the specified deadline for that transition. Guard, action, and deadline are optional. The operation on event $E$, $op(E)$, is given by:

\[
\begin{align*}
op(E) &= \text{in}(S) \land (x = 0) \rightarrow \text{goto}(T) \parallel (x := x + 1)
\end{align*}
\]

To answer the question if WCET of transition from $S$ to $T$ takes at most $d$ cycles, we need to calculate the execution time of event $E$. For the execution time calculation we need to translate the corresponding event operation $op(E)$ into assembly code. If the target is the RISC PIC16F6xx micro-controller, the code is in PIC assembly language, which we call $picasm$. The instruction set has about 35 instructions divided into three groups, byte-oriented, bit-oriented and control operations. Most of the instructions take one processor cycle, except jump and subroutine call which take two cycles. The operation $op(E)$ can be executed if the chart is in state $S$, written as $\text{in}(S)$ and guard $x = 0$ holds. The effect of transition is execution of if-branch $x := x + 1$. The chart is going to state $T$, written as $\text{goto}(T)$. The action $x := x + 1$ and $\text{goto}$ statements are specified as parallel (independent) composition. The WCET of $op(E)$ is:
The composition \( \text{goto}(T) \parallel (x := x + 1) \) is well-defined only if the variables assigned to \( \text{goto}(T) \) and \( (x := x + 1) \) are disjoint, but \( \text{goto}(T) \) and \( \text{action} \) may read the variables assigned by the other. Sequentialization of multiple assignments may need extra variables for temporary storage. The time to copy values into these variables has to be taken into account. Therefore the sequentialization of multiple assignments should be done with minimum number of auxiliary variables.

Statement \( \text{goto}(T) = r := T \) and body of the transition \( a \) do not have assigned joint variables, and do not read variables assigned to each other. Therefore this parallel composition is well defined and there is no need for an extra variable in creation of sequential composition, so

\[
\text{goto}(T) \parallel (x := x + 1) = \text{goto}(T) ; x := x + 1
\]

and WCET of this parallel composition is

\[
\text{WCET}(\text{picasm}(\text{goto}(T)) \parallel (x := x + 1)) = \text{WCET}(\text{picasm}(\text{goto}(T))) + \\
\text{WCET}(\text{picasm}(x := x + 1))
\]

The generated assembly code is shown in Listing 1. Variable \( r \) represents the root state in which \( \mathcal{S} \) and \( \mathcal{T} \) are nested. Next, we calculate the number of cycles for each component. For \( \text{in}(\mathcal{S}) \) the number of cycles is 3. The WCET of transition on event \( E \) calculated by \( \text{pState} \) is

\[
\text{WCET}(\text{picasm}(\text{op}(E))) = 3 + 3 + 2 + 2 = 10 \text{cycles}
\]

If \( d \geq 10 \), we can say that WCET of the transition on event \( E \) satisfy specified requirements for particular target, RISC PIC16F6xx micro-controller.

**Note:** We may leave out the translation to assembly if understood from the context. For the transition on \( E \), we let WCET(\( \text{op}(E) \)) stands for WCET(\( \text{target asm}(\text{op}(E)) \)), and for statement \( S \), WCET(\( S \)) stands for WCET(\( \text{target asm}(S) \)).

Listing 1: Generated assembly code for \text{op}(E)

\begin{verbatim}
picasm(op(E)) =
movf r , W ; W := r
xorlw S ; W := W XOR S
btfss STATUS, 0x2 ; If Z = 1 skip
goto CONTINUE_0
GUARD_0
movf x, W ; W := x
xorlw 0 ; W := W XOR 0
btfss STATUS, 0x2 ; If Z = 1 skip
goto CONTINUE_0
ACTION_0
movlw 1 ; W := 1
addwf x, 1 ; x := W + x
movlw T ; W := T
movwf r ; r := W
CONTINUE_0
\end{verbatim}
### 3.3 WCET of Parallel Composition

A transition from state $T_1$ to $T_2$ on event $E$ is shown in Fig. 3. Operation $op(E)$ on event $E$ returns a set of prioritized guarded commands $b \rightarrow S$ according to the algorithm in [NS14]. Suppose

$$stat_i \equiv \text{if } c_i \text{ then } S_i$$

where $i \in \{1, 2\}$, $c_i$ are boolean expressions, and $S_i$ statements.

![Figure 3: Parallel composition](image)

The operation on the event $E$ for the transition shown in Fig. 3 is the parallel composition of three statements.

$$op(E) = \text{in}(T_1) \rightarrow \text{goto}(T_2) \| \text{stat}_1 \| \text{stat}_2 = \text{in}(T_1) \rightarrow \text{goto}(T_2) \| (\text{if } c_1 \text{ then } S_1) \| (\text{if } c_2 \text{ then } S_2)$$

The parallel composition of $\text{stat}_1 \| \text{stat}_2$ amounts to:

$$\begin{align*}
&(\text{if } c_1 \text{ then } S_1) \| (\text{if } c_2 \text{ then } S_2) \\
&= \text{by definition} \\
&(\text{if } c_1 \text{ then } S_1 \text{ else skip}) \| (\text{if } c_2 \text{ then } S_2 \text{ else skip}) \\
&= \text{by (2)} \\
&(\text{if } c_1 \text{ then } S_1 \| \text{if } c_2 \text{ then } S_2 \text{ else skip}) \text{ else skip} \| (\text{if } c_2 \text{ then } S_2 \text{ else skip}) \\
&= \text{by (4) and symmetry of } \| \\
&(\text{if } c_1 \text{ then } \text{if } c_2 \text{ then } S_1 \| S_2 \text{ else } S_1) \text{ else } (\text{if } c_2 \text{ then } S_2 \text{ else skip})
\end{align*}$$

For example, if Yices verifies that $(c_1 \land c_2)$ and $(\neg c_1 \land \neg c_2)$ are not satisfiable and $S_i$ are multiple assignments, the WCET of operation $E$ is calculated by $p\text{State}$ as:

$$\text{WCET}(op(E)) = \text{WCET}(\text{in}(T_1)) + \text{WCET}(\text{goto}(T_2)) + \max(\text{WCET}(c_1 \land \neg c_2 \rightarrow S_1), \text{WCET}(\neg c_1 \land c_2 \rightarrow S_2))$$

That is, choices with an unsatisfiable condition are excluded.

### 3.4 WCET Taking State Invariants into Account

State invariants are used to improve WCET calculation by providing additional information, which makes the analysis feasible and tight. Figure 1 presents the process of reading redundant digital ports. If we assume that our target is PIC16F636, we can specify $\text{readA}$ action as reading port A into predefined variable $ra$ by $ra := PORTA$, where $PORTA$ is port A of the microcontroller. Similarly, we specify reading from port B.
If the signal is lost on port A, the event toggle is generated and the system starts reading data from port B. From the generated code, the WCET of transition in state Timer is automatically calculated as $10\text{ms} + 160\text{ cycles}$. However, when PortB is in Through, PortA must be in Off and vice versa. So it is not possible to read both ports A and B at the same tick. But this is not taken into account when the WCET is calculated. By adding the invariant in PortA.Off $\lor$ in PortB.Off we specify that PortA and PortB can not be in the Through states at the same time. pState verifies this automatically and excludes one of the transition on tick when calculating the WCET. The calculated WCET is now $10\text{ms} + 148\text{ cycles}$. The complete generated assembly code as used for WCET calculation is at http://pstate.mcmaster.ca/.

3.5 WCET Taking into Account Timed Transitions

We consider two timed transitions $k$ and $l$ as in Fig. 4. Transition $k$ takes place exactly $t_1$ time units after the state $S$ is entered, if event $E$ does not occur in the meantime. Similarly, transition $l$ happens $t_2$ timed units after state $T$ is entered. For timed transitions $k$ and $l$ we need to add time of scheduling transition $l$, and time needed to cancel transition $k$.

The operation on event $E$ for the transition in Fig. 4 includes scheduling timed transition $l$ and cancelling timed transition $k$ is

$$op(E) = \text{in}(S) \land g \rightarrow (\text{goto}(T) \parallel a); \text{ schedule } l; \text{ cancel } k$$

The transition is considered to be correct if the WCET is less or equal to specified deadline $d$, $\text{WCET}(\text{picasm}(op(E))) \leq d$. A scheduler for timed transitions is created separately, which is called from the generated code for scheduling and cancelling transitions. Scheduling of timed transitions is the process of adding new events into a data structure of scheduled events, sorted by time and priority. Procedure $\text{cancel}$ contains a loop to search for the scheduled timed transition which has to be removed. When the first due-to event is removed, other events are shifted down. In the current implementation of the scheduler, the time to schedule an event if the data structure is empty is 52 processor cycles. The time to go trough the loop and sort events is 141 cycles per event. The WCET of scheduling is less or equal to the number of scheduled transitions multiplied by 141 plus 52 cycles. One run through the loop of a canceled timed transition takes 35 cycles, while the time to shift the scheduled event takes 75 cycles. In the similar way, we have that the WCET of cancelling a transition is less or equal to the number of scheduled transitions multiplied by 35 plus 75 cycles.

$$\text{WCET}(\text{canceling of timed transition}) \leq 35 \cdot \# \text{ of scheduled transition} + 75$$

For the transition on event $E$ from Fig. 4 pState calculates
\[ \text{WCET}(\text{picasm}(op(E))) \leq 10 + 52 + 35 \cdot 1 + 75 = 172 \text{ cycles} \]

Note: Specific processor cycle numbers are related to the current implementation. Any modification of the algorithm may have an impact on those numbers.

### 3.6 WCET of Timed Transition

Consider timed transition \( t \) given in Fig. 5.

![Figure 5: WCET of timed transition](image)

The operation on timed transition \( t \) is given by

\[ op(t) = in(S) \xrightarrow{100ms} \text{goto}(T) \]

The transition takes place 100\( ms \) after state \( S \) is entered, but to calculate the WCET we need to add the time required by the scheduler to start this process.

\[ \text{WCET}(\text{picasm}(op(t))) \leq 100ms + \text{WCET}(\text{picasm}(in(S))) + \text{WCET}(\text{picasm}(\text{goto}(T))) + \text{WCET}(\text{to call "}t\text{" event}) \]

To determine the time of the transition, we use TIMER0 of our target processor. It is programmed to generate an interrupt every 1\( ms \). In the interrupt service routine, the due-time for each scheduled event is decreased. This is done in a loop and the execution time depends of the number of scheduled events.

\[ \text{WCET(to call "}t\text{" event)} = \text{WCET(decrease event time)} + \text{WCET(event execution)} \]

Based on our implementation, the time to decrease the event due-time in the interrupt service routine is 98 cycles if there is only one task scheduled. Timed transition \( t \) is called from the non-terminating main loop. The loop has two parts, polling of timed trigger event and polling of input-output actions. So, the WCET to call event execution is

\[ \text{WCET(event execution)} = \text{WCET(polling timed event)} + \text{WCET(polling input events)} \]

From the assembly code we can count the WCET for polling timed events of 22 cycles, while the WCET of polling input events is only 5 cycles plus the time to execute the input event.

\[ \text{WCET(event execution)} \leq 22 + \sum (5 + \text{input event exec. time}) \cdot \# \text{ of input events} \]

For timed transition \( t \), taking into account calculations from Sec. 3.2, and assuming that (1) there is only one event to execute, and (2) there are no input events, \( pState \) calculates the WCET as:
\[ WCET(\text{picasm}(op(t))) \leq 100\text{ms} + 3 \text{ cycles [in(S)]} + 2 \text{ cycles [goto(T)]} + 98 \text{ cycles [tick]} + 22 \text{ cycles [event execution]} = 100\text{ms} + 125 \text{ cycles} \]

If the target micro-controller runs on 4MHz, the execution time of one cycle is \(\frac{1}{4\text{MHz}/4}\) which is 1\(\mu\text{s}\). In this case, \(WCET(op(t)) = 100.125\text{ms}\). If the specified deadline is \(\Delta \geq 125\mu\text{s}\), the transition execution time is satisfiable. But, if we need \(\Delta \geq 100\mu\text{s}\), we will during the specification phase find-out that the requirement can not be implemented. In that case, possible solution is to select a higher frequency of micro-controller clock. Instead of 4MHz, we can use 8MHz crystal clock, which will fix the problem. By this approach in the early phase of design during specification we can identify some hardware limitations and make appropriate design decisions.

### 4 Assembly Code Generation

Assembly code is created by translating the intermediate code representation. Translation of if-else or case statements is straightforward and it is done by just converting them into assembly syntax. The translation of parallel statements needs extra processing since these have to be converted into sequential compositions as it is shown in Secs. 3.2 and 3.3. For the sequentialization of multiple assignments:

\[ v_1, \ldots, v_n := e_1, \ldots, e_n \]

we may need to create one or more extra variables. The problem of sequentializing multiple assignments can be expressed as follows: for given \(v_i (i \in 1..n)\) and expressions \(e_i\) which are only dependent on these variables, we define the dependency relation to be a directed graph \(G = (N,E)\), where \(N = \{n_i | i \in 1..n\}\), and

\[ E = \{ (n_i, n_j) | i \in 1..n \land j \in 1..n \land e_i \text{ is dependent on } e_j \}. \]

When \(G\) is a acyclic, the sequentialization is trivial. When there is one or more cycles in \(G\), extra variables are necessary to eliminate the cycles. Introducing an extra variable for \(v_i\) will remove \(n_i\) and all connected edges from \(G\). A cycle in \(G\) is broken when any \(n_i\) in the cycle is removed. One node can be in more than one cycles, we can call the number of each \(n_i\) as cycle degree \(c_i\), then removing \(n_i\) would break these \(c_i\) cycles in \(G\). Therefore, sequentialization of multiple assignment can be reduced to the process of removing cycles form the feedback vertex set graph. That is proven to be an NP-complete problem [GJ90]. Because of this, our implementation of sequentialization provides approximate solution. Based on experimentation, for most practical cases where \(n \leq 5\), the approximation provides an optimal solution.

Chart states are generated as enumeration names of states, and variables as integers. On the chart integers are declared as subrange types with lower and upper bound, but target micro-controller allows only subrange that fit into byte. In the generation of assembly code, the technique of delayed code generation [Wir96] is used, which produces optimal addressing modes and register usage for this simple architecture in a single pass. This makes code generation sufficiently fast that WCET analysis can be done interactively.

Scheduler, initialization and I/O actions are not generated from the specification, they are write-once code. In this way we have full control over the structure of the application, similar to the approach described in [IAR99].
5 Summary and Conclusion

We described and implemented a framework for model-based WCET analysis of reactive systems. From a hierarchical representation, executable code in a low level language is generated. On the generated code, WCET of transitions can be calculated by counting the number of assembly instructions execution cycles. Precise WCET determination on complex architecture is a challenging problem, but determination of WCET on 8 and 16 bit micro-controllers is easier since features like multi-stage pipelines and caches are not present. Invariants for improving WCET is independent of the architecture of the target processor. The pState architecture allows in principle external tools for WCET to be plugged in. Existing WCET analyzers like AbsInt [Gmb15] could calculate the WCET of executable code generated by pState, but the invariants should be specified manually in the AbsInt general annotation file; the annotation could in principle be generated by pState as well.

In order to exclude infeasible paths, conditions specified by state invariants and conditions created by sequentialization of parallel compositions are verified by SMT solver Yices. In our specification we do not have loops, so the calculation of WCET is a summation of straight-line code blocks. Actions can contain external calls, e.g. to I/O libraries, that are out of control of pState, but for those timing constraints cannot be specified. Thus, if loops or recursion is necessary, these can be implemented by an external call. We plan to work on allowing loops to be expressed directly in pState.

Bibliography


http://doi.acm.org/10.1145/235321.235322


doi:10.1109/ISORC.2005.19

http://dx.doi.org/10.1007/978-90-481-9929-7


doi:10.1109/MECO.2013.6601339

doi:10.1145/2641483.2641522

http://www.omg.org/spec/SysML/1.3/

http://www.omgmar.te.org/


http://doi.acm.org/10.1145/2516821.2516841


State Distribution Policy for
Distributed Model Checking of Actor Models

Ehsan Khamespanah 1,2, Marjan Sirjani 2, MohammadReza Mousavi 3, Zeynab Sabahi Kaviani 1, and MohamadReza Razzazi 4

1 e.khamespanah@ut.ac.ir , z.sabahi@ut.ac.ir
University of Tehran, School of Electrical and Computer Engineering
2 ehsan13@ru.is, marjan@ru.is
Reykjavik University, School of Computer Science
3 m.r.mousavi@hh.se
Halmstad University, Centre for Research on Embedded Systems
4 Razzazi@aut.ac.ir
Amirkabir University of Technology, School of Computer Eng. and Information Tech.

Abstract: Model checking temporal properties is often reduced to finding accepting cycles in Büchi automata. A key ingredient for an effective distributed model checking technique is a distribution policy that does not split the potential accepting cycles of the corresponding automaton among several nodes. In this paper, we introduce a distribution policy to reduce the number of split cycles. This policy is based on the call dependency graph, obtained from the message passing skeleton of the model. We prove theoretical results about the correspondence between the cycles of call dependency graph and the cycles of the concrete state space and provide empirical data obtained from applying our distribution policy in state space generation and reachability analysis. We take Rebeca, an imperative interpretation of actors, as our modeling language and implement the introduced policy in its distributed state space generator. Our technique can be applied to other message-driven actor-based models where concurrent objects or services are units of concurrency.

Keywords: Distributed Model Checking, State Distribution Policy, Concurrent Objects, Actors, Rebeca

1 Introduction

Providing quality guarantees despite the ever-increasing complexity of computer systems has been and remains a grand challenge. Using formal methods, in general, and model checking [CES86] in particular, has been advocated as a response to this grand challenge. Model-checking tools explore the state space of the system exhaustively to make sure that a given property holds in all possible execution of a system. A major limiting factor in applying model checkers to practical systems is the huge amount of space and time required to store and explore the state space. Generating the state space of large-scale practical systems undoubtedly results in state spaces that cannot fit in the memory of a single computer.

Besides the traditional model-checking reduction techniques, distributed LTL model checking [GMS13, BHR13, VVFB11, BBC05, BBS01, BC06, BC0S06] is a well-known technique to
deal with huge state spaces. In distributed LTL model checking the state space is partitioned into some slices and each slice is assigned to a node. Theoretically, dividing cycle detection in a state space among a number of nodes increases the efficiency of model checking; however, unlike the sequential and the shared-memory parallel algorithms, the efficiency of these algorithms depends on the communication costs [OPE05]. The communication cost directly relates to the distribution policy of states among nodes, as detecting accepting cycles that span over many different nodes requires communication. Another, more fine-grained, representative of communication cost is the number of split transitions; a split transition is a transition between two states, where the hosts of source and destination states are different nodes.

In the present work, we tackle the state distribution policy problem in the state space generation of actor models [Hew72]. We introduce a new state distribution policy based on the so-called Call Dependency Graph (CDG) of actor models. A CDG represents the abstract causality relation among messages of actors (Section 2). Our abstraction is akin to the dynamic representation of actor’s event activation causality proposed by Clinger [Cli81].

The most primitive and widely used distribution policy is random state distribution [GMS13, BHR13, VVFB11, BC0506]. Random state distribution policy distributes states among nodes based on their hash values. Random distribution policy guarantees load balancing. However, it is not an effective technique as cycles are scattered over many different nodes. In [BBC05], state distribution is performed based on the Büchi automata of the properties. LTL model checkers find accepting cycles in the synchronous product of the state space and the Büchi automata of LTL specifications. Therefore, distributing states based on the strongly connected components of the property Büchi automata avoids creation of split cycles in the state space. This way, there is no need for communication among nodes for detecting accepting cycles. In practice, the corresponding Büchi automata of LTL properties do not have many strongly connected components. Hence, this approach does not work efficiently in most practical cases.

In [OPE05], another state space distribution policy is suggested to improve the locality of cycles. This policy is based on the static analysis of an abstracted model and detects may or must transition relations among states [LT88]. Based on this analysis, if two states have a must relation, they should be stored in the same node. We use a similar idea in our state distribution policy and show that using the CDG improves the locality of cycles by reducing the split transitions in the state space. In other words, we find the must relations among the states of actor models using the CDG. Our technique is applicable to other service-oriented models where the unit of concurrency can be modeled as an autonomous active object and message passing is the only way of communication. To illustrate the applicability of our method, we implement it in the distributed model checker of Rebeca, which is an actor-based language for modeling and model checking of reactive systems (Section 3). The experimental results of using CDG illustrate that the number of split transitions is reduced significantly by up to 50% (Section 4). We also discuss possible extensions of our work and possible application domains for it (Section 5).

In a nutshell, the contributions of this paper are as follows:

− Introducing the notion of call dependency graph (CDG) for actor models as an abstract representation for message passing causality,

− Presenting the relation between the cycles in the CDG and the cycles in the state space of a model,
Adapting the notion of CDG in order to define a state distribution policy,
Implementing the proposed techniques in a distributed model checking tool, and
Providing experimental results and measuring the efficiency of our technique by means of
a number of case studies.

2 Call Dependency Graph of Actor Models

The actor model [Hew72, Agh90b, AMST97, Agh90a] is a well-established paradigm for
modeling distributed and asynchronous systems. In this model, actors encapsulate the concept of
concurrent behavior. Each actor provides services that can be requested by other actors through
sending messages to the provider. Messages are put in the message buffer of the receiver; the
receiver takes the message and executes the requested service, and consequently, may send some
messages to other actors. The source code of a simple actor model is shown in Figure 1. This
model consists of two actors ac1 and ac2, each of which provides two services. To start the exe-
cution of the model, some messages must be put in the message buffer of the actors (i.e., initially
sent messages); this is specified in the main block (line 24). Sending a message is denoted by
“actor_name.service_name()” (line 3).

We illustrate our approach using a Simple Actor Modeling language, called SAM, which con-
tains the key features of the actor model. Below, we briefly introduce SAM, which is inspired by
the earlier actor models, e.g., by Agha et al. in [AMST97] and by Sirjani et al. in [SMSB04].

Definition 1 (An Actor Model) An actor in SAM is a member of type Actor = ID × P(Vars) ×
P(mtds), where P(·) denotes power set and:

− ID is the set of actor identifiers,
− Vars is the set of variable names, and
− mtds is the set of service names:

```plaintext
Actor ac1 {
  service msg1() {
    acl.msg1();
  }
  service msg2() {
    acl.msg2();
  }
  service msg3() {
    ac2.msg3();
  }
}
Actor ac2 {
  int sv = 1;
  service msg3() {
    acl.msg3();
  }
  service msg4() {
    if (sv == 1)
      sv = 4;
    else
      sv = 3;
  }
}
main {
  ac1.msg1();
}
```

Figure 1: An example of a simple actor model.
− $\textit{Mtds}$ is the set of method declarations.

In the above-given definition, the members of $\textit{Mtds}$ are tuples $(m, p, s) \in \textit{MName} \times \textit{Vars}^* \times \textit{Statement}^*$, where $m$ is the name of the message which must be served by this method, $p$ is the lists of message parameters, and $s \in \textit{Statement}^*$ is the sequence of statements compromising the method’s body. The set of statements in SAM is limited to a number of preliminary statements defined below.

**Definition 2 (SAM Statements)** The set of SAM statements is defined as $\textit{Statement} = \textit{Assignment} \cup \textit{Condition} \cup \textit{Send}$ where:

− $\textit{Assignment} = \textit{Vars} \times \textit{Expr}$ is the set of assignment statements. In Figure 1, we use $\textit{var} = \textit{expr}$ to denote the assignment statement $(\textit{var}, \textit{expr})$.

− $\textit{Condition} = \textit{BExpr} \times \textit{Statement}^* \times \textit{Statement}^*$ is the set of conditional statements. In Figure 1, we use $\textit{if} (\textit{bexpr}) \sigma \textit{else} \sigma'$ to denote the conditional statement $(\textit{bexpr}, \sigma, \sigma')$.

− $\textit{Send} = (\textit{ID} \cup \{\textit{self}\}) \times \textit{MName} \times \textit{Expr}^*$ is the set of send statements. In Figure 1, we use $a.m(e)$ to denote the send statement $(a, m, e)$.

In the aforementioned items, $\textit{Expr}$ denotes the set of integer expressions defined using usual arithmetic operators (with no side effects). $\textit{BExpr}$ denotes the set of Boolean expressions defined using usual relational and logical operators. We dispense with further details of the syntax in this definition.

Based on Definition 1 and Definition 2, a SAM model is specified by $\mathcal{P}(\textit{Actor}) \cup \textit{Send}^*$ where the $\textit{Send}^*$ term addresses the send statements of the main block (i.e. the initially sent messages). Note that since there may be more than one initial message for an actor, the send statements are ordered in a sequence not just a set of statements.

We define below the operational semantics of SAM in terms of a Labeled Transition System (LTS). In order to do this, the following assumptions and notations are required. We assume that the only possible communication mechanism among actors is asynchronous message passing. The type of messages is defined as $\textit{Msgs} = \textit{ID} \times \textit{MName} \times \textit{ID} \times (\textit{Vars} \rightarrow \textit{Vals})$, where for a message $(a_1, m, a_2, \text{arg}) \in \textit{Msgs}$, $a_1$ is the name of the sending actor, $a_2$ is the name of the receiving actor, $m$ is the name of the message, and $\text{arg}$ is a function for mapping argument names to their values. For the sake of simplicity and without loss of generality, we assume that the messages do not have arguments and are left out of the message signature in the remainder of this paper. The other assumption is that the received messages of an actor are stored in a FIFO mailbox. Hence, the mailbox of an actor is denoted by a sequence of messages, i.e., a member of $\textit{Msgs}^*$.

**Definition 3 (SAM Operational Semantics)** For a given actor model $\textit{AC}$, its labeled transition system $\textit{LTS}(\textit{AC})$, is defined as a tuple $(S, s_0, \textit{Act}, \rightarrow)$, where:

− $S$ is the global state of a SAM model defined as a function $\textit{s}: \textit{ID} \rightarrow (\textit{Vars} \rightarrow \textit{Vals}) \times \textit{Msgs}^*$, which maps an actors identifier to the local state of the actor, i.e., the values of its state variables and its mailbox content,
− $s_0 \in S$ is the initial state,
− $Act = Msgs$ is the set of action labels (sent messages).
− $\rightarrow \subseteq S \times Act \times S$ is the set of transitions, defined by the coarse-grained interleaving of actor message executions, by removing a message from their mailboxes, sending the messages in the body of the corresponding method and finally updating the global state (as the result of assignment statements). By coarse-grained interleaving, we mean that the sequence of messages in the body of a method are sent in an atomic sequence.

Using the operational semantics of actor models, Clinger’s event diagram of actor models can be created. Clinger’s event diagram comprise vertices (called dots) for each event, and edges (called arrows) that represent the activation relation of two events. Using dots and arrows, the runtime characteristics of an actor system is presented by the graph of activation relation of events. Clinger’s event diagram is typically drawn using parallel vertical swim-lanes for actors, where the dots are placed respecting their sequential execution order. Figure 2(a) presents the Clingers’ event diagram of the example actor model of Figure 1. As shown in the actor model, message $msg_1$ is the first executing message (as it is put in the queue of $ac_1$ in the main block) which results in sending $msg_2$ and $msg_3$; hence, there is one dot with label $msg_1$, which is connected by arrows to two other dots with labels $msg_2$ and $msg_3$.

Clinger’s event diagrams can be seen as detailed representations of CDG. Intuitively, a CDG represents the possible activation relations of events derived from a static analysis of the model. Hence, a CDG over-approximates the event activations in the Clinger’s event diagram. The activation relation of events in a CDG can be extracted from the source codes of actor models. For example, as shown in Figure 2(a), the static activation relation between $msg_1$ and two messages $msg_2$ and $msg_3$ can be extracted from the source code of Figure 1. In addition, the execution of $msg_3$ results in the activation of $msg_1$, hence, the loop back to the topmost state of the CDG in Figure 2(b).

**Definition 4** (Sent Messages) For a given message $msg = (a_1, m, a_2) \in Msgs$, the set of messages that can be possibly sent by $a_2$ to arbitrary actors as a result of serving $m$ (which is in turn sent by $a_1$) is denoted by $snt(msg)$.

The set $snt(msg)$ is statically determined by an evaluation of the source code of an actor model. For a given message $msg = (a_1, m, a_2)$ assume that $(m, p, s)$ is its corresponding method. The members of $snt(msg)$ is computed by the analysis of the statements of $s$, as depicted below, for a given SAM model $sam = acs \cup snt msgs$.

\[
snt(msg) = \{msg' \in Msgs | \begin{align*}
msg &= (a_1, m, a_2) \land msg' = (a_2, m', a_i) \land (a_2, vars, mtds) \in acs \rightarrow \\
\exists stmts \in \mathcal{P}(Statements) \cdot (m, arg, stmts) \in mtds \land (a_i, m', \emptyset) & \in stmts
\end{align*}
\}
\]

**Definition 5** (Call Dependency Graph (CDG)) Given an actor model $AC$ with $LTS(AC) = (S, s_0, Act, \rightarrow)$, its CDG is a finite labeled transition system $CDG(AC) = (V, v_0, Act, \rightarrow)$, where
State Distribution Policy

$V \subseteq \text{Act} \times \mathcal{P}(\text{Act})$ is the set of vertices (states), $v_0 \in V$ is the initial vertex (state), and $\rightarrow \subseteq V \times \text{Act} \times V$ is the set of edges (transitions). For two given states $u, v \in V$, there is $(u, \text{act}, v) \in \rightarrow$ if and only if $u = (\text{act}, \text{pm})$ and $\exists(a_i, m', a_i) \in \text{pm}$ such that $v = ((a_i, m', a_i), \text{snt}(a_i, m', a_i))$. This way, the initial state of CDG is defined as $v_0 = (\varepsilon, \text{pm})$ where $(\varepsilon, m, a_i) \in \text{pm}$ and there is a send statement in form of $a_i.m()$ in the main block of AC.

Next, we show that the abstract notion of the CDG reflects the cycles of the state space; more precisely, our goal is to show that each cycle in the LTS of an actor model can be projected into at least one cycle in the corresponding CDG.

**Definition 6** (Labels, Sub-Traces, Sub-Cycles, and Cycles) Given an actor model $AC$ a finite trace $tr$ of $LTS(AC) = (S, s_0, Act, \rightarrow)$ is any finite word $m_0, m_1, \ldots, m_n \in M^*$ such that there is a sequence $s_0, s_1, \ldots, s_n, s_{n+1}$ of vertices in $LTS(AC)$, where $s_0$ is the initial state and $(s_i, m_i, s_{i+1}) \in \rightarrow$ for $i = 0, 1, \ldots, n$.

The set of cycles of $LTS(AC)$, denoted by $\text{Cycles}(LTS(AC))$, is the set of all traces that start and end with the same message. A sub-trace of a cycle which starts and ends with the same message is called a sub-cycle. The set of all edge labels of a given trace $tr$ is denoted by $\text{Labels}(tr)$.

**Theorem 1** (Mapping $LTS(AC)$ Cycles into CDG Cycles) Each cycle in $LTS(AC)$ as the state space of the actor model $AC$, has a sub-cycle in $LTS(\text{CDG})$.

In order to prove the theorem, we first prove the following lemma, which establishes a link between individual messages appearing in the cycles of the state space and those appearing in the CDG. Here *appear* means that the message is the label of at least one of the transitions of the state space.

**Lemma 1** For each extended message $m$ appearing in $LTS(AC)$ of actor model $AC$, $m$ also appears in $CDG(AC)$.

**Proof.** Assume that there exist messages which appear in the state space but never appear in the CDG. Pick one such message $m$ that is reachable with the shortest trace from the initial state.
Assume that \( m \) is sent in the body of a service \( m' \). Due to the minimality assumption for \( m, m' \) should appear in the CDG and by the definition of CDG, \( m \) should appear in the CDG subsequent to the edge labeled \( m' \) (i.e., \( m' \) is the parent of \( m \)), contradicting our original assumption.

We also need the following definition.

**Definition 7** (Parent and ancestors) Assume that \( \text{LTS}(AC) \) and a trace \( m_k \rightarrow \cdots \rightarrow m_t \rightarrow m_j \rightarrow m_i \in \text{tr}(\text{LTS}(AC)) \) are given, \( m_j \) is called the parent of \( m_i \), and is denoted by \( P(m_i) \). In addition, all messages from \( m_k \) to \( m_t \) are called the ancestors of \( m_i \).

**Proof.** Consider a cycle \( c_{\text{LTS}} \in \text{Cycles}(\text{LTS}(AC)) \) and an arbitrary label \( \mathcal{L} \in \text{Label}(c_{\text{LTS}}) \); we claim that for each trace \( m \rightarrow \cdots \rightarrow m' \) in the traces of \( c_{\text{LTS}} \), there exists a sub-trace \( m \rightarrow \cdots \rightarrow m' \) in \( \text{CDG}(AC) \). Once we prove this claim, the theorem follows by taking \( m \rightarrow \cdots \rightarrow m = c_{\text{LTS}} \) as the antecedent of the claim (then, it follows from the claim that there should exist a sub-cycle of \( c_{\text{LTS}} \) in the CDG, which was to be shown).

To prove the claim, we use induction on the length of the trace \( m \rightarrow \cdots \rightarrow m' \). The base case follows from Lemma 1. Assume that the claim holds for all traces of length \( n \) or less and consider a trace \( m \rightarrow \cdots \rightarrow m' \) of length \( n + 1 \). Let \( M \) be the set of parents of \( m' \) in all cycles of CDG (by Lemma 1, \( m' \) should appear in at least one cycle of the CDG). There exists some \( m_i \in M \) such that \( m \rightarrow \cdots \rightarrow m' = m \rightarrow \cdots \rightarrow m_i \rightarrow \cdots \rightarrow m' \). The trace \( m \rightarrow \cdots \rightarrow m_i \) is of length \( n \) (or less) and hence the induction hypothesis applies and a sub-trace of it appears in \( \text{CDG}(AC) \). Since \( m_i \) is a parent of \( m' \) in the \( \text{CDG}(AC) \), an edge labeled \( m' \) follows after \( m_i \). Therefore \( m \rightarrow \cdots \rightarrow m_i \rightarrow m' \) is a trace of CDG, which was to be shown.

### 3 Using CDG in the Distributed Model Checking Algorithms

In this section, we show how CDG can be exploited to improve the efficiency of state distribution policy in distributed model checking algorithms. Besides the traditional model checking algorithms, distributed model checking is proposed to deal with huge state spaces [BBC05, BBS01, BC06, BC0S06, BCKP01]. In distributed model checking, the state space is partitioned into slices and slices are distributed among multiple nodes for exploration. Dividing the exploration of a state space among nodes increases the analysis efficiency, but the performance gain heavily depends on the communication required among the nodes. Therefore, decreasing the number of split transitions (transitions between two states of which their hosts are different) reduces the required communication and hence the model checking cost. To reduce the number of split transitions, different states distribution policies are proposed [BBS01, GHS01]. To this aim, these policies use the static analysis of the source codes of models. Here, we show that how using CDG of actor models results in a better distribution of states in the distributed model checking of actor models. In the following, we show that how the CDG-based policy is implemented for distributed BFS-based model checking algorithm.
3.1 BFS Model Checking

The BFS exploration algorithm, creates and explores the state space in a level-by-level fashion and examines the back edges of the state space graph for cycle detection (explained below). In the first step of the BFS algorithm, the Cartesian product of the initial state of the state space and the property is stored, is marked as visited, and its level is set to zero. Then, for each level the successors of the states of that level are generated by applying the successor function to both the state space and the property automaton and their level is set by increasing the current level by one; when there are no unexplored states in the next level, the algorithm terminates.

This algorithm can be implemented using two queues to manage states of each level. The first queue stores the current level states (CLQ) and the second one stores the successors of the CLQ states. The latter queue is called the next level queue (NLQ). In each iteration, the unexplored states of the CLQ are dequeued and their unvisited successors are generated. When all states of the CLQ are dequeued, the content of the NLQ is moved to the CLQ and the algorithm continues until the NLQ is empty, i.e., all successors of the states in the CLQ have been visited. There is no need to examine all visited states, because only back edges may create cycles. Figure 1 shows a pseudo code of this algorithm. The backward search algorithm done by function CYCLE-DETECTION the same as the algorithm given in [BC06].

Algorithm 1: BFS_MODEL_CHECKING(initState) traverses a given state space level by level.

Input: The initial state initState
Output: The state space of the system

1. \( CLQ \leftarrow \{\text{initState}\} \)
2. \( NLQ \leftarrow \emptyset \)
3. \( Visited \leftarrow \emptyset \)
4. while \( CLQ \neq \emptyset \) do
5.   foreach state \( s \in \text{CLQ} \) do
6.     foreach state \( s' \in \text{PREDECESSORS}(s) \) do
7.       if \( s' \notin \text{Visited} \) then
8.         visited \( \leftarrow \text{Visited} \cup \{s'\} \)
9.         \( NLQ \leftarrow NLQ \cup \{s'\} \)
10.     else
11.        CYCLE-DETECTION(s')
12.   \( CLQ \leftarrow NLQ \)
13.   \( NLQ \leftarrow \emptyset \)

3.2 Distributed BFS Model Checking Algorithm

A major difference between the centralized- and the distributed BFS model checking (BFS-MC) algorithm is in storing the next level states. In the centralized BFS-MC, all newly generated system states are stored in the NLQ but in the distributed BFS-MC, some of them should be sent
to other nodes of the cluster. In other words, each state has a host node. The host of a state is the node that is responsible for storing the state and generating its successors. Line 8 of Algorithm 2 shows host assignment based on the random distribution. After finding the host, if the newly generated state host is the same as its parent’s and it has not yet been visited, then the state is stored in NLQ. In contrast, if the newly generated state's host is another node, the state is sent to it. Then, the host node receives the new state and checks if the state is visited before. Therefore, checking whether a state is visited or not can be done locally.

The other difference between the centralized- and the distributed BFS-MC is in the cluster nodes synchronization phase at the end of each iteration (line 17). In the synchronization phase, nodes that finish processing their CLQ wait for other nodes to finish their work. Hence, after synchronization, all nodes have processed their CLQ states and are ready to continue the search for the next level. If none of the nodes have any new state to explore, the value of allFinished is set to true in line 17 to terminate the model checking.

Algorithm 2: DISTRIBUTED_BFS_MODEL_CHECKING(initState, id) traverses a given state space level by level.

Input: The initial state initState (which is null if this node is not the host of the initial state) and the node's id

Output: The state space of the system

1. \( CLQ \leftarrow \{ \text{initState} \} \)
2. \( NLQ \leftarrow \emptyset \)
3. \( Visited \leftarrow \emptyset \)
4. allFinished \( \leftarrow \) false
5. while \( \neg \) allFinished do
6.     foreach state \( s \in CLQ \) do
7.         foreach state \( s' \in \text{PREDECESSORS}(s) \) do
8.             hostId \( \leftarrow \text{HASH}\_\text{VALUE}(s') \)
9.             if id = hostId then
10.                if \( s' \notin Visited \) then
11.                   Visited \( \leftarrow \text{Visited} \cup \{ s' \} \)
12.                   NLQ \( \leftarrow \text{NLQ} \cup \{ s' \} \)
13.                else
14.                   CYCLE\_DETECTION(s')
15.             else
16.                 send(s', hostId)
17.             allFinished \( \leftarrow \text{SYNCHRONIZE\_ALL()} \)
18.         CLQ \( \leftarrow \text{NLQ} \)
19.         NLQ \( \leftarrow \emptyset \)
3.3 States Distribution Policy based on CDG

In the new state distribution policy, we find the set of active cycles for each state. The active cycles of each state are found in the CDG of the model and are based on the messages which are executed before reaching this new state. Without loss of generality, we base the definition of our distribution policy on the simple cycles in a CDG (i.e., cycles with no repetition of vertices). When exploring the transitions of a state, we store the states that belong to a cycle of the CDG on the same cluster node, (i.e., states with the same active cycles).

**Definition 8** (Active cycles of a state) Consider an actor model $A$ and its CDG $CDG(A) = (V, v_0, Act, \rightarrow)$. For a given state $v \in V$, the set of active cycles of $v$ is the subset of $Cycles(CDG(A))$ containing all cycles in which the label $m_j$ appear, where $m_j$ is a label of one of the outgoing edges of state $v$.

The implementation of the new distribution policy is given in Algorithm 3. As shown in the input section, the CDG of the model is generated before model checking and it is given as an input to the algorithm. For the sake of simplicity, only differences between Algorithms 2 and the new algorithm are shown in Algorithm 3 (the common parts are shown by $\cdots$). Namely, line 8 of Algorithm 2 is replaced with the CDG-based distribution policy of lines 5 to 8.

```
Algorithm 3: DISTRIBUTED_BFS_MODEL_CHECKING(initState, id, CDG) traverses a given state space level by level.

Input: The initial state $initState$ (which is null if this node is not the host of the initial state), the node's id, and CDG as the call dependency graph of the model

Output: The state space of the system

1  ...  
2  while ¬allFinished do  
3      foreach state $s \in CLQ$ do  
4          foreach state $s' \in PREDECESSORS(s)$ do  
5              activeCycles ← $\emptyset$  
6              foreach message $msg \in ENABLED_MESSAGE(s')$ do  
7                  activeCycles ← activeCycles $\cup$ CDG_CYCLES(CDG, msg)  
8              hostId ← CHOOSE_CYCLE(activeCycles)  
9      ...  
10 ...  
```

4 Experimental Results

We implemented CDG-based distribution policy for the BFS-based distributed model checking engine of Rebeca, an actor-based language with a Java like syntax (A brief description about Rebeca and how the CDG of a Rebeca model is obtained is described in Appendix A). We studied
the impacts of using CDG in the state space generation and the analysis against reachability properties, using the current implementation of the Rebeca distributed model checking toolset. The test platform has been Ubuntu 9.10 on a cluster of 2.2GHz Pentium 4 Core2 Duo with 2GB of RAM storage for each cluster node. We chose the size of each cluster based on the number of simple cycles in the CDG of each case study.

Three different case studies are used to compare the execution time and the memory consumption among centralized model checking, distributed model checking with random distribution policy, and distributed model checking with distribution policy based on CDG. The examples are the asynchronous resource manager (from Figure 3), dining philosophers and train controller.

In the dining philosophers model, there are a number of philosophers sitting at a round table. Between each adjacent pair of philosophers, there is a chopstick. To model such a behavior, each philosopher can be in one of the following states: thinking, hungry, or eating. A philosopher thinks for a while, and then stops thinking and becomes hungry. When the philosopher becomes hungry, she cannot eat until he owns both of the chopsticks to her left and right. When the philosopher is done eating she puts down the chopsticks and begins thinking again.

In the train controller model there are a number of trains on each side of the bridge. Trains arrive non-deterministically and the controller has to manage them in such a way that only one train passes the bridge at a time, because there is one railway on the bridge. Each train announces its arrival to the controller and the controller lets the train enter the bridge, if there is no other train on the bridge. If the bridge is full then the arrived train is put in a queue. The waiting trains will be served respectively. Each train should faithfully declare its departure to the controller. The Rebeca code of each case study can be found at the Rebeca homepage [fml].

Each example represents different pattern of communication and synchronization: dining philosophers example shows a ring topology, train controller and asynchronous resource manager show a star topology. In the dining philosophers example, each actor sends requests and responses to its left and right neighbors. In the train controller, the bridge controller behaves like a binary semaphore, whereas in the resource manager, the central node behaves like a counting semaphore.

Asynchronous resource manager is model checked for deadlock freedom with 4 to 7 clients (5 to 8 rebecs). The dining philosophers example is model checked for deadlock freedom with 2 to 5 philosophers (4 to 10 rebecs). The train controller model is model checked for deadlock freedom with 2 to 8 trains (3 to 9 rebecs). Tables 1 and 2 show the results.

In the CDG-based distribution policy in comparison to the random distribution, there is the overhead of cycle membership check and instead we have fewer split transitions and less communication among cluster nodes for cycle detection. Our results show that time-wise the gain exceeds the overhead.

As shown in Table 1 in the large enough cases, the number of split edges in the CDG-based distribution policy is 50% to 70% of the random distribution policy. In addition, memory consumption is reduced, because storing the split transitions requires storing endpoints host ids of the edges. This improvement is about 10% for the asynchronous resource manager and 5% for the train controller.

Table 2 shows the gain in the execution time that is about 8% for the asynchronous resource manager and 13% for the train controller in their largest versions. For the dining philosophers model, although the split cycles for the CDG-based policy are 52% of the random-based policy,
Table 1: Split edges in the random and the CDG-based distribution policies.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Size</th>
<th>#Transitions Random</th>
<th>#Split Transitions CDG</th>
<th>improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynch. Resource Manager</td>
<td>2 clients</td>
<td>94</td>
<td>39</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>3 clients</td>
<td>818</td>
<td>540</td>
<td>432</td>
</tr>
<tr>
<td></td>
<td>4 clients</td>
<td>7,76K</td>
<td>5,83K</td>
<td>4516</td>
</tr>
<tr>
<td></td>
<td>5 clients</td>
<td>83,19K</td>
<td>66,52K</td>
<td>50,46K</td>
</tr>
<tr>
<td></td>
<td>6 clients</td>
<td>1,02M</td>
<td>850,74K</td>
<td>635,14K</td>
</tr>
<tr>
<td></td>
<td>7 clients</td>
<td>14,34M</td>
<td>12,30M</td>
<td>9,01M</td>
</tr>
<tr>
<td>Dining Philosophers</td>
<td>2 phils</td>
<td>408</td>
<td>196</td>
<td>107</td>
</tr>
<tr>
<td></td>
<td>3 phils</td>
<td>10,30K</td>
<td>6,97K</td>
<td>4,81K</td>
</tr>
<tr>
<td></td>
<td>4 phils</td>
<td>206,00K</td>
<td>154,76K</td>
<td>75,86K</td>
</tr>
<tr>
<td></td>
<td>5 phils</td>
<td>3,78M</td>
<td>3,02M</td>
<td>1,60M</td>
</tr>
<tr>
<td>Train Controller</td>
<td>2 trains</td>
<td>86</td>
<td>46</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>3 trains</td>
<td>620</td>
<td>433</td>
<td>296</td>
</tr>
<tr>
<td></td>
<td>4 trains</td>
<td>4,46K</td>
<td>3,35</td>
<td>2,29K</td>
</tr>
<tr>
<td></td>
<td>5 trains</td>
<td>33,30K</td>
<td>26,66K</td>
<td>18,17K</td>
</tr>
<tr>
<td></td>
<td>6 trains</td>
<td>265,89K</td>
<td>221,61K</td>
<td>148,32K</td>
</tr>
<tr>
<td></td>
<td>7 trains</td>
<td>2,30M</td>
<td>1,97M</td>
<td>1,30M</td>
</tr>
<tr>
<td></td>
<td>8 trains</td>
<td>21,83M</td>
<td>19,11M</td>
<td>12,40M</td>
</tr>
</tbody>
</table>

the execution time remains the same. This is due to the many back edges and the big cycles in the state space. In this case the CDG-based policy reduces the split edges but the effect of the reduction is negligible comparing to the time spent for backward search in detecting accepting cycles.

We also measured load balancing of cluster nodes for the two distributed policies. Random distribution results in balanced load for each node. In our experimental results, we could see that in random distribution, the deviation from the best distribution starts from 9% in small models and reduces to less than 1% for the larger one. In the CDG distribution this deviation starts from 13% and reduces to 1.28% for larger models.

In general the experimental results show that our technique outperforms random distribution when the size of the model is large enough. The gain increases as the size of the model grows. Also in our approach, the load balancing of cluster nodes converges to the optimum point in larger examples.

5 Discussion, Conclusion and Future Work

In this paper we introduced the Call Dependency Graph (CDG) for the actor-based modeling language Rebeca. The CDG is generated by a static analysis of the model and is an abstract graph capturing the causality of message passing among actors. We designated and proved a relation between the cycles in the CDG and the cycles in the state space. We devised a distribution policy for the distributed model checker of Rebeca based on the CDG. The new distribution policy increases the efficiency of distributed model checking by increasing the locality of the accepting cycles. Our new policy is implemented as an extension of breadth first search distributed model checking. Experimental evidence supports that this new policy improves cycle locality, and decreases model checking time and memory in practice.

As future work, we plan to improve our algorithm by duplicating states to avoid split cycle
Table 2: Time consumption for centralized and distributed model checking with the random and the CDG-based distribution policies.

creation such that all cycles can be detected locally. This comes at the cost of more memory consumption, and we need to define a set of criteria to balance between the increase in the size of state space, due to duplicating states, and the decrease in the verification time, due to localizing cycles. Moreover, we look for property classes for which our distribution policy guarantees localized cycles. Finally, we would like to investigate the effect of incorporating CDG into other analysis and reduction techniques such as slicing.

Acknowledgments. We thank the anonymous reviewers of AVoCS 2015 for their useful comments. The work of M.R. Mousavi has been partially supported by the Swedish Research Council (Vetenskapsrådet) with award number 621-2014-5057 (Effective Model-Based Testing of Parallel Systems) and the Swedish Knowledge Foundation (Stiftelsen för Kunskaps- och Kompetensutveckling) in the context of the AUTO-CAAS project.

Bibliography


A Rebeca

Rebeca is an incarnation of the actor model. It comes equipped with an on-the-fly explicit-state LTL model-checking engine called Modere [JMS06]. Rebeca has a Java-like syntax and an operational semantics [Sir06, SMSB04]. Each Rebeca model consists of a number of reactive classes, each describing the type of a number of actors (called rebecs in Rebeca). We describe Rebeca language constructs using a simple resource manager model (see Figure 3).

```plaintext
reactiveclass CentralNode(3) {
    knownrebecs {Client c1, c2, c3;}
    statevars {int max;}
    msgsrv initial() {
        max = 5;
    }
    msgsrv register(int cnt) {
        max = max - cnt;
        if(sender == c1)
            c1.ack();
        else if(sender == c2)
            c2.ack();
        else if(sender == c3)
            c3.ack();
    }
    msgsrv return(int cnt) {
        max = max + cnt;
        if(sender == c1)
            c1.start();
        else if(sender == c2)
            c2.start();
        else if(sender == c3)
            c3.start();
    }
}
reactiveclass Client(2) {
    knownrebecs {CentralNode cn;}
    statevars {
        byte id;
        boolean asked;
    }
    msgsrv initial(byte id2) {
        id = id2;
        self.start();
    }
    msgsrv start() {
        asked = true;
        cn.register(id);
    }
    msgsrv ack() {
        asked = false;
        cn.return(id);
    }
}
main {
    CentralNode cn(c1, c2, c3):();
    Client c1(cn):();
    Client c2(cn):();
    Client c3(cn):();
}
```

Figure 3: Rebeca model for an asynchronous resource manager.

In this model, there are two reactive classes CentralNode and Client. Each reactive class declares a set of state variables, whose valuations define the local state of the actors of that reactive class. Following the actor model, communication takes place by actors sending asynchronous messages to each other. Each actor has a set of known rebecs to which it can send messages. For example, an actor of type CentralNode knows all the actors of type Client (line 2), to which it can send messages (e.g., lines 10, 12, and 14). Reactive classes declare the messages to which they can respond. The way an actor responds to a message is specified in its corresponding message server. An actor can change the value of its state variables through an assignment statement (line 34), make decisions through a conditional statement (line 18), and communicate with other rebecs by sending a message (line 19). Since communication is asynchronous, each actor has a message queue, from which it takes the next incoming message.
reactiveclass CentralNode(3) {
   knownrebecs{ Client c1, c2, c3; }
   msgsrv initial() { }
   msgsrv register() {
      if(sender == c1)
         c1.ack();
      else if(sender == c2)
         c2.ack();
      else if(sender == c3)
         c3.ack();
   }
   msgsrv return() {
      if(sender == c1)
         c1.start();
      else if(sender == c2)
         c2.start();
      else if(sender == c3)
         c3.start();
   }
}

reactiveclass Client(2) {
   knownrebecs { CentralNode cn; }
   msgsrv initial() {
      self.start();
   }
   msgsrv start() {
      cn.register();
   }
   msgsrv ack() {
      cn.return();
   }

main {
   CentralNode cn(c1, c2, c3):();
   Client c1(cn):();
   Client c2(cn):();
   Client c3(cn):();
}

Figure 4: Rebeca model for an asynchronous resource manager.

An actor takes the first message from its queue, executes the corresponding message server atomically, and then takes the next message (or waits for the next message to arrive).

For our resource manager, starvation-avoidance and resource-availability are two properties that are to be satisfied. Starvation-avoidance means that if a client asks for a resource, it will eventually receive it. Resource-availability property guarantees the existence of enough resources using the value of max state variable. The LTL formulas of these properties are given below.

- Starvation-avoidance: $G((c1.asked \rightarrow F(\neg c1.asked)) \land (c2.asked \rightarrow F(\neg c2.asked)) \land (c3.asked \rightarrow F(\neg c3.asked)))$

- Resource-availability: $G(cn.max > 0)$

A.1 Obtaining CDG of Rebeca Models

To obtain the CDG of a given Rebeca model, we first abstract away the original Rebeca model into a skeleton. This skeleton reflects the message communication structure of each reactive class together with the part of the control structure that is influenced by the signature of the message being processed. Then, we generate the CDG from the skeleton by applying Definition 5. The skeleton of the example of Figure 3 is depicted in Figure 4. The resulting CDG from the skeleton is depicted in Figure 5(a). The labels in the figure are the edge labels and the vertices are not labeled in order not to clutter the figure. A sample of vertex label, for the end points of the edge $<C_1, start, C_1>$ are shown in Figure 5(b).
Figure 5: An example CDG which is extracted from the Rebeca model of Figure 3.
Permissive strategies in timed automata and games†

Patricia Bouyer¹, Erwin Fang² and Nicolas Markey¹

¹LSV, CNRS & ENS Cachan, France
²Institute of Information Security, ETH Zurich, Switzerland

Abstract: Timed automata are a convenient framework for modelling and reasoning about real-time systems. While these models are now well-understood, they do not offer a convenient way of taking timing imprecisions into account. Several solutions (e.g. parametric guard enlargement) have been proposed over the last ten years to take such imprecisions into account. In this paper, we propose a novel approach for handling robust reachability, based on permissive strategies. While classical strategies propose to play an action at an exact point in time, permissive strategies consider intervals of possible dates when to play the selected action. In other words, the controller specifies an interval of time delays for actions to be executed in a more flexible way. With such a permissive strategy, we associate a penalty, which is the inverse of the length of the proposed interval, and accumulates along the run. We show that in that setting, optimal strategies can be computed in polynomial time for one-clock timed automata.

Keywords: timed automata, timed games, permissive strategies, multi-move, timed penalty games, timed robustness

1 Introduction

Validation of real-time embedded systems has been an active research area for many years now. Model checking real-time systems was proposed in [ACD90] as a possible approach to verify properties of such system models. Another approach to construct timed systems correctly is by synthesizing executions or winning strategies of a controller given a specification or winning condition.

There is an increasing interest in synthesis based on games within the computer science and control theory communities, since games are a suitable paradigm for modeling reactive systems that maintain a continuous interaction with the environments [FLM14]. The synthesis problem is somehow dual to verification: while in verification, one asks whether some property $\varphi$ is satisfied in a model $\mathcal{M}$, i.e., $\mathcal{M} \models \varphi$, the synthesis problem considers a property and a plant or game area as input and asks whether a strategy can be computed that controls the system in order to satisfy the property. In a game-theoretic context this corresponds to the existence of a strategy for a player. In this work, we consider timed automata, as defined by Alur and Dill [AD94], and the reachability winning objective. The main objective is to synthesize winning strategies that are robust w.r.t. to timing perturbations.

† Partly supported by FET project Cassting (FP7-601148) and ERC project EQualIS (FP7-308087).
A timed automaton is a finite automaton extended with a finite set of clocks. It is a convenient paradigm to model systems with real-time constraints and to reason about these algorithmically. Efficient model-checking tools such as HyTech [HHW97], Kronos [BDM+98] and Uppaal [LPY97] are available. Still, a drawback of timed-automata is that their semantics are idealistic: these models are assumed to have arbitrary precision for delays, and immediate transitions. This leads, among other unrealistic behaviors, to the paradox that infinitely many actions can be executed within a finite amount of time. Furthermore, timed automata also assume that time can be measured exactly. This means that a system can enforce a controller to choose punctual delays. However, these are not realistic assumptions since computers are digital and values can only be stored in variables of finite size.

Therefore, investigating on robustness issues on timed automata is crucial, and it has been an active area of research over the last ten years. The quest is to include certain meaningful notions of robustness or tolerance with respect to timing perturbations into the timed-automata model. A prominent approach is the so-called guard enlargement, i.e., the transformation of each guard of the form \( a \leq x \leq b \) into \( a - \delta \leq x \leq b + \delta \), for some parameter \( \delta > 0 \). Safety of the resulting enlarged automaton entails robust safety of the original automaton, i.e., safety even in the presence of timing perturbations. Several decidability and complexity results have been obtained for this notion of robustness. Efficient algorithms are being implemented in the tool Shrinktech [San15]. Robust reachability has also been proved to be decidable [BMS12]: there, the aim is to synthesize a strategy that will be able to counteract the (parametric) timing perturbations and reach a target location. We discuss these and other related works in more detail in Section 3.

**Our contribution.** In this paper, we also focus on robust reachability, but using permissive strategies. As opposed to strategies classically used in most kind of games, permissive strategies propose several possible moves to be played from a given configuration. In the timed setting, this is implemented by having strategies proposing an interval of possible dates at which the player allows her action to be played or executed. Each interval is assigned a penalty inversely proportional to the size of the interval. These penalties are summed up along the path until the target is reached.

In this setting, our aim is to compute the most permissive strategy for reaching a target location. We prove that the problem can be solved in polynomial time for one-clock timed automata (and games), and that an almost-optimal memoryless permissive strategy exists.

## 2 Permissive strategies and penalty games

**Timed automata.** Let \( \mathcal{C} \) be a finite set of variables (named clocks in the sequel). A clock valuation over \( \mathcal{C} \) is a mapping \( \kappa : \mathcal{C} \to \mathbb{R}_{\geq 0} \), assigning to each clock a non-negative real value. For \( t \in \mathbb{R}_{\geq 0} \), we write \( \kappa + t \) for the clock valuation that results from \( \kappa \) by adding \( t \) time units, i.e., \( (\kappa + t)(c) = \kappa(c) + t \) for all \( c \in \mathcal{C} \). For a subset \( U \subseteq \mathcal{C} \), let \( \kappa[\{U := 0\}] \) be the clock valuation that results from \( \kappa \) by resetting all clocks in \( U \), i.e., \( \kappa[\{U := 0\}](c) = \kappa(c) \) for all \( c \in \mathcal{C} \setminus U \), and \( \kappa[\{U := 0\}](c) = 0 \) for all \( c \in U \). The set \( \text{Constr}(\mathcal{C}) \) of all convex clock constraints over \( \mathcal{C} \) is defined as the set of conjunctions of atomic constraints of the form “\( c \sim n \)” for \( c \in \mathcal{C} \), \( n \in \mathbb{N} \), and \( \sim \in \{<,\leq,=,\geq,>\} \). We write \( \mathcal{I} \) for the set of all intervals of \( \mathbb{R}_{\geq 0} \).
Definition 1 A timed automaton is a tuple \( A = \langle Q, C, \text{Act}, E, \text{Inv} \rangle \), where \( Q \) is a finite set of locations; \( C \) is a finite set of clocks; \( \text{Act} \) is a finite set of actions; \( E \subseteq Q \times \text{Act} \times \text{Constr}(C) \times 2^C \times Q \) is a transition relation; \( \text{Inv} : Q \to \text{Constr}(C) \) is a mapping that assigns an invariant to each location. The transition relation is required to be deterministic, which in our setting means that for any two transitions \((q, a, g_1, r_1, q_1)\) and \((q, a, g_2, r_2, q_2)\) in \( E \) with \( q_1 \neq q_2 \), the constraint \( g_1 \land g_2 \) is unsatisfiable.

A configuration of \( A \) is a pair \( s = (q, \kappa) \in Q \times (\mathbb{R}_{\geq 0})^C \) such that \( \kappa \models \text{Inv}(q) \). A move is a pair \((d, a) \in \mathbb{R}_{\geq 0} \times \text{Act} \) if the following conditions hold: (1.) the invariant \( \text{Inv}(q) \) holds for all \( \kappa + d' \) with \( d' \in [0, d] \), and (2.) there is a (unique) transition \( e = (q, a, g, r, q') \in E \) such that \( \kappa + d \models g \) and \( \kappa' = (\kappa + d)[r := 0] \models \text{Inv}(q') \).

When those conditions are met, we write \((q, \kappa) \xrightarrow{d,a} (q', \kappa')\), which gives rise to an infinite-state transition system. Notice that we can assume that the second condition always holds, even if it means adding an extra sink location \( q_{\text{sink}} \). We make this assumption in the sequel, as it simplifies the presentation.

A run from the initial configuration \( s_0 \) is an infinite sequence \( \rho \) of pairs \(((d_i, a_i), s_i)_{i \geq 1} \) with \( s_i \in Q \times (\mathbb{R}_{\geq 0})^C \) and \( s_{i-1} \xrightarrow{d_i, a_i} s_i \) for all \( i \geq 1 \). For a finite prefix of a run (which we abusively call finite run in the sequel) \( \pi = (\pi_j)_{1 \leq j \leq n} \), we write \( \text{last}(\pi) \) for the configuration \( s_n \) of the last element \( \pi_n \) of \( \pi \). We let \( |\pi| = n \). For a run \( \pi \) and an integer \( 1 \leq j \leq n \), we write \( \pi_{\leq j} \) for the finite prefix of \( \pi \) up to the \( j \)-th transition.

Multi-moves and permissive strategies. In this paper, we consider a modified notion of moves, which we call multi-moves. In our timed setting, a multi-move is a pair \((I, a)\) where \( I \) is a non-empty interval of \( \mathbb{R}_{\geq 0} \) and \( a \) is an action. Intuitively, a multi-move \((I, a)\) corresponds to the set of all moves \((t, a)\) for all \( t \in I \). Non-determinism is then solved by an opponent player, and the semantics of timed automata in this setting is defined as a game, as we now explain.

A multi-move \((I, a)\) is enabled in configuration \((q, \kappa)\) whenever for all \( d \in I \), the move \((d, a)\) is enabled in \((q, \kappa)\). Any multi-move \((I, a)\) enabled in \((q, \kappa)\) gives rise to a transition \((q, \kappa) \xrightarrow{I,a} (q, \kappa, I, a)\); the latter configuration is an intermediary configuration, from which the opponent can select some \( d \in I \) and activate the actual transition \((q, \kappa, I, a) \xrightarrow{d, a} (q', \kappa')\) where \((q', \kappa')\) is the unique configuration such that \((q, \kappa) \xrightarrow{d, a} (q', \kappa')\). In this setting, a play from \( s_0 \) is an infinite sequence \( \pi \) of triples \(((I_i, a_i), d_i, s_i)_{i \geq 1} \) such that \( s_{i-1} \xrightarrow{I_i, a_i} (s_{i-1}, I_i, a_i) \xrightarrow{d_i} s_i \) for all \( i \geq 1 \). A finite play is a finite prefix of a play, in the same way as finite runs. In particular, the last configuration \( \text{last}(\pi) \) is \( s_{|\pi|} \).

A permissive strategy is a mapping \( \sigma \) that associates with each finite play \( \pi \) from \( s_0 \) a multi-move \( \sigma(\pi) = (I, a) \) enabled in \( \text{last}(\pi) \). A finite play \( \pi = (\pi_j)_{1 \leq j \leq n} \), with \( \pi_j = ((I_j, a_j), d_j, s_j) \) for all \( 1 \leq j \leq n \), is compatible with a permissive strategy \( \sigma \) if \( \sigma(\pi_{\leq j}) = (I_j, a_j) \) for all \( 1 \leq j \leq n \). An (infinite) play \( \pi \) from \( s_0 \) is compatible with \( \sigma \) whenever all its finite prefixes are compatible with \( \sigma \). Such a play is then called an outcome of \( \sigma \) from \( s_0 \). In this paper, we consider reachability objectives: given a target location \( g \), a permissive strategy \( \sigma \) is said winning from \( s_0 \) whenever all its outcomes eventually visit location \( g \).
Penalty of a permissive strategy. In the setting of timed robustness, our aim is to compute highly permissive strategies. A naive approach for comparing strategies is to compare the sizes of the intervals proposed by the strategies. This order would obviously not be total, and would not give rise to a notion of maximally permissive strategies. We prefer a semantic criterion, based on the quantitative measure of permissiveness.

We define the penalty of a multi-move \((I, \alpha)\) as follows:

\[
\text{penalty}(I, \alpha) = \begin{cases} \\
\frac{1}{|I|} & \text{if } I \text{ is not punctual, i.e., if } |I| > 0, \\
+\infty & \text{otherwise.}
\end{cases}
\]

With this definition, the larger the interval, the smaller the penalty. Of course, various other penalty functions could be considered. We elaborate on this point in Section 4.4.

In order to define the penalty of a permissive strategy, we extend the notion of penalty along finite plays: given a permissive strategy \(\sigma\) and a finite play \(\pi\), we define

\[
\text{penalty}_\pi(\sigma) = \sum_{j=0}^{\lfloor |\pi| - 1 \rfloor} \text{penalty}(\sigma(\pi \leq j)).
\]

(Notice that this definition does not need \(\pi\) to be an outcome of \(\sigma\), even though it will be the case in the sequel). Again, other ways of accumulating penalties along a play could be considered.

Finally, we define the penalty of a permissive strategy. In order to have only finite paths (and finite penalty), we only consider winning permissive strategies, and consider the prefixes of the plays until their first visit to the target location. For a winning permissive strategy \(\sigma\) from initial configuration \(s_0\), we define

\[
\text{penalty}_{s_0, g}(\sigma) = \sup_{\pi \in \text{Out}_{s_0}(s_0, g, \sigma)} \text{penalty}_\pi(\sigma).
\]

where \(\text{Out}_{s_0}(s_0, g, \sigma)\) is the set of finite outcomes of \(\sigma\) from \(s_0\) and ending at their first visit to \(g\). The penalty of non-winning strategies is \(+\infty\). The problem we tackle in this paper is the following:

**Definition 2** (Computing the most permissive strategy - the decision problem) Given a timed automaton \(\mathcal{A}\), a configuration \(s_0\) and a target location \(g\), and a threshold \(p \in \mathbb{Q}\), the most-permissive strategy problem asks whether there exists a winning permissive strategy \(\sigma\) in \(\mathcal{A}\) such that \(\text{penalty}_{s_0, g}(\sigma) \leq p\).

**Example 3** Figure 1 displays an example of a timed automaton with target location \(\ell_f\). Obviously, the target location \(\ell_f\) is reachable, and can even be reached with a penalty of 4 (starting from \((\ell_0, x \mapsto 0)\)); a corresponding strategy is to propose delay interval \([0, 1/2]\) in \((\ell_0, x \mapsto 0)\), and then \([0, (1 - \kappa(x))/2]\) from \((\ell_2, \kappa)\). One easily sees that the penalty of this strategy is 4 (which is reached when Player 2 selects delay 1/2 in \(\ell_0\)). As we explain after Theorem 10, better strategies exist for this example.

### 3 Related work

**Robustness.** Several previous works have proposed notions on defining robustness in timed automata. One of the first attempts was presented in [GHJ97], where a topological definition was
introduced. The idea of this “tube semantics” is to accept a run if, and only if, all “neighbouring runs” are also accepted. The aim was to find a procedure for deciding language inclusion in this setting. However, this was shown to be undecidable later in [HR00].

Guard enlargement was then proposed by Puri [Pur98]. This semantics aims at over-approximating the behaviors of implementations of timed automata over (simplified) hardware [DDR04]. Notice that makes model-checking algorithms consider more runs, contrary to the tube semantics. Hence this is mainly aimed at reasoning about robust safety which is proven to be decidable in [Pur98, DDMR04]. Guard shrinking was then introduced in [SBM11]: the aim of shrinking is to counteract the enlargement that the model will be subject to when being implemented. Hence, the shrunk model is a good candidate to implement, provided that it preserves roughly the same behaviors as the original automaton. This was proven decidable in [SBM11]. Guard enlargement was also considered for reachability objectives [BMS12]. In this case, the aim is to reach a target location despite possible timing perturbations. A natural approach is to see this as a game, where one player tries to reach the target while the opponent introduces timing perturbations. This approach is also decidable. Based on this approach, a stochastic approach to the robustness of timed systems was proposed in [ORS14].

Our approach here shares similarities with that of [BMS12]: in both approaches, the aim is to end up with a strategy to reach a target without choosing the exact date at which transitions are taken. There are several important differences however: in particular, in our approach we add up the penalties along the runs, so that we favor shorter runs. We believe that having shorter strategies is a sensible choice in a setting where the imprecisions may accumulate when the run becomes longer. Also, guard enlargement considers the same enlargement for all the transitions, while we allow different lengths for the intervals.

**Permissive strategies.** While permissive strategies are a key notion in supervisory control [RW89, ELTV14], they have not been widely considered in reactive synthesis, with the exception of [BJW02, BKK11]. In those cases however, permissiveness is measured in terms of the set of behaviours allowed by the strategy. Hence maximally-permissive strategies need not exist, depending on the type of winning objectives. Our quantitative measure of permissiveness
4 Computing optimal permissive strategies

In this section, we study some properties of the most-permissive-strategy problem, and prove that it is decidable for one-clock timed automata: we define a sequence of functions that we prove converges to the least penalty that can be achieved for reaching \( g \). We then show that for one-clock timed automata, the computation is effective and that it terminates in a finite number of steps.

4.1 Least penalty for winning in \( i \) steps

Let \( \mathcal{A} \) be a timed automaton, and \( g \) be the goal location. W.l.o.g., we assume that all the configurations of \( \mathcal{A} \), except configurations involving \( q_{\text{sink}} \), are winning for the objective of reaching location \( g \). Given \( a, b \in \mathbb{R}_{\geq 0} \), we write \( (a, b) \), with \( (\in \{[, ]\} \text{ and } ) \in \{[, ]\} \), for the interval between \( a \) and \( b \) which is either (half-)open or (half-)closed. For a clock valuation \( \kappa \) and a convex clock constraint \( \varphi \), we define

\[
\mathcal{D}(\kappa, \varphi) = \{ I \in \mathcal{I} \setminus \{\emptyset\} \mid \forall t \in I. \kappa + t \models \varphi \}.
\]

Then \( \mathcal{D}(\kappa, \text{inv}(q)) \) contains the set of intervals of delays that can be elapsed from \( (q, \kappa) \). We now define a sequence of functions \( (\mathcal{P}_i)_{i \in \mathbb{N}} \) inductively as follows: for location \( g \), we let \( \mathcal{P}_i(g, \kappa) = 0 \) for all \( i \in \mathbb{N} \) and all valuation \( \kappa \). For any location \( q \neq g \), and for any valuation \( \kappa \), we let

\[
\mathcal{P}_0(q, \kappa) = +\infty
\]

\[
\mathcal{P}_{i+1}(q, \kappa) = \min_{a \in \text{Act}} \inf_{I \in \mathcal{D}(\kappa, \text{inv}(q))} \left( \text{penalty}(I, a) + \sup_{d \in I} \mathcal{P}_i(\text{succ}(q, \kappa, d, a)) \right)
\]

where \( \text{succ}(q, \kappa, d, a) \) is the configuration \( (q', \kappa') \) such that \( (q, \kappa) \xrightarrow{d,a} (q', \kappa') \). We take the usual convention that the infimum over the empty set is \( +\infty \).

Then, we let \( \mathcal{P}(q, \kappa) = \lim_{i \to +\infty} \mathcal{P}_i(q, \kappa) \). Notice that this limit exists, as a consequence of the following lemma:

**Lemma 4** For any \( n \in \mathbb{N} \), for any configuration \( (q, \kappa) \), the mapping \( t \mapsto \mathcal{P}_n(q, \kappa + t) \) is non-decreasing and continuous, while the mapping \( i \mapsto \mathcal{P}_i(q, \kappa) \) is non-increasing.

**Proof.** We assume \( q \neq g \), as the case of location \( g \) is trivial. For the first claim, it suffices to prove that \( \mathcal{P}_n(q, \kappa) \leq \mathcal{P}_n(q, \kappa + t) \) for any \( t \geq 0 \). First notice that \( \mathcal{D}(\kappa + t, \varphi) + t \subseteq \mathcal{D}(\kappa, \varphi) \), where \( \mathcal{D}(\kappa + t, \varphi) + t \) is the set of intervals of \( \mathcal{D}(\kappa + t, \varphi) \) shifted by \( t \). Also, the set of transitions that will be enabled in the future of \( (q, \kappa) \) is a subset of the transitions that will be enabled from \( (q, \kappa + t) \). Thus for any multi-move \( (I, a) \) enabled in \( (q, \kappa + t) \), the multi-move \( (I + t, a) \) is available in \( (q, \kappa) \). Both multi-moves have the same penalty and give rise to the same sets of configurations, so that \( \mathcal{P}_n(q, \kappa) \leq \mathcal{P}_n(q, \kappa + t) \) holds.
We now prove that the function is continuous (when it has finite value). This is clearly the case of $P_0$. Now, if $P_n(q, \kappa)$ is finite, then for any $\varepsilon > 0$, there is an action $a$ and a non-singular interval $I = (\alpha, \beta)$ such that

$$\frac{1}{|I|} + \sup_{d \in I} P_{n-1}(\text{succ}(q, \kappa, a, d)) - \varepsilon \leq P_n(q, \kappa).$$

Now, there exists $\eta > 0$ such that

$$\left| \frac{1}{(\beta - \eta) - (\alpha + \eta)} - \frac{1}{\beta - \alpha} \right| \leq \varepsilon.$$

Then the move $((\alpha + \eta, \beta - \eta), a)$ can be played from any configuration $(q, \kappa + t)$ with $-\eta \leq t \leq \eta$ (provided that such a configuration exists), so that

$$P_n(q, \kappa + t) \leq \frac{1}{(\beta - \eta) - (\alpha + \eta)} \sup_{d \in (\alpha + \eta, \beta - \eta)} P_{n-1}(\text{succ}(q, \kappa + t, a, d))$$

$$\leq \frac{1}{\beta - \alpha} + \varepsilon + \sup_{d \in (\alpha, \beta)} P_{n-1}(\text{succ}(q, \kappa, a, d))$$

$$\leq P_n(q, \kappa) + 2\varepsilon.$$

For the second claim, an easy induction proves that $P_i(q, \kappa) \geq P_{i+1}(q, \kappa)$.

Next we prove the correspondence between $P_i$ and the optimal penalty of winning permissive strategies from a given configuration:

**Lemma 5** For any integer $i$ and for any $\varepsilon > 0$, there exists a winning permissive strategy $\sigma$ such that for any winning configuration $s$,

$$\text{penalty}_{s, g}(\sigma) \leq P_i(s) + \varepsilon.$$

**Proof.** We prove the result by induction on $i$, the case where $i = 0$ being trivial. Assume that the result holds for some $i$. Pick $\varepsilon > 0$. Applying the induction hypothesis, we pick a winning permissive strategy $\sigma$ such that

$$\text{penalty}_{s, g}(\sigma) \leq P_i(s) + \frac{\varepsilon}{2}$$

from any winning configuration $s$.

Pick a configuration $s = (q, \kappa)$. By definition of $P_{i+1}$, there exists an action $a_s$ and an interval $I_s$ such that

$$P_{i+1}(q, \kappa) \leq \text{penalty}(I_s, a_s) + \sup_{d \in I_s} P_i(\text{succ}(q, \kappa, d, a_s)) \leq P_{i+1}(q, \kappa) + \frac{\varepsilon}{2}.$$

We then define a new strategy $\sigma'$ as follows:

$$\sigma'(s) = (I_s, a_s)$$

$$\sigma'(s \cdot \rho) = \sigma(\rho) \quad \text{for any non-empty path } \rho$$

By construction, this permissive strategy satisfies the expected inequality. \qed
Lemma 6  For any winning configuration $s$, and for any permissive strategy $\sigma$ that is winning from $s$, it holds
\[
P(s) \leq \text{penalty}_{s,g}(\sigma).
\]

Proof: The proof is by induction on the number of steps needed by $\sigma$ to reach $g$. More precisely, we prove that for any integer $k$, for any winning configuration $s$, and for any permissive strategy all of whose outcomes from $s$ reach $g$ within at most $k$ steps, it holds
\[
P_k(s) \leq \text{penalty}_{s,g}(\sigma).
\]
The result follows from Lemma 4.

The case $k = 0$ holds trivially, since either $s = (g, \kappa)$ for some $\kappa$ and $P(s) = 0$, or there is no permissive strategy that is winning in zero steps. Assume that the result holds for some integer $k$, and consider a permissive strategy that is winning from $s = (q, \kappa)$ in $k+1$ steps. Let $(I,a) = \sigma(s)$. Then from any configuration $\text{succ}(q, \kappa, d, a)$, the strategy $\sigma'$ defined by $\sigma'(\rho) = \sigma(s \cdot \rho)$ is winning in at most $k$ steps. It follows that $P_k(\text{succ}(q, \kappa, d, a)) \leq \text{penalty}_{\text{succ}(q, \kappa, d, a), g}(\sigma')$. Then
\[
\text{penalty}_{s,g}(\sigma) = \sup_{\pi \in \text{Out}_{I,s,g}} \sum_{j=0}^{|\pi|-1} \text{penalty}(\sigma(\pi_{\leq j}))
\]
\[
= \text{penalty}(I,a) + \sup_{\kappa \in I} P_{k+1}(\text{succ}(q, \kappa, d, a), g)(\sigma')
\]
Hence $\text{penalty}_{s,g}(\sigma) \geq \text{penalty}(I,a) + \sup_{\kappa \in I} P_k(\text{succ}(q, \kappa, d, a)) \geq P_{k+1}(q, \kappa)$, as required.

\[
\square
\]

4.2 Memoryless permissive strategies for one-clock automata

Despite these good properties, the sequence $P_k(q, \kappa)$ does not provide us with an algorithm for computing (or even approximating up to some positive $\varepsilon$) the optimal penalty from a given configuration. This is for two reasons: first, $P_k(q, \kappa)$ only gives an over-approximation of $P(q, \kappa)$, and we have no information about how close this approximation is from the exact value. But more importantly, computing $P_{k+1}(q, \kappa)$ requires computing $P_k(\text{succ}(q, \kappa, d, a))$ for infinitely many moves $(d, a)$. Hence the results of the previous section are by no means effective.

In this section, we prove that for one-clock timed automata, the sequence can be computed, and that the computation terminates in finitely many steps. The proof has several stages: we first prove that any winning multi-strategy can be made to use any resetting transition at most once, without increasing its penalty. Then, we prove that any location will be visited at most once between any two resetting transition. This bounds the number of steps after which the sequence $(P_k)_k$ is constant.

4.2.1 Taking reset transitions at most once.

In this section, we prove that optimal permissive strategies can be made to visit any resetting transition at most once, along any outcome:
Lemma 7  Let $\mathcal{E}$ be the set of resetting transitions of a game on a timed automaton $G$ and let $\sigma$ be a winning permissive strategy from some configuration $s$. We can build a winning permissive strategy $\sigma'$ such that $\text{penalty}_{s,g}(\sigma') \leq \text{penalty}_{s,g}(\sigma)$ and any transition in $\mathcal{E}$ appears at most once along any finite outcome of $\text{Out}_{fin}(s,g,\sigma')$.

Proof. The proof is by induction: for a subset $E \subseteq \mathcal{E}$, we define our induction hypothesis as follows:

\[ \exists \sigma_E \text{ s.t. } \forall \pi \in \text{Out}_f(s,g,\sigma_E), \text{ any edge } e \in E \text{ is taken at most once along } \pi \text{ and } \sigma_E \text{ is winning, and } \text{penalty}_{s,g}(\sigma_E) \leq \text{penalty}_{s,g}(\sigma). \]  

($IH_E$)

Then $\sigma$ satisfies ($IH_\mathcal{E}$). Now assume that we have a strategy $\sigma_E$ satisfying ($IH_E$) for some $E$. We pick an edge $e \in \mathcal{E} \setminus E$, and, writing $E' = E \cup \{e\}$, we build a strategy $\sigma_{E'}$ satisfying ($IH_{E'}$).

For this, we first remark that because $\sigma_E$ is winning (for a reachability objective), the edge $e$ is visited finitely many times along any outcome in $\text{Out}_f(s,g,\sigma_E)$. In other terms, for any finite outcome $\rho$ ending after an occurrence of edge $e$, we can select a path $\pi$ such that $\rho \cdot \pi$ is an outcome of $\sigma_E$, ending after an occurrence of $e$, and such that there is no occurrence of $e$ in the subtree generated by $\sigma_E$ after $\rho \cdot \pi$. We write $f_{\sigma_E}(\rho)$ for the path $\rho \cdot \pi$ constructed above.

Now, we build the strategy $\sigma_{E'}$. We arbitrarily pick a finite path $\rho$. If $\rho$ does not visit edge $e$, then we let $\sigma_{E'}(\rho) = \sigma_E(\rho)$. If $\rho$ visits edge $e$ once, we write $\rho = \rho_1 \cdot \rho_2$, where edge $e$ is the last transition in $\rho_1$, and define $\sigma_{E'}(\rho) = \sigma_E(f_{\sigma_E}(\rho_1) \cdot \rho_2)$. Finally, the value of $\sigma_{E'}$ over paths that visit $e$ more than once is irrelevant.

It remains to prove that $\sigma_{E'}$ satisfies both conditions of ($IH_{E'}$). First, pick a maximal outcome $\rho$ in $\text{Out}_f(s,g,\sigma_{E'})$. If $\rho$ does not visit edge $e$, then it is also an outcome of $\sigma_E$, hence it visits any edge in $E$ at most once, and the first property follows. If $\rho$ visits $e$ at least once, then one easily proves that writing $\rho = \rho_1 \cdot \rho_2$ where $\rho_1$ ends after the first visit to edge $e$, it holds that $f_{\sigma_E}(\rho_1) \cdot \rho_2$ is an outcome of $\sigma_E$. By construction, $\rho_2$ never visits edge $e$, and $\rho_1$ is a prefix of $f_{\sigma_E}(\rho_1)$. It follows that the edges in $E'$ are visited at most once along $\rho$.

We use similar arguments for proving that the penalty of $\sigma_{E'}$ is less than or equal to that of $\sigma_E$. In order to prove this, we prove that for any outcome $\rho' \in \text{Out}_f(s,g,\sigma_{E'})$, there is an outcome $\rho \in \text{Out}_f(s,g,\sigma_E)$ such that the penalty of $\sigma_{E'}$ along $\rho'$ is higher than that of $\sigma_E$ along $\rho$. In case $\rho$ never visits edge $e$, then letting $\rho' = \rho$, and noticing that $\sigma_{E'}$ plays as $\sigma_E$ all along $\rho$, we get the result. If $\rho = \rho_1 \cdot \rho_2$, with $\rho_1$ ending after visiting edge $e$, then $\rho_1$ is an outcome of both strategies, and both strategies play the same moves along this outcome, so that they have the
same penalties; similarly, $\rho_2$ is an outcome of $\sigma_{E'}$ after $\rho_1$ and an outcome of $\sigma_E$ after $f_{\sigma_E}(\rho_1)$, and both strategies plays the same moves along those paths. Hence the penalty of $\sigma_E$ is higher than that of $\sigma_{E'}$, which completes the proof.

\[\square\]

4.2.2 No cycles between reset transitions.

We use similar arguments for proving that any location of a timed automaton $G$ is visited at most once between any two consecutive resets of the clock:

**Lemma 8** Let $\sigma$ be a winning permissive strategy from some configuration $s$. We can build a winning permissive strategy $\sigma'$ such that $\text{penalty}_{s,G}(\sigma') \leq \text{penalty}_{s,G}(\sigma)$ and for any finite outcome $\pi$ of $\text{Out}_{\text{fin}}(s,g,\sigma')$,

- if $\pi$ involves no resetting transitions, any location of $G$ appears at most once along $\pi$;
- otherwise, along $\pi$, any location of $G$ appears at most once between any two consecutive resets, before the first reset, and after the last reset.

**Proof.** The proof is by induction on a subset $L \subseteq Q$ of locations. We inductively build strategies $\sigma_L$ with non-increasing penalties, and visiting locations in $L$ at most once between any two consecutive resets, before the first reset and after the last one.

The induction is trivially initiated with $\sigma_\emptyset = \sigma$. Now, assume that $\sigma_L$ has been obtained for some $L$, and pick a location $\ell \in Q \setminus L$. Assume that there is an outcome $\rho \in \text{Out}_f(s,g,\sigma_L)$ along which $\ell$ appears twice along a segment without reset (i.e., there is at least one edge between the two occurrences, but no reset). Since $\sigma_L$ is winning (for the reachability objective), $\ell$ can be only visited finitely often in that segment. Therefore, for any finite outcome $h$ ending with a reset transition or $h$ to be the empty play, for any finite continuation $\rho$ without any occurrences of reset transitions such that $h \cdot \rho \in \text{Out}_{\text{fin}}(s,g,\sigma')$, we can select a path $\pi$ containing no clock reset such that $h \cdot \rho \cdot \pi \in \text{Out}_{\text{fin}}(s,g,\sigma')$ ends in $\ell$ and there is no further occurrence of $\ell$. We write $f_{\sigma_L}(h \cdot \rho)$ for $h \cdot \rho \cdot \pi$.

Now, we build the strategy $\sigma_{L'}$. Let $\gamma$ be an arbitrary play in $\text{Out}(\sigma_L)$. Then $\gamma$ is of the form $h \cdot \rho \cdot m$ with $h, \rho$ as indicated above and $m$ to be any continuation. We construct the new strategy as $\sigma_{L'}(\gamma) = f_{\sigma_L}(h \cdot \rho) \cdot m$ whenever $\rho$ contains $\ell$ exactly once and $\sigma_{L'}(\gamma) = \sigma_L(\gamma)$ otherwise.

It remains to prove that $\sigma_{L'}$ satisfies the two required conditions. The second condition implies that the property of visiting $\ell$ at most once remains. If $\ell$ is visited in a segment, then the strategy at the first occurrence of $\ell$ is replaced by the strategy at the last occurrence of $\ell$ within that segment. Hence, $\ell$ is visited only once in this case. The first condition actually implies from the second condition for the case when $h$ is the empty play.

**Example 9** Notice that visiting the same location several times can be necessary, even for reachability objectives with deterministic strategies. Figure 1 displays such an example.

4.2.3 Computation of $\mathcal{P}_i(q, \kappa)$.

The arguments above entail that the sequence $\mathcal{P}_i$ converges in finitely many steps. It remains to explain how we compute these functions. We write $C$ for the set of constants appearing in the
Fig. 3: An automaton where several visits to $\ell_1$ are needed

clock constraints of the automaton. Computing $\mathcal{P}_1(q, \kappa)$ is easy, as it suffices to find the action $a$ with the largest time interval $I$ for which $\text{succ}(q, \kappa, d, a) = (g, \kappa')$ for any $d \in I$. One easily notices that the lower bound of the largest $I$ is either 0 or of the form $c - \kappa(x)$, for some constant $c \in C$. Similarly, the upper-bound is of the form $c' - \kappa(x)$ or $+\infty$. It follows that $\mathcal{P}_1(q, \kappa)$ is made of finitely many pieces, on which it is either a constant, or of the form $\frac{1}{d - \kappa(x)}$.

We prove by induction that $\mathcal{P}_1(q, \kappa)$ is always piecewise of the form $b_i^n + \frac{c_i^n}{d_i^n - \kappa(x)}$ (or $+\infty$), with finitely many pieces $\langle e^n_i, f^n_i \rangle$, with rational constants when $n \leq 1$, and algebraic constants for larger $n$.

As we just showed, this is the case of $\mathcal{P}_0$ and $\mathcal{P}_1$. Suppose this is the case at step $n$. Then

$$\mathcal{P}_{n+1}(q, \kappa) = \min_{a \in \text{Act}} \inf_{I = \langle e, f \rangle} \left( \text{penalty}(\langle e, f \rangle, a) + \sup_{d \in I} \mathcal{P}_n(\text{succ}(q, \kappa, d, a)) \right)$$

From $(q, \kappa)$, several transitions may be possible, with guards of the form $x \in \langle \alpha_j, \beta_j \rangle$, leading to configurations $(q_j, \kappa_j)$, with $\kappa_j(x) \in \{0, \kappa(x)\}$ depending on whether the clock is reset along the corresponding transition.

Following Lemma 4, we have that $\sup_{d \in \langle e, f \rangle} \mathcal{P}_1(\text{succ}(q, \kappa, d, a))$ can only be achieved by taking the transition as late as possible, hence when $d$ tends to $f$ (the upper bound of $I$) or when $\kappa(x) + d$ tends to some constant in $C$ (which corresponds to taking a transition as late as possible while it is available). The same argument entails that $e$ (the lower bound of $I$) can be chosen as a constant in $C$: since the worst case is when the opponent plays as late as possible, $e$ can be taken as low as possible as long as it does not enable a new transition.

In the end, there are only finitely many values to try for the action to play and the lower bound $e$ of $I$ (satisfying $e = 0$ or $\kappa(x) + e \in C$). For those choices, $f \mapsto \sup_{d \in \langle e, f \rangle} \mathcal{P}_n(\text{succ}(q, \kappa, d, a))$ is easily computed as a function of $f$, following the remark above. It is piecewise of the form $b_i^n + \frac{c_i^n}{d_i^n - \kappa(x)}$ (but it need not be continuous at positions where $\kappa(x) + f \in C$). The function (of $f$) to optimize is then of the form

$$\frac{1}{f - (c - \kappa(x))} + b_i^n + \frac{c_i^n}{d_i^n - (\kappa(x) + f)}$$

This function is to be optimized over an interval with bounds of the form $\langle e'_i, f'_i \rangle$. The extremal

\[\text{penalty}(\langle e, f \rangle, a)\]
values can be obtained at the bounds of the interval, or at the root of a polynomial of degree 2 obtained by computing the derivative of the above function.

Finally we obtain the following:

**Theorem 10** The optimal penalty (and a corresponding almost-optimal strategy) for reaching a target location in a timed automaton can be computed in polynomial time.

**Example 11** We come back to our Example 1, and compute the optimal penalty for reaching the target. We initialize the computation by letting $\mathcal{P}_0(\ell, \kappa) = +\infty$ for all $\ell \neq \ell_f$; we also let $\mathcal{P}_1(\ell_f, \kappa) = 0$ for all $i$.

Then only configurations $(\ell_2, \kappa)$ where $\kappa(x) \leq 1$ are winning in one step. For those configurations, $\mathcal{P}_1(\ell_2, \kappa) = \frac{1}{1-\kappa(x)}$ (hence it is $+\infty$ when $\kappa(x) = 1$). The other configurations have value $+\infty$.

At step 2, any configuration $(\ell_3, \kappa)$ is winning, since the clock is reset when going to $\ell_2$.

We have $\mathcal{P}_2(\ell_3, \kappa) = \mathcal{P}_1(\ell_2, x \leftarrow 0) = 1$. Configuration $(\ell_0, \kappa)$ with $\kappa(x) \leq 1$ are also winning in two steps. The optimal penalty in two steps is computed as follows:

\[
\mathcal{P}_2(\ell_0, \kappa(x)) = \inf_{0 \leq e \leq f < 2-\kappa(x)} \left( \frac{1}{f-e} + \sup_{e \leq d \leq f} \mathcal{P}_1(\ell_2, \kappa+d) \right)
\]

\[
= \inf_{0 \leq f < 2-\kappa(x)} \left( \frac{1}{f} + \mathcal{P}_1(\ell_2, \kappa+f) \right)
\]

\[
= \inf_{0 \leq f \leq 1-\kappa(x)} \left( \frac{1}{f} + \frac{1}{1-(\kappa(x)+f)} \right)
\]

One easily obtains that the infimum is reached when $f = \frac{1-\kappa(x)}{2}$, with $\mathcal{P}_2(\ell_0, \kappa) = \frac{4}{1-\kappa(x)}$.

The only new transition to consider at step 3 is the transition from $\ell_2$ to $\ell_3$. The penalty in $\ell_2$ is computed as follows:

\[
\mathcal{P}_3(\ell_2, \kappa) = \inf_{0 \leq e \leq f < 2-\kappa(x)} \left( \frac{1}{f-e} + \sup_{e \leq d \leq f} \mathcal{P}_2(\text{succ}(\ell_2, \kappa,d,a)) \right)
\]

The successor of $(\ell_2, \kappa)$ may be either $\ell_f$ (with penalty 0) or $\ell_3$ (with penalty 1); the latter will be chosen by the opponent as soon as $\kappa(x) + f > 1$. Hence

\[
\mathcal{P}_3(\ell_2, \kappa) = \inf_{0 \leq e \leq f < 2-\kappa(x)} \left( \frac{1}{f-e} + \mathbb{1}_{(1-\kappa(x);+\infty)}(f) \right)
\]

\[
= \inf_{0 \leq f < 2-\kappa(x)} \left( \frac{1}{f} + \mathbb{1}_{(1-\kappa(x);+\infty)}(f) \right)
\]

(We denote $\mathbb{1}_I(f)$ for some interval $I$ and value $f$ as the function outputting 1 if $f \in I$ and 0 otherwise.) We optimize this function by considering two cases: for $f \leq 1 - \kappa(x)$, the penalty is $1/f$, which is minimized when $f = 1 - \kappa(x)$ with value $1/(1 - \kappa(x))$; for $1 - \kappa(x) < f < 2 - \kappa(x)$, the penalty is $1/f + 1$, which is optimized when $f$ tends to $2 - \kappa(x)$ with value $1/(2 - \kappa(x)) + 1$. In the end, when $\kappa(x) \leq x_0 = (3-\sqrt{5})/2 \approx 0.38$, the optimal multi-strategy is to play interval $[0, 1 - \kappa(x)]$, with penalty $1/(1 - \kappa(x))$; when $\kappa(x) \geq x_0$, the optimal multi-strategy
is to play interval $[0, 2 - \kappa(x))$, with penalty $1 + 1/(2 - \kappa(x))$. Notice that the optimal penalty is a continuous function of $\kappa(x)$. Also notice that this gives the optimal penalty for winning from $\ell_2$.

We now compute $P_4$. Following Lemma 7, there is no hope of improving the penalty from location $\ell_3$, so that only $\ell_0$ has to be considered. We have:

$$
P_4(\ell_0, \kappa(x)) = \inf_{0 \leq e \leq f < 2 - \kappa(x)} \left( \frac{1}{f - e} + \sup_{e \leq d \leq f} P_1(\ell_2, \kappa + d) \right)
$$

$$
= \inf_{0 \leq f < 2 - \kappa(x)} \left( \frac{1}{f} + P_3(\ell_2, \kappa + f) \right)
$$

When $\kappa(x) + f \in [0, x_0]$ (assuming $\kappa(x) \leq x_0$), the function to optimize is $1/f + 1/((1 - \kappa(x) - f)$. This function has no local minimum for $\kappa(x) + f \in [0, x_0]$. Hence over $[0, x_0 - \kappa(x)]$, the infimum is when $f = x_0 - \kappa(x)$, and its value is $1/(x_0 - \kappa(x)) + 1/(1 - x_0)$. When $\kappa(x) + f \in [x_0, 2)$, the function to optimize is $1/f + 1/(2 - \kappa(x) - f)$. The local minimum is reached when $f = (2 - \kappa(x))/2$, which indeed satisfies $\kappa(x) + f \in [x_0, 2)$ when $\kappa(x) < 2$. In the end, we obtain $P_4(\ell_0, \kappa) = P(\ell_0, \kappa) = 1 + \frac{2}{2 - \kappa(x)}$.

### 4.3 Extension to one-clock timed games

In the computations above, non-determinism is solved by an adversary with very limited capabilities. We explain below how our approach can be lifted to timed games, where the second player has more power. In (classical) timed games, at each step, both players propose a delay and an action (where the set of actions is partitioned between Player-1 actions and Player-2 actions); the player with the shortest delay (if any) then applies her move, and the game continues. Non-determinism (when both players propose the same delay) is solved by the second player [AFH+03]. This framework can be lifted to the setting of permissive strategies: the first player proposes a multi-move $(I, a)$, while the second player proposes a move $^7 (\delta, \alpha)$. In case $\delta < e$ for all $e \in I$, then the move of the second player is applied; in case $\delta > f$ for all $f \in I$, the second player selects a delay $d \in I$, and the move $(d, a)$ is applied; finally, if $\delta \in I$, Player 2 may decide to either apply her move $(\delta, \alpha)$, or to select some $d \in I$ with $d < \delta$, and to apply the move $(\delta, a)$.

Our results above for permissive strategies in timed automata can be extended to timed games, with the following changes: First, we adapt the computation of the sequence $\mathcal{D}_{i+1}$, in order to take the extended capabilities of the opponent. More precisely, instead of only maximizing $\mathcal{P}_i(\text{succ} (q, \kappa, d, a))$ when $d$ ranges over $I$, she now also has the opportunity to apply another move $(\delta, \alpha)$ for any $\delta$ in or “before” $I$ (i.e., for which there exists $t \geq 0$ s.t. $\delta + t \in I$):

$$
\mathcal{D}_i(g, \kappa) = 0
$$

$$
\mathcal{D}_0(q, \kappa) = +\infty \quad \text{for all } q \neq g
$$

$$
\mathcal{D}_{i+1}(q, \kappa) = \min_{a \in \text{Act}_1} \left( \frac{\text{penalty}(I, a) +}{\max_{d \in I} \left( \sup_{\delta \in I} \mathcal{D}_i(\text{succ}(q, \kappa, d, a)) + \sup_{\alpha \in \text{Act}_2} \mathcal{D}_i(\text{succ}(q, \kappa, \delta, \alpha)) \right)} \right)
$$

$^7$ We only consider permissive strategies of the first player in this setting, as we only want to minimize the penalty for the protagonist.
With this definition, the first statement of Lemma 4 fails: the global penalty of move \((I, a)\) from \((q, \kappa + t)\) might be better than the penalty of move \((I + t, a)\) from \((q, \kappa)\), because the latter offers more possibilities to Player 2. Still, in the one-clock setting, the result holds in case \(\kappa(x)\) and \(\kappa(x) + t\) are not separated by any constant of the automaton. In other terms, for one-clock games, \(t \mapsto \mathcal{D}_n(q, \kappa + t)\) is piecewise non-decreasing and piecewise continuous, with pieces defined by the constants of the automaton.

In the end, we can still look for the optimal choice of Player 2 within a finite set, and all the other arguments that we used in the one-player setting still apply. Finally, we obtain:

**Theorem 12** The optimal penalty (and a corresponding almost-optimal strategy) for reaching a target location in a timed automaton game can be computed in polynomial time.

### 4.4 Discussions on other ways of computing penalties

Our choice of the way we compute penalties is only one among many relevant possibilities: for instance, we believe that assigning penalty \(+\infty\) to punctual intervals is reasonable when dealing with robustness, but it might be argued that we should compare how much the player reduces the interval compared to what she is allowed to do. In other terms, if playing punctual is the only possibility, then it should have bounded penalty. This requires extra definitions and arguments, which we leave for future work.

Accumulating penalties along a run is also something we could change. It fits well with finitary objectives such as reachability, and favors short paths. Another solution would be to take the maximum penalty along the outcomes, which could be handled by a trivial modification of our algorithm. Averaging would be yet another option, which looks more complex to compute.

### Bibliography


Research Idea Papers
Automated Hazard Analysis with pMAX-SMT for Automobile Systems

Shogo Hattori¹, Shoji Yuen², Hiroyuki Seki³ and Shuichi Sato⁴

¹ hatsutori@sqlab.jp
² yuen@is.nagoya-u.ac.jp
³ seki@is.nagoya-u.ac.jp
Graduate School of Information Science, Nagoya University, Japan

⁴ shuichi-sato@mosk.tytlabs.co.jp
Toyota Central R&D Labs., Inc., Japan

Abstract: We propose a method to automatically detect the patterns of signals with unexpected data failures that lead to error states in automobile systems. We regard the target system as a discrete-state transition system, in which the system does not reach any hazardous states as it is working normally. We encode bounded behavior of the transition system as constraint of by conjunctive normal forms for the pMAX-SMT problem with “soft clauses” representing the potential signal data failures.

Keywords: Hazard analysis, pMAX-SMT, Automobile control system

Modern automobile systems are composed of dozens of computers (electronic control unit, or ECU), and they interact with others through network. Further automobile systems get connected with external networks, as seen in vehicle-to-vehicle (V2V) communications[BTD06]. While these enable us to make automobiles smarter, communication capabilities make the automobile system quite complicated. As a result, it takes much time for engineers to analyze the system safety since possible environments become diverse or even infinite in general. The trade-off between the amount of workload and the strictness of the safety analysis is a big concern in designing automobile control system. We propose an encoding of the automobile system behavior for finding the disturbed patterns of signal failures leading to error states. For encoding the behavior of system, we check the partial satisfiability of a formula consisting of clauses marked as hard and clauses marked as soft. The partial maximum satisfiability modulo theories (pMAX-SMT) problem[Arg11][DD06] is the problem for finding (if exists) an assignment to variables that makes every clause marked as hard true and makes as many as clauses marked as soft true.

Figure 1 shows an overview of the proposed method. First, the target system is automatically converted into a formula where an error state is reachable if and only if the formula is satisfiable. Boolean formula Φ for input to the pMAX-SMT solver introduces a variable for each signal in the target system, called a cushion variable, which may hold an erroneous value for the signal. If the values of cushion variables differ from that of the original variables, it means that the signal failure happens. Φ consists of hard clauses expressing the bounded trace of the target system and error states, and soft clauses specifying that the values of original signals are equal to that of cushion variables. We assume that as long as the values of the variables are in the designated ranges, the error states are never reached. That is, Φ is unsatisfiable if all the clauses are regarded
as hard clauses. Let $\Phi_1$ be the constraints for bounded traces $\Phi_1$ and the error state property be $\Phi_2$. $\Phi$ consists of (1) the conjunction of equality between signal variables and corresponding cushion variables (as $\Phi_3$), and (2) the copy of $\Phi_1 \land \Phi_2$ replacing signal variables with the corresponding cushion variables. By the assumption, $\Phi_1 \land \Phi_2$ is unsatisfiable. We declare $\Phi_3$ as soft clauses and we check if $\Phi = \Phi_3 \land \Phi_4$ is pMAX-satisfiable.

Second, we input $\Phi$ constructed so far to a pMAX-SMT solver. When the solver answers "unsatisfiable", the system is guaranteed to be safe with respect to the protected variables. When the solver answers "satisfiable", the assignments to the variables in the unsatisfied clauses can be obtained, which represent a disturbed pattern. We can obtain all disturbed patterns by repeatedly adding hard clauses expressing that the value of the cushion variable is equal to that of the original variable for each signals contained in the answer from the solver and feeding the solver with the revised formula until the solver answers "unsatisfiable."

We applied our proposed method to the hazard analysis of a simplified automobile system with three ECUs and obtained disturbed signal patterns which would be difficult to be discovered by hand.

**Acknowledgements:** We would like to express our gratitude to Hideki Nomura, Yutaka Inamori, and Noriyoshi Sano for constructive discussions.

**Bibliography**


Model Transformation of high-level requirements in a Domain Specific Language into a Formal Specification Language

Suraj Ajit\textsuperscript{1}, Oyindamola Olajubu\textsuperscript{1}, Scott Thomson\textsuperscript{2} and Mark Edwards\textsuperscript{2}

\textsuperscript{1}University of Northampton, UK  
\textsuperscript{2}GE Aviation Systems Ltd, Cheltenham, UK

Abstract: This paper provides a brief context of current work being done in an industrial research project that involves automated transformation of high-level requirements expressed in a Domain Specific Language into test cases. The paper introduces a research idea/approach and outlines relevant research questions/challenges. Finally, the paper briefly discusses related work in this area.

Keywords: DSL, Model Transformation, Requirements

1 Introduction

The industrial research project discussed here is collaborative work between GE Aviation Systems and the University of Northampton. GE Aviation Systems primarily uses natural language (textual ‘shall’ statements) to express software requirements. These textual statements are often ambiguous, untestable, incorrect, missing detail, etc. Finding this out late in the development lifecycle proves very expensive. To this end, GE Aviation Systems has turned to Model-Based Development (MBD). They still use textual statements to express software requirements, but they supplement the requirements writing activity with modelling and simulation so that engineers gain a better understanding of the requirements and their faults. The models can then be refined and improved and serve as the Software Design artefacts that are used to auto-generate code. Currently, this means they have to write tests manually to test the design against the requirements. Within the joint research project, our work has focused on developing a Domain Specific Language (DSL) that can be used by domain experts to express high-level requirements in textual form. The use of DSL should minimise or eliminate the shortcomings of using a natural language that were stated earlier in this section. DSL should also be easier for a domain expert to understand, learn and use, when compared to a formal specification language. We then apply model transformations to transform the requirements into test cases [OAJ’15].

2 Research Challenge

The DSL discussed in the previous section offers several advantages over natural language to express high-level requirements. However, it lacks many benefits when compared to using a formal specification language such as Z [SA92]. These benefits may include software verification, theorem proving and model checking. For instance, it is important to detect any inconsistencies among requirements expressed by the DSL. Many systems developed in the aviation industry including GE Aviation Systems tend to be critical systems, which need to meet stringent dependability requirements (safety-critical, security-critical, business-critical, performance-critical, etc.). Software development practices within GE Aviation Systems need to adopt guidance provided by document standards such as DO178C, DO331 and DO333 from
RTCA (http://www.rtca.org/) for getting assurance, certification and acceptance. Hence, it may be useful for industries such as GE Aviation Systems to use formal methods for the specification, development and verification of software systems. Our research idea is to develop a framework that supports transformation of requirements expressed in DSL to a formal language specification. The DSL has been currently developed using Xtext, a language development tool based on the Eclipse Modelling Framework (https://eclipse.org/Xtext/). The CZT tool is an eclipse-based tool for Z specification in both Latex and Unicode formats. To meet the objective, we intend to investigate the application of model transformation techniques to transform the meta-model of a DSL to a meta-model of the Z language. This should then result in automated (or semi-automated) model transformations from the DSL to Z specifications that can be used by the CZT tool. To summarise, the underlying research questions are as follows: Could we apply model transformation techniques to transform high-level requirements in a DSL into a formal specification language? If so, could the process be fully automated? What are the benefits (if any)? What tools exist to support automated verification of high-level requirements? The next section discusses related work in this area.

3 Related Work

In [ASBC13], requirements expressed in natural language are transformed into Event-B models. The transformation is partially automated, where intermediate models are initially constructed from pre-classified requirements. Event-B models are then automatically generated from these intermediate models. In [KCD01], a model-based approach is used for transformation of UML models into formal Object Z models. In [BFLM05], the B formal method is used for verification of DSL specifications. The B method is used for refining specifications in Bossa, a language specific to the railway domain.

Bibliography


A Verified Virtual Machine for Safety-Critical Java

James Baxter and Ana Cavalcanti

University of York, UK

Abstract: The Safety-Critical Java (SCJ) specification has been created to facilitate development of certifiable Java programs, but as yet there has been no work on verification of SCJ virtual machines. We describe a framework for verification of an SCJ virtual machine and present some preliminary results.

Keywords: Safety-Critical Java, virtual machine, verification, Java bytecode

Introduction Since its release in 1995, the Java programming language [GJS+13] has increased in popularity and is now in use on a wide variety of platforms, in particular, embedded systems. Many of these are also safety-critical and require certification. To allow development of such systems, the Safety-Critical Java (SCJ) specification [L+13] has been created as a subset of the Real-Time Specification for Java (RTSJ) [GB00]. SCJ restricts the RTSJ to features that facilitate static reasoning, so features such as the garbage-collected heap and dynamic class loading are absent from SCJ. This has encouraged the development of several techniques for showing that SCJ programs are safe and correctly fulfil their specifications.

To show that SCJ programs are executed correctly, however, the Java Virtual Machine (JVM) running the programs must be shown to be correct as well. Since SCJ differs from standard Java in areas such as scheduling and memory management, a specialised virtual machine is needed. Additionally, as SCJ programs are targeted at embedded systems, an SCJ virtual machine (SCJVM) must be small and fast. This requirement is typically met by precompiling the Java bytecode of the program to a language such as C. These features can be seen in existing SCJVMs, such as Fiji VM [PZV09] and icecap HVM [SKR12]. However, as far as we know, no SCJVM has been formally verified and little work appears to have been done in this area.

Research Proposal We propose to construct a framework for verification of an SCJVM. The first component required in this framework is a formal specification of an SCJVM, which shapes the rest of the work. There is at present no clear specification of what is required of an SCJ virtual machine or how it differs from a standard JVM. The specification of requirements needs to consider the requirements imposed, both explicitly and implicitly, by the SCJ specification [L+13]. It is also helpful to consider the approach taken by some existing SCJVMs on points where the SCJ specification is unclear. The SCJVM must also meet the standard JVM specification [LYBB14] on points such as how to interpret Java bytecode instructions.

The second component required is a verified compilation strategy from Java bytecode to a language from which native code can be generated and integrated with existing code for embedded systems. We consider C as the target language as it meets those requirements and is already the target language for several existing SCJVMs. The compilation strategy can be verified using standard compiler verification techniques, which fall into two main approaches: the commuting diagram approach, which was identified by Lockwood Morris [Mor73] and can be seen in
much of the work on compiler verification, and the algebraic approach, which was proposed by Hoare [Hoa91] and further developed by Sampaio [Sam93]. The compilation strategy must be formally specified to permit proof of its correctness and, as part of that specification, a formal semantics of Java bytecode and C is required. There is already some work in that area that can be leveraged but the semantics of Java bytecode may differ slightly for SCJ due to the absence of dynamic class loading and the differing scheduling and memory models.

Both the specification of requirements and the compilation strategy needs to be written in some notation that can be machine-checked. This allows proofs of the specification’s correctness to be performed in an automated theorem prover, thereby providing greater certainty that the proofs are free from mistakes. This means that the framework will eventually consist of the specification of requirements and compilation strategy together with machine-checked proofs of their correctness. It is envisioned that an SCJVM can then be developed and shown to fulfil the specification; it will be correct by virtue of implementing a valid specification.

**Preliminary Results** We have already identified the requirements for services that need to be offered by an SCJVM besides bytecode execution, covering areas such as scheduling and memory management. We have stated these requirements in the Circus specification notation [OCW09] and a draft of the specification can be found in [Bax15]. The effort of proving consistency of the specification using Z/Eves is ongoing. We will specify a compilation strategy from Java bytecode to C using the algebraic approach to compiler verification in our later work.

**Bibliography**

http://www.cs.york.ac.uk/circus/publications/techreports/reports/scjvm-requirements.pdf


https://jcp.org/aboutJava/communityprocess/edr/jsr302/index2.html


Structural Resolution for Automated Verification

Ekaterina Komendantskaya\textsuperscript{1}, Peng Fu\textsuperscript{2,*} and Patricia Johann\textsuperscript{3}

\textsuperscript{1} katya@computing.dundee.ac.uk
\textsuperscript{2} p.fu@dundee.ac.uk
University of Dundee, UK

\textsuperscript{3} johannp@appstate.edu
Appalachian State University, USA

Abstract: We pose a research question: Can the newly-developed structural resolution be used to extend coinductive methods in automated theorem proving?

Keywords: Induction, Coinduction, Recursion, Corecursion, Structural Resolution

1 Coinductive methods in ITP and ATP

Mathematical induction is pervasive in programming and program verification. It arises in data definitions (e.g., it describes some algebraic data structures), it underlies program semantics (e.g., it explains how to reason about finite iteration and recursion), and it gets used in proofs (e.g., it supports lemmas about data structures used in inductive proofs). Coinduction, too, is important in programming and program verification. It arises in infinite data definitions (e.g., lazily defined infinite streams), semantics (e.g., of concurrency), and proofs (e.g., of observational equivalence, or bisimulation, of potentially infinite processes). It is thus desirable to have good support for both induction and coinduction in systems for reasoning about programs.

The first implementations of coinduction were pioneered in interactive theorem proving (ITP) [Gim98], where the duality of inductive and coinductive methods was achieved by distinguishing inductive from coinductive types, recursive functions consuming inputs of inductive types from corecursive functions producing outputs of coinductive types, and methods for constructing inductive proofs from those for constructing coinductive proofs.

Recently, the rapid development of automated theorem proving (ATP) in general, and SAT/SMT solvers in particular, has opened the way to introducing induction and coinduction to ATP [LM14, RB15, S\textsuperscript{+}07], and thus to bridging the previously existing gap between coinductive methods in ITP and ATP. Some coinductive methods of ITP can be easily translated to ATPs. For example, definitions of (inductive and) coinductive types in ITP translate naturally into fixed-point definitions in ATP. However, some coinductive methods in ITP are much trickier to adapt to ATP. In particular, the notion of program and function \textit{productivity} that is so central to the theory of corecursive functions in ITP has, until recently, been virtually absent from ATP.

2 Why ITP Theory of Productivity is a challenge for ATP?

In ITP, productivity plays a role for coinductive computations dual to that of termination for inductive ones. To safely use potentially non-terminating programs defined by corecursion, a

\textsuperscript{*} This author is sponsored by the EPSRC Grant EP/K031864/1.
system must be able to guarantee that they are productive, in the sense that each part of their (potentially) infinite coinductive output will be generated in finite time. In the absence of productivity, the soundness of systems supporting corecursion is not ensured. Some systems, including Coq and Agda, guarantee productivity via syntactic guardedness checks [Gim98] which ensure that any recursive call to a corecursive function occurs under a call to a constructor of the program’s output data type, and thus that the program’s output yields finite observations by terminating recursive programs.

In ITP, productivity and guardedness depend crucially on types and type constructors, as well as on reductions by pattern matching computations. Indeed, the very definition of a productive function in ITP requires that the type of its output data be coinductive. But in the untyped setting of ATP there is no way to ensure that a function’s output is coinductive, and thus that coinductive reasoning will be sound for it. In addition, ITP guardedness checks do not work if reductions by pattern matching are replaced by derivations via (SLD-)resolution, as commonly used in ATP.

Some approaches to coinduction in ATP, such as those of [RB15, S+07], can construct coinductive proofs only for terms that can be represented as rational (or regular) trees. The corresponding regular corecursion is relatively easy to handle operationally via cycle detection [S+07], and yields cyclic closed terms. This is crucial for SAT/SMT solvers [RB15]. Other approaches “guard” corecursion by imposing a recursive observational measure on corecursive functions — thus effectively viewing corecursion as a form of recursion [LM14]. All such additional restrictions are unnecessary in ITP, and, due to their ad hoc nature, work for only restricted cases of corecursion. In the general case, they do not actually capture the essence of ITP productivity.

3 Can Structural Resolution help?

Structural Resolution [JKK15, JKF+15] is a newly-proposed resolution method that supports a very natural definition of productivity, as well as semi-decidable guardedness checks for it.

The propositional resolution rule underlying most modern ATPs is given by \( C \lor A \neg B \lor D \), where \( A, C, D \) are propositions. The standard (first-order) resolution rule is then (1) \( \frac{C \lor A \neg B \lor D \theta(C \lor B \lor D)}{C \lor D} \), where \( A, B, C, D \) are (first-order) terms and \( \theta \) is a unifier of \( A \) and \( B \) (i.e., \( \theta(A) = \theta(B) \)), and the pattern matching reduction used in ITP is given by the restriction of the above rule to (2) \( \frac{C \lor A \neg B \lor D \theta(C \lor B \lor D)}{C \lor D} \), where \( A, B, C, D \) are (first-order) terms and \( \theta \) is a matcher of \( A \) and \( B \) (i.e., \( \theta(A) = B \)). The restricted rule (2) is of course incomplete relative to rule (1), and requires a further rule to emulate the effect of standard resolution by rule (1). This rule is given by (3) \( \frac{C \lor A \neg B \lor D \theta(C \lor B \lor D)}{C \lor A, \theta(\neg B) \lor \theta(D)} \), where \( A, B, C, D \) are (first-order) terms and \( \theta \) is a unifier of \( A \) and \( B \).

Subject to careful definitions, derivations comprising rules (2) and (3) can be shown to emulate the effect of standard resolution by rule (1) [FK15]. These are called derivations by structural resolution. Importantly, and perhaps surprisingly, structural resolution bears properties that are key to the theory of coinduction and productivity for resolution-based methods. Logic programs that correspond to productive corecursive functions in ITP are precisely those for which reductions steps by rule (2) always terminate (give finite observations), and reductions by rule (3) can be applied infinitely (thus accounting for the coinductive nature of a program’s “output”). These two properties can be used to define productivity via structural resolution in LP. We ask: Can structural resolution give a theory of productivity for other resolution-based ATPs?
Bibliography


Model Checking Web Applications using SPIN and UPPAAL

Mohammed Alzahrani

Department of Computer Science, Heriot-Watt University

Abstract: With the complex nature of web applications and their unpredictable environment, developers need new methods to ensure web applications’ usability, security and stability on early stages of the development. Modelling web applications behaviour should not only check the static HTML links but also the dynamic features that changes with each execution triggered by different input from the user or the server side. Users have the ability to change the flow of navigation at any point during the execution. Therefore, a web application need to be evaluated as complete software product. Authentication, navigation and security properties are linked together in order to obtain a final secure web application. Eliminating any of the properties during the verification stage could lead to vulnerabilities and design error. This paper present a method to use the model checking tools SPIN and UPPAAL to model a web application dynamic navigation flow. We show how to simulate different scenarios of an online banking web application. We then verify navigation and related security properties by examining the sequence of actions.

Keywords: Model Checking, Web Applications, Security, SPIN, UPPAAL.

1 Introduction

Web applications have become common in today’s economic and social life. The complex structure and rapid development of web applications lead to vulnerabilities that expose sensitive information to both users and business. According to a report by [Cen14], 96% of tested web applications in 2013 have vulnerabilities which are considered high risk. In addition, the average number of vulnerabilities per web application were up to 14 in each in 2013. A recent report by [Hof13] showed that in 2012 alone, there were more than 800 reported hacking incidents, and 70% of those were carried out through web application vulnerabilities. Therefore, detecting flaws at first stages of development cycle ensures usable and secure web applications.

As a result, there is an ongoing need to support the design, implementation and verification of web applications to be developed and applied. A successful verification technique is model checking [Cla08]. Model checking provides a fully automatic analysis and verification of the initial system design. By constructing a model of a system and its desirable behaviour, a model checker can carry out a verification task. If an error is found, the tool shows under which circumstances the error accrued. Traditionally, model checking has been widely used for verification of hardware systems. More recently model checkers have been increasingly used for the verification of communication, security protocols and software development [Cla08].
2 Modelling using SPIN and UPPAAL

We use the SPIN model checker [Hol04]. The input language PROMELA is expressive to model systems as processes communicating to each other via channels or through shared memory, represented as global variables. By modelling the possible sequence of actions can lead to more secure and usable web applications. In SPIN we can use the simulator to create graphical trace diagrams showing the sequence of interactions between the user and the server. This present the dynamic features of a web application when a different input leads to different page. Also at this phase we include assertions to insure the correct flow of our model. The second phase, is to use the verifier, SPIN uses the Liner Temporal Logic (LTL) which is suitable for our case, as it used for for reasoning about dynamic scenarios. We also show how use the predefined trace is used to verify the sequences of messages exchanged between the user and the server.

In SPIN The models are written in Promela and demonstrate the system components as processes. As a model checker SPIN has been successfully applied in simulating and formally verifying security protocols and web applications. Since SPIN does not support the modelling of time, we extend our model with discrete time macros. We add a separate process that models a ticking clock.

The second tool that we use is UPPAAL, which is used for automated verification of safety and bounded liveness properties of real-time systems. The models in UPPAAL are designed as networks of timed automata which are extended with data variables, structured data types, user defined functions, and channel synchronisation [UU11]. In contrast to SPIN, UPPAAL uses real time modelling.

We model a secure and a compromised user’s interaction with a web application in both tools. we introduce an attacker which interferes with the communication at various stages (weak points) in the message exchange, for example by intercepting the message exchange or altering the message. We use a time stamp to model a sequence of actions. If an attacker is active and attempts to impersonate the client, the sequence of time-stamped actions is different than the intended one. For example, during a man in the middle attack, the compromised model is more complex than the secure one and the sequences of actions is different than the intended one. We achieve comparable results for the analysis of our models in both tools with respect to speed and expressiveness.

3 Related Work

There are different approaches for analysing web applications, for example [DDMP03, DHM01, CFB00] which use purposefully developed tools. In contrast, we do not rely on specially developed tools, but use standard model checkers. In [HPS04, HITT11, ZMSZ12]they propose methods to model web applications as finite state automata, we model the complete web application as pages and transitions with the Server along with the timing concept. The SPIN model checker have been used in many successful software verification [KCCW03, MS02, RA00]. Our model approach of adding the timing and using the expressive language of SPIN can improve in creating secure and accurate web applications.
Bibliography


Revisiting modelling and analysing railway control systems in CSP

Mike Smith¹, Hoang Nga Nguyen², Markus Roggenbach¹

¹ Swansea University, Wales, UK ² Nottingham University, England, UK

Abstract: Various authors have modelled and analysed railway control systems in CSP. While first results published often appeared promising, astonishingly in most cases these were dismissed or not followed up by the authors. In this work we report on past attempts and introduce our fresh approach in CSP which appears to be more than competitive to endeavours in other modelling languages.

Keywords: Railway Control Systems, CSP, Modelling, Model-Checking.

1 Introduction

Formal verification of railway control software has been identified as one of the “Grand Challenges” of Computer Science [Jac04]. The automated verification of interlocking systems (on the design level) is an active research topic, investigated by several research groups. Here, we focus on approaches based on the process algebra CSP (Communicating Sequential Processes), see, e.g., [Ros10]. CSP provides a well established formalism to describe concurrent systems. CSP has various tool support, including the CSP Refinement Checker FDR3 [TGR14]. FDR3 analyses processes written in CSPm, which combines the operators of Hoare’s CSP with a functional programming language.

Historically, there have been three major attempts to model and verify interlockings in CSP. In 1997, Simpson et al. [SWD97] check an interlocking model for safety invariants, involving a number of abstractions to make model checking feasible. They conclude with the hope that their approach can be fully automated. However, to the best of our knowledge, their ideas have not been followed any further. In 2002, Winter describes how to model railway control systems in CSP for proving their safety [Win02]. She formalises the track plan as a graph in a functional language for data description. State changing entities of a railway system such as signals, points, and trains become CSP processes. However, in their follow up publication [WR03] Winter et al. dismiss their CSP modelling for a, in their view, more promising approach based on ASMs and supported by the NuSMV model checker. In 2012, Isobe et al. [IMNR12], work from our own group, followed and improved on Winter’s CSP modelling. However, as even simple examples could barely be verified due to state space explosion (the double junction considered in the table below was already too large and could not be analysed with model checking), this line of work was dismissed and replaced by successfully modelling and verifying interlockings in CSP|B, see, e.g., [JMN+14].

Based on the experience gained through various modelling exercises in different specification languages, including CASL, CSP|B, Event-B, and Real Time Maude, we missed the modularity (especially when it came to proofs) provided by CSP and felt that we gave up too early. Therefore, we decided to revisit modelling and analysing railway control systems in CSP.
To this end we utilise the methodology for railway modelling established in previous exercises. This involves the following key components: using a Domain Specific Language (DSL) in order to describe the verification problem at hand; analysing the components and information flow involved; developing a systematic translation from the DSL description into our specification language CSP; and systematically validating the model by establishing meaningful counter-examples from error-injection. We illustrate the translation table for the event “move”, which describes how a train moves from its current position “cp” to its new position “np”:

<table>
<thead>
<tr>
<th>Event</th>
<th>Pre-condition</th>
<th>Pre-condition encoding</th>
<th>Post-condition</th>
<th>Post-condition encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>move.cp.np</td>
<td>No red signal on cp</td>
<td>Enforced by Signal and Train synchronisation</td>
<td>Train position is np</td>
<td>Train state is updated to np</td>
</tr>
<tr>
<td></td>
<td>If cp is a point p, np is given by p</td>
<td>Enforced by Point and Train synchronisation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Our full translation table is concise enough to fit on one whiteboard.

The resulting verification speed was astonishing to us. To establish the property that no two trains collide on rail yards takes less than a second with a moderately sized state space. For comparison, we add the verification times from our work on CSP||B as published in [MNR+12]:

| Rail Yard    | 2015 CSP modelling | 2012 CSP||B modelling |
|--------------|--------------------|----------------|
|              | Time Taken  States Checked | Time Taken  States Checked |
| Station      | 0.11s    5156       | 21.34s  2441       |
| Single Junction | 0.17s   15,336    | 82.75s  8645       |
| Double Junction | 0.72s  984,720   | 4056.66s 240655    |

Several factors contribute to this speed up: different specification languages (CSP and CSP||B); different model checkers (FDR3 for CSP and ProB for CSP||B); a more refined modelling approach (CSP modelling builds upon experience from CSP||B modelling).

Our 2011 CSP modelling of the Double Junction cannot be verified either with FDR2 or FDR3; whereas the 2015 CSP modelling checks all three examples in less than a second. Furthermore, checking our 2015 CSP models with FDR2 also needs only verification time in the seconds. FDR2 dates back to the 1990s, ProB was built in the 2000s – in this sense we consider them to have comparable model checking technology. Therefore, we interpret the achieved speed-up mostly as an improvement on the modelling side. A better understanding of the railway domain allowed us to minimise the number of events and processes, and also to model the point behaviour more abstractly.

Further experiments in which we expanded our modelling to include timing constraints in the process algebra Timed-CSP lead to similar results for capacity analysis. While we were limited to single lines in the Double Junction example in [IMNR12], with our new approach, we are able to analyse the Double Junction as a whole.

---

1 All verification results are on a PC running Xubuntu 14.04.2 with i7 4790 @3.60Ghz and 32GB RAM.
Acknowledgement  The authors would like to thank Siemens Rail Automation, Chippenham, UK, for funding the first author’s MRes studies. Further thanks go to Erwin Catesbeiana (Jr) pointing out inconsistencies in CSP synchronisation sets.

Bibliography


