Imperial College London
Department of Computing

Scalable Techniques for Analysing and Testing Asynchronous Software Systems

Pantazis Deligiannis

April 2017

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

This thesis and the work it presents are my own except where otherwise acknowledged.

Pantazis Deligiannis

Abstract

This thesis is about *scalable* analysis and testing techniques for *asynchronous* programs. Due to their highly-concurrent nature, the number of states that such programs can reach grows exponentially (in the worst case) with program size, leading to *state-space explosion*. For this reason, searching the state-space of real-world asynchronous programs to find bugs is computationally expensive, and thus it is important to develop analysis and testing techniques that can scale well. However, techniques typically scale by sacrificing precision. Detecting only true bugs — versus reporting a large number of false alarms — is crucial to increasing the productivity of developers. Alas, finding the fine balance between precision and scalability is a challenging problem. Motivated by this challenge, we focus on developing tools and techniques for analysing and testing two different kinds of real-world asynchronous programs: device drivers and distributed systems. In this thesis we make the following three original contributions:

- A novel symbolic lockset analysis that can detect all potential data races in a device driver. Our analysis avoids reasoning about thread interleavings and thus scales well. Exploiting the race-freedom guarantees provided by our analysis, we achieve a sound partial-order reduction that significantly accelerates CORRAL, an industrial-strength bug-finder for concurrent programs.

- The design and implementation of *P#*, a new programming language for building, analysing and testing asynchronous programs. The novelty of P# is that it is comes with a static data race analysis and controlled testing infrastructure, which makes a strong move towards building highly-reliable asynchronous programs.

- A new approach for partially-modelling and testing production-scale asynchronous distributed systems using P#. We report our experience of applying P# on Azure Storage vNext, a cloud storage system developed inside Microsoft, and show how P# managed to detect a subtle liveness bug that had remained unresolved even after months of troubleshooting using conventional testing techniques.

Acknowledgements

I am offering my utmost gratitude to my advisor, Alastair F. Donaldson, who supported me throughout my thesis with his guidance and knowledge. I could not possibly wish for a better or friendlier PhD advisor than Ally. During my studies, Ally gave me the freedom to work on my own ideas, and connected me with researchers who (together with Ally) played an instrumental role in shaping this thesis. For example, if it was not for Ally, I would have probably never met Akash Lal, do an internship with him in Bangalore and kickstart the P# project (which ended up being a large part of my thesis). Ally, thank you for offering me the opportunity to do a PhD with you in the Multicore Programming group, it was an incredible experience!

I would also like to thank Akash Lal and Shaz Qadeer, my Microsoft Research mentors. Working with Akash and Shaz during my internships in India and USA (and also remotely after returning to Imperial) was so rewarding and fun. I still remember brainstorming with them for hours (either on the whiteboard or via Skype) trying to design the next big feature in P#. Akash and Shaz not only taught me how to do research in an industrial setting, but also how to believe in an idea even if it sounds a bit crazy. I really appreciate both of them for their time, mentorship and expertise. Akash and Shaz, without you (and Ally), this thesis would not be possible, so thank you.

I am really grateful to my academic colleagues and co-authors, Jeroen Ketema, Zvonimir Rakamarić and Paul Thomson, for contributing to this thesis. I deeply thank Jeroen, not only for working tirelessly towards our PLDI paper, but also for being a good mentor and friend during my studies. Zvonimir invited me to Utah for a four week research visit, which was an amazing experience and helped me grow as a researcher. I thank Paul for his friendship and offering useful insights on how to improve testing with P#. I would also like to thank Cheng Huang, for the close collaboration during my summer 2015 internship in MSR Redmond, which formed the basis for the fifth chapter of this thesis.

I thank the remaining members (old and new) of the Multicore Programming group at Imperial for their friendship and fun conversations: Ethel Bardsley, Adam Betts, Nathan Chong, Andrei Lascu, Christopher Lidbury, Daniel Liew, Tyler Sorensen and John Wickerson. Special thanks to Andrei for being a good friend, and for sharing some great moments

in India. I would also like to thank my friends and former collaborators in Scotland: Evgenij Belikov, Hans-Wolfgang Loidl and Prabhat Totoo. Sharing an office with Evgenij was so much fun (and I still miss our pub nights). I deeply thank Hans for being a good mentor during my first years in the UK, and for supporting me when I decided to move to London to start a PhD at Imperial. Prabhat, I cannot thank you enough for your support and for being a great friend all these years.

I also want to mention some incredible people that I met during my MSR internships: Nimantha Thushan Baranasuriya, Surya Teja Chavali, Xin Chen, Ankush Desai, Dharmesh Kakadia, Yi Li, Matt McCutchen, Rashmi Mudduluru, Matthias Schlaipfer, Tatsunori Taniai, Deepak Vijaykeerthy and Chien-Chih Wang (among many others). Thanks a lot guys! The internships would not have been as fun without you. I would also like to thank my long-time friends Nick Triantafyllidis, Tasos Andronidis and George Saviolakis for their friendship and all the good times!

I deeply thank my parents who educated me and gave me everything, and my brother for being the best brother ever. Last but not least, I thank my wife Yi-Wen (Julie) Cheng for her never-ending love, support and understanding throughout this thesis. Julie, I am the luckiest person in the world to have you.

Dedicated to my wife, my parents and my brother.

Contents

List of Tables

List of Figures

1 Introduction

1.1 Motivation

This thesis is about the design and implementation of *analysis* and *testing* techniques for helping programmers build *correct* asynchronous software systems. Such systems typically receive an unbounded number of requests from their environment, which they have to handle in a timely manner using a predefined set of action handlers [Pro02, Dah09, Mic16b]. To increase responsiveness and performance, asynchronous programs execute their action handlers concurrently. However, if these handlers are not coordinated properly, their interleaving can cause subtle, but serious, bugs [Gra86, MQB$^+$08]. Even if a bug manifests only in a very rare execution path, that bug might have catastrophic consequences if it is not found and fixed before a system goes into production [Ama12, Tre14].

Due to their highly-concurrent nature, the number of states that asynchronous systems can reach grows exponentially, in the worst case, with program size; this is known as the *state-space explosion problem* [God96, ABH$^+$97, Val98]. For this reason, exhaustively searching the state-space of real-world asynchronous systems to find bugs is computationally infeasible, both in terms of time and memory. Thus, the need for developing analysis and testing techniques that are *scalable* becomes apparent.

Over the past decade, multiple concurrency analysis tools have been developed that use techniques such as over-approximation and data-flow analysis to scale to real-world systems (e.g. [BBC$^+$06, PFH06, VJL07, NKA11]). However, these analysers sacrifice precision in order to scale, which results in a significant amount of false bug reports (also known as false positives). We believe that sacrificing precision can hurt the productivity of programmers, as they will have to waste substantial development time to manually filter a long list of false positives [BBC$^+$10]. Having this in mind, we focus on designing techniques that yield a low false positive ratio.

Other state-of-the-art concurrency analysers achieve scalability by exploring only a subset (chosen nondeterministically or systematically) of all possible executions (e.g. [EFG$^+$03, MQB$^+$08, BKMN10, EQR11, DGJ$^+$13]), or by enforcing a bound on the number of explored context-switches (e.g. [QW04, LR08, LQL12]). Although bounded exploration techniques have the advantage of reporting only true bugs, they can potentially also miss bugs. However, prior studies argue that most concurrency bugs manifest in executions with a small number of context-switches [QW04, MQ07a, TDB16]; this increases our confidence in using bounded exploration techniques in this work.

In this thesis, we aspire to create tools and techniques that are, foremost, useful to programmers. However, finding the fine balance between precision, scalability and practicality is a challenging problem. Motivated by this challenge, we focus on developing tools and techniques for analysing and testing two different kinds of real-world asynchronous programs: (i) device drivers, which are system-level programs responsible for the interface between an operating system and the hardware; and (ii) distributed applications and systems, which are high-level programs typically deployed on a computer cluster.

After discussing background and related work in Chapter 2, we present the core thesis contributions in Chapters 3, 4 and 5 (summarised in §1.2), and then conclude by discussing open problems and interesting future directions in Chapter 6.

1.2 Thesis Contributions

The original contributions of this thesis are discussed in the following three chapters:

- In Chapter 3, we discuss Whoop, a novel *symbolic lockset analysis* that can detect all *potential* data races in a device driver. Our analysis scales well, as it avoids reasoning about thread interleavings, but can also report false races. To increase the precision of Whoop, we invoke Corral [LQL12], an industrial-strength bug-finder. Alas, the scalability of Corral suffers in the presence of concurrency. Exploiting race-freedom guarantees provided by Whoop, we achieve a sound partial-order reduction that can significantly accelerate Corral, without sacrificing precision. An experimental evaluation of Whoop and Corral on 16 Linux drivers shows that our combined technique scales well. We also used Whoop to analyse 1016 concurrent C programs from the 2016 Software Verification Competition (SVCOMP'16), showing that our verification approach is not specific to device drivers.

- In Chapter 4, we present the design and implementation of P#, a new asynchronous event-driven programming language *co-designed* with a static data race analysis and controlled testing infrastructure. The goal of P# is to unify language design, analysis and testing under a common framework for developing highly-reliable asynchronous software systems. First, we formulate a static ownership-based analysis for proving race-freedom in programs written in P#. Next, we show how we can leverage the P#

language and static analysis to develop an efficient controlled testing infrastructure. Finally, we experimentally evaluate P# against 14 distributed algorithms, including widely-used protocols such as ChainReplication [vRS04] and Raft [OO14], showing that the P# analysis and testing techniques improve on previous work in terms of scalability, precision and bug-finding ability.

- In Chapter 5, we discuss how P# can be used to model, specify and test real-world distributed systems. We focus on a particular case study of applying P# to Azure Storage vNext [DMT$^+$16], a distributed storage system developed by Microsoft, and show how P# managed to detect a subtle, but serious, liveness bug that had remained unresolved even after months of troubleshooting using conventional testing techniques.

1.3 Publications

Most of the material discussed in this thesis has been originally published by the author in three co-authored conference papers and one co-authored workshop paper.

- The Whoop symbolic lockset analysis technique, discussed in Chapter 3, is based on material originally published in the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15) [DDR15].[1] An early sketch of Whoop was originally published in the 2014 Imperial College Computing Student Workshop (ICCSW'14) [DD14].[2]
- The P# asynchronous programming language, presented in Chapter 4, is based on material originally published in the 36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15) [DDK$^+$15].[3]
- The approach for testing distributed systems using P#, discussed in Chapter 5, was originally published in the 14th USENIX Conference on File and Storage Technologies (FAST'16) [DMT$^+$16].[4]

[1]`http://dx.doi.org/10.1109/ASE.2015.30`
[2]`http://dx.doi.org/10.4230/OASIcs.ICCSW.2014.36`
[3]`http://dx.doi.org/10.1145/2737924.2737996`
[4]`https://www.usenix.org/conference/fast16/technical-sessions/presentation/deligiannis`

1.4 Formal Acknowledgements

I gratefully acknowledge and thank Alastair F. Donaldson and Zvonimir Rakamarić for their significant contributions to the Whoop [DDR15] project, which is discussed in Chapter 3. In particular, Alastair played an instrumental role in formalising the symbolic lockset analysis, and gave invaluable insights for improving the technique (e.g. by using watchdog race-checking instrumentation). The partial-order reduction for accelerating Corral was conceived together with Zvonimir, while I was visiting his research group (Software Analysis Research Laboratory, University of Utah) for four weeks during March 2015. Most of the text from our ASE'15 publication [DDR15] transitioned into this thesis. Although the majority of this work was done by me, there are some sentences and paragraphs that were written by my co-authors, Alastair and Zvonimir. I am also grateful to Montgomery Carter for modelling the Linux locks used in the Chapter 3 experiments, and conducting the SVCOMP'16 experiments in Figure 3.12.

I also deeply thank and acknowledge my internship mentors Akash Lal and Shaz Qadeer, for their significant contributions to the P# project [DDK$^+$15, DMT$^+$16], which formed the basis for Chapters 4 and 5 of this thesis. Akash and Shaz played an important role in helping P# transition from a research prototype into a tool used by Microsoft researchers, engineers and interns. More specifically, I would like to gratefully acknowledge and thank the following individuals for their contributions to the P# project:

- Akash Lal, for his significant contributions to designing the P# language and ecosystem of tools. I prototyped P# in close collaboration with Akash during my summer 2014 internship at the Programming Languages and Tools (PLATO) group of Microsoft Research India. Since then, we are having regular meetings to coordinate and push the P# project forward.

- Shaz Qadeer, for playing a key role in adding liveness checking capabilities to P#, and helping improve the general infrastructure, during my summer 2015 internship at the Research in Software Engineering (RiSE) group of Microsoft Research Redmond. Since then, Shaz has been a key advocator of using P# in Microsoft.

- Alastair F. Donaldson and Jeroen Ketema, who played an instrumental role in designing the static data race analysis for P# (discussed in Chapter 4). I am especially thankful to Jeroen for writing the semantics for the P# language and painstakingly formalising the P# static analysis. I gratefully acknowledge that Alastair and Jeroen authored a significant part of the "Language and Semantics" and "Checking for Data

Races" sections in our PLDI'15 publication [DDK+15], which I have largely reused (with some edits) in this thesis.

- Cheng Huang, for embracing P#, using it to test Microsoft Azure Storage vNext (in close collaboration with me, during my summer 2015 Microsoft Research internship), and advocating its use to his colleagues in Azure Storage. I gratefully acknowledge that Cheng authored a significant part of the "Case Study: Azure Storage vNext" section in our FAST'16 publication [DMT+16], which I have largely reused (with some edits) in this thesis.

- Paul Thomson, for his numerous contributions in improving the P# controlled testing infrastructure, sending me bug reports, and having useful discussions on various aspects of the project. Paul used P# to model Azure Service Fabric and test CScale (a distributed system built on top of Azure Service Fabric) during his summer 2015 internship at Microsoft Research India. This contribution is included in our FAST'16 publication [DMT+16]. Although I present results from this work in Chapter 5 for completeness reasons, ownership of this contribution belongs to Paul.

- Matt McCutchen, for using P# during his summer 2015 internship at Microsoft to develop and test the Live Table Migration protocol [DMT+16], and discovering the Live Table Migration bugs presented in Figure 5.2. Although I discuss results from this work in Chapter 5 for completeness reasons, ownership of this contribution belongs to Matt.

- Ankush Desai, for giving us access to programs written in P, so that we can port them to P# and use them for the Chapter 4 experiments, and having useful discussions on how to improve the P# testing infrastructure.

2 Background and Related Work

In this chapter we review analysis and testing techniques for finding bugs in asynchronous software systems. In §2.1 we briefly discuss asynchronous programming and present the main challenges associated to analysis and testing. We then focus on previous work related to our three core chapters: race analysis for asynchronous device drivers (see Chapter 3); and race analysis and controlled concurrency testing for asynchronous distributed systems (see Chapters 4 and 5).

2.1 Asynchronous Programming

Modern computing infrastructure offers multiple computational resources, through multi-core and distributed clusters, and leaves software systems responsible for exploiting concurrency for responsiveness (to minimise latency) and performance (to maximise throughput). To achieve low latency and high throughput, developers often use *asynchronous* programming to split long-running sequential tasks into multiple shorter tasks, and schedule them *concurrently* without blocking [Pro02, Dah09, App16, Mic16b]. This approach differs from *synchronous* programming, where only a single task can execute at a time (all other tasks block until the running task terminates).

Asynchrony can significantly increase responsiveness and performance, since it allows a system to continue performing other tasks (e.g. handling requests from some client), while asynchronous operations (e.g. reading from disk) execute concurrently in the background; when an asynchronous task terminates, it notifies its caller thread using a callback. Alas, asynchrony is at odds with correctness as it makes testing and analysis much harder than in the sequential case: if asynchronous tasks are not coordinated properly, their interleaving can be a source of subtle, but serious, bugs [Gra86, MQB$^+$08].

2.1.1 Programming Models, Languages and Frameworks

Many programming models, languages and frameworks have been proposed to help address coordination issues in asynchronous systems. Notable programming models include *process calculi*, which is a family of approaches (e.g. CSP [Hoa78] and CCS [Mil80]) for formally modelling and verifying concurrent systems, and *actors* [Agh86], which are asynchronously executing processes that communicate with each other by exchanging messages. Actors are available in languages such as Erlang [VWW96] and Scala [OSV08], and in frameworks such as Orleans [BBG+14] and Akka [Typ16]. Building large-scale, complex asynchronous distributed systems using actors has found a lot of success in industry; recent examples include the cloud services for the Halo 4 game [BBG+14], Azure Service Fabric [Mic16c], and SpatialOS [Imp16].

Actor-based languages and frameworks have mostly focused on simplifying the challenging task of programming asynchronous systems by moving towards a declarative paradigm, but cannot guarantee the absence of bugs (e.g. unexpected failures in a distributed system, data races due to the exchange of heap references, as well as other asynchrony-related race conditions). To improve the reliability of asynchronous systems, researchers have designed and implemented languages such as Pony [CDBM15] and P [DGJ+13].

Pony is an actor language for writing data race free asynchronous programs. To achieve this, Pony introduces a *capabilities-secure* type system that indicates which operations are allowed on references to the same heap object. This essentially allows a Pony program to safely share heap references between actors without data races (thus avoiding deep-copying or marshalling, which can severely hurt application performance [KSA09]). Alas, the type system of Pony is arguably complicated to use, and the language does not provide any guarantees regarding functional (i.e. application-specific) bugs, such as safety and liveness property violations.

P is a domain-specific language for modelling and specifying asynchronous event-driven protocols. The language provides first-class support for modelling asynchrony, specifying safety and liveness properties, and checking for property violations. To find protocol bugs, the P compiler can generate a model of a P program and check it using the Zing [AQR+04] model checker (see §2.2.3). The P compiler can also generate executable C [KRE88] code. P was used in Microsoft to implement the core of the USB device driver stack that ships with Windows 8. Using the P tools, the developers were able to find and fix hundreds of protocol bugs [DGJ+13].

P provides a simple surface syntax for writing actors, and only supports operations over scalar values, as well as (nested) tuples, lists, and maps of scalar values. To get around its limited language features, P exposes a foreign-function interface for encoding functionality that cannot be implemented in P. However, *only* program components written in P, *not* the foreign code, can be checked using Zing; during model checking, the P compiler substitutes the foreign code with models provided by the user.

To tackle the limitations of P, we designed and implemented P# [DDK+15], an actor-based .NET [Mic16e] language with embedded static analysis and testing capabilities. We give a detailed discussion of P# in Chapter 4 and present an industrial case study of using our language and tools in Chapter 5.

2.1.2 Challenges in Analysing and Testing Asynchronous Systems

Reasoning about the correctness of asynchronous systems is notoriously difficult, because the asynchronous programming model combines concurrency with *reactivity*. Concurrency is responsible for the nondeterministic scheduling of asynchronously executing operations; this leads to state-space explosion [God96, ABH^{+}97, Val98], since the number of execution paths increases exponentially, in the worst case, as the number of scheduling points also increases. Reactivity amplifies this challenge, as it is responsible for the interaction of an asynchronous system with an external nondeterministic environment; this can dramatically increase the state-space. Dealing with concurrency and reactivity requires techniques that are *scalable*. We consider a technique as scalable if it can analyse a program of interest in a relatively short amount of time and memory, despite a large associated state-space.

Scalability on its own, however, does not suffice. Analysis and testing techniques should be also *precise*, i.e. they should be able to identify bugs without reporting a large amount of false alarms. This is important as it can save developers a significant amount of time and effort from manually searching for real bugs among many false ones. Precision is directly related to *soundness* and *completeness*: a sound technique never misses true bugs, whereas a complete technique never reports false bugs. However, a technique that is both sound and complete cannot exist due to Rice's theorem [Ric53], which states that all non-trivial program properties are undecidable. In practice, different techniques try to reach different levels of precision. For example, dynamic analysers typically report only real bugs at the cost of soundness, whereas static verifiers are more conservative and aim for soundness at the cost of completeness.

Alas, achieving scalability is at odds with achieving precision; this is because increased precision comes at the cost of increased time and space complexity. Therefore, designing practical analysis or testing tools requires thoughtful trade-offs between scalability and precision at the cost of potentially missing bugs or reporting false ones [FLL^{+}02]. In this work, we focus on developing practical techniques that can achieve scalability without sacrificing too much precision. We achieve this by exploiting domain-specific knowledge for device drivers and reusing industrial-strength verification tools (see Chapter 3), and by implementing a new asynchronous programming language with embedded analysis and testing techniques (see Chapters 4 and 5).

2.2 Reasoning about Program Correctness

We now discuss important techniques for reasoning about program correctness. Although the focus of this thesis is on finding concurrency-related bugs in asynchronous systems, we first discuss what software reliability is, and give an overview of foundational techniques for sequential program analysis, and then proceed to §2.3, where we discuss state-of-the-art techniques for checking concurrency-related properties.

2.2.1 Software Reliability

The standard definition of *reliability* is "the ability of a system or component to perform its required functions under stated conditions for a specified period of time" [ISO10]. There are many factors that could adversely affect the ability of a system to function as expected (such as unpredictable hardware faults or even natural disasters), however, most software failures are caused by programming errors [Gra90, KKI99], which introduce unintentional behaviour during program execution. Programming errors, which are commonly known as *bugs*, pose a serious risk to the global economy, security and public health [Off92, LT93, Age96, Tas02]. Ensuring the absence of bugs is a prerequisite for building software systems that execute reliably and according to their specifications.

2.2.2 Foundations of Program Analysis

Manual testing is one of the most commonly used techniques for finding bugs. However, manual testing has many limitations [DRP99]: it can be very time consuming; prone to human errors; and can offer no guarantees that a program is free from defects. To address these limitations, researchers have been actively developing program analysis tools and techniques that can *automatically* detect bugs or verify their absence. Apart from software reliability, program analysis has also multiple applications in areas such as compiler optimisation [LA04], software security [LL05] and reverse engineering [CC10].

An important class of program analysis techniques is *formal verification*, which aims to prove (or disprove) the absence of bugs with respect to a given set of program specifications and properties by using mathematical-based methods. Floyd [Flo67] and Hoare [Hoa69] laid down the foundations of software verification in the 1960s by defining the *Floyd-Hoare logic*, a set of formal rules for reasoning about the execution of programs. These rules are based on the concept of the *Hoare triple*, which has the form $\{x\}\ P\ \{y\}$. The Hoare triple is interpreted as follows: assuming that a program P starts executing in a state where the precondition x holds, if P terminates then the postcondition y will be true afterwards.

Programs can be annotated (manually or automatically) with Hoare triples to construct program proofs that can be checked to establish program correctness.

Building on the Floyd-Hoare logic, Dijkstra [Dij75] introduced the *predicate transformer semantics* as a mechanism for generating program proofs. Predicate transformers translate a program annotated with pre-/post-conditions and invariants into a logical formula known as a *verification condition*, the validity of which implies program correctness. A verification condition can be discharged to an automated theorem prover, such as a *satisfiability modulo theories* (SMT) solver [BSST09, DMB08, BCD$^+$11], which can generate a *counterexample* if the verification condition does not hold. A counterexample is essentially an example that disproves a false logical formula.

In 1977, Cousot and Cousot introduced abstract interpretation [CC77], a technique for reasoning about program correctness by checking an *approximation* rather than checking directly the original source code. There are two ways to create a program approximation: by *over-approximating* (i.e. adding program behaviours), which can introduce false positives (i.e. produce false bug reports); and by *under-approximating* (i.e. removing program behaviours), which can introduce false negatives (i.e. miss real bugs). The motivation behind abstract interpretation is that the verification of real-world programs, which usually contain a large amount of procedure calls and loops, does not easily scale. In contrast, analysing an approximation is often simpler and faster. However, as discussed in §2.1.2, there is a trade-off between precision and scalability. Although loss of precision is unavoidable when approximating a program, significant effort should be put into producing sound approximations to avoid giving a false impression of correctness.

All these discoveries acted as the catalyst for achieving fully automatic reasoning about program correctness,[1] and enabled the creation of industrial-strength static analysers, such as the Boogie [BCD$^+$06] verification engine, CBMC [CKL04], CodeSonar [Gra13], Corral [LQL12], Coverity [BBC$^+$10] and SLAM [BMMR01].

2.2.3 Model Checking

Model checking [CE82, QS82] is a well-established verification technique used to determine whether the *model* of a system meets a set of formal specifications. A model is a directed graph, where nodes represent states, and edges represent transitions between states. The specifications, also known as properties, are expressed in temporal logic [Pnu77], which is a logic for reasoning about state-transitions over time. Model checking tools find bugs by exhaustively examining the reachable states of a model. If a model checker reaches a

[1]In practice, fully automatic program verification is never fully achieved; all program analysis tools have trade-offs between precision, scalability and practicality.

state that violates the property of interest, it generates a counterexample in the form of a sequence of state-transitions that leads to the bug. Counterexample generation is a key strength of model checking as it helps developers trace and reproduce bugs. In this section we will focus on software model checking.

There are two approaches to representing states in model checking: *explicit* and *symbolic*. In explicit-state model checking, a model is represented as a state-transition graph, where each node is a snapshot of the global program state (e.g. the contents of the stack and the heap). This directed graph can be explored using depth-first search, breadth-first search or other traversal algorithms. Explicit representation, however, does not scale due to state-space explosion and memory constraints. In contrast, symbolic model checking [BCM+92] represents set of states as boolean expressions and explores them using *binary decision diagrams* (BDDs) [Bry86]. This representation proved to be very efficient and allowed the verification of models with more than 10^{20} states [BCM+92].

An important problem when constructing a BDD, is how to choose an optimal variable ordering (i.e. the order in which variables appear when following a path from the BDD root node) in order to minimise the size of the resulting BDD. This problem is widely known to be NP-complete [BW96]. Finding an optimal variable ordering in large programs can be very time consuming and often requires manual intervention. To tackle this limitation, researchers proposed the use of *propositional decision procedures* (SAT) [DP60], which also operate on boolean expressions, but do not use canonical forms and thus are more space efficient than BDDs [BCCZ99].

The use of SAT solving in symbolic model checking, instead of BDDs, became the basis for a technique known as *bounded model checking* (BMC) [BCCZ99]. The main idea behind BMC is to search for a counterexample only in states reachable within a *bounded* number of state-transitions. If no such counterexample is found, the exploration bound increases until either a predefined upper bound is reached, a counterexample is found, or the problem becomes intractable. An important limitation of BMC is that it is typically unable to prove the absence of bugs, since a counterexample might be longer than the exploration bound. However, BMC has been widely successful (especially in the semiconductor industry) as it is able to quickly discover many bugs using a small exploration bound [ADK+05].

Although hardware has a finite amount of states, which guarantees that a model checker will eventually terminate exploration, software typically has an infinite amount of potential execution paths, due to loops, recursion and concurrency (to name a few reasons). State-space explosion is the main limitation of software model checking; trying to fully explore an infinite state-space is impossible. To tackle this issue, model checking techniques are often combined with abstraction techniques, which allow a model checker to reason about program correctness using a finite state-space abstraction of the original program.

One such abstraction technique, is *predicate abstraction* [GS97], which over-approximates a program with respect to a set of predicates. The result is an abstract program, which only contains the predicates represented as boolean variables. Due to over-approximation, if the abstraction is correct, then the original program is also correct. However, a counterexample in the abstraction might not correspond to a real bug, since the over-approximation might introduce new program behaviours. If the counterexample is indeed spurious, an iterative technique known as *counterexample-guided abstraction refinement* (CEGAR) [CGJ$^+$00] can be repeatedly applied to refine the abstraction until the false bug is eliminated.

Software model checking formed the basis of many successful software verification tools: Spin [Hol97], which has been used to verify distributed system specifications written in the PROMELA [Hol91] verification language; BLAST [HJMS02], which introduced the concept of lazy abstraction, a technique for refining parts of a program's abstraction on demand to achieve the necessary precision required for verification; SATABS [CKSY04], which uses SAT solving during the predicate abstraction and refinement process; SLAM [BMMR01], which successfully found bugs in Windows device drivers for over a decade [BLR11] using predicate abstraction and CEGAR-based refinement; and CORRAL [LQL12], a bounded model checker for (concurrent) Boogie [DL05] programs.

The SLAM verification engine automatically checks whether a driver respects a set of temporal safety properties, which specify that something bad will *never* happen (e.g. the driver will not dereference a null pointer, or acquire a lock that is already acquired). Until recently [LQ14], SLAM was the core verification engine in Static Driver Verifier [BBC$^+$06] (SDV), a static analyser for Windows drivers. SDV checks the behaviour of a driver against a set of Windows driver API usage rules. Each time a rule is violated, SLAM generates a counterexample for reproducing the bug. Although SLAM can find many serious sequential bugs in device drivers, it cannot detect concurrency bugs, since it has no built-in notion of threads. Attempts to overcome this limitation include *sequentialisation* techniques, such as the KISS [QW04] algorithm (see §2.3.5).

CORRAL, which recently replaced SLAM as the core verification engine in SDV [LQ14], uses techniques such as variable abstraction and stratified inlining to prove bounded (in terms of the number of loop iterations and recursion depth) sequential reachability of a bug in a goal-directed, lazy fashion. This helps CORRAL to postpone state-space explosion when analysing the (translated to Boogie) source code of large device drivers. To handle concurrency, CORRAL reduces a concurrent Boogie program to a sequential Boogie program that under-approximates the behaviours of the original concurrent Boogie program (see §2.3.5). We further discuss CORRAL in §3.5, where we show how we used the tool as part of our Whoop symbolic lockset analysis toolchain.

2.2.4 Symbolic Execution

Symbolic execution [Kin76, Cla76] is a program analysis technique for efficiently exploring execution paths to find bugs. The main idea behind this technique is to treat the program input as *symbolic* (i.e. as a set of unknown values) rather than concrete (i.e. as a set of real values). While a program executes, a symbolic execution tool gathers *constraints* on this symbolic input. Each time a branch is reached, the tool invokes an underlying constraint solver to decide which path to follow (multiple paths might be feasible). If a bug is found (e.g. an assertion violation or uncaught exception), the gathered constraints are solved to generate a test case, which can be used to reproduce the same bug, assuming that the program is deterministic. An advantage of symbolic execution is that it actually executes the program-under-test, and thus cannot report false positives.

KLEE [CDE08], a popular open-source symbolic execution tool, showcased the potential of symbolic execution by discovering three serious bugs in GNU Coreutils, one of the most heavily tested application suites in UNIX. Other notable symbolic execution tools include DART [GKS05], CUTE [SMA05], JPF-SE [APV07], Pex [TDH08], SAGE [GLM$^+$08] and Symbooglix [LCD16]. Symbolic execution tools have proven to be effective in finding bugs in real-world programs; however, they can rarely offer any guarantees of correctness, since exhaustively exploring all execution paths in non-trivial programs is infeasible.

Due to challenges associated with handling concurrency, reactivity, and other sources of nondeterminism, symbolic execution cannot be easily applied to asynchronous programs. Despite these difficulties, prior work demonstrated that symbolic execution can successfully detect bugs in asynchronous distributed systems [SA06, SLAW08, BUZC11]. In the case of device drivers, tools that combine symbolic execution with environment emulation have been promising. Examples of such tools are DDT [KCC10] and SymDrive [RKS12], which are both based on KLEE and the QEMU [Bel05] machine emulator.

DDT is a symbolic execution tool for testing closed-source binary Windows drivers. The tool consists of the following components: a KLEE-based symbolic execution engine; a set of dynamic bug checkers; and a QEMU-based emulator, which generates CPU instructions and hardware interrupts. Each time a driver calls a kernel function, DDT injects a *symbolic interrupt* before and after the call. Whenever QEMU encounters such a symbolic interrupt, it blocks the currently running driver process and invokes the interrupt handler. The bug checkers are then used to detect bugs such as memory corruption, resource leaks and race conditions. A limitation of this approach, is that DDT can detect concurrency bugs only related to interrupt handling. Another limitation is that DDT requires manually written annotations to work, which limits its applicability in practice.

SymDrive is a symbolic execution tool for finding bugs in Linux and FreeBSD drivers. The tool is based on the S^2E [CKC11] virtualisation platform, which uses modified versions of KLEE and QEMU. SymDrive consists of the following components: a static analyser for identifying the entry points of a driver; a framework for writing checkers that can validate the driver behaviour; and an execution tracer for logging execution paths (used to compare different revisions of a driver). SymDrive can find many types of bugs, including misuse of kernel APIs, memory leaks, and blocking while holding a spinlock. An important limitation of SymDrive is that it cannot detect data races between threads, and thus can miss a lot of serious concurrency issues.

2.3 Finding Bugs in Asynchronous Programs

We proceed by discussing techniques for finding concurrency bugs in asynchronous programs. Without attempting to provide an exhaustive survey, we will focus on the following topics: reasoning about the order of events in an asynchronous system (§2.3.1); analysing programs that use locks (§2.3.2); finding concurrency bugs by controlling the underlying thread scheduler (§2.3.3); tackling the state-space explosion problem with partial-order reduction (§2.3.4); and finding bugs in concurrent programs using sequential analysis techniques (§2.3.5).

2.3.1 Ordering of Events in an Asynchronous System

A fundamental concept relevant to our work is the *happens-before* relation [Lam78], which denotes a partial-order between all events occurring during the execution of asynchronous software systems. The happens-before relation can be defined as follows:

Definition 2.1. *Let E_1 and E_2 be two events occurring during the execution of a concurrent asynchronous program. If E_1 and E_2 are in the same thread and E_1 occurs before E_2, or if E_1 and E_2 are in different threads and there is a synchronisation mechanism that prevents a thread schedule in which E_2 occurs before E_1, then $E_1 \rightarrow E_2$, where the symbol $\rightarrow$ denotes that E_1 happens-before E_2.*

Note that happens-before is transitive, i.e. for any three events E_1, E_2 and E_3, if $E_1 \rightarrow E_2$ and $E_2 \rightarrow E_3$, then $E_1 \rightarrow E_3$.

If two events E_1 and E_2 occur in different threads, and neither $E_1 \rightarrow E_2$ nor $E_2 \rightarrow E_1$ holds, then E_1 and E_2 *happen-concurrently*, which leads to a *race condition*, a potentially

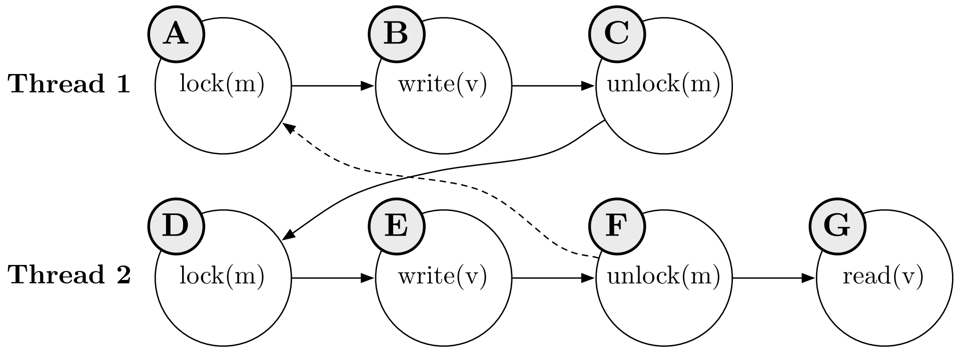

Figure 2.1: Example of a simple data race between statements B and G. The race is missed by a happens-before tool that checks only this specific interleaving. The dashed line denotes an alternative thread interleaving.

undesirable situation where E_1 and E_2 occur in nondeterministic order. A race condition can result into a bug, if the nondeterministic order of E_1 and E_2 affects the correctness of the program. A *data race* (which is a special case of a race condition) occurs if two events E_1 and E_2 happen-concurrently, E_1 and E_2 access the same shared variable, and at least one of the two accesses is a write. In such case, there are thread interleavings in which the execution order of the threads changes, causing E_1 and E_2 to access the same shared variable in a nondeterministic order.

In some languages (e.g. C [ISO11, p. 38]), data races are classed as undefined behaviour, and thus always indicate bugs (although in practice system-level code sometimes exhibits intentional data races as seen in §3.6.1); whereas in some other languages (e.g. Java) data races have well-defined semantics, and thus whether they are indicative of program bugs is considered application-specific.

Figure 2.1 shows a simple example of applying happens-before reasoning on two concurrently running threads. The three statements {A, B, C} in thread 1 execute one after the other, and thus all of them are ordered by the happens-before relation. The same applies for each of the four statements {D, E, F, G} in thread 2. Statement C in thread 1 releases lock m, and subsequently allows D in thread 2 to acquire m. Thus, C $\rightarrow$ D. This means that for this specific thread interleaving {A, B, C} $\rightarrow$ {D, E, F, G}.

Happens-before allows reasoning about the ordering of events in an asynchronous system using *logical clocks* [Lam78] instead of physical clocks, which may be absent or provide inaccurate timing information due to communication delays. Logical clocks are associated with a monotonically increasing numerical value, instead of the real time of the day. Each thread T_i has its own clock C_i, which is a function that assigns a clock value $C_i\langle E\rangle$ to any event E in T_i. If E_A is an event in T_1, and E_B an event in T_2, and $E_A \rightarrow E_B$, then the relation of the clock values of the two events is $C_1\langle E_A\rangle < C_2\langle E_B\rangle$.

A limitation of using the above logical clock system is that it provides only one of many possible event orderings depending on a particular thread interleaving. This information might not suffice to prove that two events are causally related, i.e. knowing that $C_1\langle E_A\rangle < C_2\langle E_B\rangle$ holds, does not imply that $E_A \rightarrow E_B$ holds. Figure 2.1 illustrates this limitation. If a tool checks only the depicted interleaving, it will miss a read-write data race between statements B and G, because B $\rightarrow$ G in this execution path. The race can be found only in an execution where F $\rightarrow$ A (denoted with a dashed line). In such an execution, B and G are not mutually protected by m, and neither B $\rightarrow$ G nor G $\rightarrow$ B holds, which means that B and G happen-concurrently. Due to this limitation, a tool based on happens-before requires a scheduler that produces as many different thread interleavings as possible to increase execution path coverage and find all hidden bugs.

Many dynamic analysers [OC03, PS03, FFY08, MQB+08, FF09] use the happens-before relation to detect concurrency bugs. Modern analysers combine happens-before reasoning with other successful concurrency analysis techniques, such as lockset analysis [SBN+97] and controlled scheduling [God97, MQB+08], to increase scalability and precision.

2.3.2 Lockset Analysis

Lockset analysis is a lightweight race detection method for programs that contain locks. It was proposed in the context of Eraser [SBN+97], a dynamic data race detector. The main idea is to track the set of locks (*lockset*) that are *consistently* used to protect a shared memory location during execution. An empty lockset suggests that a memory location *may* be accessed simultaneously by two or more threads, due to being inconsistently protected by locks (or accessed without protection). Consequently, the analysis reports a *potential* data race on that memory location.

Lockset analysis for an asynchronous program starts by creating a *current* lockset CLS_T for each thread T of the program, and a lockset LS_s for each shared variable s used in the program. Every CLS_T is initially empty because threads do not hold any locks when they start executing. In addition, every LS_s is initialised to the set of all locks manipulated by the program since initially each access to s is (vacuously) protected by every lock. The program is executed as usual (with threads scheduled according to the OS scheduler), except that instrumentation is added to update locksets as follows. After each *lock* and *unlock* operation by T, CLS_T is updated to reflect the locks currently held by T. When T accesses s, LS_s is updated to the intersection of LS_s with CLS_T, which removes any locks

	Program	$\mathbf{CLS_{T1}}$	$\mathbf{CLS_{T2}}$		$\mathbf{LS_A}$
Initial		{ }	{ }		{ M, N }
T1	lock (M);	{ M }			{ M, N }
	lock (N);	{ M, N }			{ M, N }
	write (A);	{ M, N }			{ M, N }
	unlock (N);	{ M }		**compute lockset** $\longleftarrow$	{ M, N }
	write (A);	{ M } $\longrightarrow$		**intersection at access points** $\longrightarrow$	{ M }
	unlock (M);	{ }			{ M }
T2	lock (N);		{ N }	**warning: access**	{ M }
	read (A);		{ N }	**to A may not** $\longrightarrow$	{ }
	unlock (N);		{ }	**be protected**	{ }

Figure 2.2: Applying lockset analysis on two concurrently running threads.

that are not common to the two locksets. If LS_s becomes empty as a result, a warning is issued that the access to s may be unprotected.

Figure 2.2 shows an example of applying lockset analysis to a concurrent program consisting of two threads T_1 and T_2, both accessing a shared variable A. Initially, LS_A, which is the lockset for A, contains all possible locks in the program: M and N. During execution of T_1, the thread writes A without holding lock N, and thus N is removed from LS_A. During execution of T_2, the thread reads A without holding M, and thus LS_A becomes empty. As a result, a potential read-write data race is reported because the two threads do not consistently protect A.

In contrast to data race analyses that encode a *happens-before* relation [Lam78] between threads, lockset analysis is lightweight, and thus can scale well. The technique, however, can report false alarms since a violation of the locking discipline does not always correspond to an actual data race (e.g. shared variables are typically initialised without holding a lock). A promising approach is to combine lockset analysis with happens-before reasoning to gain the advantages of both techniques [DS91, PS03, OC03, YRC05, EQT07]; the scalability of the former and the precision of the latter.

An important limitation of dynamic lockset analysis tools, such as Eraser, is that code coverage is restricted by the execution paths that are explored under a given scheduler. To tackle this problem, tools such as Locksmith [PFH06] and RELAY [VJL07] have explored the idea of applying lockset analysis statically, on the actual source code, using dataflow analysis. In this work, we created a novel symbolic lockset analysis that involves abstracting an asynchronous program, generating verification conditions, and then discharging them to an SMT solver (see Chapter 3).

2.3.3 Controlled Concurrency Testing

Traditional testing techniques (e.g. unit and integration testing) are ineffective in finding and reproducing concurrency bugs, due to scheduling nondeterminism. An important class of techniques for detecting such bugs is *controlled concurrency testing* [God97, MQB$^+$08, BKMN10, EQR11, NBMM12, YNPP12, TDB16]. When attached to a concurrent program, controlled concurrency testing tools take control of the thread scheduler and serialise the program execution; the controlled scheduler then decides (based on some scheduling strategy) which thread to execute at each scheduling point, dictating when interleavings are performed. Controlling all possible sources of scheduling nondeterminism allows the scheduler to record all interleavings that led to a bug, facilitating deterministic bug reproduction.[2] The strategies used to control the thread scheduler can be classified as either *systematic* or *randomised*.

Systematic scheduling strategies [God97, MQB$^+$08, EQR11] work by repeatedly executing a concurrent program from start to finish, exploring a *different* schedule in each testing iteration. This can be achieved by treating the schedule-space as a tree, where each node is a scheduling point and its branches are the threads that can be scheduled at this scheduling point. The scheduler traverses this tree using depth-first search; this allows any unexplored scheduling decisions to be efficiently stored in a stack data-structure. In each iteration, the scheduler replays the previous schedule up to the most recent scheduling point, and then chooses the next—previously unchosen—thread to schedule, and continues scheduling threads until the program reaches a terminal state. When all possible schedules have been explored, the testing process terminates. Exhaustively exploring the schedule-space of real-world systems, however, is infeasible (e.g. due to state-space explosion, or due to cycles in the schedule-space). To tackle this problem, systematic testing tools introduce exploration bounds (e.g. a maximum number of allowed interleavings per schedule), at the cost of missing bugs.

Randomised scheduling strategies [BKMN10, NBMM12, TDB16] explore the schedule-space by randomly choosing a thread to schedule at each scheduling point, rather than systematically exploring a schedule tree. In contrast to systematic scheduling, randomised scheduling does not require caching of previously explored schedules and thus has $\mathcal{O}(1)$

[2]In reality, programs might exhibit additional sources of nondeterminism, e.g. an asynchronous distributed system might deal with nondeterminism arising from its environment (timers, clients and node failures). These non-scheduling-related sources of nondeterminism must also be captured by the controlled testing tool to enable deterministic reproduction of bugs, as we discuss in Chapter 5.

space complexity. Subsequently, a randomised scheduler cannot offer any guarantees regarding exhaustive exploration (although this is infeasible in practise, as discussed above). A randomised scheduler might also waste time repeatedly exploring the same schedules. However, in our own experience, random scheduling is effective at finding bugs, and can outperform systematic strategies (see §4.6.2). This confirms findings from previous studies [BKMN10, TDB16].

Controlled concurrency testing has been implemented in a number of research tools, including Verisoft [God97], CHESS [MQB+08], Inspect [YCGK08] and Maple [YNPP12]. In this thesis, we created P#, an asynchronous event-driven programming language with embedded controlled concurrency testing capabilities (see Chapter 4).

2.3.4 Partial-Order Reduction

Partial-order reduction [Pel93, God96] (POR) is a classic technique for tackling state-space explosion. POR exploits the commutativity of asynchronous operations, which can result in the same program state regardless of their execution order, to (significantly) reduce the number of schedules that an analysis or testing tool has to explore. If POR is implemented soundly, then no bugs are missed during schedule exploration.

One of the first implementations of POR is available in the SPIN [Hol97] model checker. Java PathFinder [VHB+03] from NASA combines POR with lockset analysis, as well as other static analysis techniques, to find concurrency errors in avionics systems and spacecraft controllers. Flanagan and Godefroid introduced the concept of *dynamic partial-order reduction* (DPOR) [FG05], which dynamically tracks the interaction between threads to identify additional schedules that should be explored during controlled concurrency testing. Recent studies [MQ07b, HF11, CMM13] investigated the combination of DPOR with schedule bounding, showing promising results. Controlled concurrency testing tools, such as CHESS, avoid exploring the same schedules by caching the happens-before graph, resulting in a similar reduction to POR [MQB+08].

In this thesis, we use POR-based techniques to accelerate race checking in device drivers (see §3.5.2), and to reduce the schedule-space when (systematically) testing asynchronous distributed protocols and systems (see §4.5).

2.3.5 Sequentialisation and Context-Bounded Analysis

A promising approach for tackling state-space explosion, is to reduce a concurrent program into a sequential program that simulates a subset of the behaviours of the original program. Properties of the sequential program can then be checked using a sequential analyser. This technique, which is known as *sequentialisation*, was first introduced by Qadeer and Wu in KISS [QW04],[3] which sequentialises a concurrent Windows driver under a fixed bound of 2 context-switches,[4] and then passes it to the SLAM model checker (see §2.2.3).

Lal and Reps built upon KISS and introduced a technique for sequentialising a concurrent program under *any* given context-bound [LR08]. One of the benefits of the Lal-Reps technique is that it scales linearly with the number of threads in a program, whereas prior context-bounded analysis techniques scaled exponentially [QR05, BFQ07, LTKR08]. Lal and Reps achieved this as follows:

Let T_1 and T_2 be two threads of a concurrent program, and let $T_1; T_2; T_1; \cdots$ denote one of many possible thread schedules under a context-bound. When the context switches from T_1 to T_2, the local state of T_1 is stored. When the second context-switch happens, T_1 must resume execution from its previously stored local state, taking into account any changes in shared memory caused by executing T_2. However, this approach is problematic, because a concurrency analyser must store the local state of each thread in each context-switch point, even if the local state of a thread does not change while another thread executes. This results into an exponential increase in space complexity. Lal and Reps observed that the order that threads are analysed does not matter as long as T_1 *assumes* any effects that T_2 might have on shared memory, apply this assumption while T_1 executes, and then check this assumption after analysing T_2. Subsequently, a thread interleaving $T_1; T_2; T_1$ can be represented as $T_1; T_1; T_2$, eliminating the need to use the local state of T_1 for the analysis of T_2, and thus scaling linearly with the number of threads.

Examples of tools based on the Lal-Reps sequentialisation method are STORM [LQR09] and CORRAL [LQL12]. Both tools were built on top of the Boogie [BCD+06] verification engine, and use verification condition generation and SMT solving to find bugs in device drivers. Although tools using the above sequentialisation never report false bugs, they can miss bugs, as the analysis is based on a context-switch bound.

In this work, we verify data race freedom in device drivers by using a sequentialisation technique inspired by the two-thread reduction that has been used in graphical processing unit (GPU) kernel verification tools such as PUG [LG10] and GPUVerify [BCD+15]. Our sequentialisation reduces a concurrent driver into a sequential driver by analysing only one pair of driver entry points at a time, and over-approximates the shared state to model the effect of additional threads (see §3.3). In contrast to the Lal-Reps approach, our pair-wise analysis is sound, but imprecise: it cannot miss data races, but can report false positives. To increase precision, we combined our analysis with CORRAL, as discussed in §3.5.

[3]KISS is an acronym for "keep it simple and sequential".

[4]Context-switching is the process of passing execution control from one thread to another.

2.4 Summary

We have reviewed state-of-the-art techniques for finding bugs in software systems. We first discussed foundational techniques for (sequential) program analysis, and then focused on asynchronous programs. We presented classic techniques for detecting concurrency-related bugs, including happens-before analysis, lockset analysis, controlled concurrency testing, partial-order reduction and sequentialisation.

We will now proceed with the main contributions of this thesis (see Chapters 3, 4 and 5), and then conclude by discussing open problems and future work (see Chapter 6).

3 Fast and Precise Symbolic Lockset Analysis of Concurrency Bugs

In this chapter we are concerned with the problem of detecting (or verifying the absence of) data races in system-level asynchronous programs. We target device drivers [CRKH05], because bugs in driver source code are the main cause of errors in operating systems [CYC$^+$01, SBL03, RCKH09]. Reasoning about race-freedom in drivers is a challenging activity. This is because drivers are low-level (typically written in C [KRE88]), complex (often consisting of thousands of lines of source code) and highly-asynchronous (many threads can simultaneously invoke a driver to access the available shared resources).

Several techniques have been successfully used to analyse drivers in the past [BBC$^+$06, CKSY04, ECCH00, HNJ$^+$02, CPR06, KCC10, RKS12, LQL12]. However, most of these techniques focus on generic sequential program properties and protocol-level bugs. Linux kernel analyzers, such as sparse [Cor04], coccinelle [PLHM08] and lockdep [Cor06], can find deadlocks in kernel source code, but are unable to detect more sophisticated concurrency bugs, such as data races, as well as other bugs caused by race conditions.

Techniques that aim to detect concurrency bugs in device drivers face significant scalability issues due to the exponentially large state-space of concurrent programs. To tackle this problem, concurrency-bug analysers typically sacrifice precision for scalability, but can report false bugs as a consequence [EA03, PFH06, VJL07]. A promising approach to finding concurrency bugs without losing precision, is to *sequentialise* the originally concurrent program, *bound* the sequentialised program (e.g. by number of thread interleavings), and finally apply existing sequential analysis techniques (see §2.3.5). Although tools based on bounded sequentialisation (e.g. Corral [LQL12]) are typically precise (i.e. only report true bugs), they can be unsound (i.e. can miss true bugs), because they under-approximate the program behaviour. Furthermore, such tools typically do not scale well, as they have to consider exponentially many thread interleavings (even with small bounds).

We present a novel *symbolic lockset analysis* that can detect all *potential* data races in the source code of a device driver. Our analysis scales well, because it avoids reasoning about thread interleavings, but can also report (a large number of) false races. We implemented this analysis in Whoop, a practical tool that can be applied to many different types of Linux drivers. To increase the precision of Whoop, we invoke Corral, an industrial-strength bug-finder. Alas, the scalability of Corral suffers in the presence of concurrency. Exploiting race-freedom guarantees provided by Whoop, we created a sound partial-order reduction that can significantly accelerate Corral, without missing any bugs. Combining Whoop and Corral, we analysed 16 drivers from the Linux 4.0 kernel, achieving 1.5–20$\times$ speedups over standalone Corral. We also used our toolchain to analyse 1016 concurrent C programs from the 2016 Software Verification Competition (SVCOMP'16), showing that our symbolic analysis is not specific to device drivers.

The main contributions of this chapter are:

- A scalable technique, based on symbolic lockset analysis, for automatically verifying the absence of data races in device drivers.
- The design and implementation of Whoop, a practical tool that leverages our symbolic lockset analysis to detect data races in Linux device drivers.
- A sound partial-order reduction that exploits race-freedom guarantees, provided by Whoop, to accelerate Corral, an industrial-strength bug-finder.
- An experimental evaluation of Whoop on 16 Linux device drivers and 1016 concurrent C programs from SVCOMP'16, which shows that our technique can significantly accelerate Corral for arbitrary concurrency bug-finding.

Related Published Work The material presented in this chapter is based on work that was originally published in the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15) [DDR15]. An early sketch of this work was presented in the 2014 Imperial College Computing Student Workshop (ICCSW'14) [DD14].

3.1 Bugs in Device Drivers

Device drivers exist at the thin boundary between hardware and software. They are complex system-level asynchronous programs responsible for providing an interface between an operating system and any hardware devices that are attached to a computer. Device drivers are notoriously hard to develop and even harder to debug [CRKH05]. This has a negative impact on hardware product releases, as time-to-market is commonly dominated by driver development, verification, and validation schedules [Yav12].

Even after a driver has shipped, it typically has many undetected errors: Chou et al. [CYC+01] gathered data from Linux kernel releases spanning a seven year period and found that the relative error-rate in driver source code is up to 10 times higher than in any other part of the kernel, while Swift et al. [SBL03] found that 85% of the system crashes in Windows XP are due to faulty drivers. Regarding *concurrency* bugs, a recent study [RCKH09] established that they account for 19% of the total bugs in Linux drivers, showcasing their significance. The majority of these concurrency bugs were *data races* or *deadlocks* in various configuration functions and hot-plugging handlers.

Concurrency bugs are exacerbated by the complex and hostile environment in which device drivers operate [CRKH05]. An operating system can asynchronously invoke a driver from multiple threads, a hardware device can issue interrupt requests that cause running driver processes to block and switch execution context, and the user may remove a device (hot-plugging) while some driver operation is still running. These scenarios can cause data races if insufficient synchronisation mechanisms are in place to control concurrent access to shared resources. Data races are a source of undefined behaviour in C [ISO11, p. 38], and lead to nondeterministically occurring bugs that can be hard to reproduce, isolate and fix, especially in the context of complex operating systems.

In §3.2 we give background information related to Linux device drivers and concurrency in the Linux kernel. In §3.3 we present the new technique that we developed for verifying data race freedom in device drivers. In §3.4 we discuss how we implemented our technique in Whoop. In §3.4.9 we discuss practical assumptions related to the soundness of Whoop. In §3.5 we give a sound partial-order reduction that accelerates Corral. Finally, in §3.6 we evaluate our technique on 16 device drivers from the Linux kernel and 1016 concurrent C programs from SVCOMP'16.

3.2 Device Drivers in the Linux Kernel

To begin, we give a brief background introduction to Linux device drivers. Although we will discuss relevant to this work background material, an in-depth study of the Linux kernel and its drivers is outside the scope of this thesis. The interested reader might also want to refer to a number of excellent external resources, such as the official Linux kernel documentation,[1] Linux kernel development textbooks [CRKH05, Lov10], as well as other classic textbooks on operating systems [SPG91, TW06].

[1]`http://lxr.free-electrons.com`

3.2.1 The Linux Kernel and its Components

The Linux kernel [Lov10, CRKH05] is a free and open-source monolithic[2] kernel for UNIX-like operating systems.[3] It was initially created by Linus Torvalds in 1991, but since then it has received contributions by thousands of developers around the globe. An operating system kernel is responsible for managing input/output (I/O) requests from user applications, and providing access to the available system resources (e.g. memory, processing power and network connectivity) and the underlying hardware devices. The Linux kernel can be split into the following components (see Figure 3.1):

Process management which is responsible for creating, executing and terminating processes. The kernel schedules processes for execution on available central processing units (CPUs), and also provides support for inter-process communication and synchronisation (e.g. via pipes and signals).

Memory management which deals with all memory-related functionality, such as creating and exposing a virtual address space, applying memory policies, and allocating and deallocating the requested memory (e.g. via the `kmalloc` and `kfree` functions).

Virtual File System (VFS) which provides a common interface for opening, reading, writing and closing files in a large variety of filesystems (e.g. ext4, FAT and NTFS).

Networking which deals with all network-related activities, such as receiving, identifying and transmitting network packets.

Device drivers which are responsible for the interaction between the operating system and any available hardware devices. Device drivers implement a kernel-specific programming interface that can be used to access, manage and control hardware devices. This interface abstracts away all the low-level hardware details, and is used not only by user applications, but also by other components of the Linux kernel.

An important feature of the Linux kernel is that it provides the ability to add (or remove) kernel functionality while the kernel is running. This can be achieved by creating a *loadable kernel module*, which is object code that can be dynamically linked to (or unlinked from) the kernel at runtime. The Linux kernel provides support for various types of modules, with device drivers being one of them.

[2]In monolithic architectures, the entire operating system, with its provided services and core functionality, runs in kernel-space, i.e. has full privileges.

[3]`https://www.kernel.org`

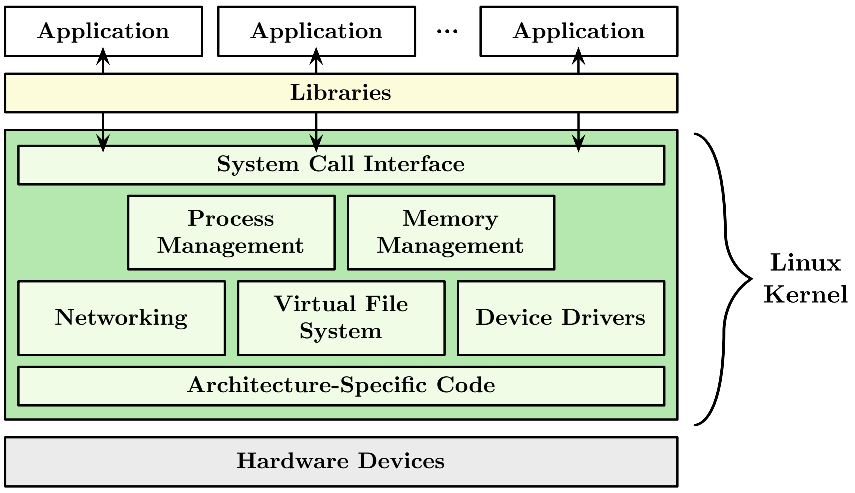

Figure 3.1: Architectural view of the Linux kernel and its components [CRKH05].

Most device drivers, as well as the rest of the Linux kernel, operate in *kernel-space*. This is a privileged mode of execution where a program executes in a protected memory space and has direct access to all hardware devices. In contrast, user applications and libraries operate in *user-space*, a mode of execution with limited privileges. The *system call interface* (see Figure 3.1) provides an application programming interface (API) that allows user-space applications and libraries to interact with kernel-space.

The Linux kernel was designed to be independent of any particular computer architecture. To achieve this, the kernel developers separated all architecture-specific code (e.g. pertaining to x86 and ARM) from the actual kernel logic, and placed it in architecture-specific directories of the kernel source tree. Then, depending on the underlying architecture, an appropriate architecture-specific directory is used.

3.2.2 Types of Linux Device Drivers

Device drivers are first class citizens in the Linux kernel. Many of them are part of the official kernel source tree[4] and evolve alongside it. Although there are many different types of drivers available, the Linux kernel typically classifies them as follows [CRKH05]:

Character (char) device drivers enable access to a char device as a stream of bytes. Char drivers typically implement the `read` and `write` functions, among other system calls. Examples of char devices are serial ports, keyboards, displays, graphics processing units (GPUs) and the FireWire serial bus interface.

[4] `https://github.com/torvalds/linux`

Block device drivers enable random-access to a block device. Each block device has an I/O request queue. Block drivers implement the `request` function, which is responsible for reading requests from the queue and processing them one at a time. Examples of block devices are flash memory cards, hard drives and optical disk drives.

Network device drivers cannot be accessed as streams of bytes, and thus are fundamentally different from char and block drivers. Network drivers are essentially interfaces that define how data is received and transmitted over a network. Instead of exposing `read` and `write` functions, a network driver implements operations related to packet transmission. Examples of network devices are Ethernet and InfiniBand.

The aforementioned are only the very fundamental types of device drivers available with the Linux kernel. They can be further subcategorised depending on what kernel interfaces they are implementing (e.g. Ethernet). A previous study [KS12] identified 72 unique types of device drivers in the 2.6.37.6 release of the Linux kernel.

Although most device drivers run in kernel-space, the Linux kernel also supports drivers running in user-space. Such drivers can be usually implemented when only a single process will ever compete for the same device. Benefits of user-space drivers include ease of development, increased reliability (e.g. a bug cannot easily crash the kernel) and faster debugging [EG04]. However, kernel-space device drivers benefit from increased performance and easier communication with the hardware and the rest of the kernel. In this work, we focus on device drivers running in kernel-space.

3.2.3 Device Driver Entry Points

The main purpose of a driver is to act as the software interface for a particular hardware device. To achieve this, each driver has to implement a set of *operations* that allow the operating system to access, manage and control the corresponding device. In the case of Linux, when a driver gets initialised, the driver has to register itself and its operations with the kernel. This subsequently allows the Linux kernel to invoke any registered operation as a callback function to gain access to the device managed by the driver. These callback functions are commonly known as the *entry points* of the driver.

There are many different types of driver entry points, ranging from generic ones (e.g. for power management), to specialised ones (e.g. for sending and receiving network packets). Each driver has to implement the entry points that are related to its functionality. For example, the RealTek r8169 driver, which is available in the Linux kernel 4.0 source tree,[5]

[5]`http://lxr.free-electrons.com/source/drivers/net/ethernet/realtek/r8169.c?v=4.0`

```
static const struct ethtool_ops rtl8169_ethtool_ops = {
  .get_drvinfo = rtl8169_get_drvinfo,
  .get_regs_len = rtl8169_get_regs_len,
  .get_link = ethtool_op_get_link,
  .get_settings = rtl8169_get_settings,
  .set_settings = rtl8169_set_settings,
  .get_msglevel = rtl8169_get_msglevel,
  .set_msglevel = rtl8169_set_msglevel,
  .get_regs = rtl8169_get_regs,
  .get_wol = rtl8169_get_wol,
  .set_wol = rtl8169_set_wol,
  .get_strings = rtl8169_get_strings,
  .get_sset_count = rtl8169_get_sset_count,
  .get_ethtool_stats = rtl8169_get_ethtool_stats,
  .get_ts_info = ethtool_op_get_ts_info,
};
```

Figure 3.2: Ethernet-related entry point declarations in the RealTek r8169 Linux 4.0 Ethernet driver.

```
static int rtl8169_get_settings(struct net_device *dev, struct ethtool_cmd *cmd) {
  struct rtl8169_private *tp = netdev_priv(dev);
  int rc;

  rtl_lock_work(tp);
  rc = tp->get_settings(dev, cmd);
  rtl_unlock_work(tp);

  return rc;
}
```

Figure 3.3: Entry point from the RealTek r8169 Linux 4.0 Ethernet driver.

is an Ethernet driver, and thus has to implement Ethernet-related entry points (among some other types of entry points). Figure 3.2 shows how the r8169 driver sets its Ethernet-related entry points as callback functions.

Figure 3.3 shows the source code of `rtl8169_get_settings`, a simple entry point from the r8169 driver. This entry point is called whenever the Linux kernel wants to read the current Ethernet settings of the underlying Ethernet device. The `rtl8169_get_settings` entry point takes as input a `net_device`, which is a data structure for network devices, and an `ethtool_cmd`, which is a data structure associated with Ethernet-specific features. The `net_device` structure is important for two reasons: (i) it is unique for each registered with the kernel network driver, and thus makes sure that the kernel will only operate the specific driver; and (ii) allows an entry point to access the driver private data by calling the `netdev_priv` function. The `rtl8169_get_settings` entry point proceeds by writing the current device settings to the `ethtool_cmd` structure and exits with an error code. The entry point also uses the `rtl_lock_work` and `rtl_unlock_work` functions for synchronisation, which is explained in §3.2.4.

3.2.4 Concurrency in Device Drivers

Modern operating systems address the demand for responsiveness and performance by providing multiple sources of concurrency [CRKH05]. For example, the Linux kernel is able to invoke the entry points of a device driver concurrently. However, this can be a source of subtle concurrency-related errors, such as *data races* (see Definition 3.1), if no proper synchronisation mechanisms are in place.

Definition 3.1. *A* data race *occurs in a device driver when two distinct threads access a shared memory location, at least one of the accesses is a write access, at least one of the accesses is non-atomic, and there is no mechanism in place to prevent these accesses from being simultaneous.*

Figure 3.4 shows the source code for the `nvram_llseek` entry point of the generic_nvram Linux 4.0 driver,[6] which can be used to modify the current read/write position in the given file, and then return the modified position. This entry point might be invoked concurrently by more than one threads, with the same `file` struct passed as an input. This can lead to data races, if the threads try to simultaneously access the `f_pos` field of `file`.

The most common method for avoiding data races is to protect all simultaneous shared memory accesses with *mutexes* (the term mutex is derived from "mutual exclusion"), also simply known as *locks*, forming *critical sections* [Dij65b]. A critical section is a sequence of program statements that cannot be executed by more than one thread at the same time. Figure 3.5 shows how to use a lock to eliminate the data races in the example of Figure 3.4. Likewise, in Figure 3.3, the `rtl8169_get_settings` entry point avoids data races by acquiring a lock (implemented in the `rtl_lock_work` function), before it reads the current device settings.

The two most common types of mutexes in the Linux kernel are *semaphores* [Dij65a] and *spinlocks* [And90]:

Semaphores are locking primitives that consist of a signed integer value n and a wait-queue. Controlling a semaphore is achieved using a pair of functions historically known as `P` and `V`. When a thread tries to enter a critical section it calls `P`: if $n > 0$,

[6] `http://lxr.free-electrons.com/source/drivers/char/generic_nvram.c?v=4.0`

```
static loff_t nvram_llseek(struct file *file, loff_t offset, int origin) {
  switch (origin) {
    case 0:
      break;
    case 1:
      offset += file->f_pos; // can potentially race
      break;
    case 2:
      offset += nvram_len;
      break;
    default:
      offset = -1;
  }

  if (offset < 0)
    return -EINVAL;

  file->f_pos = offset; // can potentially race
  return file->f_pos; // can potentially race
}
```

Figure 3.4: Entry point in the generic_nvram Linux 4.0 device driver. The read and write accesses to the `f_pos` field of the `file` parameter can potentially race if more than one threads invoke the entry point concurrently and pass a pointer to the same `file` struct as an argument.

n is decremented by 1 and the thread continues execution; else, if $n <= 0$, the thread is added to the wait-queue and yields execution to another thread. When a thread exits the critical section it calls `V`: n is incremented by 1 and, if the wait-queue is not empty, the next waiting thread starts executing. To achieve mutual exclusion, a semaphore must always be initialised with the value 1, which is also the maximum value as only one thread at a time is allowed to hold the lock.

Spinlocks are locking primitives that transition between two states: locked and unlocked. When a thread tries to enter a critical section, it checks the state of the spinlock. If the spinlock is unlocked, the thread acquires the spinlock by locking it, and continues executing. If, instead, the spinlock is already locked, the thread *busy-waits*: it enters a tight loop where it repeatedly checks if the spinlock has been released. When the spinlock is finally released, the thread exits the loop, acquires the spinlock and enters the critical section.

An important difference between a spinlock and a semaphore is that the former is based on busy-waiting, while the latter is based on blocking and context-switching. Spinlocks generally offer higher performance when they are held infrequently, or for a small number

```
static loff_t nvram_llseek(struct file *file, loff_t offset, int origin) {
  mutex_lock(&nvram_mutex); // lock
  switch (origin) {
    case 0:
      break;
    case 1:
      offset += file->f_pos;
      break;
    case 2:
      offset += nvram_len;
      break;
    default:
      offset = -1;
  }

  if (offset < 0) {
    mutex_unlock(&nvram_mutex); // unlock
    return -EINVAL;
  }

  file->f_pos = offset;
  loff_t res = file->f_pos; // store result
  mutex_unlock(&nvram_mutex); // unlock
  return res;
}
```

Figure 3.5: Using a mutex to eliminate the data races in Figure 3.4. Note that because the `return` statement can potentially race on the `f_pos` field, we store the result in a temporary variable `res` inside the critical section.

of instructions, as its more efficient for a thread to spend a few CPU cycles busy-waiting instead of performing a context-switch [KLMO91]. If, however, there are multiple threads competing for the same lock, or the lock is held for a long time, using a semaphore can increase performance due to better CPU utilisation.

Careless use of locks has many well-known pitfalls [SL05]: coarse-grained locking might hurt performance as it limits the opportunity for concurrency, whereas fine-grained locking can easily lead to *deadlocks* (see Definition 3.2).

Definition 3.2. *A deadlock occurs when a thread has locked a shared resource that one or more other threads are waiting to access, the thread holding the lock is waiting for a shared resource to be released from one of the other waiting threads, and thus no thread is able to progress.*

Although the Linux kernel provides lock-free synchronisation mechanisms [CRKH05, p. 123], such as atomic read-modify-write operations, in this work we focus on locks as they are commonly used in device drivers. In this work, we treat lock-free synchronisation operations soundly by regarding them as not providing any protection between threads; although this can lead to false alarms, it can never miss real bugs.

3.3 Symbolic Lockset Analysis for Device Drivers

We now present a novel *symbolic lockset analysis* for detecting data races in device drivers. Our technique considers, for a given driver, every pair of entry points that can potentially execute concurrently (see §3.4.2). For each such pair, we use symbolic verification to check whether it is possible for the pair to race on a shared memory location. We soundly model the effects of additional threads by treating the shared state of the driver abstractly: when an entry point reads from the shared state, a nondeterministic value is returned.

Restricting to pairs of entry points, rather than analysing all entry points simultaneously, exploits the fact that data races occur between pairs of threads and limits the complexity of the generated verification conditions (VCs) [BL05]. The trade-off is that a quadratic number of entry point pairs must be checked.[7] In §3.4 we discuss optimisations based on Linux device driver domain knowledge to reduce the number of pairs to some extent.

Symbolic verification of a pair of entry points works by (i) instrumenting each entry point with additional state to record locksets, and (ii) attempting to verify a sequential program that executes the instrumented entry points in sequence. A post-condition for the sequential program asserts, for every shared location, that the locksets for each entry point with respect to this location have a non-empty intersection (see given background on lockset analysis in §2.3.2). Verification of the resulting sequential program can be undertaken using any sound method; in practice we employ the Boogie verification engine [BCD^{+}06], which requires procedure specifications and loop invariants to be generated, after which VCs are generated and discharged to an automated theorem prover.

We now discuss the approach in a semi-formal manner, in the context of a simple concurrent programming model.

[7]In principle, our approach could be applied at a coarser level of granularity, such as by considering all entry points one after the other, taking into account that an entry point can race with itself. This would reduce the total number of generated VCs, potentially speeding up verification. We decided to implement the pair-wise approach because it fits with the theoretical presentation of the soundness of our symbolic lockset analysis. Experimenting with larger combinations of entry points would be interesting future work. Note that using pairwise analysis has the advantage that it enables us to easily run the symbolic lockset analysis for each pair in parallel (for performance), although we also leave this for future work.

Statement	Notes
$x := e;$	private assignment, where $x \in V_T$ and e is an expression over V_T
$x := f(\overline{e});$	procedure call, where $x \in V_T$, $\overline{e}$ is a sequence of expressions over V_T, and f is the name of a procedure in $procs_T$
$s := e;$	shared write, where $s \in V_s$ and e is an expression over V_T
$x := s;$	shared read, where $x \in V_T$ and $s \in V_s$
$\mathsf{lock}(m);$	mutex lock, where $m \in M$
$\mathsf{unlock}(m);$	mutex unlock, where $m \in M$

Figure 3.6: The allowed statements in our simple programming model.

3.3.1 Concurrent Programming Model

Let a *concurrent program* be described by a finite set of *shared* variables V_s, a finite set of mutexes M, and a finite set of *thread templates*. A thread template T consists of a finite set of procedures $procs_T$, and a finite set of private variables V_T.[8] A designated procedure $main_T \in procs_T$ denotes the starting point for execution of T by a thread. Each procedure of $procs_T$ is represented by a control flow graph of basic blocks, where each block contains a sequence of statements. A basic block either has a single successor or a pair of successors. In the latter case, an *exit condition* over thread-private variables determines the successor to which control should flow on block exit.

The allowed statements are shown in Figure 3.6, and include designated statements for reading from and writing to the shared variables of the concurrent program. In particular, shared variables may not appear in arbitrary expressions. This restriction simplifies our presentation of lockset instrumentation (see §3.3.3) and can be trivially enforced by pre-processing. We do not specify the form of expressions, nor the types of variables, assuming a standard set of data types and operations.

We consider a concurrent programming model where an *unbounded* number of threads execute a set of pre-defined thread templates. At any given point of execution a certain number of threads are active, each thread executing a particular template. In the context of device drivers, a thread template corresponds to a driver entry point. Multiple instances of the same thread template may execute concurrently, just as multiple invocations of a single driver entry point may be concurrent. Further threads may start executing at any

[8]We use the definition of thread templates to simplify our discussion of lockset instrumentation in §3.3.3 and sequentialisation in §3.3.5.

point during execution; in the context of device drivers this corresponds to the operating system invoking additional driver entry points.[9]

For ease of presentation only, our programming model does not feature aggregate data types, pointers, or dynamic memory allocation. These *are* handled by our implementation, and in §3.4 we discuss interesting practical issues arising from the handling of a full-blown language, which is C in the case of Linux device drivers.

3.3.2 Semantics

Let $\mathcal{I}$ be an infinite set from which dynamic thread ids can be drawn. The state of a running concurrent program consists of: a valuation of the shared variables V_s (a valuation is a mapping from V_s to D, which is a designated set of values); a mapping that associates each mutex in M with a thread id from $\mathcal{I}$, recording which thread currently holds the mutex, or with a special value $\bot \notin \mathcal{I}$ to indicate that the mutex is not currently held by any thread; and a list of *thread states*. Each thread state has an associated *thread template* T, which is the type of the thread (multiple threads may have the same associated template), and consists of an id (drawn from $\mathcal{I}$), an index indicating the next statement of T to be executed by the thread, and a valuation of the thread private variables V_T. If multiple threads are instances of the same template T, then each thread carries a *separate* valuation of the private variables for this template.

Initially, the valuation of the shared variables V_s is arbitrary, no mutexes are held (i.e. each mutex maps to $\bot$), and the list of thread states is empty. At any point of execution, a new thread state may be added to the list of thread states. This involves selecting a thread template T and a thread id $i \in \mathcal{I}$ that has not been previously used during program execution, setting the point of execution for the new thread state to be the first statement of $main_T$, and choosing an arbitrary valuation for the private variables V_T.

We consider a standard interleaving model of concurrency: at any execution point, a thread may execute its current statement, unless this statement has the form $\mathsf{lock}(m)$ and mutex m is held by some thread. Executing a statement updates the thread-private and shared state in a standard manner. For example, if a thread following the template T executes $s := e$, where $s \in V_s$ and e is an expression over V_T, the shared variable valuation is updated so that s has the value determined by evaluating e in the context of the thread's private variable valuation. Because our interest is in data race analysis for race-free programming, we are not concerned with weak memory behaviour, since race-free

[9]We do *not* consider the case where one thread spawns another thread, which does not typically occur in the context of drivers; rather we aim to capture the scenario where additional threads are launched by the environment.

Original Statement	**Instrumented Statement**
$s := e;$	$W_i := W_i \cup \{s\};$ $LS_{s,i} := LS_{s,i} \cap CLS_i;$
$x := s;$	$R_i := R_i \cup \{s\};$ $LS_{s,i} := LS_{s,i} \cap CLS_i;$ $\mathsf{havoc}(x_i);$
$\mathsf{lock}(m);$	$CLS_i := CLS_i \cup \{m\};$
$\mathsf{unlock}(m);$	$CLS_i := CLS_i \setminus \{m\};$

Figure 3.7: Instrumenting statements for lockset analysis.

programs exhibit only sequentially consistent behaviours [AH90].

A thread terminates if it reaches the end of $main_T$, in which case its state is removed from the list of thread states. Since our interest is in analysing device drivers, which are reactive programs, we do not consider the notion of global program termination.

3.3.3 Lockset Instrumentation

For thread templates T and U, which can be equal, we want to check whether it is possible for a thread executing T to race with a thread executing U, in the presence of arbitrarily many further concurrently executing threads. To this end, we first *instrument* T and U for lockset analysis (see §2.3.2 for the relevant background). Given an arbitrary symbol i, we define the instrumentation of T with respect to i, denoted T_i. There are two aspects to this instrumentation phase: *renaming* and *lockset instrumentation.*[10]

Renaming is straightforward: each occurrence of a private variable $x \in V_T$ used in a procedure of T is replaced with a renamed variable x_i in T_i, and every procedure $f \in procs_T$ is renamed (both at its declaration site and at all call sites) to f_i in T_i. The purpose of renaming is to ensure that when we analyse a pair of templates, T and U, both templates execute distinct procedures and operate on distinct private data. This is vital in the case where T and U are the same.

Lockset instrumentation introduces: a read set $R_i \subseteq \mathcal{P}(V_s)$ and write set $W_i \subseteq \mathcal{P}(V_s)$ to track the shared variables that have been read from and written to, respectively, by the thread executing T; a current lockset $CLS_i \subseteq \mathcal{P}(M)$ to record the mutexes currently held by the thread; and, for each shared variable $s \in V_s$, a lockset $LS_{s,i}$ to record the mutexes that are consistently held when the thread accesses s.

The statements of each procedure in T_i that access shared variables and mutexes are

[10]Renaming and lockset instrumentation are symmetric; they happen for both T and U thread templates, but in §3.3.3 we only describe the case for T.

```
$CLS_i := \emptyset$; $R_i := \emptyset$; $W_i := \emptyset$;
$CLS_j := \emptyset$; $R_j := \emptyset$; $W_j := \emptyset$;
for $s \in V_s$ do $\{LS_{s,i} := M;\ LS_{s,j} := M;\}$
$main_{T_i}()$;
$main_{U_j}()$;
assert $\forall s \in V_s$ .
  $s \in W_i \cap (R_j \cup W_j) \lor s \in W_j \cap (R_i \cup W_i) \implies$
    $LS_{s,i} \cap LS_{s,j} \neq \emptyset$;
```

Figure 3.8: The sequential program to be analysed to prove absence of data races in a pair of thread templates T and U.

instrumented to manipulate the above sets, as shown in Figure 3.7. For a shared variable assignment $s := e$, we first record in W_i that s has been written to, and then update $LS_{s,i}$ with the intersection of $LS_{s,i}$ and CLS_i, which eliminates any mutexes that are not currently held (i.e. those mutexes that are not in CLS_i). A shared variable read $x := s$ is instrumented analogously, with an additional havoc command (discussed in §3.3.4). Instrumentation of mutex manipulation commands, lock(m) and unlock(m), involves updating CLS_i to add and remove mutex m, respectively.

3.3.4 Shared State Abstraction

Recall that while our aim is to perform race analysis for pairs of threads, we must be sure to account for possible side-effects due to other threads that are running concurrently. The instrumentation of Figure 3.7 achieves this via *nondeterminism*: when reading from a shared variable s, a nondeterministic value is returned, creating an over-approximation. This is reflected by the use of a havoc[11] command, which sets its argument to an arbitrary value. Because all shared state accesses are abstracted in this fashion, it is possible to completely dispense with the shared variables after the lockset instrumentation has been performed. As a result, when instrumenting a shared variable write, the effect of the write is not explicitly modelled.

3.3.5 Sequentialisation

Figure 3.8 shows the sequential program that we analyse to prove race-freedom for a pair of thread templates T and U. Assuming that both templates have been instrumented using distinct symbols i and j, yielding T_i and U_j respectively (see §3.3.3), the sequential program operates as follows. First, the read, write, and current locksets for T_i and U_j are

[11] Note that although havoc is not an executable statement, it is straightforward to capture during symbolic analysis.

initialised to be empty, and for each shared variable s, the locksets $LS_{s,i}$ and $LS_{s,j}$ are initialised to the full set of mutexes M. The main procedures of the instrumented thread templates, $main_{T_i}$ and $main_{U_j}$, are then executed in turn (the order does not matter due to renaming). Finally, an assertion checks for consistent use of mutexes: if s is written during execution of T_i and accessed during execution of U_j, or vice-versa, then the locksets $LS_{s,i}$ and $LS_{s,j}$ must contain at least one common mutex.

3.3.6 Soundness Sketch

We sketch an argument that if the program of Figure 3.8 is correct (i.e. the assertion, described in §3.3.5, holds invariantly at the end of the program), then it is impossible for a thread executing template T to race with a thread executing template U, under the assumption that the threads are guaranteed to terminate.[12]

Let us assume that the program of Figure 3.8 is correct, and suppose (by way of contradiction) that a thread executing T can in fact race with a thread executing U, on some shared variable s. By our hypothesis that the program is correct, and that the threads terminate, the assertion checked at the end of the program guarantees that at least one mutex, say m, belongs to both $LS_{s,i}$ and $LS_{s,j}$. By the definition of a lockset (and according to the manner in which shared accesses are instrumented in Figure 3.7), this means that m is held during every access to s by both T_i and U_j. As a result, the mutex m must be unlocked and locked between the two shared memory accesses, which contradicts that the pair of accesses is racing.

In the presence of non-termination the assertion at the end of Figure 3.8 may not be reached (e.g. if the thread executing T gets stuck in an infinite loop). The termination analysis problem for device drivers has been widely studied (see for example the work on TERMINATOR [CPR06]), and in the remainder of the chapter we do not consider termination issues, assuming that the drivers we analyse in our experimental evaluation (see §3.6) always terminate.

3.4 Implementation in Whoop

The simple concurrent programming model of §3.3.1 is deliberately idealistic to ease the description of our symbolic verification technique. In practice, though, Linux device drivers are written in C, our technique does not know up-front which are the driver entry points,

[12]Although a race is in context of the original program, every trace of the original program is still captured by our sequential program. This is because using the `havoc` statement (see §3.3.4) allows us to simulate the behaviours of the original program.

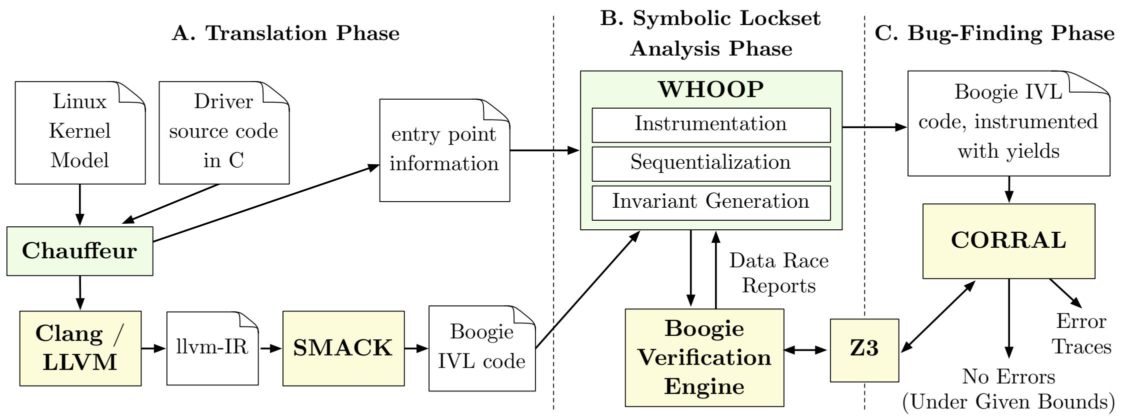

Figure 3.9: The Whoop architecture, which is empowered by industrial-strength compilation (LLVM and SMACK) and verification (Boogie and Corral) tools.

drivers might not work with a cleanly specified set of named locks, and rather than having a given set of named shared variables, we have arbitrary memory accesses via pointers. In this section, we explain how we have taken the conceptual ideas from §3.3 and used them to build Whoop, a practical, automated tool for detecting data races in drivers.

3.4.1 The Whoop Architecture

Figure 3.9 depicts the high-level architecture of Whoop. The input to the toolchain is the source code of a Linux device driver, which is written in C, together with an environmental model of the Linux kernel,[13] which is required to "close" the device driver so that it can be analysed for data races.

Initially, Whoop uses three LLVM-based tools [LA04, LLV], Chauffeur, Clang [CLA], and SMACK [RE14, SMA], to translate the Linux driver and its environmental model into an abstract program written in the Boogie intermediate verification language (IVL) [DL05] (see Figure 3.9–A). Boogie is a simple imperative language with well-defined semantics that is used as the input to a number of cutting-edge verifiers (e.g. the Boogie verifier and Corral). Next, Whoop instruments and sequentialises the program to perform symbolic lockset analysis (see Figure 3.9–B and §3.3) using the Boogie verification engine. After the verification phase completes, Whoop can exploit any inferred race-freedom guarantees to accelerate precise race-checking with Corral (see Figure 3.9–C and §3.5).

We engineered the Chauffeur and Whoop components of our toolchain (denoted with green boxes in Figure 3.9). For the remaining components, we were able to reuse industrial-

[13]The environmental model consists of stub header files modelling relevant Linux kernel APIs.

```
const struct file_operations nvram_fops = {
  .llseek = nvram_llseek,
  .read = read_nvram,
  .write = write_nvram,
  .unlocked_ioctl = nvram_unlocked_ioctl
};
```

Figure 3.10: Entry point declarations in the generic_nvram Linux 4.0 driver.

strength tools that are robust and battle-proven via their use in many complex software projects. WHOOP has been open-sourced and can be downloaded from its public GitHub repository.[14]

3.4.2 Extracting Entry Point Information

Chauffeur is a Clang-based tool that traverses the abstract syntax tree (AST) of the input driver and extracts all entry point identifier names,[15] together with the identifier names of their corresponding kernel functions. Linux drivers declare entry points in a standard way. Although changes in the Linux kernel APIs could potentially break these semantics, and thus Chauffeur, this is unlikely, since the Linux kernel developers put a lot of effort in keeping the driver-related APIs stable across kernel releases.

Figure 3.10 shows an example of how the generic_nvram Linux 4.0 driver declares the entry points for operations specific to accessing files. Chauffeur identifies these declarations in the AST and outputs the relevant information in an XML file, which can be parsed and used by WHOOP during the instrumentation phase.

3.4.3 Translation for Verification

Next, the device driver source code is compiled by Clang to LLVM-IR, a low-level, assembly-like language in single static assignment (SSA) form. Function calls (e.g. for locking and unlocking) are preserved in the translation and, hence, we do not need to keep track of them separately. SMACK then translates the driver from LLVM-IR to Boogie IVL, which is the input language of the WHOOP symbolic verifier. An important feature of SMACK is that it leverages the pointer-alias analyses of LLVM to efficiently model the heap manipulation operations of C programs in Boogie IVL. Thus, WHOOP does not need to directly

[14]`https://github.com/smackers/whoop`

[15]Chauffeur contains a list of known Linux entry point declaration identifiers. We have currently hard-coded this list inside Chauffeur, and extend it as we encounter new types of entry points. In the future, we plan to expose a configuration API so that users can register new identifiers.

deal with pointers and alias analysis, a hard problem in its own right, which allows us to reuse robust existing techniques and focus instead on verification efforts.

To achieve scalability, SMACK uses a *split* memory model that exploits an alias analysis to soundly partition memory locations into equivalence classes that do not alias. This has been shown to lead to more tractable verification compared with a *monolithic* memory model where the heap is considered to be an array of bytes [RH09]. The split model is based on *memory regions*, which are maps of integers that model the heap. A benefit of using the split model is that distinct memory regions denote disjoint sections of the heap. We leverage this knowledge inside Whoop to guide and optimise our lockset instrumentation and analysis, and to create a fine-grained context-switch instrumentation (see §3.5).

3.4.4 Identifying Locks

When the instrumentation phase begins, Whoop performs an inter-procedural static analysis (at Boogie IVL level) to identify all available locks, and rewrite them (both at declaration and at all access sites) to a unique Boogie constant (one for each lock). Representing all locks statically, instead of using their original SMACK pointer-based representation, makes it easier for Whoop to perform lockset instrumentation and lockset-based invariant generation, since the tool can manipulate the Boogie AST, without having to infer each time which pointers refer to the same lock.

If Whoop cannot reliably infer a unique lock instance (e.g. because it is stored in an unbounded data structure of locks, such as an array or a list), it will exit with a warning. However, we have rarely encountered this in practice since a small, fixed number of locks is advocated by Linux experts as good practice when developing drivers [CRKH05, p. 123]. Currently, Whoop only supports Linux kernel mutex and spinlock operations, but it is easy to enhance the tool with knowledge of other locking primitives.

3.4.5 Watchdog Race-Checking Instrumentation

Data race detection is performed by introducing sets containing the locks that are consistently held when accessing each shared variable, and sets containing all shared variables that are read and written (see §3.3.3 and the instrumentation of Figure 3.7). These sets can be modelled directly in Boogie as characteristic functions, using maps. However, this requires the use of quantifiers to express properties related to set contents. For example, to express that a set X of elements of type A is empty, where X is represented as a map from A to Bool, we would require the quantified expression $\forall a : A \, . \, \neg X[a]$. It is well known that automated theorem proving in the presence of quantified constraints is challenging [BCD+15], and that theorem provers such as Z3 [DMB08] are often much more effective when quantifiers are not present.

To avoid quantifiers and the associated theorem proving burden, we use instead a *watchdog* race-checking instrumentation, adapted from previous work [BBC+14]. Suppose we are analysing entry points T and U, and that after translating them into Boogie these two entry points share a common memory region MR (see §3.4.3). When analysing T and U for data races, we introduce an unconstrained symbolic constant $watched_{MR}$, representing some unspecified index into MR; we call this index the *watched offset* for MR. We then attempt to prove that it is impossible for T and U to race on MR at the index $watched_{MR}$. If we can succeed in proving this, we know that T and U must be race-free for the *whole* MR, since the watched offset was arbitrary.

This technique of choosing an arbitrary index to analyse for each map manipulated by an entry point pair can be seen as a form of quantifier elimination: rather than asking the underlying theorem prover to reason for all indices of MR, in a quantified manner, we eliminate the quantifier in our encoding, and instead ask the theorem prover to reason about a single, but arbitrary, index of MR.

3.4.6 Generating Loop and Procedure Summaries

Early in the development of Whoop, we experimented with analysing recursion-free Linux drivers by inlining all procedures at their call sites. We found that full procedure inlining did not scale to large drivers, and is also problematic (i.e. running out of memory) when drivers exhibit recursion, e.g. the r8169 Ethernet driver (see §3.6).

To make our analysis scale without sacrificing precision, and to support recursion, we use the Houdini algorithm [FL01] to automatically compute summaries (pre-/post-conditions and loop invariants). Given a pool of *candidate* pre-conditions, post-conditions, and loop invariants, Houdini repeatedly attempts to verify each procedure. Initially, the entire candidate pool is considered. If verification fails due to an incorrect candidate, this candidate is discarded. This process repeats until a fixpoint is reached. At this point, the (possibly empty) set of remaining candidates has been proven to hold, and thus can be used to summarise calls and loops during further program analysis. Note that Houdini does *not* generate the initial pool of candidates: Whoop generates them using a set of heuristics, and passes them to Houdini as a starting point. This is done based on syntactic patterns extracted from an inter-procedural pass over the code for an entry point.

We give two examples; for clarity we use notation from the simple concurrent programming model of §3.3.1. If we observe syntactically that procedure f of entry point T may write to, but does not read from, a shared variable s, then when instrumenting T with symbol i, we guess $s \in W_i$ and $s \notin R_i$ as post-conditions for f_i. These guesses may be incorrect, for instance if the potential write to s turns out to be in dead code, or if a read from s has already been issued on entry to f_i. Similarly, if syntactic analysis indicates that f may unlock mutex m, we guess $m \notin CLS_i$ as a post-condition for f_i. This guess may be incorrect, for instance if the unlock operation is not reachable, or if a subsequent lock operation acquires the mutex again.

We stress that guessing incorrect candidate invariants does not compromise the soundness of verification: WHOOP is free to speculate candidates that are later deemed to be incorrect, and thus discarded by Houdini. The balance we try to strike is to have WHOOP generate sufficient candidates to enable precise lockset analysis, without generating so many candidates that the speed of the Houdini algorithm is prohibitively slow.

3.4.7 Verification and Error Reporting

For each pair of entry points, the instrumented sequential program, equipped with procedure and loop summaries, is given to the Boogie verification engine. For each procedure in the program, Boogie generates a VC and discharges it to the Z3 theorem prover. In particular, the verification for the root-level procedure, implementing the sequential program sketched in Figure 3.8, encodes the race-freedom check for the corresponding pair. Successful verification implies that the pair is free of data races, while an error (i.e. counterexample) denotes a *potential* race and is reported to the user.

To improve usability, WHOOP has a built-in error reporter that matches counterexamples to source code. The following is a race that WHOOP found and reported for the example of Figure 3.4:

```
generic_nvram.c: error: potential read-write race:
  read by entry point nvram_llseek, generic_nvram.c:54:2
    return file->f_pos;
  write by entry point nvram_llseek, generic_nvram.c:53:2
    file->f_pos = offset;
```

Engineering the error reporter was significantly challenging, because we had to solve the problem of accurately matching an assertion failure at the Boogie level (see §3.3.5) with the source code of the original C program. This was harder than we anticipated: although we know that a data race occurred between two accesses in shared memory region *MR* (since the corresponding Boogie assertion for *MR* failed, resulting into a counterexample), we do *not* know which are the accesses that caused the data race. To tackle this problem, we use the *state capture* facility provided by Boogie to ask the SMT solver to provide a valuation of all program variables after each synchronisation operation and shared memory access. This allows us to traverse the counterexample trace and report all pairs of shared memory accesses that potentially raced, causing the Boogie assertion to fail. To accurately report the data races in context of the original C program, we store source code information in attributes attached to each shared memory access in the Boogie program.

3.4.8 Optimisations

We have implemented various optimisations to increase the precision and performance of Whoop. We comment on the two most effective ones.

First, we enriched Whoop with information regarding *kernel-imposed serialisation* to increase precision. The Linux kernel is able to serialise calls to entry points, thus forcing them to run in sequence instead of an interleaved manner. For example, a large number of networking entry points are mutually serialised with RTNL, a network-specific lock in the Linux kernel. We discovered this when Whoop reported many races between a number of networking entry points of the r8169 driver (see §3.6); when we investigated the source of these races, we found out that these entry points could not race in reality because of RTNL. Whoop exploits this knowledge and does not create pairs for entry points that are mutually serialised by the kernel. This is an ongoing manual effort: the more drivers we study, the more such properties we discover to make Whoop more precise.

Second, we soundly reduce the number of memory regions that are analysed for data races. If memory region *MR* is accessed by only one entry point in a pair then, trivially, the pair cannot race on *MR*. We thus disable lockset analysis for *MR*. This can significantly reduce the complexity of VCs that need to be solved by the theorem prover, speeding up the verification process.

3.4.9 Practical Assumptions Related to Soundness

In practise, Whoop is *soundy*[16] [LSS+15] rather than sound: the tool aims in principle to perform a sound analysis that can prove absence of races, but suffers from some known sources of unsoundness, which we now comment on.

We assume that the formal parameters of a device driver entry point do not alias, and thus cannot race. This is a potentially unsound feature of Whoop that can be turned off by using a command line option. Without this assumption we have observed Whoop reporting false alarms, and in our experience so far we have not missed any data races by assuming non-overlapping parameters. We also rely on the soundness of our best-effort

[16]`http://soundiness.org/`

environmental model, and on exploiting domain-specific knowledge related to entry point serialisation by the Linux kernel.

In addition to inheriting soundness issues arising from currently unknown bugs in Whoop and in the external tools that Whoop relies on, we acknowledge that: (i) the 1.5.0 release of SMACK, which we used in our toolchain, is subject to sources of unsoundness, e.g. it models integers as an infinite set (rather than as a finite set of bit-vectors), and its memory model can potentially be unsound in (typically rare) situations where programs use unaligned byte-level memory accesses;[17] and (ii) that the combination of Clang and SMACK commits our approach to specific choices related to undefined and implementation-defined aspects of the C language when translating to Boogie. However, Whoop makes no fundamental assumptions related to these translation decisions, meaning that a more accurate C-to-Boogie translation would automatically lead to a more accurate analysis with Whoop.

3.4.10 Limitations of Whoop

As a lockset analyser, Whoop can be imprecise since a violation of the locking discipline does not always correspond to a real data race (e.g. when lock-free synchronisation is used instead of locking). Whoop also uses over-approximation, which can be another source of imprecision. Furthermore, the tool does not check for locks stored in unbounded data structures (e.g. arrays or lists) or for locks provided by external libraries, although the latter could be addressed by providing a mechanism for users to declare custom locks. We also do not currently treat interrupt handlers in a special way; we just assume that they can execute concurrently with any entry point. One way to address this is to model interrupt-specific kernel functions (e.g. for enabling/disabling interrupts).

Another limitation of Whoop is that it is unable to verify drivers designed to be accessed by only a single process at a time. This *single-open device* [CRKH05, p. 173] mode can be enforced by atomically testing (at the beginning of an entry point) a global flag that indicates device availability: if the flag is set, then the checking entry point executes, else it blocks. Because Whoop over-approximates read accesses to shared variables, and thus this flag, it can falsely report a pair as racy. However, experts [CRKH05, p. 173] advise against serialising drivers in this way, as it “inhibits user ingenuity” (e.g. a user might expect that a driver can be accessed concurrently for performance).

Statically analysing drivers requires “closing” the environment by abstracting away the low-level implementation details. To this end, we developed a simple model for the Linux

[17]Note that currently bit-vectors and unaligned byte-level memory accesses are soundly and precisely handled by SMACK (as of release 1.5.1).

kernel that consists of (nondeterministic) stub functions. A limitation of our model is that it can be inaccurate, leading to semantic mismatches that can in turn lead to false positives and/or false negatives. However, we currently only focus on finding data races, and thus can get away with over-approximating large parts of the Linux kernel, without losing too much precision. Making our model more precise is an ongoing manual effort that requires Linux kernel expertise. We argue that further work on the model is orthogonal to the contributions of this thesis. Also, even if our symbolic analysis results in false positives, Whoop can still use the results to significantly speedup a more precise bug-finder, as seen in §3.5 and §3.6.

3.5 Accelerating Precise Race-Checking

Whoop is a soundy (see §3.4.9) but imprecise data race analyser. For developers who deem false alarms as unacceptable, we consider a method for leveraging the full or partial race-freedom guarantees provided by Whoop to accelerate Corral [LQL12], a precise bug-finder used by Microsoft for analysing Windows drivers [LQ14]. Because Corral operates on Boogie programs, it was easy to integrate it into our toolchain (see Figure 3.9–C). Our technique, though, is general and capable in principle of accelerating any bug-finder that operates by interleaving threads at shared memory operations.

Corral is a symbolic bounded model checker (see §2.2.3) for Boogie programs that uses the Z3 SMT solver to statically reason about program behaviours. It checks for violations of provided assertions, and reports a precise counterexample if an assertion violation is found. Corral performs bounded exploration of a concurrent program in two steps. First, given a bound on the number of allowed context-switches, the concurrent program is appropriately *sequentialised*, and the generated sequential version preserves all reachable states of the original concurrent program up to the context-bound [LQR09, LR08] (see §2.3.5). Then, Corral attempts to prove bounded (in terms of the number of loop iterations and recursion depth) sequential reachability of a bug in a goal-directed, lazy fashion to postpone state-space explosion when analysing a large program. Corral uses two key techniques to achieve this: (i) variable abstraction, where it attempts to identify a minimal set of shared variables that have to be precisely tracked in order to discharge all assertions; and (ii) stratified inlining, where it attempts to inline procedures on-demand as they are required for proving program assertions.

Whoop leverages our lockset analysis (see §3.3) to accelerate race-checking via a sound partial-order reduction. We first give a simple race-checking instrumentation for Corral (see §3.5.1), and then discuss two optimisations that Whoop permits (see §3.5.2).

3.5.1 Race-Checking Instrumentation

To detect data races, CORRAL requires an input Boogie program instrumented with `yield` and `assert` statements. Each `yield` statement denotes a nondeterministic context-switch, and is used by CORRAL to guide its sequentialisation (see §2.3.5). To interface with CORRAL, and assuming no race-freedom guarantees, WHOOP instruments a Boogie program with a simple, but effective encoding of data race checks [EMBO10].

Whenever there is a write access to a shared variable s, WHOOP instruments the Boogie program as follows:

```
s := e; // original write
yield; // allow for a context-switch
assert s == e; // check written value has not changed
```

Likewise, whenever there is a read access to the shared variable s, WHOOP instruments the Boogie program as follows:

```
x := s; // original read
yield; // allow for a context-switch
assert x == s; // check read value has not changed
```

The above instrumentation instructs CORRAL to consider all possible context-switches at each `yield` statement, up to the specified context-bound. CORRAL will then check these interleavings for data races. CORRAL is inherently unsound (i.e. can miss real data races), because it performs bounded verification. However, CORRAL is precise and, assuming a precise environmental model, it will only report true data races. WHOOP takes advantage of this precision to report only feasible races.

Note that our current instrumentation ignores some benign races: it does not report a read-write race if the write access updates the shared memory location with the same value that already existed; it also does not report a write-write race if the two write accesses update the shared memory location with the same value (which can be different from the pre-existing value). However, even benign races lead to undefined behaviour according to the C standard, and it is well known that undefined behaviours can lead to unexpected results when combined with aggressive compiler optimisations. In future work, we plan to experiment with `yield` instrumentations that can report such benign races.

In this work, we use CORRAL to analyse individual pairs of entry points. We do not use any abstraction to model additional threads, as we want CORRAL to report only true races. However, because we only analyse pairs, CORRAL will miss races that require more than two threads to manifest. We could extend our setup so that more than two threads are considered by CORRAL, but because the number of threads that an operating system kernel might launch is unknown in general, we are inevitably limited by some fixed maximum thread count.

3.5.2 Sound Partial-Order Reduction

Using the naive `yield` instrumentation described in §3.5.1, Whoop instruments a `yield` statement after each shared memory access of each entry point, and after every lock and unlock operation.[18] Whoop then sends this instrumented program to Corral, which leverages sequentialisation to explore all possible thread interleavings up to a predefined bound. Our approach to accelerating Corral is simple and yet effective: if, thanks to our symbolic lockset analysis, we know that a given statement that accesses shared memory cannot be involved in a data race, then we do not instrument a `yield` after this statement, and we also omit the `assert` statement that would check for a race.

The above approach is a form of *partial-order reduction* [God96] (see §2.3.4), and reduces the number of context-switches that Corral must consider in a *sound* manner: there is no impact on the bugs that can be found by the tool (see experiments in §3.6). This is because each shared memory access for which a `yield` is suppressed is guaranteed to be protected by a lock (a consequence of lockset analysis). If the access is a write, its effects are not visible by the other entry point in the pair until the lock is released. If the access is a read, the value of the shared location cannot change until the lock is released. The fact that a `yield` is placed after each unlock operation suffices to take account of the communication between entry points via the shared memory location.

We have implemented two different `yield` instrumentations in Whoop: Yield-EPP and Yield-MR. The first instruments `yield` statements in a binary fashion: if Whoop proves an entry point pair (EPP) as race-free, then it will instrument a `yield` statement only after each lock and unlock statement of the pair; else if Whoop finds that a pair might race, then it will instrument a `yield` after all visible operations of the pair. Yield-MR is a finer-grained instrumentation: it instruments a `yield` only after each access to a memory region (MR) that might race in the pair (regardless if the pair has not been fully proven as race-free), and after each lock and unlock statement. In our experiments (see §3.6), Yield-MR is significantly faster than Yield-EPP.

In general, Whoop is able to accelerate Corral for *arbitrary* bug-finding in concurrent programs. To demonstrate this, we applied our symbolic lockset analysis on 1016 concurrent C programs from SVCOMP'16, and managed to accelerate Corral without missing any bugs (see §3.6). In the case of device drivers, we only checked for data races using the

[18] We acknowledge that in the presence of data races and relaxed memory, even considering all interleavings of shared memory accesses may be insufficient to find all bugs.

above instrumentation; identifying useful safety properties to check is challenging since Linux drivers typically contain no assertions.

3.6 Experimental Evaluation

We performed experiments to validate the usefulness of WHOOP (see §3.4) and its combination with CORRAL (see §3.5). We first present race-checking results from running our toolchain on 16 Linux 4.0 device drivers. We then evaluate the runtime performance and scalability of CORRAL using different yield instrumentations and context-switch bounds on our driver benchmarks, and also on 1016 concurrent C programs taken from the 5th International Competition on Software Verification (SVCOMP'16). Our results demonstrate that WHOOP can efficiently accelerate concurrency bug-finding with CORRAL.

Experimental Setup We performed all experiments on a 3.40GHz Intel Core i7-2600 CPU with 16GB RAM running Ubuntu Linux 12.04.5 LTS, LLVM 3.5, SMACK 1.5.0, Z3 4.3.2, Boogie rev. 4192 and CORRAL rev. 534. We also used Mono 4.1.0 to run Boogie and CORRAL. For the Linux driver benchmarks, we configured CORRAL with a time budget of 10 hours (T.O. denotes timeout), a context-switch bound (csb) of 2, 5, and 9, and the default recursion depth bound of 1. For the SVCOMP'16 benchmarks, we configured CORRAL with a time budget of 900 seconds, and a csb of 2 and 3.

Benchmarks We evaluated our methodology against 16 device drivers taken from the Linux 4.0 kernel.[19] We chose non-trivial drivers of several types, including block, char, Ethernet, near field communication (nfc), universal serial bus (usb), and watchdog (see Table 3.1). For all these different types of drivers, we had to understand their environment and manually model it; this required approximately two months of work. We also evaluated the combination of WHOOP and CORRAL on 1016 concurrent C programs (the majority of which use POSIX threads [But97]) taken from the SVCOMP'16 competition.[20] To be able to handle these concurrent C programs, we had to extend WHOOP with understanding of the POSIX threads concurrency primitives. However, the core symbolic lockset analysis remained the same as discussed in §3.3 and §3.4.

[19] `https://www.kernel.org`
[20] `http://sv-comp.sosy-lab.org/2016/index.php`

3.6.1 Race-Checking

Table 3.1 presents statistics for our device driver benchmarks: lines of code (LoC); number of possible entry point pairs (#Pairs); number of SMACK memory regions (#MRs); number of racy pairs (#Racy Pairs) and number of racy memory regions (#Racy MRs) reported by Whoop; and number of data races discovered by Corral using a csb of 2 (#Races Found).[21] Using a csb larger than 2 did not uncover any additional races; this suggests that all data races in our benchmarks can manifest with a csb of 2, or that Corral hit its default recursion depth bound of 1 before discovering a deeper bug. Although we experimented with larger recursion depth bounds, we were unable to discover any races that could not be exposed with the default recursion depth bound.

We can see in Table 3.1 that Whoop reports more data races than Corral does. This is because Whoop employs an over-approximating shared state abstraction to conservatively model the effects of additional threads when analysing an entry point pair, and because lockset analysis is inherently imprecise; both factors can potentially lead to false positives. On the other hand, Corral is precise, but can miss races since it considers only a limited number of context-switches. Another issue with Corral is loop coverage due to unsound loop unrolling. To tackle this, we enabled the built-in loop over-approximation described in previous work [LQ14]. This can potentially lead Corral to report false bugs, but we have not seen this in practice. Furthermore, when we apply Corral to an entry point pair, we just check the specific pair and do not account for the effects of other threads (see §3.5); this can also cause Corral to miss some races. Note that standalone Corral did not discover any races that Whoop did not already report. This is expected, as Whoop aims for soundness, and increases our confidence in the implementation of Whoop.

Most of the races that Whoop and Corral discovered fall into two categories. The first is about accessing a global counter (or a flag) from concurrently running entry points, without holding a common lock. This might be for performance, and indeed a lot of the races we found might be benign (which however can be problematic as discussed in §3.5.1). The second is about an entry point modifying a field of a struct (either global or passed as an argument) without holding a lock. This can lead to a data race if another entry point simultaneously accesses the same field of the same struct.

As an example of the second category, we found the following race in the generic_nvram driver (see Figure 3.4): the `llseek` entry point accesses the file offset `file->f_pos` without holding a lock (`file` is passed as a parameter to `llseek`). This causes a race if the kernel

[21]The number of racy memory regions can be less than the number of races found by Corral: Whoop might find that a memory region is racy, but the same memory region might race in several program statements.

Table 3.1: Program statistics and race-checking results from applying Whoop and Corral to the Linux 4.0 device driver benchmarks.

					Whoop		**Corral**
id	**Drivers**	**LoC**	**#Pairs**	**#MRs**	**#Racy Pairs**	**#Racy MRs**	**#Races Found**
1	generic_nvram	283	14	39	7	2	4
2	pc8736x_gpio	354	27	55	13	6	5
3	machzwd	457	10	49	6	3	1
4	ssu100	568	7	27	✗	✗	✗
5	intel_scu_wd	632	10	45	5	1	2
6	ds1286	635	15	49	5	3	✗
7	dtlk	750	21	53	10	6	✗
8	fs3270	883	15	54	9	1	✗
9	gdrom	890	94	41	21	2	✗
10	swim	996	28	80	15	7	8
11	intel_nfcsim	1272	10	24	10	2	✗
12	ps3vram	1499	4	32	1	1	✗
13	sonypi	1729	30	62	19	4	2
14	sx8	1751	2	47	2	1	1
15	8139too	2694	46	37	40	4	✗
16	r8169	7205	192	50	88	1	✗

invokes `llseek` from another thread, while passing the same `file` object as a parameter. We observed that another char driver, using the same APIs, *does* use a lock to protect the offset access in its respective `llseek` entry point, leading us to suspect that the race we found in generic_nvram is not benign.

3.6.2 Accelerating Corral

Table 3.2 presents runtime results from using Whoop, standalone Corral, and the combination of Whoop and Corral to analyse our driver benchmarks, while Figure 3.11 plots the runtime speedups that Corral achieves using Whoop with the Yield-MR instrumentation on the same benchmarks. Standalone Corral uses Yield-ALL, which instruments a context-switch (i.e. a `yield`) after all visible operations, while Whoop + Corral uses Yield-EPP and Yield-MR, which are finer-grained instrumentations (see §3.5.2). Table 3.2 also shows the number of context-switches per instrumentation (#Y). All reported times are in seconds and averaged over three runs.

Whoop uses over-approximation to scale and, as expected, executes significantly faster than standalone Corral in all our driver benchmarks. For example, Corral times out in all configurations (with and without Whoop) when trying to analyse the r8169 Ethernet

Table 3.2: Comparison between different yield instrumentation granularities and context-switch bounds in the Linux 4.0 device driver benchmarks.

	Whoop	**Corral**				**Whoop + Corral**							
		Yield-ALL — Time (sec)				**Yield-EPP — Time (sec)**				**Yield-MR — Time (sec)**			
id	**Time (sec)**	**#Y**	**csb = 2**	**csb = 5**	**csb = 9**	**#Y**	**csb = 2**	**csb = 5**	**csb = 9**	**#Y**	**csb = 2**	**csb = 5**	**csb = 9**
1	2.3	92	32.0	67.7	197.5	47	17.7	24.2	132.1	29	13.5	16.9	49.3
2	4.1	691	169.3	595.1	27337.6	500	79.4	432.4	22514.1	167	41.3	79.4	1358.9
3	2.8	104	38.1	45.6	78.4	51	26.1	31.2	55.2	22	15.7	17.3	23.9
4	2.9	82	11.1	13.8	37.3	✗	3.1	3.1	3.2	✗	3.1	3.1	3.1
5	2.4	314	22.8	130.3	1571.7	217	14.7	70.0	748.0	196	13.5	66.1	689.3
6	4.1	513	35.0	40.2	51.8	245	22.3	25.7	33.0	129	14.5	16.1	19.3
7	5.4	801	182.6	263.7	793.0	664	91.2	150.7	633.2	286	41.6	52.9	104.6
8	3.2	321	81.5	419.4	T.O.	239	62.6	405.7	33468.4	211	40.7	295.3	8883.0
9	9.5	2058	388.0	390.8	392.1	1143	104.0	105.5	107.1	812	99.2	102.4	107.6
10	5.8	1270	271.0	2746.6	T.O.	996	164.9	2309.8	T.O.	805	95.5	1847.9	T.O.
11	3.6	732	39.8	85.0	1539.9	732	44.4	89.6	1543.4	601	21.0	32.0	278.0
12	4.5	189	99.6	2376.8	T.O.	176	96.2	2249.0	T.O.	156	55.8	1698.6	T.O.
13	10.6	1745	1966.5	T.O.	T.O.	1566	1924.4	T.O.	T.O.	1024	906.0	T.O.	T.O.
14	2.8	227	12.9	15.2	21.4	227	16.1	18.5	24.5	217	15.9	18.5	24.3
15	18.9	7151	548.2	14469.0	T.O.	6266	474.7	12677.3	T.O.	4964	359.8	5664.5	T.O.
16	144.5	16035	T.O.	T.O.	T.O.	11723	T.O.	T.O.	T.O.	10535	T.O.	T.O.	T.O.

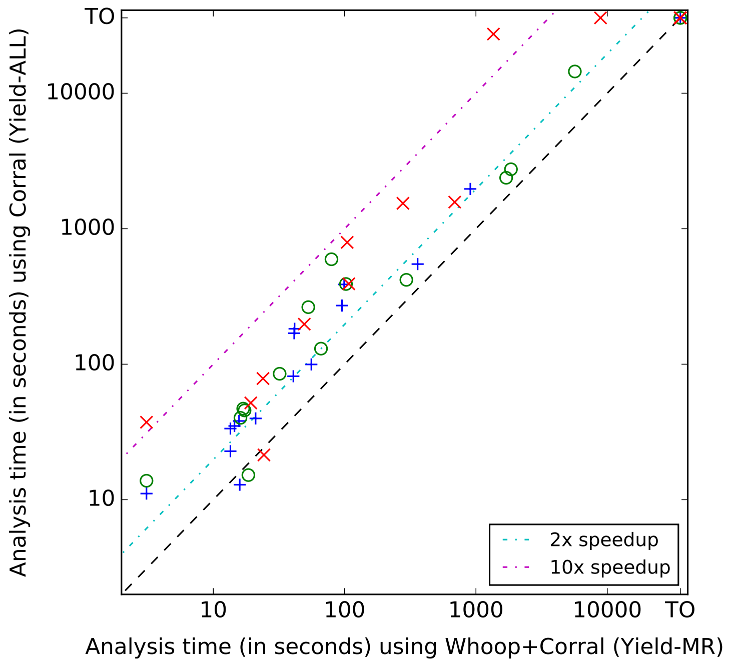

Figure 3.11: Scatter plot showing the runtime speedups that Corral achieves using Whoop with the Yield-MR instrumentation when applied to the 16 Linux 4.0 device driver benchmarks. The symbols $+$, $\circ$, and $\times$ represent a context-switch bound of 2, 5, and 9, respectively.

driver, while Whoop manages to analyse the same driver in 144.5 seconds. We believe the reason behind this is that r8169 has deeply-nested recursion in some of its entry points, which puts burden on the stratified inlining that Corral performs. This is not an issue for Whoop, which uses procedure summarisation. This shows that Whoop has value as a standalone analyser.

Using the race-freedom guarantees from Whoop, we managed to significantly accelerate Corral in most of our Linux driver benchmarks; the best results were achieved using Yield-MR. Figure 3.11 shows that most speedups using Yield-MR are between 1.5$\times$ and 10$\times$; in ssu100 and pc8736x_gpio with a csb of 9 we managed to achieve a speedup of 12$\times$ and 20$\times$, respectively. We noticed that a larger csb typically results in greater speedups when exploiting Whoop. This is expected as time complexity grows exponentially with csb, and hence applying Whoop helps more at a larger csb.[22]

However, there are also cases where Whoop might slow down Corral. We noticed this

[22]Note, though, that this result is a bit artificial, since in our experience most bugs can be found with a small csb.

Table 3.3: Aggregate results from running Corral and Whoop + Corral on 1016 concurrent C programs from SVCOMP'16. The competition score for the concurrency category (where these benchmarks are taken from) has a maximum score of 1240. This score drastically reduces with every false result. Read the official competition rules (`http://sv-comp.sosy-lab.org/2016/rules.php`) for more information about the scoring.

Configurations	**csb**	**Time (sec)**	**#False Results**	**SVCOMP Score**
Corral	2	111394.3	12	784
Whoop + Corral		90760.8	8	918
Corral	3	358728.4	7	748
Whoop + Corral		129258.2	5	993

in the sx8 Linux driver (and also in some SVCOMP'16 benchmarks, see below): Whoop verified 46 out of 47 memory regions, but did not fully verify any of the pairs; Corral, on the other hand, analysed the only two pairs of the driver in just 21.4 seconds (csb of 9). We believe that in this case the overhead of running Whoop outweighed the benefits of using Yield-EPP or Yield-MR.

Table 3.3 shows aggregate results from running standalone Corral and the combination of Whoop and Corral (using the Yield-MR instrumentation) on the 1016 SVCOMP'16 benchmarks. The combination of Whoop and Corral analysed these benchmarks 2.78× faster than Corral with a csb of 3, and 1.23× faster with a csb of 2. Figure 3.12, which plots the corresponding speedups, shows that a lot of the SVCOMP'16 benchmarks are analysed slower with Whoop + Corral, than with just Corral, when using a csb of 2. Corral analyses these SVCOMP'16 benchmarks relatively fast (typically < 100 seconds per benchmark) when using a csb of 2, and thus we believe that Whoop causes additional overhead, without providing tangible benefits. However, when increasing the csb to 3, we notice that the number of programs that run slower with Whoop + Corral significantly decreases. As discussed in the case of our Linux driver benchmarks, we believe that as the csb increases, the benefit of applying Whoop also increases.

In the SVCOMP'16 experiments, we disabled the race-checking instrumentation that Whoop performs (see §3.5.1), since the SVCOMP'16 benchmarks include their own safety-checking assertions. In Table 3.3, we see that using Whoop reduces the number of false results reported by Corral *without* missing any real bugs, which increases the overall competition score.[23] Whoop achieves this by eliminating many of the timeouts that we

[23]Note that increasing the context-switch bound also helps to reduce the number of false results (as seen

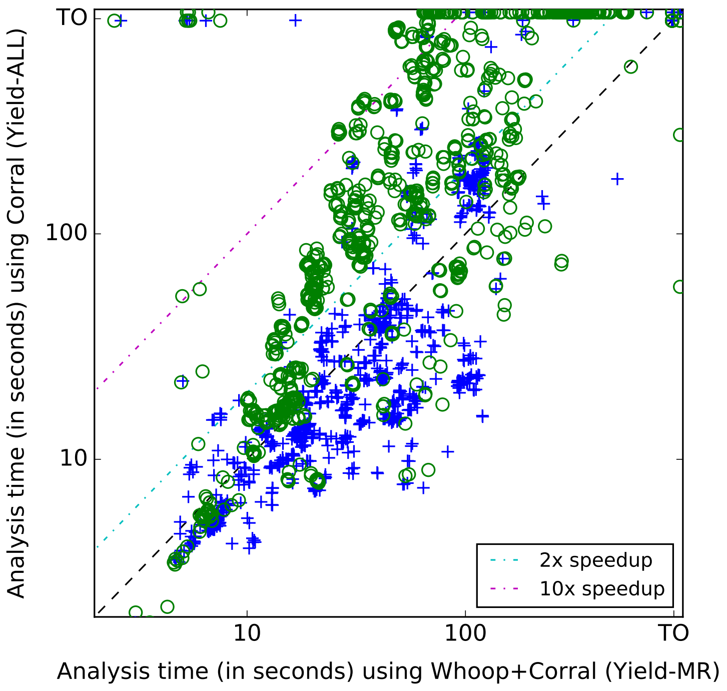

Figure 3.12: Scatter plot showing the runtime speedups that CORRAL achieves using WHOOP with the Yield-MR instrumentation when applied to the 1016 concurrent C programs from the SVCOMP'16 competition. The symbols + and ○ represent a context-switch bound of 2 and 3, respectively.

experienced by using standalone CORRAL (see Figure 3.12). Our experimental results show that WHOOP is valuable not just for detecting data races, but also for speeding up generic concurrency bug-finding, when combined with a tool such as CORRAL.

3.7 Related Work

Regression testing [MSB11] is widely used for checking device driver functionality. Examples of regression testing suites for drivers include the Linux USB testing project [Pro07] and Microsoft's NDISTest [Mic13]. Regression testing, however, is rarely exhaustive. The trickiest bugs can be easily missed if the test harness does not check all possible execution paths or does not simulate all possible environmental behaviours. The later is especially important for inter-system interactions such as in the case of device drivers. Another disadvantage of regression testing is that it is not well suited for discovering concurrency bugs. Such bugs typically manifest in few specific execution paths, and without any control over

in Table 3.3), as it allows CORRAL to investigate deeper paths, and thus find additional bugs.

thread scheduling, the threads will interleave in a nondeterministic way [MQB+08].

Static data race analysis is a promising technique for detecting data races in drivers. Warlock [Ste93] and LockLint [Ora10] are two notable static race analysers. In comparison to Whoop, these two tools rely heavily on user annotations. Prior works on static data race checking also include the following. Choi et al. [CLL+02] combine static analysis and runtime access caching to speed up dynamic race detection. Kahlon et al. [KYSG07] use a divide-and-conquer algorithm that partitions all pointers of a program that do not alias in disjoint sets to achieve scalability; more recently, they used abstract interpretation to achieve a sound partial-order reduction on the set of thread interleavings and statically reduce the number of false race warnings [KSG09]. Das et al. [DSR15] employ interprocedural alias analysis and verifiable user annotations to split programs into disjoint sections, based on non-communicating accesses of shared data, and eliminate redundant checks during dynamic data race detection. Whoop uses symbolic lockset analysis, which involves generating verification conditions and discharging them to a theorem prover, and then employs Corral to filter out false races.

Most related to our lockset analysis are the static lockset analysers RELAY [VJL07] and Locksmith [PFH06]. However, both tools have several limitations. RELAY found 5022 warnings when analysing the Linux 2.6.15 kernel, with only 25 of them being true data races. To tackle this issue, RELAY employs unsound post-analysis filters and, hence, can also filter out real bugs. Locksmith was successfully applied to several small applications that use POSIX threads and also to 7 medium-sized Linux device drivers, but the authors reported that the tool was unable to run on several large programs, hinting at its limited scalability. Whoop aims to achieve scalability *and* precision: the first via novel symbolic lockset analysis, and the second by accelerating Corral, an industrial-strength precise bug-finder for concurrent programs.

The Goblint [VAR+16] static data race detector is a very recent approach, that shares some similarities with Whoop. The main advantage of Goblint is that it provides support for dealing with complex locking schemes (e.g. involving an array of locks), which Whoop does not currently support. However, the use of complex locking schemes in device drivers is discouraged by Linux experts and is rarely encountered (see discussion in §3.4.4).

Our pairwise approach to analysing driver entry points, employing abstraction to model additional threads, was inspired by the *two-thread reduction* used by GPUVerify to analyse data-parallel OpenCL and CUDA kernels [BBC+14, BCD+15]. The idea of pairwise analysis of components in complex systems has been broadly applied, notably in model checking of cache coherence protocols [McM99, CMP04, TT08].

3.8 Summary

In this chapter we presented Whoop, a new automated approach for detecting all possible data races in device drivers. Compared to traditional data race detection techniques that are based on a happens-before analysis and typically attempt to explore as many thread interleavings as possible (and thus face code coverage and scalability issues), Whoop uses a novel over-approximation and symbolic lockset analysis, which scales well. Exploiting the race-freedom guarantees provided by Whoop, we showed that we can achieve a sound partial-order reduction that can significantly accelerate Corral, an industrial-strength concurrency bug-finder.

4 Safer Asynchronous Event-Driven Programming with P#

In this chapter, we present the design and implementation of P# [DDK$^+$15, DMT$^+$16], a new asynchronous event-driven programming language that is *co-designed* with static data race analysis and controlled testing techniques. P# programs execute in shared memory, and exchanged events can contain heap references. This enables efficient message-passing without the need for deep-copying or marshalling, but it also leads to aliasing that can result in hard-to-detect data races. This problem has been acknowledged in many previous actor-based message-passing systems [KSA09]. Existing solutions include runtime support for on-the-fly data race detection [GB13], which can limit system responsiveness by imposing a runtime overhead, and type systems that disallow data races by design [SM08, HO10, CDBM15], which are often complex and restrictive, hampering reuse of legacy code in a fully-featured programming language.

To tackle this problem, we designed a static data race analysis that leverages the semantics of P# to track *ownership* of heap objects. We show experimentally that our analysis is effective in detecting (and proving absence of) data races in practical examples, and has higher precision than SOTER, a similar static analysis proposed in previous work [NKA11]. A race-free P# program can still suffer from serious bugs (e.g. uncaught exceptions and assertion violations); the nondeterminism due to the asynchronous message-passing can make such bugs hard to detect. To address this problem, we embedded a *controlled concurrency testing* [God97, MQB$^+$08, TDB16] framework inside P#, which serialises the asynchronous execution of a P# program and explores different event-handler interleavings in a controlled manner, facilitating deterministic replay of bugs.

The goal of this work is to unify language design, data race analysis and testing under a common framework for building highly-reliable asynchronous reactive programs. We have used P# to develop, analyse and test multiple widely-used distributed protocols, including Paxos [Lam98] and Raft [OO14]. P# has been also successfully used inside Microsoft by a number of researchers, developers and interns to detect subtle bugs in complex distributed systems built on top of Microsoft Azure [DMT$^+$16]. In this chapter, we focus on the P#

language and the accompanying analysis and testing techniques, while in Chapter 5 we discuss an approach for testing real-world distributed systems with P#.

The main contributions of this chapter are:

- We present P#, a new asynchronous event-driven language that is co-designed with a static data race analysis and controlled testing infrastructure.
- We formulate a static analysis for proving absence of data races in programs written in P#, and show that our analysis improves on previous work in terms of scalability and precision.
- We develop systematic and controlled concurrency testing techniques that leverage the P# language and static data race analysis to achieve higher efficiency than a state-of-the-art concurrency testing tool.

Related Published Work The material presented in this chapter is based on work that was originally published in the 36th Annual ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15) [DDK+15].

4.1 Overview of the P# Approach

The programming model of P# is based on *communicating state-machines*. These state-machines are similar to the actors [HBS73, Agh86] used in languages like Scala and Pony. A P# program consists of one or more *state-machines* that communicate by asynchronously sending and receiving *events* (also known as *messages*), an approach that is commonly used in building asynchronous reactive programs, such as embedded systems, web services, mobile applications and distributed systems.

A P# state-machine is similar to a class in object-oriented programming: it contains methods and fields, it can be instantiated, and it supports inheritance. However, a machine also differs from a class: it can only execute sequential code (i.e. intra-machine concurrency is not allowed); it must isolate its local data (i.e. cannot contain fields and methods with public visibility); and it must declare one or more machine *states* (also known as *roles* in other actor programming languages). Each machine state can register *actions* to handle incoming events (one action per event).

Each P# machine instance contains an *event queue* that is used to enqueue any event that is being sent to this machine. P# machines run concurrently with each other, each executing an event-handling loop that dequeues the next event from the queue, and handles

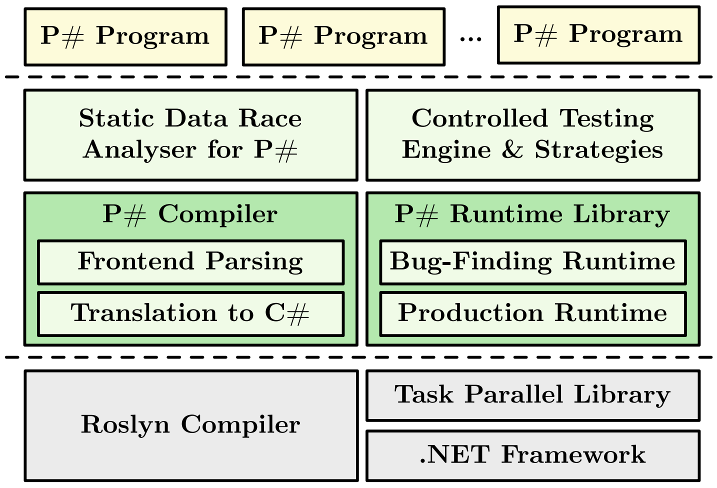

Figure 4.1: The high-level architecture of the P# project.

it by invoking the registered action. An action might transition the machine to a new state, create a new machine instance, send an event to a machine, access a field or call a method. In P#, creating a machine instance and sending an event are *non-blocking* operations. In the case of the send operation, the event is simply enqueued into the queue of the target machine, which will asynchronously dequeue and handle the event.

Figure 4.1 shows the core components of the P# framework, which include the compiler, the static analyser, the runtime library, and the controlled testing engine. The P# compiler, which is built on top of the Microsoft Roslyn compiler,[1] is responsible for parsing a P# program and compiling it into a .NET executable. The static analyser (also built on top of Roslyn) is responsible for detecting data races in P# programs (see §4.4). The P# runtime library, which is built on top of the .NET Task Parallel Library [LSB09], is responsible for implementing the operational semantics of P#. Finally, the testing engine (see §4.5) is responsible for exploring P# program executions to detect bugs (e.g. assertion failures and uncaught exceptions). The P# language has been open-sourced, and can be dowloaded from its source code repository on GitHub.[2]

In its essence, P# is an extension of the mainstream C# [Mic16a] language for highly-reliable asynchronous programming. Because P# is built on top of C#, it allows developers to easily blend P# and C# source code; this not only lowers the overhead of learning a new language, but also allows P# to integrate with existing .NET projects.

[1]`https://github.com/dotnet/roslyn`
[2]`https://github.com/p-org/PSharp`

4.1.1 Static Data Race Analysis

By default, a P# program executes in shared memory and exchanged events can contain references to heap objects.[3] Although this enables efficient message-passing communication without the need for deep-copying or marshalling, it leads to aliasing that can result in hard-to-detect data races between concurrently executing event-handlers. To tackle this problem, we co-designed P# with a sound static data race analysis (see §4.4) that leverages (i) the state-machine structure of the language and (ii) the absence of intra-machine concurrency to track *ownership* of objects. A machine assumes ownership of any object it creates or receives from another machine, and it gives up ownership of any object it sends to another machine. As long as each object has a unique owner, data races between machines cannot occur. In §4.6, we show experimentally that our static analysis is effective in detecting (and proving absence of) data races in practical examples, and has higher precision than a similar static analysis proposed in earlier work [NKA11].

4.1.2 Controlled Testing

A race-free P# program can still suffer from serious bugs (e.g. assertion violations and uncaught exceptions); the nondeterminism due to the asynchronous communication between state-machines can make such bugs hard to detect. To deal with this problem, the P# runtime comes with an embedded controlled concurrency testing [God97, MQB$^+$08, TDB16] framework, which serialises the execution of a P# program and explores different event-handler interleavings in a controlled manner, facilitating deterministic replay of bugs.

We optimise controlled testing by leveraging the P# semantics and the results from our static data race analysis: for race-free programs it suffices to consider interleavings at event-level granularity, not at the granularity of individual memory accesses.[4] This leads to improved performance compared to the industrial-strength CHESS controlled testing tool [MQB$^+$08], which also supports the analysis of C# (and thus P#) programs.

As evidence of our claim that P# enables high-reliability asynchronous programming, we evaluate P# and our analysis and testing infrastructure using several distributed protocols, and compare the P# controlled testing engine with CHESS (see §4.6). In Chapter 5, we present a case study of applying P# to Azure Storage vNext, an industrial-scale distributed storage system that is being developed by Microsoft on top of the Azure cloud computing

[3]The existing P# runtime only supports message-passing in shared memory; the developer has to use external networking libraries to execute a P# program in a distributed setup.

[4]Our current implementation does not support exploring interleavings at the granularity of individual memory accesses. In this chapter, we test a P# program *only* after proving it as race-free. In Chapter 5, we do not use the static analyser; instead, we assume that P# programs are race-free.

platform; P# managed to detect a previously undiscovered serious bug in vNext (see §5.3). P# has also been used by two other Microsoft teams to successfully detect bugs in their systems [DMT+16].

4.1.3 Relation to the P Programming Language

Our work is inspired by P [DGJ+13], a domain-specific language for writing high-reliability asynchronous reactive programs. The P language was used inside Microsoft to implement the core of the USB device driver stack that ships with the Windows 8 operating system. During the development process, hundreds of bugs were found and fixed [DGJ+13].

P provides a simple surface syntax for writing communicating state-machines, which P# borrows (see §4.2.1). Note that P only supports operations over scalar values, as well as (nested) tuples, lists, and maps of scalar values. In contrast, P# supports the use of arbitrary C# data types. To get around its limited language features, P exposes a foreign-function interface for encoding functionality that cannot be implemented in P. The selling point of P is that developers can test P programs using the Zing model checker [AQR+04]. However, *only* program components written in P, *not* the foreign code, can be checked using Zing; during testing, the P compiler substitutes the foreign code with user-provided *model* state-machines.

Although the P approach suffices for detecting protocol-level bugs, it provides no guarantees about the actual foreign code implementation; in contrast, the P# approach enables *whole-program* controlled testing. Moreover, since P# is built on top of C#, our approach offers a similar programming experience as other .NET languages, including debugging support in Visual Studio (Microsoft's integrated development environment).

4.2 Writing Asynchronous Programs in P#

In this section, we give an overview of how to develop asynchronous event-driven programs using P#. First, we present the core syntax of our language (§4.2.1). We then discuss how C# can interoperate with P# (§4.2.2). Next, we give a simple example of a P# program, and describe how it executes (§4.2.3). Finally, we discuss how a P# program is compiled to an executable using the P# compiler (§4.2.4).

4.2.1 Syntax of the P# Language

We present below the core syntax of P#, which is based on the syntax of the P [DGJ+13] language. In both P and P#, state-machines are first-class citizens (similar to how classes are first-class citizens in C#). A machine can be declared as follows:

```
machine Server { ... }
```

The above code snippet declares a P# machine named `Server`. Machine declarations are similar to class declarations in C#, and thus can contain an arbitrary number of fields and methods. For example, the below code snippet declares the field `client` of type `machine`. An object of this type contains a reference to a machine instance.

```
machine Server {
  machine client;
}
```

The main difference between a class and a machine declaration is that the latter must also declare one or more *states*:

```
machine Server {
  machine client;
  start state Init { ... }
  state Active { ... }
}
```

The above declares two states in the `Server` machine: `Init` and `Active`. The P# developer must use the `start` modifier to declare an *initial* state, which will be the first state that the machine will transition to upon instantiation. In this example, the `Init` state has been declared as the initial state of `Server`. Note that only a single state is allowed to be declared as an initial per machine. A `state` declaration can optionally contain a number of state-specific actions, as seen in the following code snippet:

```
state SomeState {
  entry { ... }
  exit { ... }
}
```

A code block indicated by `entry` { ... } denotes an action that will be executed when the machine transitions to the state, while a code block indicated by `exit` { ... } denotes an action that will be executed when the machine leaves the state. Actions in P# are essentially C# methods without input parameters and a `void` return type. P# actions can contain arbitrary P# and C# statements. However, since we want to explicitly declare all sources of asynchrony using P#, we only allow the use of *sequential* C# code inside a P# machine.[5] An example of an `entry` action is the following:

[5]In practise, we just assume that the C# code is sequential, as it would be very challenging to impose this rule in real life programs (e.g. a developer could use an external library).

```
entry {
  this.client = create(Client, this);
  send(this.client, Ping);
  raise(Unit);
}
```

The above action contains the three most important P# statements. The `create` statement is used to asynchronously create a new instance of a machine (in this case the `Client` machine). A reference to this instance is stored in the `client` field. Next, the `send` statement is used to send an event (in this case the event `Ping`) to a target machine (in this case the machine whose address is stored in the field `client`). When an event is sent from one machine to another, it is enqueued in the event queue of the target machine, which can then, at a later point in time, dequeue the received event and handle it asynchronously from the sender machine. Finally, the `raise` statement is used to send an event to the caller machine (i.e. to itself). When a machine raises an event, the raised event is not enqueued as in the case of `send`; instead, the machine terminates execution of the enclosing code block and handles the event immediately.

In P#, events (e.g. `Ping` and `Unit` in the above example) can be declared as follows (either inside or outside the scope of a machine):

```
event Ping;
event Unit;
```

A P# machine can send data (scalar values or references) to another machine, either when creating a new machine instance (using `create`) or when sending an event (using `send`). A machine can also send data to itself (e.g. for processing in a later state) using `raise`. In the previous example, `Server` sends `this` (i.e. a reference to the current machine instance) to `client`. The receiver (in our case `client`) can retrieve the sent data by using the keyword `payload` and casting it to its expected type (in this case the `machine` type). As discussed earlier, the `create` and `send` statements are non-blocking. The P# runtime will take care of all the underlying asynchrony using the Task Parallel Library and thus the developer does not need to explicitly create and manage tasks.

Besides the `entry` and `exit` declarations, all other declarations inside a P# state are related to *event-handling*, which is a key feature of P#. An event-handler declares how a P# machine should *react* to a received event. One such possible reaction is to create one or more machine instances, send one or more events, or process some local data. The two most important event-handling declarations in P# are the following:

```
state SomeState {
  on Unit goto AnotherState;
```

```
    on Pong do SomeAction;
  }
```

The declaration `on Unit goto AnotherState` indicates that when the machine receives the `Unit` event in `SomeState`, it must handle `Unit` by exiting the state and transitioning to `AnotherState`. The declaration `on Pong do SomeAction` indicates that the `Pong` event must be handled by invoking the action `SomeAction`, and that the machine will remain in `SomeState`. P# also supports *anonymous* event-handlers. For example, the declaration `on Pong do` { ... } is an anonymous event-handler, which states that the block of statements between the braces must be executed when event `Pong` is dequeued. Each event can be associated with at most one handler in each particular state of a machine. If a P# machine is in a state `SomeState` and dequeues an event `SomeEvent`, but no event-handler is declared in `SomeState` for `SomeEvent`, then P# will throw an appropriate exception.

Besides the above event-handling declarations, P# also provides the capability to *defer* and *ignore* events in a particular state:

```
  state SomeState {
    defer Ping;
    ignore Unit;
  }
```

The declaration `defer Ping` indicates that the `Ping` event should not be dequeued while the machine is in the state `SomeState`. Instead, the machine should skip over `Ping` (without dropping `Ping` from the queue) and dequeue the next event that is not being deferred. The declaration `ignore Unit` indicates that whenever `Unit` is dequeued while the machine is in `SomeState`, then the machine should drop `Unit` without invoking any action.

P# also supports specifying invariants (i.e. assertions) on the local state of a machine. The developer can achieve this by using the `assert` statement, which accepts as input a predicate that must always hold at that specific program point, e.g. `assert(k == 0)`, which holds if the integer `k` is equal to 0. In §4.5, we describe how the P# runtime explores executions of a P# program to detect assertion violations. In Chapter 5, we discuss how P# can be used to specify and check global safety and liveness properties.

4.2.2 Interoperability Between P# and C#

Because P# is built on top of the C# language, the entry point of a P# program (i.e. the first machine that the P# runtime will instantiate and execute) must be explicitly declared inside a host C# program (typically in the `Main` method), as follows:

```
  using Microsoft.PSharp;
```

```
public class HostProgram {
  static void Main(string[] args) {
    PSharpRuntime.Create().CreateMachine(typeof(Environment));
    Console.ReadLine();
  }
}
```

The developer must first import the P# runtime library (`Microsoft.PSharp.dll`), then create a `PSharpRuntime` instance, and finally invoke the `CreateMachine` runtime method to instantiate the first P# machine (`Environment` in the above example).

The `CreateMachine` method is part of the .NET interoperability API (a set of methods for calling P# from native C# code) that is exposed by `PSharpRuntime`. This method accepts as a parameter the type of the machine to be instantiated, and returns an object of the `MachineId` type, which contains a reference to the created P# machine. Because `CreateMachine` is an asynchronous method, we call the `Console.ReadLine` method, which pauses the main thread until a console input has been given, so that the host C# program does not exit prematurely.

The `PSharpRuntime` .NET interoperability API also provides the `SendEvent` method for sending events to a P# machine from native C#. This method accepts as parameters an object of type `MachineId`, an event, and an optional payload. Although the developer has to use `CreateMachine` and `SendEvent` to call P# code from native C#, the opposite is straightforward, as it only requires accessing a C# object from P# code.

4.2.3 An Example Program

The following P# program shows a `Client` machine and a `Server` machine that communicate asynchronously by exchanging `Ping` and `Pong` events:

```
event Ping; // The Client sends this event to the Server
event Pong; // The Server sends this event to the Client
event Unit; // This event is used for local state-transitions

machine Client {
  machine server;

  start state Init {
    entry {
      this.server = (machine)payload; // Receives reference to Server
      raise(Unit); // Sends an event to itself
    }
    on Unit goto Active; // Performs a state-transition
14   }
15
16   state Active {
17     entry {
18       SendPing();
19     }
20     on Pong do SendPing; // Handles Pong with the SendPing action
21   }
22
23   void SendPing() {
24     send(this.server, Ping, this); // Sends a Ping event to the Server
25   }
26 }
27
28 machine Server {
29   start state Active {
30     on Ping do {
31       machine client = (machine)payload; // Receives reference to Client
32       send(client, Pong); // Sends a Pong event to the Client
33     };
34   }
35 }
36
37 machine Environment {
38   start state Init {
39     entry {
40       machine server = create(Server); // Instantiates the Server
41       machine client = create(Client, server); // Instantiates the Client
42     }
43   }
44 }
45
46 public class HostProgram {
47   static void Main(string[] args) {
48     PSharpRuntime.Create().CreateMachine(typeof(Environment));
49     Console.ReadLine();
50   }
51 }
```

In this example, the program starts by creating an instance of the `Environment` machine (line 48). The implicit constructor of each P# machine initialises all machine-related data structures, including the event queue, a set of available states, and a map from events to event-handlers per machine state. Note that these data structures are accessible only from inside the P# runtime; the user has no direct access.

After Environment has been instantiated, the P# runtime executes the entry action of the initial (Init) state of Environment, which first creates an instance (server) of the Server machine (line 40), and then creates an instance (client) of the Client machine with the server reference as a payload (line 41). The Environment machine is essentially used to kickstart the program.

After server has been instantiated, it starts executing asynchronously. The initial state (Active) of the Server machine does not contain an entry action, and thus server waits until it receives a Ping event (line 30). For performance, the P# runtime ensures that a machine waits without blocking an underlying thread if its event queue is empty.

After client has been instantiated, the P# runtime executes the entry action of the initial (Init) state of the Client machine, which stores the received payload (which is a reference to the Server machine) in the server field (line 10), and then raises Unit (line 11). As mentioned earlier, when a machine calls raise, it exits the currently executing action, and immediately handles the raised event (bypassing the queue). In this case, the Client machine handles Unit by transitioning to the Active state (line 13). In the new state, client calls the SendPing method to send a Ping event to server (line 24).

When server receives and dequeues a Ping event, it stores the event payload (which is a reference to the Client machine) in the client local variable (line 31), and then sends Pong to client (line 32). The Client machine handles Pong by sending a new Ping event to server. This asynchronous exchange of Ping and Pong continues indefinitely.

4.2.4 Compiling P# Programs

To compile the previous example (as well as any other P# program), the developer must use the Roslyn-based P# compiler (see Figure 4.1). The compilation process consists of two phases: *parsing* and *rewriting*. In the parsing phase, the input P# program is parsed using a recursive-descent parser [ALSU06] (which we manually implemented on top of the Roslyn compiler APIs) to produce an abstract syntax tree (AST). In the rewriting phase, the P# compiler traverses the produced AST, and rewrites all P# statements to native (intermediate) C# code (all C# statements are skipped, since they are already written in C#). Finally, the P# compiler invokes the Roslyn compiler to build the intermediate C# program, link it with the P# runtime library, and produce a .NET executable.

$$
\begin{array}{lll}
\text{stmt} & s & ::= \mathsf{send}_{dst}\ evt(v) \\
 & & \quad \mid \mathsf{raise}\ evt(v) \\
 & & \quad \mid \mathsf{return}\ v \\
 & & \quad \mid v := v \\
 & & \quad \mid v := c \\
 & & \quad \mid v := v\ op\ v \\
 & & \quad \mid \mathsf{this}.v := v \\
 & & \quad \mid v := \mathsf{this}.v \\
 & & \quad \mid v := \mathsf{new}\ class \\
 & & \quad \mid v := v.m(\overrightarrow{v}) \\
 & & \quad \mid \mathsf{if}\ (v)\ ss\ \mathsf{else}\ ss \\
 & & \quad \mid \mathsf{while}\ (v)\ ss \\
\text{stmts} & ss & ::= s; ss \mid \varepsilon \\
\text{vdecl} & vd & ::= \text{type}\ v; \\
\text{mdecl} & md & ::= \text{type}\ m(\overrightarrow{vd})\ \{\overrightarrow{vd}\ \ ss\} \\
\text{cdecl} & cd & ::= \mathsf{class}\ class\ \{\overrightarrow{vd}\ \ \overrightarrow{md}\}
\end{array}
$$

Figure 4.2: The syntax of a simple asynchronous event-driven, object-oriented language.

4.3 Semantics of the P# Language

In this section, we present the core semantics of P#, on which we build when formulating the static data race analysis for P# programs in §4.4. The semantics is based on the *event-driven automata* [DGM14] formalism, which can be seen as a core calculus of P [DGJ+13] minus (for simplicity) the ability to dynamically instantiate new automata (i.e. machines). The actual P# implementation has a feature set on par with the P language, and supports dynamic machine instantiation.

4.3.1 A Simple Object-Oriented Language

We have implemented P# machines as classes in the mainstream object-oriented language C#. In this section, we formulate only the core semantics of our language using a simple asynchronous event-driven, object-oriented language (see Figure 4.2), which is based on the core syntax of P# (see §4.2.1). For brevity, we omit language features such as handling received events with actions, deferred/ignored events, exceptions and inheritance, although the implementation of P# supports all these features. We use vector notation to denote sequences of declarations (e.g. $\overrightarrow{vd}$ denotes a sequence of variable declarations).

Each class declaration (cdecl) consists of a sequence of member variable declarations $\overrightarrow{vd}$, followed by a sequence of method declarations $\overrightarrow{md}$. The type of each variable is either scalar (ensuring that the variable can store primitive values such as integers or floats), or a reference (such that the variable can store references to heap-allocated objects). Member

```
class node {
  int val; node next;

  int get_val() {                    |   node get_next() {
    int ret;                         |     node ret;
    ret := this.val;                 |     ret := this.next;
    return ret;                      |     return ret;
  }                                  |   }
                                     |
  void set_val(int v) {              |   void set_next(node n) {
    this.val := v;                   |     this.next := n;
  }                                  |   }
}
```

Figure 4.3: Class implementing a node of a linked list. For brevity, we omit return statements from the bodies of methods with void return type.

variables are not directly accessible: a member v of the current class can only be accessed through an expression of the form this.v, while a member of another class is only accessible via appropriate method calls.

A method declaration (mdecl) contains a sequence of formal parameters $\overrightarrow{vd}$, and its body consists of a sequence of local variable declarations $\overrightarrow{vd}$ followed by a sequence of statements ss, with ε denoting the empty sequence. In each statement (stmt), v refers to a local variable, a formal parameter of a method, or this, with this being a constant reference to the class in which the statement occurs. We denote by c any literal scalar value and by op any operation over scalars. The $:=$ symbol denotes an assignment.

Using the syntax of our simple object-oriented language, we build a running example of a machine (represented as a class, see §4.3.3) that is responsible for managing a linked list. We use this example to illustrate the static data race analysis for P# programs in §4.4. We start by describing a class that implements a linked list node (see Figure 4.3). Each node contains two member variables: an integer value and a reference to the next node in the linked list; an accessor and a mutator are defined for both.

4.3.2 Semantics of Basic Statements

We now present the small-step operational semantics [Plo04] of the individual statements in our simple object-oriented language. We omit the rules for send$_{dst}$ $evt(v)$ and raise $evt(v)$, which we discuss in the context of state-machine transitions (see §4.3.3).

All rules that we describe below are defined over tuples (ℓ, h, S, ss). The *local store* ℓ is a map from local variables to values (where values have either scalar or reference type), the *heap* h is a map from (reference, member variable)-pairs to values, S is a *call stack*, and ss is a *sequence of statements* to be executed. A call stack is a sequence of tuples (ℓ_s, v_s, ss_s), with ℓ_s the local store of the caller, v_s the variable in which the *return value* of the callee should be stored, and ss_s the *sequence of statements* to be executed after the callee returns to the caller.

In our rule notation, $\ell[v' \mapsto z]$ denotes a map where all keys keep their existing value, besides v', which is updated to z. The $\rightarrow$ symbol denotes a transition from the program state before a rule has been applied, to the program state after the associated rule has been applied. The $:$ symbol (e.g. in $(\ell_s, v_s, ss_s) : S$) denotes a concatenation. Finally, the $;$ symbol (e.g. in $s; ss$) denotes a sequence.

In accordance with the above notation, the Return rule pops the top frame from the call stack, updates v_s and ensures ss_s is executed next:

$$\frac{\ell' = \ell_s[v_s \mapsto \ell(v)]}{(\ell, h, (\ell_s, v_s, ss_s) : S, \mathsf{return}\, v; ss) \rightarrow_s (\ell', h, S, ss_s)} \ (\textsc{Return})$$

The Var-Assign rule is used for local variable assignment. The rule updates the local variable v with the value of local variable v':

$$\frac{\ell' = \ell[v \mapsto \ell(v')]}{(\ell, h, S, v := v'; ss) \rightarrow_s (\ell', h, S, ss)} \ (\textsc{Var-Assign})$$

Similar to the previous rule, the Const-Assign rule updates the local variable v with the constant c:

$$\frac{\ell' = \ell[v \mapsto c]}{(\ell, h, S, v := c; ss) \rightarrow_s (\ell', h, S, ss)} \ (\textsc{Const-Assign})$$

The Op-Assign rule applies an operation *op* (see Figure 4.2) to the values of the local variables v_1 and v_2, and updates the local variable v with the result:

$$\frac{c = \ell(v_1)\ op\ \ell(v_2) \qquad \ell' = \ell[v \mapsto c]}{(\ell, h, S, v := v_1\ op\ v_2; ss) \rightarrow_s (\ell', h, S, ss)} \ (\textsc{Op-Assign})$$

The Mbr-Assign-To rule is used for member variable assignment. The rule updates the value of the member variable v of the object in the heap that is pointed to by this with the value of the local variable v':

$$\frac{h' = h[(\mathsf{this}, v) \mapsto \ell(v')]}{(\ell, h, S, \mathsf{this}.v := v'; ss) \rightarrow_s (\ell, h', S, ss)} \ (\textsc{Mbr-Assign-To})$$

Conversely, rule Mbr-Assign-From updates the value of the local variable v with the value of the member variable v':

$$\frac{\ell' = \ell[v \mapsto h(\mathsf{this}, v')]}{(\ell, h, S, v := \mathsf{this}.v'; ss) \rightarrow_s (\ell', h, S, ss)} \text{ (Mbr-Assign-From)}$$

The New-Assign rule creates a new instance of a class *class* that gets associated with an unassigned reference *ref*. Given *ref*, all member variables $\mathrm{mv}(\mathit{class})$ of *class* are allocated by setting them to an undefined value $\bot$, and v (which is the new instance of *class*) is updated with *ref*. No constructor arguments are passed upon instance creation. Instead, we assume that instance creation is always followed by one or more method calls that set the members to properly defined values, thus making the exact value of $\bot$ irrelevant.

$$\frac{\begin{array}{c}\neg\exists v : (\mathit{ref}, v) \in \mathrm{dom}(h) \\ h' = h[(\mathit{ref}, mv) \mapsto \bot]_{mv \in \mathrm{mv}(\mathit{class})} \qquad \ell' = \ell[v \mapsto \mathit{ref}]\end{array}}{(\ell, h, S, v := \mathsf{new}\ \mathit{class}; ss) \rightarrow_s (\ell', h', S, ss)} \text{ (New-Assign)}$$

Assuming a method m is defined as follows:

$$\text{type } m(\text{type}_1\, \mathit{fp}_1, \ldots, \text{type}_n\, \mathit{fp}_n)\ \{\overrightarrow{v_d}\ \ ss_m\},$$

the Method-Call rule creates a new local store starting from the empty map $\bot$ (not to be confused with the undefined value that we also denote by $\bot$). The rule sets this to the value of v' (which is the object whose method m is being invoked), initializes each formal parameter fp_i with the value of v_i, and sets each local variable v_d to some undefined value $\bot$. Finally, a new stack frame is created for the caller and execution continues with the statements ss_m of the callee. As in the case of class instantiation, we assume that each local variable is assigned a well-defined value by the callee before making use of the stored value.

$$\frac{\ell_{\mathit{fp}} = \bot[\mathsf{this} \mapsto \ell(v')][\mathit{fp}_i \mapsto \ell(v_i)]_{1 \le i \le n} \qquad \ell' = \ell_{\mathit{fp}}[v_d \mapsto \bot]_{v_d \in \overrightarrow{v_d}}}{(\ell, h, S, v := v'.m(v_1, \ldots, v_n); ss) \rightarrow_s (\ell', h, (\ell, v, ss) : S, ss_m)} \text{ (Method-Call)}$$

$$\frac{\ell(v)}{(\ell, h, S, \mathsf{if}\ (v)\ ss_t\ \mathsf{else}\ ss_f; ss) \rightarrow_s (\ell, h, S, ss_t; ss)} \text{ (If-True)}$$

$$\frac{\neg\ell(v)}{(\ell, h, S, \mathsf{if}\ (v)\ ss_t\ \mathsf{else}\ ss_f; ss) \rightarrow_s (\ell, h, S, ss_f; ss)} \text{ (If-False)}$$

The If-True and If-False rules are standard; the value of v is retrieved and the correct branch is executed. The While-True and While-False rules are similar.

$$\frac{\ell(v)}{(\ell, h, S, \mathsf{while}\ (v)\ ss_b; ss) \rightarrow_s (\ell, h, S, ss_b; \mathsf{while}\ (v)\ ss_b; ss)}\ (\textsc{While-True})$$

$$\frac{\neg\ell(v)}{(\ell, h, S, \mathsf{while}\ (v)\ ss_b; ss) \rightarrow_s (\ell, h, S, ss)}\ (\textsc{While-False})$$

4.3.3 Semantics of Machines and Transitions

This section presents the semantics of machines and state-transitions. Given a set of event names *Evt*, a set of classes $\mathcal{C}$, and the set of all possible values *Val* that can be assigned to variables, a *machine* m is defined as a tuple $(class_m, q_m, Q_m, T_m)$, where:

- $class_m \in \mathcal{C}$ is the *class* whose methods define the statements that need to be executed on entry to each state of m;
- q_m is the *initial state* of m and is represented by a method of $class_m$ that does *not* have any arguments;
- Q_m is the set of *non-initial* states of m, each of which is represented by method of $class_m$ that has a *single* argument;
- T_m is the *transition function* of machine m. Given a state $q \in Q_m$ and a queue of events $E \in \mathsf{queue}(Evt \times Val)$, T_m finds the first event $evt(val)$ in E that machine m can handle in state q. The transition function T_m then yields: (i) the next state q' as a function in q and evt; (ii) val; and (iii) an event queue E' identical to E, but with $evt(val)$ removed.

The semantics of an asynchronous reactive *system* (i.e. a program) composed of multiple machines is defined in terms of transitions between *system configurations* (h, M), where h is a heap shared between the machines in the system, and M is a map from (machine) identifiers id to machine configurations. A *machine configuration* is a tuple (m, q, E, ℓ, S, ss), where m is a machine, q is the current state of the machine (represented by a method of $class_m$, as discussed above), E is an event queue, ℓ is a variable store, S is a call stack (as in our simple object-oriented language discussed above), and ss is a sequence of statements that needs to be executed before the next event in E can be handled.

Given a set of (machine) identifiers id, an *initial system configuration* is any system configuration (h, M) such that for each $i \in id$ there exists a machine m with $M(i) = (m, q_m, \varepsilon, \ell_m, \varepsilon, ss_m)$, where ε represents both an empty event queue and an empty call stack, where $\ell_m = \bot[v_m \mapsto \bot]$ for some variable v_m, and where ss_m is defined as the sequence of statements $v_m := \mathsf{new}\ class_m; v_m.q();$. The machine definition m occurs in the tuple, as the RECEIVE and RAISE rules defined below need access to T_m. The sequence of statements ss_m initializes the machine by creating an instance of the appropriate class, retaining a reference in v_m, and executing the method corresponding to the initial state. Once the statements in ss_m have been executed, the machine is ready to handle incoming events by invoking methods of v_m.

We now define the rules for the transitions between machine configurations. The INTERNAL rule employs the rules presented in §4.3.2 to execute a statement for some i. The rule updates the shared heap, and the local variables, call stack, and sequence of statements of i appropriately. Note that this rule requires ss to be non-empty.

$$\frac{\begin{array}{c} M(i) = (m, q, E, \ell, S, ss) \\ (\ell, h, S, ss) \to_s (\ell', h', S', ss') \qquad M' = M[i \mapsto (m, q, E, \ell', S', ss')] \end{array}}{(h, M) \to_t (h', M')} \ (\textsc{Internal})$$

The SEND rule appends the event being sent to the event queue E of the appropriate machine configuration (depending on the dst machine identifier). Note that the machine configuration of i must be updated before appending the event, since a machine i can send an event to itself (i.e. we can have $dst = i$).

$$\frac{\begin{array}{c} M(i) = (m, q, E, \ell, S, \mathsf{send}_{dst}\ evt(v); ss) \qquad M_s = M[i \mapsto (m, q, E, \ell, S, ss)] \\ M_s(dst) = (m', q', E', \ell', S', ss') \qquad M' = M_s[dst \mapsto (m', q', E' : evt(\ell(v)), \ell', S', ss')] \end{array}}{(h, M) \to_t (h, M')} \ (\textsc{Send})$$

A machine m transitions to a new state once no more statements remain to be executed in the current state of m. The RECEIVE rule takes care of performing the transition by employing the transition function T_m of m, and invoking the appropriate method obtained through this function. Note that in this rule, we use a variant of the call statement that accepts a value instead of a variable as an argument.

$$\frac{\begin{array}{c} M(i) = (m, q, E, \ell, S, \varepsilon) \\ T_m(q, E) = (q', val, E') \qquad M' = M[i \mapsto (m, q', E', \ell, S, v_m.q'(val))] \end{array}}{(h, M) \to_t (h, M')} \ (\textsc{Receive})$$

```
class list_manager {
  node list;

  void init() {
    this.list := null;
  }

  void get(id payload) {
    node tmp;
    tmp := this.list;
    send_payload eReply(tmp);
  }

  void add(node payload) {
    node tmp;
    tmp := this.list;
    payload.set_next(tmp);
    this.list := payload;
  }
}
```

Figure 4.4: A class implementing a state-machine that manages a linked list. For brevity, we omit $v := \ldots$ when calling methods with a void return type, and we similarly omit return statements from the bodies of these methods. The special type id denotes a machine identifier, and the special constant null denotes a null reference.

The Raise rule appends the event being raised to the head of the event queue E of the same machine that is raising the event, which means that a raised event is always handled before any other event in the queue.[6] The sequence of statements ss after the raise $evt(v)$ statement is skipped (i.e. ss becomes ε), and the Raise rule performs the state-transition by employing the transition function of m, and invoking the appropriate method obtained through this function, similar to the Receive rule above.

$$\frac{\begin{array}{c} M(i) = (m, q, E, \ell, S, \textsf{raise}\ evt(v); ss) \\ T_m(q, evt(\ell(v)) : E) = (q', val, E) \qquad M' = M[i \mapsto (m, q', E, \ell, S, v_m.q'(val))] \end{array}}{(h, M) \to_t (h, M')} \ (\textsc{Raise})$$

Continuing the example from Figure 4.3, we define a machine *list_manager* that manages a linked list (see Figure 4.4). The *init* method defines the initial state of the *list_manager* machine. The other states of *list_manager* are defined by (i) the get method, which sends an *eReply* event that contains the whole linked list to the machine specified by *payload*, and (ii) the add method, which adds *payload* to the head of the linked list.

Assuming *list_manager* handles events *eAdd* and *eGet*, we define the transition function of *list_manager* as the total function that (i) transitions to *add* when *eAdd* is the event

[6]Note that in the actual implementation of P#, a raised event is not appended to the head of the event queue; instead, the raised event is handled immediately, or an exception is thrown if no appropriate event-handler exists, as discussed in §4.2.1.

at the head of the event queue, and (ii) transitions to *get* when *eGet* is the event at the head of the event queue. Observe that the machine can potentially suffer from a data race, since a reference to the linked list is still held by the machine after being used as a payload in the send statement of the *get* method. In §4.4, we discuss how we can statically detect data races in P# programs.

4.4 Checking for Data Races

Two transitions in a P# program are said to *race*, if they (i) originate from different state-machine instances, (ii) are not separated by any other transition, and (iii) both access the same field of an object with at least one of the accesses being a modifier. Formally, (iii) requires the common field to be accessed using the Mbr-Assign-To and Mbr-Assign-From rules, with at least one being Mbr-Assign-To (see §4.3.2).

If a P# program is free from data races then we can reason about the possible behaviours of the program by assuming that machines interleave with event-level granularity; there is no need to consider interleavings between individual heap access operations. Furthermore, the ability to detect or prove absence of races is valuable since races between machines are typically unintentional and erroneous, because they break the event-based communication paradigm of the language.

To detect possible data races, or to show their absence, we employ a modular, *ownership*-based static data race analysis. Objects are owned by machines, and ownership of an object o is transferred from a machine M to a machine M' upon M sending an event to M' with a (direct or indirect) reference to o. Once ownership has been transferred, M is no longer allowed to access o, as this would violate the fact that M' has ownership of o. If ownership is respected then data races cannot occur between machines. Violations of ownership *may* indicate that data races do occur.

To make our static ownership analysis both machine- and method-modular (thus ensuring scalability when we apply the analysis on large systems), we define a *gives-up set* for each method m. For each formal parameter *fp* of m, if there exists an object o such that (i) o is *reachable* from *fp* (either directly or by transitively following references), and (ii) ownership of o is transferred by m, then *fp* is in the gives-up set for m.

Once the gives-up set for each method has been computed, data race-freedom can be established by considering all scenarios where:

1. a formal parameter *fp* is in the gives-up set for a method m';
2. a method m calls m', passing a variable v for *fp*;

3. after calling m', m goes on to access a variable v'.

It suffices to show that in all such scenarios, no object o exists in the heap of the program that can be accessed (either directly or indirectly) through both v and v'. Based on this observation, we define the central theorem of this section as follows:

Theorem 4.4.1. *If each state of a machine m respects the ownership of each method invocation and send statement (as defined in §4.4.4), then no data race can occur.*

4.4.1 Assumptions

We assume that formal parameters of methods cannot be assigned to, and that the variables passed as actual parameters during method calls are pairwise distinct (thus avoiding implicit creation of aliases through parameter passing). These two assumptions simplify the presentation and can be satisfied through preprocessing.

Because scalar variables are passed by value, our race analysis only needs to be concerned with reference variables. Henceforth, and unless stated otherwise, we use *variable* to only mean a variable of reference type.

Finally, we represent each method as a single-entry, single-exit *control flow graph* (CFG), where each CFG node consists of a *single* statement. The entry and exit nodes are denoted Entry and Exit. Employing CFGs allows us to treat conditionals, loops and sequences of statements in a uniform manner. In the actual implementation, we use CFG nodes that can consist of multiple statements. The CFG representation of a P# program is standard and straightforward, and thus we omit it here for brevity.

4.4.2 Heap Overlap

Our static data race analysis depends on knowing whether the same object o is reachable from multiple variables, i.e. our race analysis depends on an analysis that soundly resolves *heap overlap* between variables at different CFG nodes (which, for simplicity, we refer as nodes in the remaining of this section). Let us assume that we have a heap overlap analysis at our disposal, in the form of a predicate may_overlap such that for a method m:

- $\text{may\_overlap}_{m,\text{in}}^{(N,v)}(N', v')$ holds if there may exist a heap object o that is reachable from variable v' on entry to node N' and is also reachable from variable v on entry to node N.

- $\text{may\_overlap}_{m,\text{out}}^{(N,v)}(N', v')$ holds if there may exist a heap object o that is reachable from variable v' on exit from node N' and is also reachable from variable v on entry to node N.

Observe in both cases that we consider the heap objects reachable from v on *entry* to node N. The reason for this is that we are interested in objects that are reachable immediately before they are given up in node N. In practice, we implemented the may_overlap predicate using an inter-procedural *taint-tracking* analysis, which is flow-sensitive, context-sensitive and path-insensitive.

Given a tainted variable v, the taint-tracking analysis *taints* a variable v' if there *may* exist some object o that is reachable from v' and is also reachable from v. The output of the analysis consists of a function that summarises how tainting propagates throughout methods. Given a method m, a local variable or formal parameter v of m, and a node N of m, our analysis yields a summary function $\text{tainted}_m^{(N,v)}$ that maps nodes N' of m to variables v' such that if v is assumed to be tainted on entry to N, then v' is tainted on exit from N'. In other words, $v' \in \text{tainted}_m^{(N,v)}(N')$ implies that there may exist an object o such that v' can reach o on exit from N' if v can reach o on entry to N.

Our summary function is member variable *insensitive*, i.e. when our analysis finds that a member of an object should become tainted, it taints the whole object instead. Although this design choice made the analysis easier to implement, it reduces the overall precision and can result in false alarms (although we did not observe any in our experiments, see §4.6.1). In future work, we plan to improve the precision of our analysis by keeping track of individual member variables, and investigate if finer-grained taint tracking reduces the scalability of our static race detection approach.

The may_overlap predicate can now be defined as follows:

- $\text{may\_overlap}_{m,\text{in}}^{(N,v)}(N', v') \triangleq v' \in \bigcup_{P \in \text{pred}(N')} \text{tainted}_m^{(N,v)}(P)$
- $\text{may\_overlap}_{m,\text{out}}^{(N,v)}(N', v') \triangleq v' \in \text{tainted}_m^{(N,v)}(N')$

where $\text{pred}(N')$ denotes the set of predecessor nodes of N', and the $\triangleq$ symbol denotes that the left hand side is defined as the right hand side.

In the example of Figure 4.3, only this can become tainted in the *get_val* and *set_val* methods. Note that this can only become tainted if it was tainted in the first place, as all other variables in these methods are scalar. For *get_next*, we have $\text{tainted}_{get\_next}^{(\mathsf{Exit}, ret)}(\mathsf{Entry}) = \{\mathsf{this}\}$ (*ret* is not included in the set, as its value is overwritten in the second line of the method). The summary for *set_next* is similar.

Heap overlap is closely related to the well-studied problem of computing heap reachability, i.e. the problem of establishing whether a heap object o is reachable from a variable v. Hence, instead of using our custom taint-tracking analysis, we could have tried to adapt the Thresher analysis [BCS13], which is a path-, flow- and context-sensitive heap reachability analysis that currently seems to be among the most accurate and scalable. However, Thresher is based on advanced verification techniques (such as separation logic [Rey02]), which make it challenging to implement, and its scalability benefits for our problem are not immediately clear. For this reason, we opted instead to develop a simpler taint-tracking analysis, which in our experience is accurate enough for P# programs (see §4.6.1).

We generalised the implementation of our taint-tracking analysis, and made it available as a separate project in the P# GitHub repository,[7] so that it can be used to build static analyses for arbitrary C# programs, not just P#. In our case, we used this taint-tracking analysis to build the gives-up ownership (see §4.4.3) and respects ownership (see §4.4.4) analyses for programs written in P#.

4.4.3 Gives-up Ownership Analysis

We now define for which formal parameters of a method m ownership is *given up* by m; we denote the set of these parameters by $\text{gives\_up}(m)$. Initially, the gives-up set for each method is set to the empty set. The gives-up sets are then computed as follows:

```
1: repeat
2:     for all $m$ do
3:         $\text{gives\_up}(m) := \bigcup_{N \in m} \text{gives\_up}^{\mathit{fp}}_{m}(N)$
4:     end for
5: until gives_up no longer changes
```

The function $\text{gives\_up}^{\mathit{fp}}_{m}(N)$ is defined in Figure 4.5, where $\text{fp}(m)$ denotes the set of formal parameters of m. Ownership of a formal parameter *fp* is given up either when

1. some object reachable from *fp* on entry to m is also reachable from the variable passed as value argument to a send statement of m, or when

2. some object reachable from *fp* on entry to m is also reachable from the ith argument of a method m' invoked by m, where m' gives up its ith argument.

The $\text{gives\_up}(m)$ function is formulated as a fixed-point computation, because methods may be mutually recursive. Termination occurs, since the number of methods and formal parameters is finite. The function identifies all formal parameters of a method m from which heap objects may be reachable that are also reachable from a variable occurring in a send statement.

[7] `https://github.com/p-org/PSharp/tree/master/Libraries/AddOns/DataFlowAnalysis`

$$\text{gives\_up}^{fp}_{m}(N) = \begin{cases} \{w \in \text{fp}(m) \mid \text{may\_overlap}^{(N,v)}_{m,\text{out}}(\mathsf{Entry}, w)\} & \text{if } N = \mathsf{send}_{dst}\ evt(v) \\ \cup_{1 \le i \le n}\{w \in \text{fp}(m) \mid & \text{if } N = v := v'.m'(v_1, \ldots, v_n) \\ \quad fp_i \in \text{gives\_up}(m') \land \text{may\_overlap}^{(N,v_i)}_{m,\text{out}}(\mathsf{Entry}, w)\} & \\ \emptyset & \text{otherwise} \end{cases}$$

Figure 4.5: Computing the formal parameters given up by m.

For the methods in the example of Figure 4.3 and 4.4, no formal parameters are given up. However, if we would let the *add* method of *list_manager* forward *payload* instead of adding it to the linked list, i.e. if we would replace the method body with the $\mathsf{send}_{dst}\ eAdd(payload)$ statement, then the method *add* would give up *payload*.

4.4.4 Respects Ownership Analysis

We now define the conditions under which the nodes of a method *respect* the ownership of objects. The interesting cases are when nodes represent a send operation or a method call, because these statements have the potential to erroneously transfer ownership between machines. All other types of nodes trivially respect ownership.

Suppose that N is a node of a method m, and that N represents either a method call or a send operation. If N is a method call, of the form $v := v'.m'(v_1, \ldots, v_n)$, then N *respects ownership* if ownership is respected at N for each actual parameter v_i such that the corresponding formal parameter fp_i of m' is in the gives-up set of m', i.e. $fp_i \in \text{gives\_up}(m')$. If N is a send operation, of the form $\mathsf{send}_{dst}\ evt(v)$, then N *respects ownership* if ownership is respected at N for v.

We now describe the conditions under which object ownership is respected by a variable in a node. Let N be a node in method m, and let variables(N) denote the set of variables occurring in N. For a variable w we say that ownership is respected for w at node N if, for every node N' of method m, the following three conditions hold:

1. If there is a path from Entry to N through N', then

$$\neg\text{may\_overlap}^{(N,w)}_{m,\text{out}}(N', \mathsf{this})\,.$$

2. If $N' = N$, then $w \neq \mathsf{this}$ and

$$\{v \in \text{variables}(N') \mid \text{may\_overlap}^{(N,w)}_{m,\text{in}}(N', v)\} = \{w\}\,.$$

3. If there is a path from N to Exit through N', then

$$\{v \in \text{variables}(N') \mid \text{may\_overlap}_{m,\text{in}}^{(N,w)}(N', v)\} = \emptyset\,.$$

We discuss the above three cases in turn. In the first case, we check whether some object reachable from this is also reachable from w. If this is the case, we may be able to access a given up object in a later machine state (through a field of the machine).

In the second case, we check whether w is equal to this. If it is, then again we may be able to access a given up object in a later machine state. We also check that no variable other than w has access to an object reachable from w. This takes care of potential aliasing: if an object may also be accessed through other variables, then the method we invoke may be able to access the objects given up after it has given them up.

In the third case, we check whether any variable that may be subsequently used to give up w can reach an object that was also reachable through w at the point at which w was given up. We forbid the use of any such variable. Observe that if some variable v is given up in a node N' on a path from Entry to node N, and if an object o exists that is accessible through both v and w, then by symmetry an error will be flagged through the third case, as node N is on a path from N' to Exit.

The correctness of Theorem 4.4.1 now follows by the above observations regarding our respects ownership definition and gives-up ownership analysis. Observe that implementations of the may_overlap predicate do not need to cater for aliasing of formal parameters, because (i) our top-level methods representing states only have a single argument (which trivially implies that no aliasing between arguments occurs), and (ii) we check the requirement on variables in the second case above (recall that all variables passed to a method are assumed to be pairwise distinct).

Continuing our running example from §4.3, for the method *get* in Figure 4.4, an ownership violation will be flagged, as any object accessible through *tmp* in the send statement is accessible through this. This violates the first case described above.

4.4.5 Cross-State Analysis

Most false alarms in our experiments (see §4.6.1) originate from the payload p of an event being constructed in one state of machine m and only being sent from a later state of m. This is achieved by temporarily storing p in a field of m. Sending the payload p will lead to an ownership violation, as the sent objects may still be accessible through this.

To suppress these false alarms, we created a *cross-state analysis* (xSA), which we run upon detection of an ownership violation. This analysis is based on the observation that each state-machine in P# can be seen as a CFG, where at the end of each method representing a state we nondeterministically call one of the methods representing an immediate successor state. Our analysis can now be performed on this overarching CFG once we lift all machine fields to be parameters of the methods. Since all payloads are now passed as parameters, the aforementioned false alarms no longer occur. That xSA is sound is an immediate consequence of the soundness of our ownership analysis.

We are now able to "repair" the example of Figure 4.4 by adding the statement this.*list* := null; after the send statement, i.e. we *reset* the *list* member. Once we have repaired the *get* method, we apply xSA to determine race-freedom, as *list* is a member variable. Race-freedom easily follows once we lift the member variable to be a parameter of all methods in the *list_manager* machine.

4.4.6 Implementation Details

We implemented our static data race analysis on top of the Microsoft Roslyn compiler (see Figure 4.1). Because Roslyn does not (currently) provide direct access to the underlying CFG of each method, we construct our own CFG by querying the Roslyn AST interface. Once the CFG of each method has been constructed, we perform our taint-tracking, gives-up ownership and respects ownership analyses on each P# machine in isolation.

We handle calls to library methods of which the source code is not available in a conservative manner (for soundness) by assuming that each heap object reachable before the call is reachable from all variables involved in this call once the call returns (and also report an appropriate warning to the user). Inheritance is handled soundly by creating the union of the summaries of all possible methods that can be invoked at a call site.

In practise, the P# static data race analyser is "soundy" rather than sound, similar to Whoop from Chapter 3 (see §3.4.9). The P# static data race analyser aims in principle to perform a sound analysis, but suffers from some known sources of unsoundness: we assume that no multi-threading constructs and reflection are used to develop P# programs, and assume that any imported external libraries do not use these either; we also do not support exception handling (which could be dealt with by adding appropriate edges to the CFG) due to scalability issues.

4.5 Controlling the Scheduler for Testing

Due to the asynchronous nature of programs written in P#, bugs might manifest only when P# machines are scheduled in a particular order. Such bugs are typically hard to detect, diagnose and fix [Gra86, MQB$^+$08]. Inspired by the success of controlled concurrency testing [God97, MQB$^+$08, TDB16] (see §2.3.3), we designed a bug-finding version of the P# runtime, in which execution is *serialised* and the schedule is *controlled*.

The controlled testing engine of P# (see Figure 4.1) executes a P# program on top of this bug-finding runtime for a specified number of testing iterations. On each iteration, the testing engine executes the P# program from start to completion, following a (potentially) different schedule. This process repeats until either a bug (e.g. an assertion violation – see §4.2.1) is discovered or all testing iterations terminate. Our testing approach enables easy reproduction of bugs; if a bug is found, then the testing engine dumps a reproducible sequential trace that represents the buggy schedule.

When performing dynamic analysis of concurrent software, it is necessary to explore the interleavings of all *visible* operations (e.g. shared memory accesses and locks), else a buggy schedule might be missed [God97]. However, if a program is race-free, only synchronising operations need to be treated as visible, which greatly reduces the exploration space that must be considered. Some existing controlled testing tools (e.g. CHESS [MQB$^+$08]) exploit this fact by only exploring interleavings of synchronising operations while running with a race detector. If no data races are discovered and all interleavings are explored, then no bugs were missed due to data races. Otherwise, it is necessary to fallback to interleaving all visible operations.[8]

P# programs have several benefits that make them well-suited to controlled testing. First, we can prove race-freedom using our sound static analysis (see §4.4). This avoids the overhead of testing with a race detector enabled (as in the case of CHESS – see §4.6.2), and ensures that we only have to explore interleavings of synchronising operations. Second, since the only synchronising operations in P# are the `send` and `create` methods (see §4.2.1), which are implemented in our runtime, it is straightforward to build a controlled testing engine. This differs from most existing controlled testing tools, which use dynamic instrumentation to intercept memory accesses and synchronising operations [MQB$^+$08].

In bug-finding mode, the `send` and `create` methods call the runtime method `schedule`, which blocks the currently executing P# machine and schedules a machine from the set of all *enabled* machines (which can contain the machine that just got blocked). We consider that a P# machine is enabled when it has at least one event in its input queue waiting to be handled. As observed in previous work on P [DGJ$^+$13], it is not necessary to insert a scheduling point (i.e. a call to `schedule`) before receiving an event, thus we achieve a simple form of partial-order reduction [God96] (see §2.3.4).

[8]In the presence of data races and relaxed memory, even considering all interleavings of shared memory accesses may be insufficient to find all bugs.

We have implemented the following three schedulers for choosing the next enabled P# machine to execute each time the `schedule` method is invoked during testing:

Random scheduler which nondeterministically chooses the next machine to schedule from a list of enabled P# machines. Although this is arguably a naive scheduler, in our experience it is effective in finding bugs (see §4.6).

Randomised priority-based scheduler which always schedules the highest priority enabled P# machine. The initial machine priorities are chosen randomly, as are $d-1$ priority change points (which are chosen uniformly over the length of the execution, and are inserted at the corresponding scheduling points).[9] Each time the scheduler reaches a priority change point, it lowers the priority of the currently executing machine, and schedules the highest priority machine. This scheduler is based on the probabilistic concurrency testing (PCT) algorithm discussed in the [BKMN10] study.

Depth-first search (DFS) scheduler which explores the schedule-tree of a P# program in a depth-first manner. Each node is a schedule prefix and the branches are the P# machines that are enabled in the program state reached by the schedule prefix.

The DFS scheduler is *systematic*: it explores a different schedule each time, thus ensuring exhaustive exploration of all schedules (assuming an acyclic exploration space, and given enough time and resources). In contrast, the random and randomised priority-based schedulers choose a random machine to execute after each `send` and `create`, and do not keep track of explored schedules (thus might replay previously explored schedules). In our experience, however, the random and randomised priority-based schedulers find significantly more bugs than the DFS scheduler (see §4.6.2).

The P# testing approach is fully automatic and has no false positives.[10] However, it requires a closed environment, i.e. when testing a distributed system we need to define and instantiate additional machines that model the environment of the system. Like any real environment, the additional machines may be nondeterministic, but this nondeterminism is captured and controlled by the P# runtime during testing (e.g. see §5.3.3 and §5.3.4). More details on using P# to model the nondeterministic environment of a large-scale distributed system developed inside Microsoft, and then test it using our controlled schedulers, are provided in Chapter 5.

[9]This scheduler requires the execution length and d as input parameters. In our current implementation, we estimate the execution length by executing the P# program three times before calculating an average length, which the user can override by providing a length value.

[10]If the environment is not modelled properly, false positives can be introduced. Moreover, in Chapter 5 we discuss that liveness checking using P# can report false alarms due to heuristics.

4.6 Experimental Evaluation

We experimentally evaluate our static data race analysis (see §4.4) and controlled testing infrastructure (see §4.5) against 14 distributed algorithms, implemented in P# as shared-memory (i.e. running on a single-box) asynchronous event-driven programs.

Benchmarks Our benchmark suite is available online in the Samples directory of the P# GitHub repository,[11] and includes:

- PingPong, which consists of a server and a client that communicate by exchanging ping and pong events;
- BoundedAsync, a generic scheduler that communicates with multiple processes under a user-defined bound;
- LeaderElection [DKR82], a protocol for electing a leader among a set of candidate processes connected using a unidirectional ring;
- PiCompute, which computes an approximation of π by distributing computational tasks among a pool of worker processes;
- German [Ger04], a cache coherence protocol;
- FailureDetector [CT96], a mechanism for detecting node failures in a distributed system;
- Chameneos [KPP03], a simulation game where creatures called Chameneos meet through a broker and change skin colours;
- TwoPhaseCommit [Gra78], a distributed commit protocol;
- Chord [SMK+01], a distributed hash-table used for creating peer-to-peer lookup services;
- ReplicatingStorage, a simplified model of a replicating distributed storage system used in Microsoft Azure.
- MultiPaxos [CGR07], an advanced version of Lamport's Paxos consensus protocol [Lam98];
- Raft [OO14], a consensus protocol for managing a replicating log;

[11] `https://github.com/p-org/PSharp/tree/master/Samples`

Table 4.1: Program statistics for the P# benchmarks. We report: lines of P# source code (LoC); number of state-machine declarations (#SM); number of event-handler declarations that result into a state-transition; and number of event-handler declarations that result into an invoked machine action.

			#Event-handlers	
Benchmarks	**LoC**	**#SM**	**Transitions**	**Actions**
PingPong	118	2	3	6
BoundedAsync	166	2	5	6
LeaderElection	174	2	1	4
PiCompute	195	3	2	6
German	356	3	36	16
FailureDetector	392	4	5	11
Chameneos	398	2	9	16
TwoPhaseCommit	618	5	18	22
Chord	711	3	5	22
ReplicatingStorage	728	8	18	36
MultiPaxos	922	7	24	28
Raft	969	6	20	46
ChainReplication	1276	7	24	37
Swordfish	1465	6	22	62

- ChainReplication [vRS04], a fault-tolerance protocol;
- Swordfish, which implements a protocol for performing transactions using an automatic teller machine (ATM);

Table 4.1 presents program statistics for the benchmarks. PingPong, BoundedAsync, German, FailureDetector, TwoPhaseCommit, MultiPaxos and ChainReplication were ported from open source P implementations [DQS15]. Chord, ReplicatingStorage and Raft were implemented from scratch in P#. Finally, LeaderElection, PiCompute, Chameneos and Swordfish are direct ports of the four worst-performing (in terms of analysis precision) Java actor programs from the SOTER study [NKA11].

Our benchmarks are essentially single-box simulations of the actual distributed algorithms. To simulate the external environment of these algorithms (e.g. failures, timeouts and client requests), we developed additional nondeterministic P# machines, which are controlled by the P# runtime during testing. This matches the approach used in previous studies [DQS15, DJP^{+}15] to evaluate testing techniques for distributed algorithms.

Experimental Setup We performed all experiments on a 1.9GHz Intel Core i5-4300U CPU with 8GB RAM running Windows 10 Pro 64-bit and the latest available version of

CHESS (commit 69631).

4.6.1 Detecting Data Races in P# Programs

We evaluate the soundness, precision and scalability of our static analysis with respect to our benchmark suite.

Soundness and Precision To evaluate the soundness of the P# static analyser, we ran it on our benchmarks (which contained a mix of accidental and injected data races). After manually inspecting the source code of the 14 benchmarks and each reported data race, we confirmed that no races were missed, which is unsurprising, since the static race analysis was designed to be sound. We have also developed a regression suite that contains more than 125 complex cases of racy and non-racy P# programs (e.g. with aliasing, loops and inheritance).[12] We have extensively exercised our analysis on these test cases to increase confidence in its soundness and fine-tune its precision.

To evaluate the precision of the static analysis, we first fixed the data races in all 14 benchmarks, and then ran the analyser in two different modes: (i) with *cross-state analysis* (xSA) disabled; and (ii) with xSA enabled (see §4.4.5). Table 4.2 shows the results for these two experiments. Our tool failed to verify seven benchmarks with xSA disabled. The majority of the reported races were related to either (a) giving up ownership of a machine field, (b) storing a reference to be given up in such a field, or (c) giving up ownership of the same reference to more than one machine. With xSA enabled, the analyser managed to verify two more benchmarks, and significantly reduced the false positives in three of the remaining unverified benchmarks. This shows that xSA is useful.

Although xSA discarded 16 out of 36 false positives, it did not manage to discard the remaining twenty spurious data race reports, the majority of which correspond to one of the following two cases:

1. A machine broadcasts the same reference to two or more machines. This can be a source of data races, if the receiver machines try to access the sent reference at the same time.

2. A machine M_1 in a state S_1 stores a reference in a machine field f, and then sends the reference to a machine M_2. In a later state S_2, M_1 sends the contents of f to a machine M_3 without first updating f to point to a new memory location. This could potentially lead to a data race as this gives each of M_1, M_2 and M_3 access to the same memory location.

[12]`https://github.com/p-org/PSharp/tree/master/Tests/StaticAnalysis.Tests.Unit`

Table 4.2: Results from applying the P# static data race analysis to our 14 benchmarks. All reported analysis times are in seconds and averages of 10 runs.

	Analysis Time (sec)		**#Races Reported**		
Benchmarks	**TaintTracking**	**Ownership**	**xSA-off**	**xSA-on**	**Verified?**
PingPong	0.126	0.002	✗	✗	✓
BoundedAsync	0.136	0.002	✗	✗	✓
◇ LeaderElection	0.142	0.002	✗	✗	✓
◇ PiCompute	0.142	0.059	1	✗	✓
German	0.157	0.001	✗	✗	✓
FailureDetector	0.163	0.034	1	✗	✓
◇ Chameneos	0.180	0.002	✗	✗	✓
TwoPhaseCommit	0.177	0.045	3	3	✗
Chord	0.249	0.134	8	7	✗
ReplicatingStorage	0.200	0.001	✗	✗	✓
MultiPaxos	0.212	0.124	8	2	✗
Raft	0.324	0.105	5	2	✗
ChainReplication	0.233	0.147	10	6	✗
◇ Swordfish	0.324	0.039	✗	✗	✓

However, when manually inspecting the source code of the unverified P# benchmarks, we found that the sent references are only ever read. Hence, we believe that we can suppress these false positives by introducing a *read-only* analysis.

Previous work on static ownership-based, data race analysis for message-passing programs includes the SOTER analyser [NKA11] for actor programs in Java. We ported four of the worst-performing (in terms of analysis precision) benchmarks from the SOTER study to P# (denoted by ◇ in Table 4.2). While the P# static analysis verifies all four benchmarks, SOTER reports a large number of false alarms (e.g. 70 in Swordfish) [NKA11]. SOTER differs from our work in that it uses a fundamentally different static analysis (e.g. their alias analysis is flow-insensitive, and thus less accurate than the alias analysis used in P#, which is flow-sensitive). SOTER tracks ownership not just through the user source code, but also through the underlying actor framework, whereas P# only analyses the user source code (see §4.7).

Performance and Scalability Table 4.2 also shows the time spent analysing each of the 14 benchmarks. The reported times include xSA. P# performed taint-tracking analysis on each benchmark in less than 324 milliseconds, and ownership analysis in less than 147 milliseconds. This was expected, since we designed the P# ownership analysis to be fully modular, and use pre-computed information (in the form of method summaries) from our custom taint-tracking analysis (see §4.4.2). These results show that the P# static analyser scales well across programs of varying size and complexity.

4.6.2 Finding Bugs in P# Programs

In this section, we evaluate the P# controlled testing engine against CHESS [MQB+08], an industrial-strength controlled testing tool. Because P# is built on top of C#, we were able to apply CHESS to all our benchmarks with minimal effort.

We first show that P# achieves better testing throughput (in terms of explored schedules per second) than CHESS, and that the built-in data race detector of CHESS adds unnecessary overhead when applied to programs written in P# (see Table 4.3). We then compare the bug-finding ability of P# and CHESS, using seven buggy benchmarks (see Table 4.4). Most bugs are real mistakes that we made while implementing the algorithms (which we fixed in the benchmark versions used in §4.6.1). In the case of Raft, we injected a real-world bug that was found by a previous study [SPB+07].

Any benchmark that required input was given a fixed input that was constant across executions. For the ported benchmarks, we used the same inputs as in their original Java or P implementations. For Chord, ReplicatingStorage and Raft, the input is irrelevant for finding the corresponding bugs.

Comparing Testing Throughput We measured the *testing throughput* (expressed in number of explored schedules/second) of the CHESS and P# randomised priority-based schedulers. For this experiment, we considered non-buggy benchmarks, since we were only interested in comparing the testing speed of P# and CHESS. Table 4.3 shows that P# was between 3.2× and 16.5× faster (in terms of explored schedules/second) than CHESS (with the CHESS data race detection turned off).[13]

We believe that CHESS is slower for two reasons. First, CHESS uses dynamic binary instrumentation, a technique which is well-known for adding overhead [GLSS05, UCY+06], whereas the P# testing infrastructure is embedded in our runtime. Second, CHESS inserts scheduling points before several synchronisation operations (e.g. runtime locks), whereas the P# scheduler only needs to schedule before send and create-machine operations (since it has knowledge of the runtime), which greatly reduces the schedule-space without missing any bugs (this is the same idea as the sound partial-order reduction discussed in §3.5.2). In Raft, for example, we found that CHESS inserted 1333 scheduling points, whereas P# only inserted 166 scheduling points.

[13]Note that disabling the CHESS data race detection improves the performance of CHESS.

Table 4.3: Results for applying the CHESS and P# randomised priority-based schedulers to our benchmarks. We report: number of concurrently running state-machines per execution (#SM); number of scheduling points in the first execution (#SP); and testing throughput (in term of number of explored schedules per second). We ran CHESS in two modes: with its built-in data race detection enabled (RD-on); and with its built-in data race detection disabled (RD-off).

		CHESS			**P#**	
			Schedules/sec			
Benchmarks	**#SM**	**#SP**	**RD-on**	**RD-off**	**#SP**	**Schedules per second**
PingPong	2	164	39.16	145.03	19	1185.04
BoundedAsync	4	1515	4.29	24.43	223	147.86
LeaderElection	6	391	12.31	62.92	45	374.26
PiCompute	7	309	15.75	73.36	38	413.08
German	5	383	6.15	23.28	79	181.19
FailureDetector	7	751	6.79	40.52	105	287.91
Chameneos	14	631	9.49	48.01	102	293.03
TwoPhaseCommit	7	445	10.02	47.89	65	247.82
Chord	5	143	23.58	93.40	63	302.31
ReplicatingStorage	12	12164	0.19	5.69	444	94.41
MultiPaxos	20	1747	1.26	15.49	446	38.62
Raft	18	1333	1.44	17.24	166	34.27
ChainReplication	9	2518	1.21	13.56	412	76.44
Swordfish	5	1621	2.68	15.48	229	67.04

Measuring Race Detection Overhead We measured the overhead of enabling data race detection in CHESS. For this experiment, we ran the randomised priority-based scheduler of CHESS on the 14 non-buggy benchmarks with and without the built-in data race detector of CHESS enabled. Table 4.3 shows that CHESS runs between 3.7× and 29.9× faster (in terms of explored schedules/second) when its data race detection facilities are disabled. Since the P# static data race analysis has determined the absence of races, we can run CHESS *without* race detection while being confident that no bugs are missed due to data races; thus, we can benefit from the increase in testing throughput. With data race detection enabled, CHESS did not find any data races in the benchmarks, which provides additional confidence that our static data race analysis is sound.

Assessing Bug-Finding Ability We compared the bug-finding ability of CHESS[14] and P# on seven buggy versions of our benchmarks. Table 4.4 shows the number of schedules

[14]We only used the randomised priority-based and DFS schedulers of CHESS, because the simple random scheduler of P# (discussed in §4.5) was not available in CHESS.

Table 4.4: Results from testing buggy versions of the P# benchmarks using the DFS and randomised priority-based CHESS schedulers, and the DFS, random and randomised priority-based P# schedulers. The bug depth value refers to the number of nondeterministically-chosen priority change points (see §4.5) required to trigger a bug, which is only relevant for the two priority-based schedulers.

		#Schedules Explored Until Bug				
		CHESS		**P#**		
	Bug Depth	**DFS**	**Prioritised**	**DFS**	**Random**	**Prioritised**
FailureDetector	6	✗	✗	✗	69135	594283
TwoPhaseCommit	1	✗	✗	✗	52	137
Chord	1	✗	✗	✗	46	98
ReplicatingStorage	2	✗	4	✗	22	2
MultiPaxos	0	1	5	1	7	4
Raft	2	✗	✗	✗	112	107982
ChainReplication	2	✗	40318	✗	✗	39106

explored until each bug was found (each table row represents one bug). Each scheduler was configured to run for a maximum of 3600 seconds. P# found all seven bugs, whereas CHESS only discovered the ReplicatingStorage, MultiPaxos and ChainReplication bugs. We believe that CHESS did not perform as well as P#, because it spent a lot of time exploring interleavings inside the P# runtime, instead of focusing on the interleavings at the level of state-machines, which are the ones triggering the bugs.

Table 4.4 also shows that the random and randomised priority-based schedulers discovered significantly more bugs than the two DFS schedulers, which only managed to discover the MultiPaxos bug. This finding confirms previous claims that randomised concurrency testing is effective at finding bugs [BKMN10, TDB16].

The bugs in FailureDetector and ChainReplication are deep bugs; we found them only after exploring thousands of schedules. The ChainReplication bug is especially interesting, since it was discovered by the P# and CHESS randomised priority-based schedulers, but not by the random P# scheduler. This bug seems to occur only if the P# state-machines stay scheduled for a large number of scheduling points in a row. We believe that the random scheduler is not suitable for detecting this particular bug, because the scheduler has a high-probability to schedule a different machine in each scheduling point. In contrast, the buggy schedule is significantly easier to occur with the randomised priority-based schedulers, as they schedule a new P# machine only in the following two cases: (i) when the currently scheduled P# machine terminates or blocks, or (ii) when the runtime reaches a priority change point (see §4.5). The bug was discovered with two priority change points (denoted as bug-depth in Table 4.4).

4.7 Related Work

Static Analysis Sen and Viswanathan [SV06] proved that the control state reachability problem for asynchronous programs with finitely many global states is decidable. Building on this result, Jhala and Majumdar [JM07] presented the first safety verifier for unbounded asynchronous programs. A recent study [GNK$^+$15] used an approach based on rely/guarantee [Jon83] to verify asynchronous C programs written using the libevent [Pro02] library. Although promising, the approach requires significant developer effort (i.e. to manually annotate and verify programs), and thus cannot be easily applied to real-world asynchronous systems. In our case, we only attempt to statically detect data races in P# programs, and then we use controlled testing techniques for detecting concurrency-related bugs.

Static Data Race Detection Most closely related to our work is SOTER [NKA11], an ownership-based static analyser for Java actor programs. Although both SOTER and P# perform a static ownership-based analysis, they are fundamentally different. SOTER builds upon a field-sensitive, flow-insensitive points-to analysis, which is non-modular and does not leverage an understanding of the underlying (actor) framework, and is also less precise than the P# flow-sensitive alias analysis. In §4.6.1, we show that our novel analysis can achieve higher precision than SOTER, without sacrificing scalability.

Controlled Concurrency Testing Our testing methodology is inspired by controlled concurrency testing [God97, MQB$^+$08, BKMN10, EQR11, TDB16] techniques (see §2.3.3), and our initial intent was to reuse CHESS [MQB$^+$08], since it can already analyse .NET programs. However, in practice, we found that we can significantly improve testing performance by leveraging the domain-specific nature of P# through a custom controlled testing engine, which we built inside the P# runtime (see §4.6.2).

4.8 Summary

In this chapter we presented P#, a new language for high-reliability, asynchronous event-driven programming. P# is co-designed with static race analysis and controlled testing techniques. We experimentally showed that our static analysis scales well across programs of varying size and complexity, and is more accurate than SOTER, a static race analyser for Java actor programs. We also showed that P# achieves better testing throughput than

CHESS, and that the built-in race detector of CHESS adds unnecessary overhead when applied to asynchronous programs written in our language.

5 Uncovering Bugs in Distributed Systems using P#

In this chapter, we are concerned with the problem of efficiently finding bugs in industrial-scale distributed systems. Due to the asynchronous communication between system components and numerous other sources of nondeterminism (e.g. unexpected failures), bugs in distributed systems might only manifest under extremely rare conditions [Gra86, CGR07, MQB^{+}08, Hen09, McC15, LLLG16]. Detecting such bugs is challenging, since the number of states that distributed systems can possibly reach is exponentially large.

Conventional testing techniques, such as unit, integration and stress testing, are ineffective in preventing serious, but subtle, bugs from reaching production, as they do not typically control sources of nondeterminism. Systematic testing tools such as dBug [vBG11] and MoDist [YCW^{+}09] focus on testing *unmodified* distributed systems, but this can lead to state-space explosion when trying to exhaustively explore the state-space of industrial-scale distributed systems. On the other hand, formal techniques, such as TLA+ [Lam02] have been successfully used in industry to *verify* the high-level specifications of complex distributed systems at the level of logic-based models, but fall short of checking the actual executable code [NRZ^{+}15]. More recent formal techniques, such as Verdi [WWP^{+}15] and IronFleet [HHK^{+}15], show a lot of promise as they can verify high-level specifications *and* generate verified executable code. However, both techniques focus on new systems written in non-mainstream languages, and are unable to check properties of existing systems.

In this chapter, we are proposing a solution between the above two extremes (i.e. testing unmodified systems or verifying high-level specifications). Our approach applies controlled testing techniques to thoroughly check whether the executable code adheres to its high-level specifications, which are written in P# (see §4). Using P# for testing involves writing a test harness, which *models* the nondeterministic environment of the distributed system-under-test, and is responsible for driving the system into potentially interesting execution scenarios that might hide bugs. During testing (see §4.1.2), the P# runtime controls and explores all *declared* sources of nondeterminism in the given test harness. We found that this approach can significantly improve coverage of important system behaviours.

Our approach is *flexible*: developers can choose which real components of a large system they want to test at a time, and provide models for the remaining components. The benefit of *partial-modelling* is that it helps P# detect bugs by exploring a (much) reduced state-space. Testing using P# does not come for free; developers must invest effort and time modelling the environment (and/or parts) of their system. However, developers already spend significant time in building comprehensive test suites prior to deployment. We argue that the P# approach augments this effort; by investing extra time in modelling, it offers dividends by finding more bugs (see §5.4). In principle, our approach is *not* specific to P# and .NET, and can be used in combination with any other programming framework that has equivalent capabilities.

P# has been so far successfully applied to three distributed systems built on top of the Microsoft Azure cloud computing platform [DMT$^+$16]. In the process, numerous bugs were identified, reproduced, confirmed and fixed. These bugs required a subtle combination of concurrency and failures, making them extremely difficult to find with conventional testing techniques. In this chapter, we focus on a particular case study of applying P# to Azure Storage vNext [DMT$^+$16],[1] and show how P# was used to detect a subtle liveness bug that could not be troubleshooted for months before using P#.

The main contributions of this chapter are:

- We present an approach for partially-modelling, specifying properties of correctness (both safety and liveness), and testing real-world asynchronous distributed systems using the P# programming language and testing infrastructure.

- We discuss our experience of using P# to test Azure Storage vNext, a distributed storage system developed inside Microsoft, and detect a subtle, but serious, liveness property violation.

- We evaluate the costs and benefits of using our approach, showing that the P# controlled testing engine can detect complex bugs while producing easy to understand error traces.

Related Published Work The material presented in this chapter is based on work that was originally published in the 14th USENIX Conference on File and Storage Technologies (FAST'16) [DMT$^+$16].

[1]Applying P# to Azure Storage vNext was a joint effort between the architect of vNext and the author of this thesis (acknowledged in §1.4).

5.1 Motivation

Distributed systems are notoriously hard to design, implement and test [Cav13, LHJ$^+$14, GHL$^+$14, LAdS$^+$15, Mad15]. This challenge is due to many well-known sources of *nondeterminism* [CGR07, Hen09, McC15, LLLG16], such as unexpected failures, races in the asynchronous interaction between system components, data losses due to unreliable communication channels, and interaction with clients. All these sources of nondeterminism can create *Heisenbugs* [Gra86, MQB$^+$08], corner-case bugs that are difficult to detect, reproduce, diagnose and fix. Such bugs might hide inside a code path that can only be triggered by a specific interleaving of distributed events and only manifest under extremely rare conditions [Gra86, MQB$^+$08], but the consequences can be catastrophic [Ama12, Tre14].

Developers of production distributed systems use many testing techniques, such as unit testing, integration testing, stress testing, and fault injection [McC15]. Despite the extensive use of these testing methods in industry, many bugs that arise from subtle combinations of concurrency and failure events are missed during testing and get exposed only in production. However, allowing serious bugs to reach production can cost organisations a lot of money [Tas02] and lead to customer dissatisfaction [Ama12, Tre14].

Our collaborators in Microsoft Research interviewed technical leaders and senior managers in Microsoft Azure regarding the top problems in distributed system development. The consensus was that one of the most critical problems today is how to improve *testing coverage* so that bugs can be uncovered *during testing* and *not in production* [DMT$^+$16]. The need for better testing techniques is not specific to developer teams inside Microsoft; other companies, such as Amazon and Google, have acknowledged [CGR07, NRZ$^+$15] that testing methodologies have to improve to be able to reason about the correctness of increasingly more complex distributed systems that are used in production.

Recently, the Amazon Web Services (AWS) team used formal methods "to prevent serious but subtle bugs from reaching production" [NRZ$^+$15]. The gist of their approach is to extract the high-level logic from a complex distributed system, represent this logic as specifications in the expressive TLA+ [Lam02] language, and finally verify the specifications using a model checker. While highly effective, as demonstrated by its use in AWS, the TLA+ approach falls short of "verifying that executable code correctly implements the high-level specification" [NRZ$^+$15], and the AWS team admits that it is "not aware of any such tools that can handle distributed systems as large and complex as those being built at Amazon" [NRZ$^+$15].

In our experience, checking high-level specifications is necessary but not sufficient, due to the (often wide) gap between the specifications and the implementation. Our goal is to narrow this gap. We propose an approach that validates high-level specifications directly on the executable code. Our approach is different from prior work that required developers to build the whole system using a domain specific language [KAJV07, DJP+15, WWP+15]. Instead, we allow developers to test *existing* production code by writing test harnesses in the P# asynchronous programming language (see Chapter 4).[2] P# is built on top of C#, and thus can be applied to any .NET system, which significantly lowered the acceptance barrier for adoption of our approach by engineers in Microsoft Azure.

In §5.2, we discuss how P# can be used to partially-model, specify properties of correctness, and test distributed systems. In §5.3, we present a case study of using P# inside Microsoft to test the Azure Storage vNext system. Finally, in §5.4, we evaluate the use of P# for testing vNext, as well as other distributed systems.

5.2 Overview

The goal of the work described in this chapter is to find bugs in distributed systems *before* these bugs reach production and affect the users. Distributed systems typically consist of multiple components that interact with each other via message passing. If messages—or unexpected failures and timeouts—are not handled properly, they can lead to subtle, but serious, bugs. To find these bugs, we first *partially-model* a distributed system in P# (see §5.2.2 for an example), and then use the P# controlled testing engine (see §4.5) to explore interleavings between the *distributed events* (e.g. client requests, failures and timeouts) of the partially-modelled system.

Modelling a distributed system in P# involves three core activities. First, the developer must modify the original system so that messages are not sent through the real network, but are instead dispatched via the P# `send` method (see §4.2.1). Such modification does not need to be invasive, as it can be performed using dynamic method dispatch, a C# language feature widely used in industry for testing. Second, the developer must write a P# *test harness* that drives the system towards interesting behaviours by nondeterministically triggering distributed events (see §5.2.2). The test harness is essentially a model of the environment of the distributed system. The purpose of the first two modelling activities is to explicitly declare all known sources of nondeterminism in the system using P#; our built-in testing engine can then capture, control and explore these declared nondeterministic choices during testing. Finally, the developer must specify the criteria for correctness

[2]Note that P# can be used to develop an entire system from scratch, similar to previous work [KAJV07, DJP+15, WWP+15]. However, in this chapter we focus on an approach where real components of an existing distributed system (written in C#) are tested using a test harness written in P#.

of an execution of the system-under-test. Specifications written in P# can encode either *safety* or *liveness* [Lam77] properties (see §5.2.3 and §5.2.4).

During testing, P# is aware of all sources of nondeterminism that were declared during modelling, and exploits this knowledge to create a scheduling point each time a nondeterministic choice has to be taken (see §5.3.3 and §5.3.4). The P# testing engine will serialise (in a single-box and single-process) the distributed system, and repeatedly execute it from start to completion, each time exploring a (potentially) different set of nondeterministic choices (as well as P# machine scheduling decisions, as discussed in §4.5), until it either reaches a user-supplied bound (e.g. in number of explored schedules or time), or it hits a safety or liveness property violation. The P# testing process is fully automatic and has no false positives for safety properties (assuming an accurate test harness). However, P# might report false positives for liveness properties due to our use of heuristics (see §5.2.4). After a bug is discovered, P# generates a trace that represents the buggy schedule, which can then be replayed to reproduce the bug. In contrast to logs typically generated during production, the P# trace provides a global order of all communication events, and thus is easier to debug.

5.2.1 An Example Distributed System

Figure 5.1 presents the pseudocode of a simple distributed storage system that was contrived for the purposes of explaining our P# testing methodology. The system consists of a client, a server and three storage nodes (SNs). The client sends the server a `ClientReq` message that contains data to be replicated (`DataToReplicate`), and then waits to get an acknowledgement (by calling the `receive` method) before sending the next request. When the server receives `ClientReq`, it first stores the data locally (in the `Data` field), and then broadcasts a `ReplReq` message to all SNs. When an SN receives `ReplReq`, it handles the message by storing the received data locally (by calling the `store` method). Each SN has a timer installed, which sends periodic `Timeout` messages. Upon receiving `Timeout`, an SN sends a `Sync` message to the server that contains the storage log. The server handles the `Sync` message by calling the `isUpToDate` method to check if the SN log is up-to-date. If it is not, the server sends a repeat `ReplReq` message to the outdated SN. If the SN log is up-to-date, then the server increments a replica counter by one. Finally, when there are three replicas available, the server sends an `Ack` message to the client.

There are two bugs in the above example. The first bug is that the server does not keep track of unique replicas. The replica counter increments upon each up-to-date `Sync`, even if the syncing SN is already considered a replica. This means that the server might send

Server

```
receive msg {
  case ClientReq:
    this.Data = message.Val;
    // Replicate data to all nodes
    foreach (sn in this.Nodes)
      sn.send(ReplReq, this.Data);

  case Sync:
    Node node = message.Id;
    Log log = message.Log;
    doSync(node, log);
}
```

```
doSync (Node sn, Log log) {
    // If the storage log is not
    // up-to-date, replicate
    if (!isUpToDate(log))
      sn.send(ReplReq, this.Data);
    else {
      this.NumReplicas++;
      if (this.NumReplicas == 3)
        this.Client.send(Ack);
    }
}
```

Storage Node

```
receive msg {
  case ReplReq:
    // Store received data
    store(message.Val);

  case Timeout:
    // Send server the log
    // upon timeout
    this.Server.send(Sync,
      this.Id, this.Log);
}
```

Client

```
while (hasNextRequest()) {
  this.Server.send(ClientReq,
    this.DataToReplicate);
  receive(Ack); // Wait for ack
}
```

Timer

```
// Send timeout to node
if (this.Countdown == 0)
  this.SN.send(Timeout);
```

Figure 5.1: Pseudocode of a simple distributed storage system that is responsible for replicating the data sent by a client.

an `Ack` message when fewer than three replicas exist, which is erroneous behaviour. The second bug is that the server does not reset the replica counter to 0 upon sending an `Ack` message. This means that when the client sends another `ClientReq` message, it will never receive `Ack`, and thus block indefinitely.

5.2.2 Modelling the Example System

To test the example of Figure 5.1 in a controlled manner, the developer must first create a P# test harness, and then specify the correctness properties of the system. Figure 5.2 illustrates a test harness that can find the two bugs discussed in §5.2.1. Each box in the figure represents a concurrently running P# state-machine. We use three kinds of boxes: (i) a box with rounded corners and thick border denotes a real component wrapped inside a P# machine; (ii) a box with thin border denotes a modelled component; and (iii) a box with dashed border denotes a *monitor*, a special P# machine used for safety or liveness checking (see §5.2.3 and §5.2.4, respectively). A solid arrow represents an event being sent between two non-monitor machines, while a dashed arrow represents a notification event being sent from a non-monitor machine to a monitor.

We do not model the server component since we want to test its actual implementation.

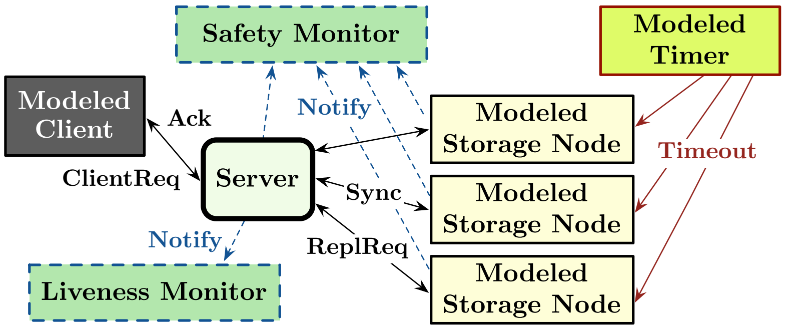

Figure 5.2: The P# test harness for the example in Figure 5.1.

The server is wrapped inside a P# machine, which is responsible for (i) sending the system messages (as payload of a P# event) via the P# `send(...)` method (see §4.2.1), instead of the real network, and (ii) delivering received messages to the wrapped component. We model the SNs so that they store the replicated data in memory rather than on disk (since the latter can be inefficient during testing). We also model the client so that it can drive the system by repeatedly sending a nondeterministically generated `ClientReq` event, and then waiting for an `Ack` event. Finally, we model the timer so that P# takes control of all time-related nondeterminism in the system. This allows the P# testing engine to control when a `Timeout` event will be sent to the SNs during testing, and (systematically) explore different schedules.

P# uses object-oriented language features such as interfaces and dynamic method dispatch to connect the real code with the modelled code. Developers in industry are used to working with such features, and heavily employ them in testing production systems. In our experience, this significantly lowers the bar for engineering teams inside Microsoft to embrace P# for testing.

In §5.2.3 and §5.2.4, we discuss how safety and liveness specifications can be expressed in P# to check if the example system is correct. The details of how P# was used to model and test a real distributed system in Microsoft are covered in §5.3. Interested readers can also refer to the P# GitHub repository[3] to find a manual and samples (e.g. Paxos [Lam98], ChainReplication [vRS04] and Raft [OO14]).

5.2.3 Specifying Safety Properties in P#

Safety property specifications generalise the notion of source code assertions; a safety property violation is a finite trace leading to an erroneous state. P# supports the usual

[3]`https://github.com/p-org/PSharp`

assertions for specifying safety properties that are local to a P# machine (see §4.2.1), and also provides a way to specify global assertions in the form of a *safety monitor* [DJP+15], a special P# machine that can receive, but not send, events.

A safety monitor maintains local state that is modified in response to events received from ordinary (non-monitor) machines. This local state is used to maintain a history of the computation that is relevant to the property being specified. An erroneous global behaviour is flagged via an assertion on the private state of the safety monitor. Thus, a monitor cleanly separates the instrumentation state required for specification (inside the monitor) from the program state (outside the monitor).[4] Safety monitors are reminiscent of the rule specifications used to define and check temporal safety properties in the Static Driver Verifier (SDV) project [BBC+06].

The first bug in the example of §5.2.1 is a safety bug. To find it, the developer can write a safety monitor (see Figure 5.2) that contains a map from unique SN ids to a Boolean value, which denotes if the SN is a replica or not. Each time an SN replicates the latest data, it notifies the monitor to update the map. Each time the server issues an `Ack`, it also notifies the monitor. If the monitor detects that an `Ack` was sent without three replicas actually existing, a safety violation is triggered. The following code snippet shows the P# source code for this safety monitor:

```
monitor SafetyMonitor {
  // Map from unique SNs ids to a boolean value
  // that denotes if a node is replica or not
  Dictionary<int, bool> replicas;

  start state Checking {
    entry {
      var node_ids = (HashSet<int>)payload;
      this.replicas = new Dictionary<int, bool>();
      foreach (var id in node_ids) {
        this.replicas.Add(id, false);
      }
    }

    // Notification that the SN is up-to-date
    on NotifyUpdated do {
      var node_id = (int)payload;
      this.replicas[node_id] = true;
    };
```

[4]An alternative (to the use of monitors) approach would be to add and maintain instrumentation variables in the distributed system source code. However, this approach would be intrusive. For this reason, we opted to borrow the concept of monitors from the P language [DJP+15], and implement it in P#.

```

    // Notification that the SN is out-of-date
    on NotifyOutdated do {
      var node_id = (int)payload;
      this.replicas[node_id] = false;
    };

    // Notification that an Ack was issued
    on NotifyAck do {
      // Assert that 3 replicas exist
      assert(this.replicas.Count(n => n.Value) == 3);
    };
  }
}
```

5.2.4 Specifying Liveness Properties in P#

Liveness property specifications generalise nontermination; a liveness property violation is an infinite trace that exhibits lack of progress. Typically, a liveness property is specified via a temporal logic formula [Pnu77, Lam94]. We take a different approach and allow the developers to write a *liveness monitor* [DJP+15]. Similar to a safety monitor, a liveness monitor can receive, but not send, events.

A liveness monitor contains two special states: the *hot* and the *cold* state. The hot state denotes a point in the execution where progress is required, but has not happened yet; e.g. a node has failed, but a new one has not launched yet. A liveness monitor transitions to the hot state when it is notified that the system must make progress. A liveness monitor leaves the hot state and enters the cold state when it is notified that the system has progressed. An infinite execution is erroneous if the liveness monitor stays in the hot state for an infinitely long period of time. Our liveness monitors can encode arbitrary temporal logic properties.

A liveness property violation is witnessed by an *infinite* execution in which all concurrently executing P# machines are *fairly* scheduled. Since it is impossible to generate an infinite execution by executing a program for a finite amount of time, our implementation of liveness checking in P# approximates an infinite execution using a simple, yet effective, heuristic: we consider an execution longer than a large (user-supplied) bound as an "infinite" execution [KAJV07, MQ08]. An advantage of using this heuristic is that checking for fairness is not relevant, due to our pragmatic use of a large bound. However, our approach has two limitations: (i) bounding the execution can generate false positives, because the liveness property might get satisfied only after the bound has been reached; and (ii) the reported trace can be long, requiring further minimisation (which P# does not currently perform). We plan to address these limitations in future work (see §6.2).

The second bug in the example of §5.2.1 is a liveness bug. To detect it, the developer can write a liveness monitor (see Figure 5.2) that transitions from a hot state, which denotes that the client sent a `ClientReq` and waits for an `Ack`, to a cold state, which denotes that the server has sent an `Ack` in response to the last `ClientReq`. Each time a server receives a `ClientReq`, it notifies the monitor to transition to the hot state. Each time the server issues an `Ack`, it notifies the monitor to transition to the cold state. If the monitor is in a hot state when a finite execution terminates, or remains in a hot state for longer than the large (user-supplied) bound, a liveness violation is triggered. The following code snippet shows the P# source code for this liveness monitor:

```
monitor LivenessMonitor {
  start hot state Progressing {
    // Notification that the server issued an Ack
    on NotifyAck do {
      raise(Unit);
    };
    on Unit goto Progressed;
  }

  cold state Progressed {
    // Notification that server received ClientReq
    on NotifyClientRequest do {
      raise(Unit);
    };
    on Unit goto Progressing;
  }
}
```

5.3 Case Study: Microsoft Azure Storage vNext

Microsoft Azure Storage [CWO+11] is a production cloud storage system that provides customers the ability to store seemingly limitless amounts of data.[5] It has grown from tens of petabytes in 2010 to exabytes in 2015, and the total number of stored objects is exceeding 60 trillion [Gre15].

Azure Storage vNext [DMT+16] is the next generation storage system currently being developed for Microsoft Azure, where the primary design target is to scale the storage

[5]https://azure.microsoft.com/en-gb/services/storage/

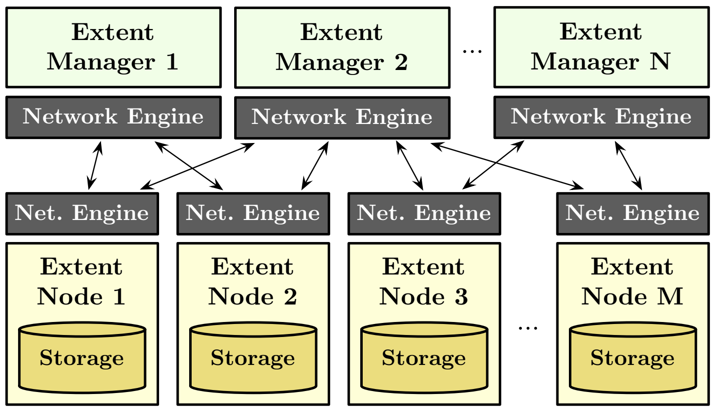

Figure 5.3: Top-level components for extent management in vNext.

capacity by more than 100×. Similar to the current system, vNext employs containers, called *extents*, to store data. Extents can be several gigabytes each, consist of many data blocks, and are replicated over multiple *Extent Nodes* (ENs). However, in contrast to the current system, which uses a Paxos-based [Lam98], centralized mapping from extents to ENs [CWO+11], vNext achieves scalability by using a *distributed mapping*. In vNext, extents are divided into partitions, with each partition managed by a lightweight *Extent Manager* (ExtMgr). This partitioning is illustrated in Figure 5.3.

One of the main responsibilities of an ExtMgr is to ensure that every managed extent maintains enough *replicas* in the system. To achieve this, an ExtMgr receives frequent periodic *heartbeat* messages from every EN that it manages. EN failure is detected by missing heartbeats. An ExtMgr also receives less frequent, but still periodic, *synchronisation reports* from every EN. A synchronisation report lists all the extents (and associated metadata) stored on the corresponding EN. Based on these two types of messages, an ExtMgr can identify which ENs have failed, and which extents are affected by the failure and are missing replicas as a result. The ExtMgr then schedules tasks to repair the affected extents, and distributes these tasks to the remaining ENs. The ENs then repair the extents from the existing replicas, and lazily update the ExtMgr via their next periodic synchronisation reports. All the communications between an ExtMgr and the ENs occur via network engines installed in each component of vNext (see Figure 5.3).

To ensure correctness, the developers of vNext have instrumented extensive, multiple levels of testing [DMT+16]:

1. *Unit testing*, in which emulated heartbeats and synchronisation reports are sent to an ExtMgr. These unit tests check that the messages are processed as expected.

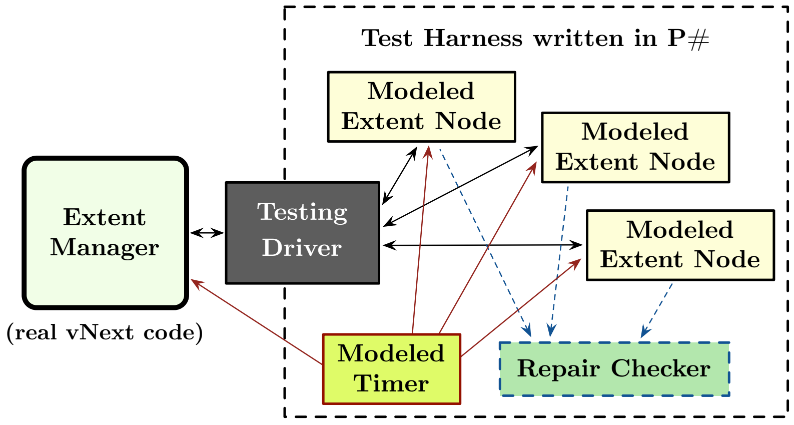

Figure 5.4: Real Extent Manager with its P# test harness (each box represents one P# state-machine).

2. *Integration testing*, in which an ExtMgr is launched together with multiple ENs. An EN failure is subsequently injected. These integration tests check that the affected extents are eventually repaired.

3. *Stress testing*, in which an ExtMgr is launched together with multiple ENs and many extents. These stress tests keep repeating the following process: inject an EN failure, launch a new EN, and check that the affected extents are eventually repaired.

Despite the extensive testing efforts, the vNext developer team was plagued for months by an elusive liveness bug in the ExtMgr logic [DMT+16]. All unit tests and integration tests successfully passed on each run. However, the stress tests failed *from time to time* after very long executions: the vNext developers noticed that in these cases, certain replicas of some extents failed without subsequently being repaired. This liveness bug proved difficult to identify, reproduce and troubleshoot for a number of reasons. First, the techniques used to test Azure Storage vNext are not suitable for detecting liveness bugs. Second, this bug seemed to manifest only in very long executions. Finally, by the time that the bug did manifest, very long execution traces had been collected, which made manual inspection tedious and ineffective.

To uncover the liveness bug in Azure Storage vNext, we collaborated with the developers of vNext and wrote a test harness in P#. The developers expected that it was more likely for the bug to occur in the ExtMgr logic, rather than in the EN logic. Hence, we focused on testing the real ExtMgr component using modelled in P# ENs. The vNext test harness consists of the following P# machines (as shown in Figure 5.4):

ExtentManager acts as a thin wrapper for the real ExtMgr component in Azure Storage

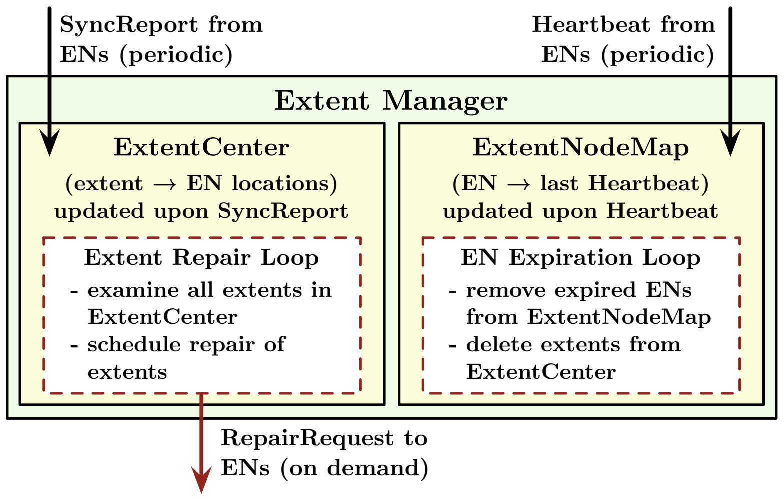
SyncReport from ENs (periodic)
Heartbeat from ENs (periodic)
Extent Manager
ExtentCenter
(extent → EN locations) updated upon SyncReport
Extent Repair Loop
- examine all extents in ExtentCenter
- schedule repair of extents
ExtentNodeMap
(EN → last Heartbeat) updated upon Heartbeat
EN Expiration Loop
- remove expired ENs from ExtentNodeMap
- delete extents from ExtentCenter
RepairRequest to ENs (on demand)

Figure 5.5: Internal components of the real Extent Manager in Microsoft Azure Storage vNext.

vNext (see §5.3.1).

ExtentNode is a simple model of an EN (see §5.3.2).

Timer exploits the nondeterministic choice generation available in P# to model timeouts in the system (see §5.3.3).

TestingDriver is responsible for driving testing scenarios, relaying messages between machines, and injecting failures (see §5.3.4).

RepairChecker collects EN-related state to check if the desired liveness property is *eventually always* satisfied (see §5.3.5).

5.3.1 The ExtentManager Machine

The real ExtMgr contains two important data structures (see Figure 5.5): `ExtentCenter` and `ExtentNodeMap`. The `ExtentCenter` maps extents to their hosting ENs. It is updated upon receiving a periodic synchronisation report from an EN. Recall that a synchronisation report from an EN lists all the extents stored at that particular EN. The purpose of the synchronisation report is to update the ExtMgr's possibly out-of-date view of the EN with the ground truth. The `ExtentNodeMap` maps ENs to their latest heartbeat times.

Internally, ExtMgr runs a periodic EN *expiration loop* that is responsible for removing ENs that have been missing heartbeats for an extended period of time, as well as cleaning up the corresponding extent records in `ExtentCenter`. In addition, ExtMgr runs a periodic extent *repair loop* that examines all the records in `ExtentCenter`, identifies extents with missing replicas, schedules extent repair tasks and sends them to the ENs.

The real ExtMgr uses a network engine to asynchronously send messages to ENs (see Figure 5.3). The P# test harness models the network engine by overriding its original implementation, as shown in the following code snippet:

```
// Network interface in vNext
class NetworkEngine {
  public virtual void SendMessage(Socket s, Message msg);
}

// Modelled engine for intercepting Extent Manager messages
class ModelNetEngine : NetworkEngine {
  public override void SendMessage(Socket s, Message msg) {
    // Intercept and relay Extent Manager messages
    PSharpRuntime.SendEvent(TestingDriver, new ExtMgrMsgEvent(s, msg));
  }
}
```

The modelled network engine intercepts all outbound messages from ExtMgr, and invokes the `PSharpRuntime.SendEvent` method (see §4.2.2) to asynchronously relay the messages to the `TestingDriver` machine, which is responsible for dispatching these messages to the corresponding ENs (see §5.3.4).

To connect the real ExtMgr (which is the vNext component we are interested in testing) with the P# test harness, we wrapped the ExtMgr component inside the `ExtentManager` machine and replaced the real vNext network engine with the modelled network engine, as illustrated in the following code snippet:

```
machine ExtentManager {
  ExtMgr extMgr; // Real vNext extent manager

  start state Init {
    entry {
      this.extMgr = new ExtMgr();
      // Replaces the network engine with a modelled version
      this.extMgr.networkEngine = new ModelNetEngine();
      this.extMgr.DisableTimers(); // Disables internal timers
      raise(Unit);
    }
    on Unit goto Active;
  }

  state Active {
```

```
16      // Relay messages from Extent Node to Extent Manager
17      on ExtentNodeMessageEvent do {
18        ExtentNodeMessage msg = (ExtentNodeMessage)payload;
19        this.extMgr.ProcessMessage(msg);
20      };
21
22      // Extent repair loop driven by external timer
23      on TimeoutEvent do {
24        this.extMgr.ProcessEvent(new ExtentRepairEvent());
25      };
26    }
27  }
```

Intercepting all network messages and dispatching them via P# is important for two reasons. First, it allows P# to control and explore the interleavings between all asynchronous event-handlers in the system. Second, the modelled network engine could leverage the support for controlled nondeterministic choices in P#, and choose to drop the messages in a nondeterministic fashion, in case emulating message loss is desirable (we do not show the corresponding code snippet for brevity).

Messages coming from `ExtentNode` machines do *not* go through the modelled network engine; they are instead delivered to the `ExtentManager` machine and trigger an action that invokes the messages on the wrapped ExtMgr by calling `extMgr.ProcessMessage` (see line 20 of the `ExtentManager` code snippet). The benefit of this approach is that the real ExtMgr can be tested without modifying the original communication code; the ExtMgr is simply unaware of the P# test harness and behaves as if it is running in an actual distributed environment and communicating with real ENs.

5.3.2 The ExtentNode Machine

The `ExtentNode` machine is a simplified version of the original EN. This machine omits most of the complex details of a real EN, and only models the necessary logic for testing.[6] This modelled logic includes: repairing an extent from its replica, and sending synchronisation reports and heartbeat messages periodically to `ExtentManager`.

The P# test harness leverages components of the real vNext whenever it is appropriate. For example, `ExtentNode` re-uses the `ExtentCenter` data structure, which is used inside a real EN for extent bookkeeping. In the modelled extent repair logic, `ExtentNode` takes action upon receiving an extent repair request from the `ExtentManager` machine. It sends

[6]Note that deciding which details to omit from a model requires domain expertise, and can lead to missed bugs or false alarms if done incorrectly.

a copy request to a source `ExtentNode` machine where a replica is stored. After receiving an `ExtentCopyResponseEvent` event from the source machine, it updates the `ExtentCenter`, as illustrated in the following code snippet:

```
machine ExtentNode {
  machine extentManager;
  // We leverage real vNext components whenever appropriate
  ExtentNode.ExtentCenter extentCenter;
  ...
  state Active {
     // Extent copy response from source replica
    on ExtentCopyResponseEvent do {
      ExtentCopyResponse response = (ExtentCopyResponse)payload;
      if (this.HasCopySucceeded(response)) {
        var record = this.GetExtentRecord(response);
        this.extentCenter.AddOrUpdate(record); // Updates extentCenter
      }
    };

    // Extent node synchronisation logic
    on TimeoutEvent do {
      // Prepares synchronisation report
      var report = this.extentCenter.GetSyncReport();
      send(this.extentManager, ExtentNodeMessageEvent, report);
    };

    // Extent node failure logic
    on FailureEvent do {
      // Invokes the synchronous P# monitor(...) method to
      // notify the liveness monitor that this EN failed
      monitor<RepairChecker>(ENFailedEvent, this)
      raise(halt); // Raising this special event terminates the machine
    };
  }
}
```

In the modelled EN synchronisation logic, the `ExtentNode` machine is driven by an external timer modelled using P# (see §5.3.3). `ExtentNode` prepares a new synchronisation report by invoking the `extentCenter.GetSyncReport` method, and then asynchronously sends this report to the `ExtentManager` machine using the P# `send`. The `ExtentNode` machine also includes failure-related logic (see §5.3.4).

5.3.3 The Timer Machine

System correctness should *not* hinge on the frequency of any individual timer. Hence, it makes sense to delegate all nondeterminism due to timing-related events to P#. To achieve this, all the internal timers of ExtMgr are disabled (see line 9 of the `ExtentManager` code snippet in §5.3.1), and the EN expiration loop and the extent repair loop are driven instead by timers modelled in P#, an approach also used in previous work [DJP+15].[7] Similarly, `ExtentNode` machines do *not* have internal timers either. Their periodic heartbeats and synchronisation reports are also driven by timers modelled in P#.

The following code snippet shows the `Timer` machine in the P# test harness:

```
machine Timer {
  machine target;
  ...
  state Loop {
    on TimerTickEvent do {
      // Nondeterministic choice controlled by P#
      if (*) {
        send(this.target, TimeoutEvent);
      }
      raise(TimerTickEvent); // Loop
    };
    on TimerTickEvent goto Loop;
  }
}
```

The `Timer` machine uses the special `*` guard expression (available in P#), which nondeterministically flips a coin that is controlled by P# during testing. Using `*` allows `Timer` to nondeterministically send a timeout event to its target (which in our case is either the `ExtentManager` or an `ExtentNode` machine). The P# testing engine has the freedom to schedule arbitrary interleavings between these timeout events and all other regular system events, to drive the system towards potentially buggy execution schedules. Controlling the timeouts enables fast detection of bugs that might otherwise take a long time to actually appear if the test harness was using real timers.

5.3.4 The TestingDriver Machine

The `TestingDriver` machine drives the following two testing scenarios:

[7]We had to make a minor change to the real ExtMgr source code to facilitate modelling: we added the `DisableTimers` method, which disables the internal timers of ExtMgr so that they can be replaced with timers modelled in P#.

1. In the first testing scenario, TestingDriver launches one ExtentManager and three ExtentNode machines, with a single extent on one of the ENs. It then waits for the extent to be replicated at the remaining ENs.

2. In the second testing scenario, TestingDriver fails one of the ExtentNode machines and launches a new ExtentNode. It then waits for the extent to be repaired on the newly launched ExtentNode machine.

The following code snippet illustrates how the TestingDriver machine injects nondeterministic failures in a manner than can be controlled by the P# runtime:

```
machine TestingDriver {
  HashSet<machine> extentNodes; // EN machines
  ...
  state InjectingFailure {
    entry {
      // Nondeterministically chooses an EN using P#
      machine node = PSharpRuntime.Choose(this.extentNodes);
      // Fails the chosen EN
      send(node, FailureEvent);
    }
  }
}
```

TestingDriver calls the PSharpRuntime.Choose method to nondeterministically choose an ExtentNode machine, and then sends FailureEvent to the chosen machine to emulate an EN failure. The PSharpRuntime.Choose method is similar to the * expression (which was discussed in §5.3.3). The main difference between these two APIs is that * is used to generate a nondeterministic boolean, whereas PSharpRuntime.Choose is used to choose an object of type machine from a collection of such objects. As shown in line 24 of the ExtentNode code snippet in §5.3.2, the chosen ExtentNode machine processes FailureEvent, notifies the liveness monitor of its failure (see §5.3.5) and terminates itself by raising the special P# event halt.[8] P# not only enumerates the interleavings of asynchronous event-handlers, but also the values returned by calls to PSharpRuntime.Choose, thus enumerating different failure scenarios.

[8]When a machine handles the special P# event halt (in our case immediately due to raising, as discussed in §4.2.1), any resources taken by the machine are released, the machine stops handling any events in its queue, the P# runtime drops any events sent to the machine, and finally the runtime removes all references to the halted machine instance from its internal data structures, so the machine can get garbage collected by the .NET runtime.

5.3.5 The RepairChecker Liveness Monitor

`RepairChecker` is a liveness monitor (see §5.2.4). Whenever an EN fails, `RepairChecker` is notified with an `ENFailedEvent` event (see line 27 of the `ExtentNode` machine code snippet in §5.3.2). As soon as the number of available extent replicas falls below a specified target (three replicas in our P# test harness), the monitor transitions into the hot `Repairing` state, waiting for all missing replicas to be eventually repaired. Whenever an extent replica is repaired, `RepairChecker` is notified with an `ExtentRepairedEvent` event. When the replica number reaches again the target, the monitor transitions into the cold `Repaired` state, as illustrated in the following code snippet:

```
monitor RepairChecker {
  HashSet<machine> extentNodesWithReplica;
  ...
  start cold state Repaired {
    on ENFailedEvent do {
      machine node = (machine)payload;
      this.extentNodesWithReplica.Remove(node);
      if (this.replicaCount < TestHarness.REPLICA_COUNT_TARGET) {
        raise(Unit);
      }
    };
    on Unit goto Repairing;
  }

  hot state Repairing {
    on ExtentRepairedEvent do {
      machine node = (machine)payload;
      this.extentNodesWithReplica.Add(node);
      if (this.replicaCount == TestHarness.REPLICA_COUNT_TARGET) {
        raise(Unit);
      }
    };
    on Unit goto Repaired;
  }
}
```

During testing, P# checks that the `RepairChecker` monitor should eventually always end up in the cold state. Otherwise, `RepairChecker` is stuck in the hot state for *infinitely* long. This indicates that the corresponding execution sequence results in an extent replica never being repaired, which is the liveness bug that we are trying to detect.

5.3.6 Liveness Bug in Azure Storage vNext

It took approximately ten seconds for the P# testing engine to report the first occurrence of a liveness bug in Azure Storage vNext (see §5.4). Upon examining the debug trace, the vNext developers were able to quickly confirm the bug. However, the original trace did not include sufficient details to allow the developers to identify the root cause of the problem. Fortunately, running the P# test harness did not require much time, which enabled the developer team to quickly iterate and add more refined debugging outputs in each iteration. After several iterations, the developers managed to pinpoint the exact culprit and propose a solution for fixing the bug. Once the proposed solution was implemented, the developers ran the test harness again. No bugs were found during 100,000 testing iterations, a process that only required a couple of minutes. We believe that one reason why this testing process worked so well is because the developers of vNext already had a hunch about this bug. This hunch helped to guide the modelling of the system.

The liveness bug occurs in the second testing scenario (see §5.3.4), where `TestingDriver` fails one of the `ExtentNode` machines and launches a new one. `RepairChecker` transitions to the hot repairing state, and is stuck in that state for infinitely long.

The following is one particular execution sequence resulting in this liveness bug: (i) EN_0 fails and is detected by the EN expiration loop; (ii) EN_0 is removed from `ExtentNodeMap`; (iii) `ExtentCenter` is updated and the replica count drops from 3 (which is the target) to 2; (iv) ExtMgr receives a *delayed* synchronisation report from EN_0; (v) `ExtentCenter` is updated and the replica count increases again from 2 to 3. However, this is problematic since the replica count is now equal to the target, and thus the extent repair loop will never schedule any repair task. At the same time, there are only two *true* extent replicas in vNext, which is one less than the target. This execution sequence leads to one missing replica; repeating this process two more times would result in all replicas missing, but ExtMgr would still think that all replicas are healthy. If released to production, this bug would have caused a very serious incident of customer data unavailability.

The culprit is in step (iv), where ExtMgr receives a synchronisation report from EN_0 *after* deleting EN_0. This interleaving is quickly exposed by the P# testing engine, since it is able to explore arbitrary interleavings between events. The interleaving may also occur, albeit much less frequently, during stress testing due to messages being delayed in the network. This explains why the liveness bug only occurs from time to time during stress testing and requires long executions to manifest. In contrast, P# allows the bug to manifest quickly, the developers to iterate rapidly, the culprit to be identified promptly, and the fix to be tested effectively and thoroughly, all of which have the potential to vastly

Table 5.1: Statistics from modelling the environment of the three Azure-based distributed systems, and the number of bugs found in each system. The ⋆ symbol denotes a bug that is "awaiting confirmation".

	System		**P# Test Harness**			
System-under-test	**#LoC**	**#B**	**#LoC**	**#M**	**#ST**	**#A**
Azure Storage vNext Extent Manager	19,775	1	684	5	11	17
Live Table Migration	2,267	11	2,275	3	5	10
CScale & Azure Service Fabric Libraries	31,959	⋆1	6,534	13	21	87

increase the productivity of distributed system development.

5.4 Evaluation

We report our experience of applying P# to Microsoft Azure Storage vNext (see §5.3). We also include results from using P# to model and test Live Table Migration,[9] a protocol for transparently migrating a key-value data set between two Azure Tables [Hog15] while an application is accessing this data set, and CScale [FRR+12], a big data stream processing system built on top of Azure Service Fabric [Mic16c]. Live Table Migration and CScale were modelled and tested by our collaborators (acknowledged in §1.4); we discussed these two case studies in a recent publication [DMT+16]. The main motivation behind including results from Live Table Migration and CScale in this thesis, is to show that our distributed systems modelling and testing methodology using the P# language is general.

In this section, we aim to answer the following two questions:

1. How much human effort is required to model the environment of a distributed system using P#?

2. How much computational time was spent in testing a distributed system with P#?

Furthermore, the experience in real-world distributed systems testing that we gained during this work has shed light on some important issues. In §5.4.3, we report on the main lessons learned in the hope that they may help to guide similar future efforts.

5.4.1 Cost of Environment Modelling

Environment modelling is a core activity of using P#. It is required for *closing* an asynchronous distributed system to make it amenable to controlled testing (see §4.5). Table 5.1

[9]`https://github.com/jerickmsft/MigratingTable`

presents statistics for the three Azure-based case studies. The two columns under "System" refer to the real system-under-test, while the four columns under "P# Test Harness" refer to the test harness written in P#. We report: lines of source code for the real system (#LoC); number of bugs found in the real system (#B); lines of P# source code for the test harness (#LoC); number of machines (#M); number of state transitions (#ST); and number of actions (#A).

We now discuss the cost of environment modelling for each case study:

Azure Storage vNext Modelling the environment of the Extent Manager in Azure Storage vNext required approximately two person weeks of part-time developing (joint effort between the vNext architect and the author of this thesis). The P# test harness for this system is the smallest (in lines of source code) from the three case studies. This was because this modelling exercise aimed to reproduce the particular liveness bug that was haunting the vNext developers (see §5.3.6), rather than model the entire vNext system.

Live Table Migration Developing both the protocol and its corresponding P# test harness took approximately five person weeks by a single developer who had just started learning P#. The harness was developed in parallel with the actual system. This differs from the other two case studies, where the modelling activity occurred independently and after the development process.

CScale & Azure Service Fabric Modelling Azure Service Fabric (to test CScale) required approximately five person months by a team of four researchers and research interns who had expertise in P#, but no expertise in CScale and Service Fabric. Although modelling Service Fabric required a significant amount of time, it is a one-time effort, which only needs incremental refinement with each release. The advantage of having a P# model of Service Fabric is that it can be used to test arbitrary services built on top of it (e.g. CScale).

Note that trying to measure the cost of environmental modelling in person time is arguably hard. On one hand, the researchers and research interns mentioned above (as well as the author of this thesis) are P# experts, but lack significant practical experience with the above distributed systems. On the other hand, the distributed system developers, are not experts in P#, but have expertise in the systems they developed. In our experience (from modelling and testing Azure Storage vNext), close collaboration between a domain expert and a P# expert enables faster progress.

5.4.2 Cost of Controlled Testing

P# managed to uncover 13 serious bugs in the case studies. These bugs were hard to find with traditional testing techniques, but P# managed to quickly find and reproduce them. According to the developers, the P# traces were useful, as it allowed them to understand the bugs and timely fix them. After all the discovered bugs were fixed, we added flags to allow them to be individually re-introduced, for purposes of evaluation.

Table 5.2 presents the results from running the P# controlled testing engine on each case study with a re-introduced bug. The CS column shows which case study corresponds to each bug: "1" is for Azure Storage vNext; and "2" is for Live Table Migration. P# found a potential bug in CScale, but we do not include it in the table as it has not yet been confirmed. We performed all experiments on a 2.50GHz Intel Core i5-4300U CPU with 8GB RAM running Windows 10 Pro 64-bit. We configured the P# controlled testing engine to perform 100,000 testing iterations. All reported times are in seconds.

We decided to use the random and randomised priority-based [BKMN10] P# schedulers (see §4.5), because random scheduling has proven to be efficient for finding concurrency bugs [TDB16, DDK$^+$15]. The random seed for the two schedulers was generated using the current system time. The priority-based scheduler was configured with a budget of 2 random machine priority change switches per execution (see §4.5).

For the vNext case study, both schedulers managed to reproduce the ExtentNodeLivenessViolation bug within 11 seconds. Note that the number of nondeterministic choices (#NDC) for finding this bug is much higher than the rest of the bugs in Table 5.2. This is because ExtentNodeLivenessViolation is a liveness bug; as discussed in §5.2.4 we leave the program to run for a long time, before checking if the liveness property holds, to emulate an "infinite" execution.

For the Live Table Migration case study, we evaluated the corresponding P# test harness (discussed in [DMT$^+$16]) on eleven bugs, including eight *organic* bugs that actually occurred in development and three *notional* bugs (denoted by $*$), which are code changes that the protocol developers regarded as interesting ways of making the system incorrect. The test harness found seven of the organic bugs, which are listed first in Table 5.2. The remaining four bugs (marked with $\diamond$) were not caught with the default P# test harness in the 100,000 executions. We believe this is because the inputs and schedules that trigger them are too rare in the used distribution. To confirm this, the developers wrote a custom test harness for each bug with a specific input that triggers it, and were able to quickly reproduce these four bugs; Table 5.2 shows the results of these runs. Note that the random scheduler only managed to trigger seven of the Live Table Migration bugs; we had to use

Table 5.2: Results from running the P# random and priority-based controlled testing schedulers for 100,000 executions. We report: whether the bug was found (BF?) (✓ means it was found, ◇ means it was found only using a custom test harness, and ✗ means it was not found); time in seconds to find the bug; and number of nondeterministic choices made in the first buggy execution (#NDC).

		P# Controlled Testing Schedulers					
		Random			Randomised Priority-based		
CS	Bug Identifier	BF?	Time (s)	#NDC	BF?	Time (s)	#NDC
1	ExtentNodeLivenessViolation	✓	10.56	9,000	✓	10.77	9,000
2	QueryAtomicFilterShadowing	✓	157.22	165	✓	350.46	108
2	QueryStreamedLock	✓	2,121.45	181	✓	6.58	220
2	QueryStreamedBackUpNewStream	✗	-	-	✓	5.95	232
2	DeleteNoLeaveTombstonesEtag	✗	-	-	✓	4.69	272
2	DeletePrimaryKey	✓	2.72	168	✓	2.37	171
2	EnsurePartitionSwitchedFromPopulated	✓	25.17	85	✓	1.57	136
2	TombstoneOutputETag	✓	8.25	305	✓	3.40	242
2	QueryStreamedFilterShadowing	◇	0.55	79	◇	0.41	79
*2	MigrateSkipPreferOld	✗	-	-	◇	1.13	115
*2	MigrateSkipUseNewWithTombstones	✗	-	-	◇	1.16	120
*2	InsertBehindMigrator	◇	0.32	47	◇	0.31	47

the priority-based scheduler to trigger the remaining four bugs.

5.4.3 Main Lessons Learned

We summarise the main take-aways from our experience applying the P# asynchronous programming language on real-world distributed systems.

Modelling is challenging Modelling the environment of a complex distributed system can be as hard as building the actual system itself. One of the main challenges is how to choose which components of a system to model, and which implementation details to safely omit from a model. Although modelling each and every component might allow for scalable testing (since a lot of low-level implementation details are omitted), it is often not accurate enough and can lead to false negatives (since bugs in the actual implementation might not be reflected in the model). Indeed, we spent considerable effort (approximately two person weeks of work, as discussed in §5.4.1) implementing and refining our model of Azure Storage vNext before being able to detect the liveness bug described in §5.3.6.

Domain expertise In our experience, modelling a distributed system in close collaboration with domain experts (e.g. the developers of the system) can make a huge difference. Indeed, in the case of Azure Storage vNext, its developers had a hunch about the bug, which guided the modelling process and allowed us to discover the bug quickly (see §5.3.6). We believe that without the direct help from the vNext developers, the process of modelling and testing vNext would require considerably larger amount of effort and time. This can be seen in the Azure Service Fabric case study: the team responsible for implementing the model of Service Fabric had expertise in P#, but no expertise in Service Fabric; this ended up considerably slowing down the whole modelling process (see §5.4.1).

Keeping a system and its model in sync An important limitation of our approach is that it might be challenging to keep a complex system and its P# model in sync as the system evolves. We believe that automatically maintaining and evolving a model is not easily achievable with current technology (e.g. state-of-the-art automatic environment modelling techniques, such as *angelic verification* [DLLL15] are still quite restricted in what kind of properties they support, as discussed in §6.2). Instead, we currently suggest that engineering teams using P# in their workflow should dedicate development resources in manually maintaining their models.

In §6.2, we discuss further limitations of the P# approach, as well as open problems in this area, and give interesting directions for future research and development.

5.5 Related Work

Most related to our work are model checking [CE82, QS82, Hol97] (see §2.2.3) and controlled concurrency testing[10] [God97, MQB+08, BKMN10, EQR11, TDB16] (see §2.3.3), two well-established techniques that have been successfully used to find Heisenbugs in the actual implementation of asynchronous distributed systems [KAJV07, YCW+09, YKKK09, GY11, GWZ+11, vBG11, GDJ+11, LHJ+14].

State-of-the-art model checkers, such as MoDist [YCW+09] and dBug [vBG11], typically focus on testing entire, often *unmodified*, distributed systems, an approach that easily leads to state-space explosion. DeMeter [GWZ+11], built on top of MoDist, aims to reduce the state-space when exploring unmodified distributed systems. DeMeter explores individual components of a large-scale system in isolation, and then dynamically extracts interface behaviour between components to perform a global exploration. In this chapter, we try to offer a more pragmatic approach to handling state-space explosion. We first *partially* model a distributed system using P#, and then we (systematically) test the actual implementation of each system component against its P# test harness. Our approach aims to enhance unit and integration testing, techniques widely used in production, where only individual or a small number of components are tested at each time.

SAMC [LHJ+14] offers a way of incorporating application-specific information during systematic testing to reduce the set of interleavings that the tool has to explore, which is a form of partial-order reduction [God96, FG05] (see §2.3.4). Such white-box testing techniques are complementary to our approach: P# could use them to reduce the exploration state-space. Likewise, other tools can use programming language technology like P# to write models and reduce the complexity of the system-under-test.

Fate and Destini is a framework for systematically injecting failures in distributed systems [GDJ+11]. Chaos Monkey [BT12][11] is an open-source tool by Netflix, which randomly kills virtual machines and services to emulate real-world failures. In contrast, P# can be used not only for injecting nondeterministic (but controlled during testing) failures in a distributed system (running in a single-box and single process environment), but also for testing generic safety and liveness properties.

MaceMc [KAJV07] is a model checker for asynchronous distributed systems written

[10]Controlled concurrency testing is also known as *stateless* model checking.
[11]`https://github.com/Netflix/SimianArmy`

in the Mace [KAB$^+$07] language. MaceMc tries is to find liveness bugs using several heuristics: it first performs a bounded depth-first search exploration; and then randomly explores the program trying to identify executions that do not satisfy the given liveness property. Because MaceMc can only test systems written in Mace, it cannot be easily used in an industrial setting. In contrast, P# can be easily applied to legacy code written in C#, a mainstream language. As an example, we are recently working on applying P# to Project Orleans [BBG$^+$14],[12] an open-source cloud computing platform that has been used in Microsoft for implementing cloud services in the Halo 4 Xbox 360 console game. The P# models of Orleans are open-source and available on GitHub.[13]

Formal methods have been used in industry to verify the correctness of distributed protocols. A notable example is the use of TLA+ [Lam02] in Amazon Web Services [NRZ$^+$15]. TLA+ is an expressive formal specification language that can be used to design and verify concurrent programs via model checking. A limitation of TLA+, as well as other similar specification languages, is that they are applied only on a model of the system and not the actual system. Even if the model is verified, the gap between the implementation and the model is still significant, so implementation bugs are still a realistic concern.

More recent formal approaches include the Verdi [WWP$^+$15] and IronFleet [HHK$^+$15] frameworks. In Verdi, developers write and verify distributed systems in Coq [BBC$^+$97]. After the system has been successfully verified, Verdi translates the Coq code to OCaml, which can be then compiled for execution. Verdi does not currently support detecting liveness property violations, an important class of bugs in distributed systems. In IronFleet, developers can build a distributed system using the Dafny [Lei10] language and program verifier. Dafny verifies system correctness using the Z3 [DMB08] SMT solver, and finally compiles the verified system to a .NET executable. In contrast, P# performs bounded controlled testing on a system *already* written in .NET, which in our experience lowers the bar for adoption by engineering teams.

5.6 Summary

We presented a new methodology for testing distributed systems. Our approach involves using P#, an extension of the C# language that provides advanced modelling, specification and controlled testing capabilities. We reported the experience of applying P# to three distributed systems developed inside Microsoft. Using P#, the developers of these systems found, reproduced, confirmed and fixed numerous bugs.

[12]`https://github.com/dotnet/orleans`
[13]`https://github.com/p-org/PSharpModels`

6 Conclusions

The goal of this thesis was to design and implement scalable analysis and testing techniques for asynchronous software systems. To conclude, we summarise our contributions towards achieving this goal, and discuss interesting directions for future work.

6.1 Contributions

This thesis has made the following original contributions:

- In Chapter 3, we discussed a novel *symbolic lockset analysis* that can automatically verify the absence of data races in device drivers. We have implemented this analysis in Whoop, a scalable analyser that can be applied to many different types of Linux drivers [CRKH05]. Whoop scales well because it summarises procedure calls, and reasons about locksets instead of context-switches, but might also report many false alarms. To increase the precision of Whoop, we combined it with Corral [LQL12], an industrial-strength bug-finder. Exploiting race-freedom guarantees provided by our analysis, we gave a sound partial-order reduction [God96] that can significantly accelerate Corral, without missing any bugs that Corral would otherwise find. Combining Whoop and Corral, we analysed 16 drivers from the Linux 4.0 kernel, achieving 1.5–20× speedups over standalone Corral. Moreover, we used Whoop to analyse 1016 concurrent C programs from the 2016 Software Verification Competition (SVCOMP'16), showing that our approach is not specific to device drivers.

- In Chapter 4, we presented P#, a new language for safer asynchronous event-driven programming. P# is *co-designed* with static data race analysis and controlled testing techniques. We first formulated a static ownership-based analysis for proving absence of data races in programs written in P#. Next, we developed an efficient controlled testing infrastructure by leveraging the P# language and static analysis. Finally, we experimentally evaluated P# against 14 distributed algorithms, including widely-used protocols such as ChainReplication [vRS04] and Raft [OO14], showing that the

P# analysis and testing techniques improve on previous work in terms of scalability, precision and bug-finding ability.

- In Chapter 5, we illustrated an approach of using P# to model, specify and test real-world distributed systems. We discussed a particular case study of applying P# to Microsoft Azure Storage vNext [DMT+16], a complex replicating storage system, and showed how P# was used to detect a subtle, but serious, liveness bug that evaded troubleshooting for months using conventional unit-testing, integration-testing and stress-testing techniques. Besides the vNext system, P# has been also successfully used by Microsoft engineers, researchers and interns to test two other complex systems built on top of the Microsoft Azure cloud computing platform [DMT+16]. We evaluated the cost and benefits of applying our approach on these industrial case studies, and showed that P# is able to efficiently detect bugs and produce easy to understand by the developers traces.

6.2 Open Problems and Future Work

We now discuss interesting directions for future work to built upon the achievements of this thesis and address the limitations of our techniques.

Automatic environment modelling An important prerequisite for analysing asynchronous programs is to "close" their environment, i.e. to provide an implementation for all external components and library methods. This can be achieved by writing an *environment model* that is abstract enough to enable scalable analysis, but also detailed enough to cover all interesting behaviours. Manually writing and maintaining such a model is tedious and error-prone. For instance, to be able to analyse the 16 drivers in Chapter 3, we first had to spend two months understanding and modelling the Linux kernel APIs used by these drivers. Significant development time was also spent for closing the environment of the distributed systems discussed in Chapter 5.

Prior work includes techniques such as Daikon [ECGN01], which infers likely program invariants using dynamic analysis, and Houdini [FL01], which computes the largest set of inductive invariants from a set of (automatically) guessed candidates. Both techniques could be used to automatically construct an environment model from the inferred program specifications. Ammons et al. [ABL02] described a machine learning technique for automatically extracting specifications, while Ball et al. [BLX04] gave an approach for automatically refining environment models via training. However, both approaches focus only on sequential programs. CBugs [JLL12] employs differential analysis to analyse concurrent programs without requiring a precise environment model. However, CBugs can be only applied to bounded programs without loops and recursion. Angelic verification [DLLL15] is a technique that attempts to find environment assumptions to suppress false positives from imprecise (or non-existing) environment models. However, angelic verification can currently only check for null-pointer dereferences and does not work on concurrent programs.

How to automatically create environment models for complex asynchronous software systems is an important open problem. We believe that developing automated modelling techniques can significantly enhance programmer productivity, and help make concurrency analysis and testing tools more usable.

Debugging at the level of distributed events Debuggers, such as the Visual Studio debugger [Mic16d], gdb [FSF16] and lldb [LLD], are used by developers to troubleshoot errors and observe the runtime behaviour of programs. Alas, these tools do not work well in the case of distributed systems. Such systems consist of multiple components that communicate asynchronously with each other by exchanging messages over a network. These messages, as well as other nondeterministic distributed events (e.g. timers and failures), are responsible for driving the execution of a distributed system. Mainstream debuggers, however, cannot reliably attach to multiple distributed processes, and most crucially, are unable to control sources of nondeterminism and explore interleavings between distributed events.

P# (see Chapters 4 and 5) provides a controlled testing infrastructure for executing a distributed system in single-process/single-box mode. Our testing infrastructure is able to explore interleavings between distributed events, and control other declared sources of nondeterminism. The P# testing process, however, differs from the typical debugging process, as it does not allow the programmer to manually set breakpoints, step through the distributed execution, examine the heap, and observe the call stack. Prior work on distributed system debugging includes tools that can deterministically replay nondeterministic failures (e.g. [GASS06, GAM^{+}07, LLPZ07]). However, most of these tools can only be used offline, and do not control sources of nondeterminism. We envision that an ideal debugger for distributed systems would combine the useful features of mainstream debuggers with the testing capabilities of P#.

Exploiting application semantics during testing All of our controlled testing P# schedulers, i.e. random, depth-first search, and randomised priority-based [BKMN10] (see §4.5), treat every system-under-test as a *black box*. In our experience, this can be inefficient when testing complex distributed systems, because the P# schedulers might end up exploring many event-handler interleavings and sources of nondeterminism that are not relevant to the safety or liveness property being checked, thus adversely affecting the scalability of P#.

A recent study illustrated that exploiting *application semantics* can significantly improve scalability when testing real-world distributed systems [LHJ$^+$14]. Interesting future work could involve developing P# schedulers that exploit application semantics and language support for even more efficient controlled testing. A potential way to enhance the P# runtime with *white-box* testing capabilities is by overloading the P# `send` method (see §4.2.1). For example, an overloaded `send` method could enable the programmer to (i) specify that a sent P# event is related to some assertion being checked, thus notifying the scheduler to focus on exploring interleavings related to handling this event, or to (ii) specify that events of a particular type belong to the same event group, notifying P# to handle them in an atomic fashion, thus achieving a semantics-aware partial-order.

Cycle detection for efficient liveness checking Program specifications written in P# can encode either safety or liveness [Lam77] properties (see Chapter 5). Checking safety properties is straightforward; the P# testing engine explores interleavings and other sources of nondeterminism until it detects an assertion violation (see §5.2.3). Checking for liveness bugs, however, is much harder; this is because P# must detect a *fair infinite* execution that exhibits lack of progress.

Prior techniques find such executions by searching for reachable cycles during state-space exploration [GH93, GMR09, DQRS15]. However, this typically requires whole-program state-caching, which is infeasible in real-world asynchronous systems. Since it is impossible to generate an infinite execution in a finite amount of time, and to avoid state-caching, our current implementation of liveness checking in P# considers an execution longer than a large user-supplied bound (e.g. 100,000 scheduling points) as a "fair infinite" execution [KAJV07, MQ08] (see §5.2.4).

Our liveness checking approach has two main limitations: (i) bounding the execution can lead to false positives, because the liveness property might get satisfied only after the bound has been reached; and (ii) the reported trace can be long, requiring further minimisation. In future work, we aim to address these limitations by developing more precise liveness checking techniques in P#.

6.3 Summary

We have summarised the contributions of the thesis, discussed open problems, and outlined potential topics for future research and development. The scalable techniques that we have developed in this work can be used to efficiently analyse and test real-world asynchronous programs, such as device drivers and distributed systems. We hope that future researchers will built upon our results to create practical analysers and testers.

Bibliography

[ABH$^+$97] Rajeev Alur, Robert K Brayton, Thomas A Henzinger, Shaz Qadeer, and Sriram K Rajamani. Partial-order reduction in symbolic state space exploration. In *Proceedings of the 9th Workshop on Computer Aided Verification*, pages 340–351. Springer, 1997. (Cited on pages 13 and 20.)

[ABL02] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16. ACM, 2002. (Cited on page 133.)

[ADK$^+$05] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P Kurshan, and Kenneth L McMillan. An analysis of SAT-based model checking techniques in an industrial environment. In *Proceedings of the 13th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 254–268. Springer, 2005. (Cited on page 23.)

[Age96] European Space Agency. Ariane 5, flight 501 failure, report by the Inquiry Board. `http://www.esa.int/For_Media/Press_Releases/Ariane_501_-_Presentation_of_Inquiry_Board_report`, 1996. [Accessed 19-May-2016]. (Cited on page 21.)

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986. (Cited on pages 19 and 70.)

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14. ACM, 1990. (Cited on page 47.)

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Pearson Education, 2nd edition, 2006. (Cited on page 79.)

[Ama12] Amazon. Summary of the AWS service event in the US East Region. `http://aws.amazon.com/message/67457/`, 2012. [Accessed 9-May-2016]. (Cited on pages 13 and 107.)

[And90] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990. (Cited on page 41.)

[App16] Apple Inc. Grand Central Dispatch (GCD) reference. `https://developer.apple.com/library/mac/documentation/Performance/Reference/GCD_libdispatch_Ref/index.html`, 2016. [Accessed 9-May-2016]. (Cited on page 18.)

[APV07] Saswat Anand, Corina S Păsăreanu, and Willem Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 134–138. Springer, 2007. (Cited on page 25.)

[AQR$^+$04] Tony Andrews, Shaz Qadeer, Sriram K Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Proceedings of the 16th International Conference on Computer Aided Verification*, pages 484–487. Springer, 2004. (Cited on pages 19 and 73.)

[BBC$^+$97] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The Coq proof assistant reference manual: Version 6.1. `https://hal.inria.fr/inria-00069968/`, 1997. [Accessed 9-May-2016]. (Cited on page 131.)

[BBC$^+$06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 2006 EuroSys Conference*, pages 73–85. ACM, 2006. (Cited on pages 13, 24, 34, and 112.)

[BBC$^+$10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. (Cited on pages 13 and 22.)

[BBC$^+$14] Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. Engineering a static verification tool for GPU kernels. In *Proceedings of the 26th International Conference on Computer Aided Verification*, pages 226–242. Springer, 2014. (Cited on pages 53 and 67.)

[BBG$^+$14] P Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed virtual actors for programmability and scalability. Technical Report MSR-TR-2014-41, Microsoft Research, 2014. (Cited on pages 19 and 131.)

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207. Springer, 1999. (Cited on page 23.)

[BCD$^+$06] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Symposium on Formal methods for Components and Objects*, pages 364–387, 2006. (Cited on pages 22, 32, and 44.)

[BCD$^+$11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011. (Cited on page 22.)

[BCD$^+$15] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Transactions on Programming Languages and Systems*, 37(3):10, 2015. (Cited on pages 32, 53, and 67.)

[BCM$^+$92] Jerry R Burch, Edmund M Clarke, Kenneth L McMillan, David L Dill, and Lain-Jinn Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and computation*, 98(2):142–170, 1992. (Cited on page 23.)

[BCS13] Sam Blackshear, Bor-Yuh Evan Chang, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *Proceedings of the 34th ACM SIG-*

PLAN Conference on Programming Language Design and Implementation, pages 275–286. ACM, 2013. (Cited on page 90.)

[Bel05] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 41–46. USENIX Association, 2005. (Cited on page 25.)

[BFQ07] Ahmed Bouajjani, Séverine Fratani, and Shaz Qadeer. Context-bounded analysis of multithreaded programs with dynamic linked structures. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 207–220. Springer, 2007. (Cited on page 32.)

[BKMN10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–178. ACM, 2010. (Cited on pages 13, 30, 31, 95, 102, 103, 127, 130, and 134.)

[BL05] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 82–87, 2005. (Cited on page 44.)

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011. (Cited on page 24.)

[BLX04] Thomas Ball, Vladimir Levin, and Fei Xie. Automatic creation of environment models via training. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2004. (Cited on page 133.)

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the 22nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213. ACM, 2001. (Cited on pages 22 and 24.)

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 100(8):677–691, 1986. (Cited on page 23.)

[BSST09] Clark W Barrett, Roberto Sebastiani, Sanjit A Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, 2009. (Cited on page 22.)

[BT12] Cory Bennett and Ariel Tseitlin. Netflix: Chaos Monkey released into the wild. `http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html`, 2012. [Accessed 29-May-2016]. (Cited on page 130.)

[But97] David R Butenhof. *Programming with POSIX threads.* Addison-Wesley, 1997. (Cited on page 60.)

[BUZC11] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. Parallel symbolic execution for automated real-world software testing. In *Proceedings of the 2011 EuroSys Conference*, pages 183–198. ACM, 2011. (Cited on page 25.)

[BW96] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996. (Cited on page 23.)

[Cav13] Mark Cavage. There's just no getting around it: you're building a distributed system. *ACM Queue*, 11(4):30–41, 2013. (Cited on page 107.)

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977. (Cited on page 22.)

[CC10] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *Proceedings of the 2010 EuroSys Conference*, pages 167–180. ACM, 2010. (Cited on page 21.)

[CDBM15] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, pages 1–12. ACM, 2015. (Cited on pages 19 and 69.)

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs.

In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224. USENIX Association, 2008. (Cited on page 25.)

[CE82] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of the Logic of Programs Workshop*, pages 52–71. Springer, 1982. (Cited on pages 22 and 130.)

[CGJ$^+$00] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000. (Cited on page 24.)

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing*, pages 398–407. ACM, 2007. (Cited on pages 96, 105, and 107.)

[CKC11] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278. ACM, 2011. (Cited on page 26.)

[CKL04] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004. (Cited on page 22.)

[CKSY04] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004. (Cited on pages 24 and 34.)

[CLA] Clang: A C language family frontend for LLVM. `http://clang.llvm.org`. [Accessed 11-May-2016]. (Cited on page 50.)

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215–222, 1976. (Cited on page 25.)

[CLL$^+$02] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the 23rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 258–269. ACM, 2002. (Cited on page 67.)

[CMM13] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded Partial-order Reduction. In *Proceedings of the 28th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 833–848. ACM, 2013. (Cited on page 31.)

[CMP04] Ching-Tsun Chou, Phanindra K Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*, pages 382–398. Springer, 2004. (Cited on page 67.)

[Cor04] Jonathan Corbet. Finding kernel problems automatically. `https://lwn.net/Articles/87538/`, 2004. [Accessed 11-May-2016]. (Cited on page 34.)

[Cor06] Jonathan Corbet. The kernel lock validator. `https://lwn.net/Articles/185666/`, 2006. [Accessed 11-May-2016]. (Cited on page 34.)

[CPR06] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 415–426. ACM, 2006. (Cited on pages 34 and 49.)

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, 3rd edition, 2005. (Cited on pages 34, 35, 36, 37, 38, 41, 43, 52, 56, and 132.)

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996. (Cited on page 96.)

[CWO$^+$11] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 143–157. ACM, 2011. (Cited on pages 114 and 115.)

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88. ACM, 2001. (Cited on pages 34 and 36.)

[Dah09] Ryan Dahl. Node.js, evented I/O for V8 Javascript. `https://nodejs.org`, 2009. [Accessed 9-May-2016]. (Cited on pages 13 and 18.)

[DD14] Pantazis Deligiannis and Alastair F. Donaldson. Automatic verification of data race freedom in device drivers. In *Proceedings of the 2014 Imperial College Computing Student Workshop*, pages 36–39. Schloss Dagstuhl, 2014. (Cited on pages 15 and 35.)

[DDK+15] Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 154–164. ACM, 2015. (Cited on pages 15, 16, 17, 19, 69, 70, and 127.)

[DDR15] Pantazis Deligiannis, Alastair F. Donaldson, and Zvonimir Rakamarić. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, pages 166–177. IEEE, 2015. © 2015 IEEE. (Cited on pages 15, 16, and 35.)

[DGJ+13] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 321–332. ACM, 2013. (Cited on pages 13, 19, 73, 80, and 94.)

[DGM14] Ankush Desai, Pranav Garg, and P. Madhusudan. Natural proofs for asynchronous programs using almost-synchronous reductions. In *Proceedings of the 29th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 709–725. ACM, 2014. (Cited on page 80.)

[Dij65a] Edsger W. Dijkstra. Co-operating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965. (Cited on page 41.)

[Dij65b] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965. (Cited on page 41.)

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975. (Cited on page 22.)

[DJP$^+$15] Ankush Desai, Ethan Jackson, Amar Phanishayee, Shaz Qadeer, and Sanjit Seshia. Building reliable distributed systems with P. Technical Report UCB/EECS-2015-198, EECS Department, University of California, Berkeley, 2015. (Cited on pages 97, 108, 112, 113, and 121.)

[DKR82] Danny Dolev, Maria Klawe, and Michael Rodeh. An O(n log n) unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, 1982. (Cited on page 96.)

[DL05] Robert DeLine and K. Rustan M. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005. (Cited on pages 24 and 50.)

[DLLL15] Ankush Das, Shuvendu K Lahiri, Akash Lal, and Yi Li. Angelic verification: Precise verification modulo unknowns. In *Proceedings of the 27th International Conference on Computer Aided Verification*, pages 324–342. Springer, 2015. (Cited on pages 129 and 134.)

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. (Cited on pages 22, 53, and 131.)

[DMT$^+$16] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, pages 249–262.

USENIX Association, 2016. (Cited on pages 15, 16, 17, 69, 73, 106, 107, 114, 115, 116, 125, 127, and 133.)

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960. (Cited on page 23.)

[DQRS15] Ankush Desai, Shaz Qadeer, Sriram Rajamani, and Sanjit Seshia. Iterative cycle detection via delaying explorers. Technical Report MSR-TR-2015-28, Microsoft Research, 2015. (Cited on page 135.)

[DQS15] Ankush Desai, Shaz Qadeer, and Sanjit A Seshia. Systematic testing of asynchronous reactive systems. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 73–83. ACM, 2015. (Cited on page 97.)

[DRP99] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: introduction, management, and performance.* Addison-Wesley, 1999. (Cited on page 21.)

[DS91] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96. ACM, 1991. (Cited on page 29.)

[DSR15] Madan Das, Gabriel Southern, and Jose Renau. Section based program analysis to reduce overhead of detecting unsynchronized thread communication. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 283–284, 2015. (Cited on page 67.)

[EA03] Dawson R. Engler and Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 237–252, 2003. (Cited on page 34.)

[ECCH00] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation.* USENIX Association, 2000. (Cited on page 34.)

[ECGN01] Michael D Ernst, Jake Cockrell, William G Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution.

IEEE Transactions on Software Engineering, 27(2):99–123, 2001. (Cited on page 133.)

[EFG$^+$03] Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for testing multi-threaded Java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003. (Cited on page 13.)

[EG04] Kevin Elphinstone and Stefan Götz. Initial evaluation of a user-level device driver framework. In *Proceedings of the 9th Asia-Pacific Computer Systems Architecture Conference*, pages 256–269. Springer, 2004. (Cited on page 39.)

[EMBO10] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16. USENIX Association, 2010. (Cited on page 58.)

[EQR11] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 411–422. ACM, 2011. (Cited on pages 13, 30, 103, and 130.)

[EQT07] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A race and transaction-aware Java runtime. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–255, 2007. (Cited on page 29.)

[FF09] Cormac Flanagan and Stephen N Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 121–133. ACM, 2009. (Cited on page 28.)

[FFY08] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 293–303. ACM, 2008. (Cited on page 28.)

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages, pages 110–121. ACM, 2005. (Cited on pages 31 and 130.)

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods for Increasing Software Productivity*, pages 500–517, 2001. (Cited on pages 53 and 133.)

[FLL+02] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245. ACM, 2002. (Cited on page 20.)

[Flo67] Robert W Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32):1, 1967. (Cited on page 21.)

[FRR+12] Jose Faleiro, Sriram Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. CScale: A programming model for scalable and reliable distributed applications. In *Proceedings of the 17th Monterey Workshop on Development, Operation and Management of Large-Scale Complex IT Systems*, pages 148–156. Springer, 2012. (Cited on page 125.)

[FSF16] Inc. Free Software Foundation. GDB: The GNU project debugger. `https://www.gnu.org/software/gdb/`, 2016. [Accessed 11-May-2016]. (Cited on page 134.)

[GAM+07] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, pages 285–298. USENIX Association, 2007. (Cited on page 134.)

[GASS06] Dennis Michael Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 289–300. USENIX Association, 2006. (Cited on page 134.)

[GB13] Olivier Gruber and Fabienne Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In *Proceedings of the 27th European Conference* *on Object-Oriented Programming*, pages 281–301. Springer, 2013. (Cited on page 69.)

[GDJ+11] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 238–252. USENIX Association, 2011. (Cited on page 130.)

[Ger04] Steven German. Tutorial on verification of distributed cache memory protocols. In *Proceedings of the 5th International Conference on Formal Methods in Computer-Aided Design*. Springer, 2004. (Cited on page 96.)

[GH93] Patrice Godefroid and Gerard J Holzmann. On the verification of temporal properties. In *Proceedings of the 13th Symposium on Protocol Specification, Testing and Verification*, pages 109–124, 1993. (Cited on page 135.)

[GHL+14] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the 5th ACM Symposium on Cloud Computing*. ACM, 2014. (Cited on page 107.)

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005. (Cited on page 25.)

[GLM+08] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the 15th Network and Distributed System Security Symposium*, pages 151–166, 2008. (Cited on page 25.)

[GLSS05] Xiaofeng Gao, Michael Laurenzano, Beth Simon, and Allan Snavely. Reducing overheads for acquiring dynamic memory traces. In *Proceedings of the 2005 IEEE International Symposium on Workload Characterization*, pages 46–55, 2005. (Cited on page 100.)

[GMR09] Pierre Ganty, Rupak Majumdar, and Andrey Rybalchenko. Verifying liveness for asynchronous programs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 102–113. ACM, 2009. (Cited on page 135.)

[GNK$^+$15] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. Rely/guarantee reasoning for asynchronous programs. In *Proceedings of the 26th International Conference on Concurrency Theory*, pages 483–496. Schloss Dagstuhl, 2015. (Cited on page 103.)

[God96] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996. (Cited on pages 13, 20, 31, 59, 94, 130, and 132.)

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186. ACM, 1997. (Cited on pages 28, 30, 31, 69, 72, 94, 103, and 130.)

[Gra78] James N Gray. *Notes on Data Base Operating Systems.* Springer, 1978. (Cited on page 96.)

[Gra86] Jim Gray. Why do computers stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12. IEEE, 1986. (Cited on pages 13, 18, 94, 105, and 107.)

[Gra90] Jim Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, 1990. (Cited on page 21.)

[Gra13] GrammaTech. Finding concurrency errors with GrammaTech static analysis. `http://codesonar.grammatech.com/finding-concurrency-errors-with-grammatech-static-analysis`, 2013. [Accessed 29-May-2016]. (Cited on page 22.)

[Gre15] Albert Greenberg. SDN for the Cloud. Keynote in the 2015 ACM Conference on Special Interest Group on Data Communication, 2015. (Cited on page 114.)

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th Workshop on Computer Aided Verification*, pages 72–83. Springer, 1997. (Cited on page 24.)

[GWZ$^+$11] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 265–278. ACM, 2011. (Cited on page 130.)

[GY11] Rachid Guerraoui and Maysam Yabandeh. Model checking a networked system without the network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, pages 225–238. USENIX Association, 2011. (Cited on page 130.)

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international Joint Conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers, 1973. (Cited on page 70.)

[Hen09] Alyssa Henry. Cloud storage FUD: Failure and uncertainty and durability. Keynote in the 7th USENIX Conference on File and Storage Technologies, 2009. (Cited on pages 105 and 107.)

[HF11] Gerard J Holzmann and Mihai Florian. Model checking with bounded context switching. *Formal Aspects of Computing*, 23(3):365–389, 2011. (Cited on page 31.)

[HHK$^+$15] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles*. ACM, 2015. (Cited on pages 105 and 131.)

[HJMS02] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70. ACM, 2002. (Cited on page 24.)

[HNJ$^+$02] Thomas A Henzinger, George C Necula, Ranjit Jhala, Gregoire Sutre, Rupak Majumdar, and Westley Weimer. Temporal-safety proofs for systems code.

In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 526–538. Springer, 2002. (Cited on page 34.)

[HO10] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, pages 354–378. Springer, 2010. (Cited on page 69.)

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. (Cited on page 21.)

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. (Cited on page 18.)

[Hog15] Jason Hogg. Azure Storage Table design guide: Designing scalable and performant tables. `https://azure.microsoft.com/en-us/documentation/articles/storage-table-design-guide`, 2015. [Accessed 7-June-2016]. (Cited on page 125.)

[Hol91] GJ Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991. (Cited on page 24.)

[Hol97] Gerard J Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. (Cited on pages 24, 31, and 130.)

[Imp16] Improbable. SpatialOS. `https://spatialos.improbable.io`, 2016. [Accessed 7-June-2016]. (Cited on page 19.)

[ISO10] ISO/IEC/IEEE. 24765-2010 - systems and software engineering - vocabulary, 2010. (Cited on page 21.)

[ISO11] ISO/IEC. Programming languages - C. International Standard, 2011. (Cited on pages 27 and 36.)

[JLL12] Saurabh Joshi, Shuvendu K Lahiri, and Akash Lal. Underspecified harnesses and interleaved bugs. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 19–30. ACM, 2012. (Cited on page 133.)

[JM07] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 339–350. ACM, 2007. (Cited on page 103.)

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983. (Cited on page 103.)

[KAB$^+$07] Charles Edwin Killian, James W Anderson, Ryan Braud, Ranjit Jhala, and Amin M Vahdat. Mace: language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 179–188. ACM, 2007. (Cited on page 131.)

[KAJV07] Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, pages 243–256. USENIX Association, 2007. (Cited on pages 108, 113, 130, and 135.)

[KCC10] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. Testing closed-source binary device drivers with DDT. In *Proceedings of the 2010 USENIX Annual Technical Conference*, pages 159–172. USENIX Association, 2010. (Cited on pages 25 and 34.)

[Kin76] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976. (Cited on page 25.)

[KKI99] Mahesh Kalyanakrishnam, Zbigniew Kalbarczyk, and Ravishanka Iyer. Failure data analysis of a LAN of windows NT based computers. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 178–187. IEEE, 1999. (Cited on page 21.)

[KLMO91] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 41–55. ACM, 1991. (Cited on page 43.)

[KPP03] Claude Kaiser and Jean-François Pradat-Peyre. Chameneos, a concurrency game for Java, Ada and others. In *Proceedings of the 2003 ACS/IEEE International Conference on Computer Systems and Applications*. IEEE, 2003. (Cited on page 96.)

[KRE88] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C Programming Language*. Prentice Hall, 2nd edition, 1988. (Cited on pages 19 and 34.)

[KS12] Asim Kadav and Michael M. Swift. Understanding modern device drivers. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 87–98. ACM, 2012. (Cited on page 39.)

[KSA09] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009. (Cited on pages 19 and 69.)

[KSG09] Vineet Kahlon, Sriram Sankaranarayanan, and Aarti Gupta. Semantic reduction of thread interleavings in concurrent programs. In *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 124–138. Springer, 2009. (Cited on page 67.)

[KYSG07] Vineet Kahlon, Yu Yang, Sriram Sankaranarayanan, and Aarti Gupta. Fast and accurate static data-race detection for concurrent programs. In *Proceedings of the 19th International Conference on Computer Aided Verification*, pages 226–239. Springer, 2007. (Cited on page 67.)

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd IEEE / ACM International Symposium on Code Generation and Optimization*, pages 75–86, 2004. (Cited on pages 21 and 50.)

[LAdS$^+$15] Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, and Feng Qin. Debugging high-performance computing applications at massive scales. *Communications of the ACM*, 58(9):72–81, 2015. (Cited on page 107.)

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977. (Cited on pages 109 and 135.)

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978. (Cited on pages 26, 27, and 29.)

[Lam94] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994. (Cited on page 113.)

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. (Cited on pages 69, 96, 111, and 115.)

[Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley, 2002. (Cited on pages 105, 107, and 131.)

[LCD16] Daniel Liew, Cristian Cadar, and Alastair F. Donaldson. Symbooglix: A symbolic execution engine for Boogie programs. In *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation*, pages 45–56. IEEE, 2016. (Cited on page 25.)

[Lei10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010. (Cited on page 131.)

[LG10] Guodong Li and Ganesh Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 187–196. ACM, 2010. (Cited on page 32.)

[LHJ$^+$14] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 399–414. USENIX Association, 2014. (Cited on pages 107, 130, and 135.)

[LL05] V Benjamin Livshits and Monica S Lam. Finding security vulnerabilities in Java applications with static analysis. In *Proceedings of the 14th USENIX Security Symposium*, volume 14, pages 18–18. USENIX, 2005. (Cited on page 21.)

[LLD] The LLDB debugger. `http://lldb.llvm.org`. [Accessed 11-May-2016]. (Cited on page 134.)

[LLLG16] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2016. (Cited on pages 105 and 107.)

[LLPZ07] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS checker: Combating bugs in distributed systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, pages 257–270. USENIX Association, 2007. (Cited on page 134.)

[LLV] The LLVM compiler infrastructure. `http://llvm.org`. [Accessed 11-May-2016]. (Cited on page 50.)

[Lov10] Robert Love. *Linux Kernel Development*. Addison-Wesley, 3rd edition, 2010. (Cited on pages 36 and 37.)

[LQ14] Akash Lal and Shaz Qadeer. Powering the Static Driver Verifier using Corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 202–212. ACM, 2014. (Cited on pages 24, 57, and 61.)

[LQL12] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. A solver for reachability modulo theories. In *Proceedings of the 24th International Conference on Computer Aided Verification*, pages 427–443. Springer, 2012. (Cited on pages 13, 14, 22, 24, 32, 34, 57, and 132.)

[LQR09] Shuvendu K. Lahiri, Shaz Qadeer, and Zvonimir Rakamarić. Static and precise detection of concurrency errors in systems code using SMT solvers. In *Proceedings of the 21st International Conference on Computer Aided Verification*, pages 509–524. Springer, 2009. (Cited on pages 32 and 57.)

[LR08] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *Proceedings of the 20th International Conference on Computer Aided Verification*, pages 37–51. Springer, 2008. (Cited on pages 13, 32, and 57.)

[LSB09] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 227–242. ACM, 2009. (Cited on page 71.)

[LSS+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, 2015. (Cited on page 55.)

[LT93] Nancy G Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993. (Cited on page 21.)

[LTKR08] Akash Lal, Tayssir Touili, Nicholas Kidd, and Thomas Reps. Interprocedural analysis of concurrent programs under a context bound. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 282–298. Springer, 2008. (Cited on page 32.)

[Mad15] Philip Maddox. Testing a distributed system. *ACM Queue*, 13(7):10–15, 2015. (Cited on page 107.)

[McC15] Caitie McCaffrey. The verification of a distributed system. *ACM Queue*, 13(9):150–160, 2015. (Cited on pages 105 and 107.)

[McM99] Kenneth L McMillan. Verification of infinite state systems by compositional model checking. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 219–237. Springer, 1999. (Cited on page 67.)

[Mic13] Microsoft. Network Driver Interface Specification Test (NDISTest). `https://msdn.microsoft.com/en-us/library/ms834435.aspx`, 2013. [Accessed 29-May-2016]. (Cited on page 66.)

[Mic16a] Microsoft. C#. `https://msdn.microsoft.com/en-us/library/kx37x362.aspx`, 2016. [Accessed 9-May-2016]. (Cited on page 71.)

[Mic16b] Microsoft. Asynchronous programming with async and await. `https://msdn.microsoft.com/en-us/library/mt674882.aspx`, 2016. [Accessed 9-May-2016]. (Cited on pages 13 and 18.)

[Mic16c] Microsoft. Azure Service Fabric. `https://azure.microsoft.com/en-us/services/service-fabric/`, 2016. [Accessed 7-June-2016]. (Cited on pages 19 and 125.)

[Mic16d] Microsoft. Debugging in visual studio. `https://msdn.microsoft.com/en-us/library/sc65sadd.aspx`, 2016. [Accessed 11-May-2016]. (Cited on page 134.)

[Mic16e] Microsoft. .NET. `https://www.microsoft.com/net/`, 2016. [Accessed 7-June-2016]. (Cited on page 20.)

[Mil80] Robin Milner. *A calculus of communicating systems*. Springer, 1980. (Cited on page 18.)

[MQ07a] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455. ACM, 2007. (Cited on page 14.)

[MQ07b] Madanlal Musuvathi and Shaz Qadeer. Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft Research, 2007. (Cited on page 31.)

[MQ08] Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 362–371. ACM, 2008. (Cited on pages 113 and 135.)

[MQB$^+$08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280. USENIX Association, 2008. (Cited on pages 13, 18, 28, 30, 31, 67, 69, 72, 94, 100, 103, 105, 107, and 130.)

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011. (Cited on page 66.)

[NBMM12] Santosh Nagarakatte, Sebastian Burckhardt, Milo MK Martin, and Madanlal Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *Proceedings of the 33rd ACM SIGPLAN Conference* *on Programming Language Design and Implementation*, pages 543–554. ACM, 2012. (Cited on page 30.)

[NKA11] Stas Negara, Rajesh K. Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, pages 81–90. ACM, 2011. (Cited on pages 13, 69, 72, 97, 99, and 103.)

[NRZ$^+$15] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How amazon web services uses formal methods. *Communications of the ACM*, 58(4):66–73, 2015. (Cited on pages 105, 107, and 131.)

[OC03] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178. ACM, 2003. (Cited on pages 28 and 29.)

[Off92] US General Accounting Office. Patriot missile defense: Software problem led to system failure at Dhahran, Saudi Arabia. `http://www.gao.gov/products/IMTEC-92-26`, 1992. [Accessed 19-May-2016]. (Cited on page 21.)

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319. USENIX Association, 2014. (Cited on pages 15, 69, 96, 111, and 132.)

[Ora10] Oracle Corporation. Analyzing program performance with Sun WorkShop, Chapter 5: Lock analysis tool. `http://docs.oracle.com/cd/E19059-01/wrkshp50/805-4947/6j4m8jrnd/index.html`, 2010. [Accessed 11-May-2016]. (Cited on page 67.)

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala.* Artima Inc, 2008. (Cited on page 19.)

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *Proceedings of the 5th Workshop on Computer Aided Verification*, pages 409–423. Springer, 1993. (Cited on page 31.)

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster, and Michael Hicks. LOCKSMITH: Context-sensitive correlation analysis for race detection. In *Proceedings of* *the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 320–331. ACM, 2006. (Cited on pages 13, 29, 34, and 67.)

[PLHM08] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 2008 EuroSys Conference*, pages 247–260. ACM, 2008. (Cited on page 34.)

[Plo04] Gordon D Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Programming*, 60(61):17–139, 2004. (Cited on page 81.)

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977. (Cited on pages 22 and 113.)

[Pro02] Niels Provos. libevent - an event notification library. `http://libevent.org`, 2002. [Accessed 22-May-2016]. (Cited on pages 13, 18, and 103.)

[Pro07] Linux USB Project. USB testing on Linux. `http://www.linux-usb.org/usbtest/`, 2007. [Accessed 29-May-2016]. (Cited on page 66.)

[PS03] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–190. ACM, 2003. (Cited on pages 28 and 29.)

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2005. (Cited on page 32.)

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the International Symposium on Programming, 5th Colloquium*, pages 337–351. Springer, 1982. (Cited on pages 22 and 130.)

[QW04] Shaz Qadeer and Dinghao Wu. KISS: Keep it simple and sequential. In *Proceedings of the 25th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–24. ACM, 2004. (Cited on pages 13, 14, 24, and 32.)

[RCKH09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 2009 EuroSys Conference*, pages 275–288. ACM, 2009. (Cited on pages 34 and 36.)

[RE14] Zvonimir Rakamarić and Michael Emmi. SMACK: Decoupling source language details from verifier implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification*, pages 106–113. Springer, 2014. (Cited on page 50.)

[Rey02] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74. IEEE, 2002. (Cited on page 90.)

[RH09] Zvonimir Rakamarić and Alan J Hu. A scalable memory model for low-level code. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 290–304, 2009. (Cited on page 52.)

[Ric53] Henry G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953. (Cited on page 20.)

[RKS12] Matthew J Renzelmann, Asim Kadav, and Michael M Swift. SymDrive: Testing drivers without devices. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2012. (Cited on pages 25 and 34.)

[SA06] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *Proceedings of the 9th international conference on Fundamental Approaches to Software Engineering*, pages 339–356. Springer, 2006. (Cited on page 25.)

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 207–222. ACM, 2003. (Cited on pages 34 and 36.)

[SBN$^+$97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997. (Cited on page 28.)

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005. (Cited on page 43.)

[SLAW08] Raimondas Sasnauskas, Jó Ágila Bitsch Link, Muhammad Hamad Alizai, and Klaus Wehrle. Kleenet: automatic bug hunting in sensor network applications. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 425–426. ACM, 2008. (Cited on page 25.)

[SM08] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 104–128. Springer, 2008. (Cited on page 69.)

[SMA] SMACK: A bounded software verifier for C programs. `https://github.com/smackers/smack`. [Accessed 11-May-2016]. (Cited on page 50.)

[SMA05] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 263–272. ACM, 2005. (Cited on page 25.)

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160. ACM, 2001. (Cited on page 96.)

[SPB$^+$07] Colin Scott, Aurojit Panda, Vjekoslav Brajkovic, George Necula, Arvind Krishnamurthy, and Scott Shenker. Minimizing faulty executions of distributed systems. In *Proceedings of the 13th USENIX Conference on Networked Systems Design and Implementation*, pages 291–309. USENIX Association, 2007. (Cited on page 100.)

[SPG91] Abraham Silberschatz, James L. Peterson, and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, 3rd edition, 1991. (Cited on page 36.)

[Ste93] Nicholas Sterling. WARLOCK — a static data race analysis tool. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 97–106, 1993. (Cited on page 67.)

[SV06] Koushik Sen and Mahesh Viswanathan. Model checking multithreaded programs with asynchronous atomic methods. In *Proceedings of the 18th International Conference on Computer Aided Verification*, pages 300–314. Springer, 2006. (Cited on page 103.)

[Tas02] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, Planning Report 02-3*, 2002. (Cited on pages 21 and 107.)

[TDB16] Paul Thomson, Alastair F Donaldson, and Adam Betts. Concurrency testing using controlled schedulers: An empirical study. *ACM Transactions on Parallel Computing*, 2(4), 2016. (Cited on pages 14, 30, 31, 69, 72, 94, 102, 103, 127, and 130.)

[TDH08] Nikolai Tillmann and Jonathan De Halleux. Pex–white box test generation for .NET. In *Proceedings of the 2nd International Conference on Tests and Proofs*, pages 134–153. Springer, 2008. (Cited on page 25.)

[Tre14] Ben Treynor. GoogleBlog – Today's outage for several Google services. `http://googleblog.blogspot.com/2014/01/todays-outage-for-several-google.html`, 2014. [Accessed 9-May-2016]. (Cited on pages 13 and 107.)

[TT08] Murali Talupur and Mark R Tuttle. Going with the flow: Parameterized verification using message flows. In *Proceedings of the 8th International Conference on Formal Methods in Computer-Aided Design*. IEEE, 2008. (Cited on page 67.)

[TW06] Andrew S Tanenbaum and Albert S Woodhull. *Operating Systems: Design and Implementation*. Prentice Hall, 3rd edition, 2006. (Cited on page 36.)

[Typ16] Typesafe Inc. Akka actors library. `http://akka.io`, 2016. [Accessed 9-May-2016]. (Cited on page 19.)

[UCY+06] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *Proceed-*

ings of the 2006 Workshop on Binary Instrumentation and Applications. Citeseer, 2006. (Cited on page 100.)

[Val98] Antti Valmari. The state explosion problem. In *Lectures on Petri nets I: Basic models*, pages 429–528. Springer, 1998. (Cited on pages 13 and 20.)

[VAR$^+$16] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 391–402. ACM, 2016. (Cited on page 67.)

[vBG11] Jiří Šimša, Randy Bryant, and Garth Gibson. dBug: Systematic testing of unmodified distributed and multi-threaded systems. In *Proceedings of the 18th International SPIN Workshop*, pages 188–193. Springer, 2011. (Cited on pages 105 and 130.)

[VHB$^+$03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003. (Cited on page 31.)

[VJL07] Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. RELAY: Static race detection on millions of lines of code. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on The Foundations of Software Engineering*, pages 205–214, 2007. (Cited on pages 13, 29, 34, and 67.)

[vRS04] Robbert van Renesse and Fred Barry Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 91–104. USENIX Association, 2004. (Cited on pages 15, 97, 111, and 132.)

[VWW96] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, 2nd edition, 1996. (Cited on page 19.)

[WWP$^+$15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 357–368. ACM, 2015. (Cited on pages 105, 108, and 131.)

[Yav12] Raj Yavatkar. Era of SoCs. Presentation at the Intel Workshop on Device Driver Reliability, Modelling and Synthesis, 2012. (Cited on page 35.)

[YCGK08] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Inspect: A runtime model checker for multithreaded C programs. Technical Report UUCS-08-004, University of Utah, 2008. (Cited on page 31.)

[YCW$^+$09] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 213–228. USENIX Association, 2009. (Cited on pages 105 and 130.)

[YKKK09] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, pages 229–244. USENIX Association, 2009. (Cited on page 130.)

[YNPP12] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the 27th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 485–502. ACM, 2012. (Cited on pages 30 and 31.)

[YRC05] Yuan Yu, Tom Rodeheffer, and Wei Chen. RaceTrack: Efficient detection of data race conditions via adaptive tracking. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 221–234. ACM, 2005. (Cited on page 29.)