Efficient Static Analyses for Concurrent Programs

A THESIS
SUBMITTED FOR THE DEGREE OF
Doctor of Philosophy
IN THE
Faculty of Engineering

BY

Suvam Mukherjee

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

December, 2017

Declaration of Originality

I, **Suvam Mukherjee**, with SR No. **04-04-00-17-12-11-1-08865** hereby declare that the material presented in the thesis titled

Efficient Static Analyses for Concurrent Programs

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2011-2017**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Advisor Signature

© Suvam Mukherjee
December, 2017
All rights reserved

DEDICATED TO

My Parents, my brother Sourav, my sister-in-law Deboeeta, my niece Aheer, and Arpita.

Thank you for everything.

Acknowledgements

This thesis would not have seen the light of day without the continuous support, love and encouragement of a large number of people.

I would like to thank my advisor, Prof. Deepak D'Souza, for all his care, help and encouragement. His classes were my first brush with the fascinating field of Program Analysis, and I have been intrigued ever since. I have learnt a lot from him over the course of my Ph.D.– his drive to push yourself that extra bit in order to make a good work great, his penchant for ensuring that my research was mathematically rigorous and elegant, and his intuitive explanations (specially his diagrams, which I hope I learnt well!) of even the most difficult concepts. Above all, I admire most his understanding and sympathetic nature. Thank you, Sir.

I would like to thank Prof. Raghavan, Prof. Kanade and Prof. D'Souza for offering excellent courses in the area of Program Analysis and Verification. I owe all my knowledge and understanding of these areas to their exposition and insights. I thank Prof. Raghavan and Prof. D'Souza for providing the excellent laboratory facilities.

I would like to thank Prof. Narahari, Chairman of the Division of Electrical Sciences, IISc. You have been a source of inspiration for me right from the day of my interviews. I would like to thank you for the innumerable things you have unfailingly helped me out with.

I am thankful to Prof. Helmut Seidl and Prof. Mooly Sagiv, for providing me with wonderful internship opportunities in Germany and Israel, respectively. I would like to take this opportunity to thank Oded Padon, Prof. Sharon Shoham and Prof. Noam Rinetzky for all the intellectually stimulating discussions. Mooly, Oded, Sharon and Noam- thank you so much for all your support.

The CSA department is a wonderful place to do research, and I would like thank the Faculty of CSA for making this possible. Thanks to all the staff in the CSA office- Ms. Kushael, Ms. Meenakshi, Ms. Padmavathi, Ms. Nishita and Ms. Suguna, who helped me sort out various administrative issues. Many thanks to the security personnel for ensuring safety at all times. Special thanks to Sudesh Bhaiya (whose encyclopedic knowledge makes him akin to CSA Wiki)

for all your encouragement and support, right from Day 1.

During the course of my PhD at IISc, I made the most wonderful friends possible. Thanks to Anirban, Narendran, Remish, Indradeep, Chandrahas, Aastha, Saneem, Ninad, Sandeep, Manjunath, Biswaroop, Karthik, Aniket, Dilesh, Abhiruk, Sayantan and Indranil. You really were there through all my ups and downs, and I cannot thank you enough.

Thanks to all members of the Programming Languages Lab, past and present: Raveendra, Girish, Snigdha, Inzemam, Himanshu, Tejas, Vasanta, Aravind, Amogh, Raghavendra, Parixit, Suvitha, and Pallavi, for making the lab a wonderful place to be in.

I would like to thank my Father, for his love, infectious enthusiasm, support and continuous encouragement. Many thanks to my brother Sourav, for everything that you have taught me and all your advice. Thanks to my sister-in-law Deboeeta, for all your love, support and guidance. Though she may not understand this yet, many thanks to my little niece Aheer. You helped me a lot with my research, though you conversed in your incomprehensible, monosyllabic, yet beautiful, sounds. My thanks to revered Sudarshan Maharaj and my grandparents, for their love and blessings.

Finally, I would like to thank two very important people in my life. Ma– I cannot thank you enough for everything you have done for me. This thesis is a result of all your years of love, support, sacrifices, and selfless dedication to my well-being. Thank you, Ma. The other person is Arpita– thank you for your love, sacrifices, friendship and guidance. I simply could *not* have done this without you. You are amazing, and you make my life tick.

Abstract

Concurrent programs are pervasive owing to the increasing adoption of multi-core systems across the entire computing spectrum. However, the large set of possible program behaviors make it difficult to write correct and efficient concurrent programs. This also makes the formal and automated analysis of such programs a hard problem. Thus, concurrent programs provide fertile grounds for a large class of insidious defects. Static analysis techniques infer semantic properties of programs without executing them. They are attractive because they are sound (they can guarantee the absence of bugs), can execute with a fair degree of automation, and do not depend on test cases. However, current static analyses techniques for concurrent programs are either precise and prohibitively slow, or fast but imprecise. In this thesis, we partially address this problem by designing efficient static analyses for concurrent programs.

In the first part of the thesis, we provide a framework for designing and proving the correctness of data flow analysis for race free multi-threaded programs. The resulting analyses are in the same spirit as the "sync-CFG" analysis, originally proposed in De et al [22]. Using novel thread-local semantics as starting points, we devise abstract analyses which treat a concurrent program as if it were sequential. We instantiate these abstractions to devise efficient relational analyses for race free programs, which we have implemented in a prototype tool called RATCOP. On the benchmarks, RATCOP was fairly precise and fast. In a comparative study with a recent concurrent static analyzer, RATCOP was up to 5 orders of magnitude faster.

In the second part of the thesis, we propose a technique for detecting all high-level data races in a system library, like the kernel API of a real-time operating system (RTOS) that relies on flag-based scheduling and synchronization. Such races are good indicators of atomicity violations. Using our technique, a user is able to soundly disregard 99.8% of an estimated $41,000$ potential high-level races. Our tool detected 38 high-level data races in FreeRTOS (a popular OS in the embedded systems domain), out of which 16 were harmful.

Abstract

Publications based on this Thesis

- Suvam Mukherjee, Oded Padon, Sharon Shoham, Deepak D'Souza, and Noam Rinetzky. "RATCOP: Relational Analysis Tool for Concurrent Programs." In Haifa Verification Conference (HVC 2017), pp. 229-233. Springer, Cham, 2017.
- Suvam Mukherjee, Oded Padon, Sharon Shoham, Deepak D'Souza, and Noam Rinetzky. "Thread-local semantics and its efficient sequential abstractions for race-free programs." In Static Analysis Symposium (SAS 2017), pp. 253-276. Springer, Cham, 2017.
- Suvam Mukherjee, Arun Kumar, Deepak D'Souza. "Detecting All High-Level Dataraces in an RTOS Kernel". In: Bouajjani A., Monniaux D. (eds) Verification, Model Checking, and Abstract Interpretation (VMCAI 2017). Lecture Notes in Computer Science, vol 10145. Springer.

Publications based on this Thesis

Contents

Acknowledgements i

Abstract iii

Publications based on this Thesis v

Contents vii

List of Figures xi

List of Tables xv

1 Introduction 1

1.1 A Thread-Local Semantics, and its efficient Sequential Abstractions for Race Free Programs . . . 4

1.1.1 Illustrative Example . . . 7

1.2 Detecting all High-Level Data Races in an RTOS Kernel . . . 11

1.2.1 Overview of our Approach . . . 12

1.3 Structure of this Thesis . . . 14

I Thread-Local Semantics, and its Efficient Abstractions, for Race Free Programs 17

2 Concurrent Programs and their Interleaving Semantics 19

2.1 Mathematical Notations . . . 19

2.1.1 Programming Language and Programs . . . 20

2.2 Interleaving Semantics . . . 23

2.3 Data Races and the Happens-Before Relation 26

3 The Thread-Local Semantics *L-DRF* **29**

3.1 Overview 29

3.2 The thread-local *L-DRF* semantics 30

3.3 Soundness and Completeness of *L-DRF* 35

4 Sequential Abstractions of the *L-DRF* semantics **49**

4.1 Theory of Consistent Abstractions 50

4.2 *A-DRF*: A Canonical *sync*-CFG Analysis based on *L-DRF* 51

4.2.1 Thread-Local Cartesian Abstract Domain 51

4.2.2 Abstract Transitions 52

4.3 The LFP solution of *A-DRF* 54

4.4 Soundness of the Sequential Abstractions 55

4.5 Other abstractions of *L-DRF* 58

5 A Region-Parameterized version of *L-DRF* **61**

5.1 Why do we need another semantics? 61

5.2 Region Race Freedom 62

5.3 The *R-DRF* semantics 63

5.3.1 Thread-Local Abstractions of the *R-DRF* Semantics 63

5.4 Illustrative Example 63

6 Implementation and Experiments **65**

6.1 RATCOP: Relational Analysis Tool for COncurrent Programs 65

6.2 Evaluation 67

6.2.1 Porting Sequential Analyses to Concurrent Analyses. 67

6.2.2 Precision and Efficiency. 67

6.3 Comparing with a current abstract interpretation based tool. 68

7 Related Work and Discussion **71**

II Detecting all High-Level Data Races in an RTOS Kernel 75

8 The architecture of FreeRTOS **77**

8.1 Overview of FreeRTOS 77

9 Atomicity Violations, and High-Level Data Races **83**

9.1 $(\mathcal{S}, \mathcal{C})$ Races . . . 83

9.2 Examples of $(\mathcal{S}, \mathcal{C})$ races in FreeRTOS . . . 85

10 Modelling FreeRTOS in Spin **89**

10.1 Modeling M_n . . . 89

10.2 Strategy to identify *all* high-level races . . . 93

11 Reduction to $\mathcal{M}_{red}$ **97**

12 Experimental Evaluation **105**

12.1 Experimental Setup . . . 105

12.2 Evaluating M_2 . . . 106

12.3 Evaluating $\mathcal{M}_{red}$ . . . 106

12.3.1 List of Detected Races . . . 108

13 Related Work and Discussion **111**

14 Concluding Remarks **115**

Bibliography **117**

List of Figures

1.1 A simple race free multi-threaded program. The variables x, y and z are shared and initialized to 0. 7

1.2 A simple race free program with two threads t_1 and t_2, with all variables being shared and initialized to 0. The columns *L-DRF* and *R-DRF* show the facts computed by polyhedral abstractions of our thread-local semantics and its region-parameterized version, respectively. The Value-Set column shows the facts computed by interval abstractions of the Value-Set analysis of [22]. *R-DRF* is able to prove all 3 assertions, while *L-DRF* fails to prove the assertion at line 11. Value-Set only manages to prove the simple assertion at line 9. 8

1.3 The sync-CFG representation of the program in Figure 1.2 is presented on the left. On the right is an excerpt of the standard product graph representation of the same program. In a sequential setting, a control flow graph is a standard way to represent programs, where each node denotes a control location, and edges represent flow of control. Each node in the product graph represents a *combination* of possible program locations across threads, and an edge between two such nodes represents an interleaving step. Any abstract analysis running on the product graph will clearly not scale for large programs. The sync-CFG, on the other hand, is a more sparse representation, resulting in improved scalability. 9

1.4 A simple program demonstrating an atomicity violation. The variable x is global. The program is free from data races. Since it is initialized to 0, and the threads perform 2 increments to it, the expectation of a developer would be that $x = 2$ when the threads finish. However, x can, in fact, be 1 at line 6. 12

2.1 Semantic Domains. 20

2.2 Simulating the forking of a thread using `acquire` and `release` instructions. For each thread, we assume we have a unique "thread lock" associated with it (in fact, each thread object in Java does have an unique associated monitor). We can assume that the `main` thread initially holds all the thread locks. In the example above, the unique lock associated with `T1` is called `l_T1`. The left side of the code shows the actual code with `fork`, while the one on the right is an equivalent code using commands from *cmd*. The `fork` of `T1` is substituted with the `release(l_T1)`, while the `begin` in `T1` is substituted with `acquire(l_T1)`. In other words, the first command `T1` tries to execute is the acquisition of the lock `l_T1`, and `T1` can only proceed if the lock acquisition succeeds. 21

2.3 The example program from Figure 1.2. 22

2.4 A typical execution of the program in Fig. 1.1 with two threads, in accordance to the standard interleaving semantics. The execution is an interleaving of instructions from the different threads, and is indicated by the solid arrows. The dashed arrows indicate a couple of partial-orders induced by this execution: the *program-order* (po), which relates successive instructions from the same thread, and *synchronizes-with* (sw), which relates successive synchronization operations. The reflexive and transitive closure of these partial orders forms the *happens-before* (hb) relation, induced by this particular execution. The write of `x` at location 2 in thread t_1, and its subsequent read at location 11 in thread t_2, can be seen to be related by hb. 25

3.1 An execution of program Figure 1.2 with 2 threads. 33

3.2 Operation of the functions $take_x$, $take_y$, $take_z$, and *updEnv* when t_2 acquires `m`. The superscripts indicate the versions. 34

3.3 A typical execution of a program P in the *L-DRF* semantics. The solid arrows represent the interleaved execution of the instructions from different threads. The dotted arrows denote the happens-before path induced by this execution. The figure marks the sections of the happens-before path which are program-order related (po), and the transitions related by synchronizes-with(sw). 37

3.4 The proof obligation for Soundness. For any $n+1$ length trace π of program P in the standard semantics, we show that there exists a $n+1$ length trace $\hat{\pi}$ in the *L-DRF* semantics, such that $\chi(\hat{\pi}) = \pi$. 42

3.5 The proof obligation for Completeness. For any $n+1$ length trace $\hat{\pi}$ of program P in the *L-DRF* semantics, we show that $\chi(\hat{\pi})$ is a valid a $n+1$ length trace of P in the standard semantics. 46

4.1 Illustrating the *mix* on a set of containing two environments ϕ_1 and ϕ_2. Observe that the invariant $x = y$ holds in the input environments. However, since this *mix* operates at the granularity of single variables, the correlation is lost in the output states. 53

4.2 The proof obligation for proving that the analysis $\mathcal{F}_\times$ is a consistent abstraction of the *L-DRF* analysis $\mathcal{F}$. 55

4.3 A simple program demonstrated the benefit of using thread-identifiers in the abstract state. In the normal setting, the synchronizes-with edges creates a cycle in the program, and it is not possible to derive an upper bound on the value of `x`. However, if we track thread-identifiers in the state, thread `t1` observes that any state it receives from `t2` is tagged with the {`t1`}, and thus `t1` can safely drop the data flow facts. 58

5.1 Illustrating the operation of *mix* when it is aware of regions. In this example, with the regions being $\langle\{x, y\}, \{z\}\rangle$, the function maintains the correlation between x and y in the output. 64

5.2 The improved precision of the region aware *mix* derived from the *R-DRF* semantics allows it to prove the additional assertion at line 11 in Figure 1.2. . . . 64

6.1 Overview of RATCOP. 66

6.2 Running times, on a log scale, of RATCOP (**A3**) and Batman (Bm-oct) on loosely coupled threads. The number of shared variables is fixed at 6. 69

7.1 Example demonstrating that a program can be DRF, when when the accesses of a global variable (in this case, the write and read of `x` at lines 11 and 12 respectively) are not directly guarded by any lock. 73

8.1 An example FreeRTOS application and its execution 78

8.2 Kernel data-structures in FreeRTOS. The data structures within the upper rectangle are protected by the `SchedulerSuspended` flag. `RxLock` and `TxLock` protects the `WaitingToSend` and `WaitingToReceive` data structures, respectively. 80

8.3 Excerpts from FreeRTOS functions 81

9.1 Extract from the `QueueSend` function in Figure 8.3. These instructions constitute a critical write to the user-queue data structure. 84
9.2 The code on the left is a version of the `QueueSend` library function, with the interrupts *disabled* at line 2. The code on the right is the example application presented in Chapter 8. 86
9.3 Excerpt of the `IncrementTick` function outlined in Figure 8.3. 86

10.1 Model of the flow of control between the scheduler, a library function (task), and an ISR in FreeRTOS. 90
10.2 Promela model of an interrupt. If the `SchedulerSuspended` flag is set, we ensure that the control returns to the preempted task. Otherwise, after the execution of the ISR, context may switch to an arbitrary ready task. 91
10.3 The abstraction of the `QueueSend` library function in the Promela modeling language. For each shared data structure `x` we are interested in modeling, we introduce an integer variable `_x`. For example, we use `_queueData` above as an abstraction of the data component of a queue. Reads to `x` are modeled by an increment of `_x` by 1, followed by a decrement. Similarly, writes are modeled by an increase of `_x` by 2, followed by a decrease by 2. If, in addition, the accesses are made in a non-atomic section of code, the increment and decrement operations are interspersed by a call to the `interrupt` function, to model the fact that an interrupt invocation may occur while `x` is being accessed. The `Lock` and `UnLock` functions are described in Figure 10.4 and Figure 10.5 respectively. 92
10.4 The `LockQueue` function used in Figure 10.3. 93
10.5 The `UnlockQueue` function used in Figure 10.3. 93
10.6 Promela model M_n. 94

11.1 An example library where there is a potential data race between library functions `F` and `G`. However, any execution which causes this data race also needs to execute the library function `H`. Thus, a "reduction" to $\mathcal{M}_{red}$ does not suffice for this library. 97
11.2 The execution ρ and its reduction ρ_{red}. 99

12.1 Experimental Evaluation of $\mathcal{M}_{red}$ 107

13.1 Applicability of earlier work to our setting. 113

List of Tables

2.1 Set of Program Commands, *cmd* 20

6.1 Summary of the experiments. Superscript [B] indicates that the program has an actual bug. (C) indicates the use of Convex Polyhedra as abstract data domain. "*" indicates a program where we have altered/weakened the original assertion. The ✓ column indicates the number of assertions the tool was able to prove. 67

6.2 Running times of RATCOP (**A3**) and Batman (Bm-oct) on loosely coupled threads. The number of shared variables is fixed at 6. 69

12.1 Some Sample Detected Races. "H" indicates Harmful races and "PB" indicates Possibly Benign. 108

12.2 List of Harmful Races 109

12.3 List of Potentially Benign Races 110

LIST OF TABLES

Chapter 1

Introduction

Concurrent programs are ubiquitous today across the entire computing spectrum: from software running on low power micro-controllers, to programs running on powerful machines with multiple processing cores. In recent years, the adoption of concurrent programs has accelerated due to the proliferation of multi-core processors. Concurrency results in performance gains thanks to the exploitation of potential parallelism in a system. Concurrent behaviors may occur either due to the presence of "interrupts", which causes a switch of context from the currently executing task, or due to the explicit creation of threads which run in parallel. Unfortunately, writing efficient and correct concurrent programs is a notoriously hard endeavor, owing to the large number of possible behaviors due to the interactions between the concurrently executing tasks or threads. Thus, concurrent programs provide fertile ground for a large class of insidious defects: the bugs are difficult to detect, difficult to reproduce, and can, unpredictably, result in catastrophic failures.

The large number of possible program behaviors makes automated reasoning about these behaviors a very hard problem. This is also the reason why traditional testing techniques are of little efficacy for uncovering bugs in concurrent programs. The problem is exacerbated by the fact that *finding* an input which manifests a concurrency defect is not enough, because many different schedules can result for the same input. Thus, reproducibility of a detected defect is crucial for an algorithm to be useful.

While many formal and automated techniques have been proposed to argue about the behaviors of concurrent programs, we look at three broad classes: systematic testing, dynamic analyses, and static analyses.

The key to systematic concurrency testing techniques (SCT) [28, 37, 84] (synonymous to stateless model checking) is the belief that most concurrency defects manifest with only a few

number of context switches. The common idea in these approaches is to bound the concurrent interleavings in some way (for example preemption bounding [61] or delay bounding [28]), and then repeatedly executing the program while taking control of the scheduler, in order to ensure that the same interleaving is not explored twice. [78] provides an excellent comparative study of the various systematic concurrency testing techniques. Such techniques uncover real bugs which are reproducible, and provide good coverage guarantees modulo the bound.

However, an SCT-based tool must have a scheduler that is aware of all possible sources of non-determinism, since any such source is modeled as an invocation of this scheduler [47]. This makes SCT difficult to apply in many scenarios [78]. Moreover, as [78] highlights, some concurrency bugs do not always cause a crash, and requires the additional non-trivial overhead of the design of corresponding reliable and efficient checks.

In contrast to stateless model checking, [89, 80] provide techniques to model check concurrent programs with explicit-state tracking. However, such techniques can be very resource intensive: in Part II of this thesis, we discuss an explicit-state model checking based approach to finding a class of concurrency defects (which we call "high-level data races") in the kernel library functions of FreeRTOS, a popular real-time operating system. Model checking is an exhaustive check of a specific property on the entire state space of a system. Thus, for computability, certain bounds are enforced (for example, in Chapter 12, we bound the maximum number of concurrently running tasks or threads). In the case of FreeRTOS we discovered that for an application with just 2 concurrently executing tasks, a sophisticated model checker [44] required over 39GB of RAM, and ran for hours before timing out.

The other issue with both stateless and explicit-state model checking is their inherent incompleteness: such techniques cannot prove the *absence* of bugs. In the case of model-checking, it is often not possible to a priori provide a bound for the number of threads needed for a bug to manifest. Thus, even though the check is exhaustive within the bound, it is incomplete in general. In Chapter 11, we state and prove a meta claim which allowed us to use an existing explicit-state model checker for our problem of detecting "high-level data races" [1] in FreeRTOS, and yet circumvent the issues of scalability and incompleteness. Using our technique, a user is able to *soundly* disregard 99.8% of an estimated 41,000 potential high-level races in FreeRTOS. Our tool flagged 16 harmful races, and an additional 22 benign ones.

Dynamic analysis techniques take as input a concurrent program and a set of test inputs. Typically, the compiled program is instrumented with additional instructions which record

[1] While we define this more precisely later on, intuitively, high-level data races are interleavings of regions of code of concurrently executing processes. A high-level data race is a necessary condition for *atomicity violations*, which is an important class of concurrency defects.

events of interest (for example, forking of threads, accesses to shared variables, and so forth). Each execution of the program on a test input induces an "abstract trace", which is a sequence of such interesting events. The dynamic analyzer now performs reasoning over these abstract traces. A rich set of dynamic analyzers exist [73, 27, 35, 67, 66, 52, 86, 34] targeting a variety of concurrency defects like data races, deadlocks and atomicity violations. The bugs uncovered by such techniques are true positives. However, dynamic analyses suffer from some drawbacks. First, they are incomplete as well: dynamic analyses are unable to prove the absence of defects. Second, there is the additional overhead of providing input test cases to the analyzer, and it is non-trivial to devise good concurrent tests. There have been recent works [69, 68, 70, 71] which address the problem of synthesizing tests for dynamic analyzers.

Static analysis techniques, as the name suggests, analyzes a program without executing it. In this thesis, we consider static analyses based on the abstract interpretation [19] framework. Static analyses are sound: they compute an over-approximation of all the possible concrete states arising at *every* program point. Thus, a static analyzer can prove the absence of bugs. However, the computation may generate false positives in the process– an analysis may be unable to rule out a particular defect at a program point.

The first step in building such an analysis is to define a "most precise" concrete semantics, which defines the precise set of behaviors of a given program. The concrete semantics is, in general, incomputable. The abstract interpretation framework then suggests how to systematically perform suitable abstractions of the concrete semantics to devise computable *abstract* analyses. The steps here involve designing abstract states which correctly approximate a given set of concrete states. The domain of abstract states and concrete states are related by a couple of functions α and γ (α being the abstraction and γ being the concretization), which are together said to form a "Galois connection". Next, for each command in the programming language, one needs to define a sound *abstract* transformer, which interprets each command over the abstract states. The abstract analysis, which is a conglomeration of the abstract states (related via α and γ with the concrete states) along with the abstract transformers, is then said to be a *consistent* abstraction of the concrete analysis. An analysis which is a consistent abstraction of the most precise concrete analysis provides the guarantee that the solution computed by it satisfies the aforementioned notion of soundness.

Static analyzers have been successfully used to analyze and verify several large scale, and some safety-critical, systems and a rich set of static analyzers are available today [21, 13, 53, 6, 79]. However, while the problem of devising efficient static analyses for sequential programs has been well studied and is fairly well understood, doing the same for concurrent programs is still

an active area of research. The main difficulty arises from the intrinsic need of a static analyzer to capture all possible interferences *between* threads. In a typical multi-threaded program, the set of possible interferences between threads may be large. Thus, a precise analysis does not scale, while a fast analysis is quite imprecise [17].

In the first part of this thesis, we address the problem of devising efficient static analyses for an important subclass of shared-variable multi-threaded programs, namely programs which are data race free (DRF). Our starting point is a novel new *thread-local* concrete semantics for race free multi-threaded programs, which we call the *L-DRF* semantics. We show that for programs without races, the *L-DRF* semantics is equivalent to the standard concrete semantics. Using the *L-DRF* semantics, one can devise efficient consistent abstractions, which are fairly precise, and are in the same spirit as the "sync-CFG" analysis first proposed in [22]. These abstract analyses satisfy a non-standard notion of soundness– the values computed for a variable are sound only at points where they are *relevant*. The analyses thus trade soundness at all points for gains in efficiency. The *L-DRF* semantics also aids in rapidly porting existing sequential analyses to scalable analyses for DRF programs. Next, we parameterize the *L-DRF* semantics with a partitioning of the program variables into "regions" which are accessed atomically. Abstractions of the region-parameterized semantics yield more precise analyses for concurrent programs which are "region race" free (a new notion that we introduce, which is a stronger property than data race freedom). We instantiate these abstractions to devise efficient relational analyses for race free programs, which we have implemented in a prototype tool called RATCOP. On the benchmarks, RATCOP was able to prove up to 65% of the assertions, in comparison to 25% proved by a version of the analysis from [22].

In the following few sections, we introduce our contributions in more detail.

1.1 A Thread-Local Semantics, and its efficient Sequential Abstractions for Race Free Programs

Our aim in this part of the thesis is to provide a framework for developing data-flow analyses which specifically target the class of *data race free* (DRF) concurrent programs. Informally, a program is data race free if along every execution of it, all parallel accesses to a shared memory location, with at least one access being a write, are separated by some form of synchronization.

Data races are the sources of many concurrency defects. All data races, "benign" or otherwise, are considered to be errors [11, 2, 12, 5, 83]. DRF programs constitute an important class of concurrent programs because they are guaranteed to have *sequentially consistent* (SC) execution behaviors in many weak memory models [3, 56]. Non-DRF programs do not have

this guarantee, and program behaviors which do not conform to SC-semantics are often hard to comprehend and reason about (leading to insidious defects). As such, programmers are expected to write DRF programs [11].

The starting point of this work is the so-called "sync-CFG" style of analysis proposed in [22] for race-free programs. The analysis here essentially runs a sequential analysis on each thread, communicating data-flow facts between threads only via "synchronization edges" that go from a release statement in one thread to a corresponding acquire statement in another thread. The analysis thus runs on the control-flow graphs (CFGs) of the threads augmented with synchronization edges, as shown in the center of Figure 1.2, which explains the name for this style of analysis. The analysis computes data flow facts about the value of a variable that are sound *only* at points where that variable is *relevant*, in that it is read or written to at that point. The analysis thus trades unsoundness of facts at irrelevant points for the efficiency gained by restricting interference between threads to the points of synchronization alone.

However, the analysis proposed in [22] suffers from some drawbacks. Firstly, the analysis is intrinsically a "value-set" analysis, which can only keep track of the set of values each variable can assume, and not the relationships *among* variables. Any naive attempt to extend the analysis to a more precise relational one quickly leads to unsoundness. The second issue is to do with the technique for establishing soundness. A convenient way to prove soundness of an analysis is to show that it is a consistent abstraction [19] of a canonical analysis, like the collecting semantics for sequential programs or the interleaving semantics for concurrent programs. However, a sync-CFG style analysis *cannot* be shown to be a consistent abstraction of the standard interleaving semantics, due largely to the unsoundness at irrelevant points. Instead, one needs to use an intricate argument, as done in [22], which essentially shows that in the least fixed point of the analysis, every write to a variable will flow to a read of that variable via a happens-before path (that is guaranteed to exist by the property of race-freedom). Thus, while one can argue soundness of an analysis that abstracts the value-set analysis by showing it to be a consistent abstraction of the value set analysis, to argue soundness of any other proposed sync-CFG style analysis (in particular one that is more precise than the value-set analysis), one would have to resort to a similar involved proof as in [22].

Towards addressing these issues, we propose a framework that facilitates the design of different sync-CFG analyses with varying degrees of precision and efficiency. The foundation of this framework is a thread-local semantics for DRF programs, that can play the role of a "most precise" analysis, and which other sync-CFG analyses can be shown to be consistent abstractions of. This semantics, which we call *L-DRF*, is similar to the interleaving semantics

of concurrent programs [51], but keeps thread-local (or per-thread) copies of the shared state. Intuitively, our semantics works as follows. Apart from its local copy of the shared data state, each thread t also maintains a per-variable version count, which is incremented whenever t updates the variable. The exchange of information between threads is via buffers, associated with the release points in the program. When a thread releases a lock, it stores its data state to the corresponding buffer, along with the version counts of the variables. As a result, the buffer of a release point records both the local data state and the variable versions as they were when the release was last executed. When some thread t acquires a lock `m`, it compares its per-variable version count with those in the buffers pertaining to release points associated with `m`, and copies over the valuation of a variable to its local state, if it is newer in some buffer (as indicated by a higher version count). Similar to a sync-CFG analysis, the value of a shared variable in the local state of a thread may be *stale*. The *L-DRF* semantics leverages the race freedom property to ensure that the value of a variable is correct in a local state at program points where it is *read*.

It thus captures the essence of a sync-CFG analysis. The *L-DRF* semantics is also of independent interest, since it can be viewed as an alternative characterization of the behavior of data race free programs.

The analysis induced by the *L-DRF* semantics is shown to be sound for DRF programs. In addition, the analysis is in a sense the most precise sync-CFG analysis one can hope for, since at every point in a thread, the relevant part of the thread-local copy of the shared state is *guaranteed* to arise in some execution of the program.

Using the *L-DRF* semantics as a basis, we now propose several precise and efficient *relational* sync-CFG analyses. The soundness of these analyses all follow immediately, since they can easily be shown to be consistent abstractions of the *L-DRF* analysis. The key idea behind obtaining a sound relational analysis is suggested by the *L-DRF* analysis: at each `acquire` point we apply a *mix* operator on the abstract values, which essentially amounts to forgetting all correlations between the variables.

While these analyses allow maintaining fully-relational properties within thread-local states, communicating information over cross-thread edges loses all correlations due to the *mix* operation. To improve precision further, we refine the *L-DRF* semantics to take into account *data regions*. Technically, we introduce the notion of *region race freedom* and develop the *R-DRF* semantics: the programmer can partition the program variables into "regions" that should be accessed *atomically*. A program is *region race free* if it does not contain conflicting accesses to variables in the same region, that are that are not separated by ordering constraints. The

classical notion of data race freedom is a special case of region race freedom where each region consists of a single variable, and techniques to determine that a program is race free can be naturally extended to determine region race freedom (see Section 5.2). For region race free programs, *R-DRF*, which refines *L-DRF* by taking into account the atomic nature of accesses that the program makes to variables in the same region, produces executions which are indistinguishable, with respect to reads of the regions, from the ones produced by *L-DRF*. By leveraging the *R-DRF* semantics as a starting point, we obtain more precise sequential analyses that track relational properties *within regions* across threads. This is obtained by refining the granularity of the *mix* operator from single variables to regions.

We have implemented our analyses in a prototype analyzer called RATCOP (Relational Analysis Tool for Concurrent Programs), and provide a thorough empirical evaluation in Chapter 6. We show that RATCOP attains a precision of up to 65% on a subset of race-free programs from the SV-COMP15 suite. In contrast, an interval based value-set analysis derived from [22] was able to prove only 25% of the assertions. On a separate set of experiments, RATCOP turns out to be nearly 5 orders of magnitude faster than a recent abstract interpretation based tool [60].

1.1.1 Illustrative Example

We illustrate the *L-DRF* semantics, and its sequential abstractions, on the simple program in Figure 1.1.

```
Thread t1() {

1:  acquire(m);
2:  x := y;
3:  x++;
4:  y++;
5:  assert(x==y);
6:  release(m);

}
```

```
Thread t2() {

 8:  z++;
 9:  assert(z==1);
10:  acquire(m);
11:  assert(x==y);
12:  release(m);

}
```

Figure 1.1: A simple race free multi-threaded program. The variables x, y and z are shared and initialized to 0.

We assume that all variables are shared and are initialized to 0. The threads access x and y only after acquiring lock m. The program is free from data races. The *sync*-CFG representation of the program, along with the data flow facts computed by our analyses is given in Figure 1.2.

R-DRF	L-DRF	Value-Set	Thread t_1	Thread t_2	Value-Set	L-DRF	R-DRF
$0 = x = y = z$	$0 = x = y = z$	$0 = x = y = z$			$0 = x = y = z$	$0 = x = y = z$	$0 = x = y = z$
			1: acquire (m);	8: z++;			
$x = y$, $0 \le y$, $0 \le z \le 1$	$0 \le x$, $0 \le y$, $0 \le z \le 1$	$0 \le x$, $0 \le y$, $0 \le z \le 1$			$x = 0$, $y = 0$, $0 \le z \le 1$	$0 = x = y$, $z = 1$	$0 = x = y$, $z = 1$
			2: x := y;	9: assert (z = 1);			
$x = y$, $0 \le y$, $0 \le z \le 1$	$x = y$, $0 \le y$, $0 \le z \le 1$	$0 \le x$, $0 \le y$, $0 \le z \le 1$			$x = 0$, $y = 0$, $0 \le z \le 1$	$0 = x = y$, $z = 1$	$0 = x = y$, $z = 1$
			3: x++;	10: acquire (m);			
					$0 \le x$, $0 \le y$, $0 \le z \le 1$	$0 \le x$, $0 \le y$, $0 \le z \le 1$	$x = y$, $0 \le y$, $0 \le z \le 1$
			4: y++;	11: assert (x = y);			
$x = y$, $1 \le y$, $0 \le z \le 1$	$x = y$, $1 \le y$, $0 \le z \le 1$	$1 \le x$ $1 \le y$, $0 \le z \le 1$			$0 \le x$, $0 \le y$, $0 \le z \le 1$	$0 \le x$, $0 \le y$, $0 \le z \le 1$	$x = y$, $0 \le y$, $0 \le z \le 1$
			5: assert (x = y);	12: release (m); 13:			
$x = y$, $1 \le y$, $0 \le z \le 1$	$x = y$, $1 \le y$, $0 \le z \le 1$	$1 \le x$ $1 \le y$, $0 \le z \le 1$					
			6: release (m); 7:				

Figure 1.2: A simple race free program with two threads t_1 and t_2, with all variables being shared and initialized to 0. The columns *L-DRF* and *R-DRF* show the facts computed by polyhedral abstractions of our thread-local semantics and its region-parameterized version, respectively. The Value-Set column shows the facts computed by interval abstractions of the Value-Set analysis of [22]. *R-DRF* is able to prove all 3 assertions, while *L-DRF* fails to prove the assertion at line 11. Value-Set only manages to prove the simple assertion at line 9.

A state in the *L-DRF* semantics keeps track of the following components: a location map pc mapping each thread to the location of the next command to be executed, a lock map μ which maps each lock to the thread holding it, a local environment (variable to value map) Θ for each thread, and a function Λ which maps each buffer (associated with each location following a release command) to an environment. Every release point of each lock `m` has an associated buffer, where a thread stores a copy of its local environment when it executes the corresponding release instruction. In the environments, each variable x has a version count associated with it which, along any execution π, essentially associates this valuation of x with a unique prior write to it in π. As an example, the "versioned" environment $\langle x \mapsto 1^1, y \mapsto 1^1, z \mapsto 0^0\rangle$ says that x and y have the value 1 by the 1st writes to x and y, and z has not been written to. An

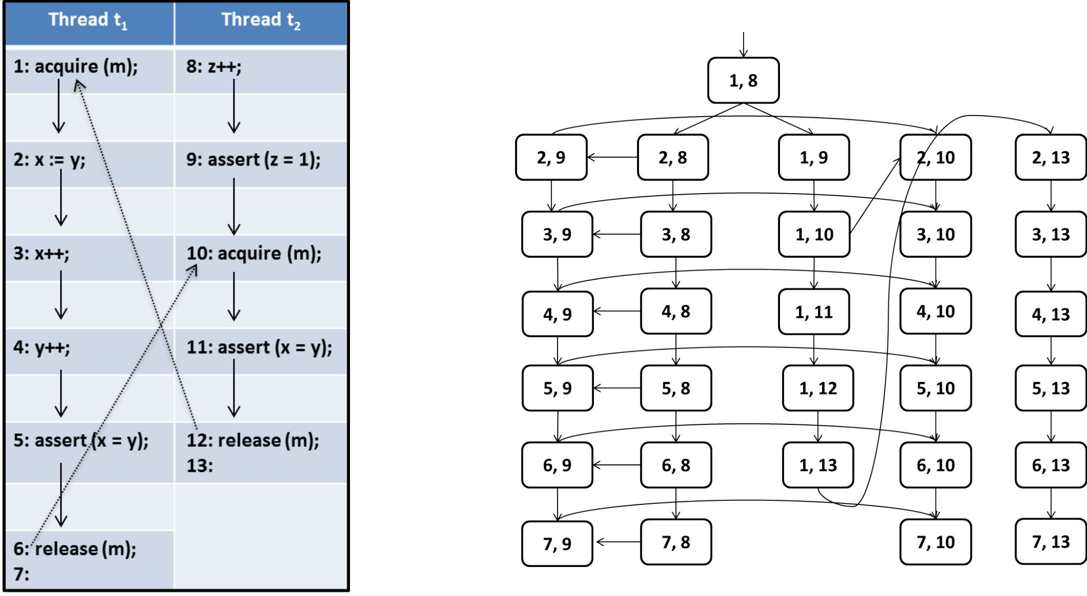

Figure 1.3: The sync-CFG representation of the program in Figure 1.2 is presented on the left. On the right is an excerpt of the standard product graph representation of the same program. In a sequential setting, a control flow graph is a standard way to represent programs, where each node denotes a control location, and edges represent flow of control. Each node in the product graph represents a *combination* of possible program locations across threads, and an edge between two such nodes represents an interleaving step. Any abstract analysis running on the product graph will clearly not scale for large programs. The sync-CFG, on the other hand, is a more sparse representation, resulting in improved scalability.

execution is an interleaving of commands from the different threads. Consider an execution where, after a certain number of steps, we have the state $pc(t_1 \mapsto 6, t_2 \mapsto 10), \Theta(t1) = \langle x \mapsto 1^1, y \mapsto 1^1, z \mapsto 0^0\rangle, \Theta(t2) = \langle x \mapsto 0^0, y \mapsto 0^0, z \mapsto 1^1\rangle, \mu(m) = t_1, \Lambda = \bot$. The buffers are all empty as no thread has executed a `release` yet. Note that the values (and versions) of x and y in $\Theta(t_2)$ are *stale*, since it was t_1 which last modified them (similarly for z in $\Theta(t_1)$). Next, t_1 can execute the `release` at line 6, thereby setting $\mu(m) = _$ and storing its current local state to $\Lambda(7)$. Now t_2 can execute the `acquire` at line 10. The state now becomes $pc(t_1 \mapsto 7, t_2 \mapsto 11)$, $\mu(m) = t_2$, and t_2 now "imports" the most up-to-date values (and versions) of the x and y from $\Lambda(7)$. This results in its local state becoming $\langle x \mapsto 1^1, y \mapsto 1^1, z \mapsto 1^1\rangle$ (the valuations of x and y are pulled in from the buffer, while the valuation of z in t_2's local state persists). The value of x and y in $\Theta(t_2)$ is no longer stale: *L-DRF* leveraged the race freedom to ensure that the values of x and y are correct when they are read at line 11.

Roughly, we obtain *sequential* abstractions of *L-DRF* via the following steps:

1. Provide a data abstraction of sets of environments
2. Define the state to be a map from locations to these abstract data values
3. Draw inter-thread edges by connecting releases and acquires of the same lock (as shown in Figure 1.2).
4. Define an abstract *mix* operation which soundly approximates the "import" step outlined earlier
5. Analyze the program as if it was a sequential program, with *inter*-thread join points (the `acquire`'s) using the *mix* operator.

The analysis in [22] is precisely such a sequential abstraction, where the abstract data values are abstractions of *value-sets* (variables mapped to sets of values). Value sets do not track correlations between variables, and only allow coarse abstractions like Intervals [18]. The *mix* operator, in this case, turns out to be the standard join. For Figure 1.2, the interval analysis only manages to prove the assertion at line 9.

A more precise relational abstraction of *L-DRF* can be obtained by abstracting the environments as, say, convex polyhedra [20]. As shown in Figure 1.2, the resulting analysis is more precise than the interval analysis, being able to prove the assertions at lines 5 and 9. However, in this case, the *mix* must forget the correlations among variables in the incoming states: it essentially treats them as value sets. This is essential for soundness. Thus, even though the `acquire` at line 10 obtains the fact that $x = y$ from the buffer at 7, and the incoming fact from 9 also has $x = y$, it fails to maintain this correlation after the *mix*. Consequently, it fails to prove the assertion at line 11.

Finally, one can exploit the fact that x and y form a data region, that is always accessed atomically by the two threads. The program is thus *region race free*, for this particular region definition. One can parameterize the *L-DRF* semantics with this region definition, to yield the *R-DRF* semantics. The resulting sequential abstraction maintains relational information as in polyhedra based analysis derived from *L-DRF*, but has a more precise *mix* operator which preserves relational facts which hold *within* a region. Since both the incoming facts at line 10 satisfy $x = y$, the *mix* preserves this fact, and the analysis is able to prove the assertion at 11.

Note that in all the three analyses, we are guaranteed to compute sound facts for variables *only* at points where they are accessed. For example, all three analyses claim that x and y are both 0 at line 9, which is clearly wrong. However, x and y are not accessed at this point. We make this trade-off for the soundness guarantee in order to achieve a more efficient

analysis. Also note that in Figure 1.2, the inter-thread edges add a spurious loop in the program graph (and, therefore, in the analysis of the program), which prevents us from computing an upper bound for the values of x and y. We show in a later section (Section 4.5) how we can appropriately abstract the versions to avoid some of these spurious loops.

1.2 Detecting all High-Level Data Races in an RTOS Kernel

In the first part of the thesis, we considered shared variable programs with explicit threads which run in parallel. A typical example of such a system is the Java programming language, with its threads library. In the second part of the thesis, we consider a different class of concurrent programs. We deal with a system library like the kernel API of a real-time operating system (RTOS). Such systems exhibit concurrent behaviors even without explicit thread creation, and while running on a single-core architecture. The reason for this is the presence of *interrupts*, which have the capability of preempting a currently executing task.

Our objective in this part of the work is to expose atomicity violations in the library functions of such an kernel API. For this purpose, we introduce the notion of "high-level data races". Intuitively, a high-level race occurs when an execution interleaves instructions corresponding to user-annotated critical accesses to shared memory structures. A high-level data race is a necessary condition for there to be an atomicity violation.

We propose a technique for detecting *all* high-level data races in a system library like the kernel API of a real-time operating system (RTOS) that relies on flag-based scheduling and synchronization. In contrast to the abstract interpretation framework which we used in Part I, our methodology here is based on model-checking, which allows our technique to have a low false positive rate. However, standard model-checking techniques suffer from two drawbacks: they are incomplete (they cannot prove the absence of defects), and suffer from scalability issues. To circumvent these issues, our technique relies on a meta-argument to bound the number of concurrently running tasks needed to orchestrate a race. This also results in gains in scalability.

We describe our approach in the context of FreeRTOS, a popular RTOS in the embedded domain. While we detail our contributions in the context of FreeRTOS, the operating system is quite representative of libraries in its class. Thus, our techniques generalize to other APIs which permit interrupts, and follow a flag-based scheduling and synchronization paradigm.

1.2.1 Overview of our Approach

Atomicity violations [36] precisely characterize the bugs in a method library that arise due to *concurrent* use of the methods in the library. An execution of an application program that uses the library is said to exhibit an atomicity violation if its behavior cannot be matched by any "serialized" version of the execution, where none of the method calls interleave with each other. As one may expect, such bugs can be pernicious and difficult to detect.

```
main() {

 1:  x = 0;
 2:  start(t1);
 3:  start(t2);
 4:  join(t1);
 5:  join(t2);
 6:  assert(x==2);

}
```

```
t1() {

 7:  acquire(m);
 8:  temp1 = x;
 9:  release(m);
10:  temp1 = temp1 + 1;
11:  acquire(m);
12:  x = temp1;
13:  release(m);

}
```

```
t2() {

14:  acquire(m);
15:  temp2 = x;
16:  release(m);
17:  temp2 = temp2 + 1;
18:  acquire(m);
19:  x = temp2;
20:  release(m);

}
```

Figure 1.4: A simple program demonstrating an atomicity violation. The variable `x` is global. The program is free from data races. Since it is initialized to 0, and the threads perform 2 increments to it, the expectation of a developer would be that $x = 2$ when the threads finish. However, x can, in fact, be 1 at line 6.

Figure 1.4 shows a simple multi-threaded program with 3 threads, which can exhibit an atomicity violation. The variable `x` is shared, and the program is free from data races. The `main` thread spawns two child threads, each of which essentially increments `x`. However, the critical section between lines $7 \rightarrow 9$ may immediately follow the critical section between lines $14 \rightarrow 16$. This causes both the temporary variables $temp1$ and $temp2$ to be set to 0. Thus, the eventual value of `x` becomes 1, even though there are *two* logical increments to it. Note that the execution which causes the value of `x` to be 1 at line 6 cannot be produced by any serial invocation of the two increment methods.

This defect can be captured by marking lines $8 \rightarrow 12$ and $15 \rightarrow 19$ as *critical accesses* to the `x`. Observe that it is necessary for these critical accesses to interleave in order to have an atomicity violation.

A necessary condition for an atomicity violation to occur in a library L is that two method invocations should be able to "race" (or interleave) in an execution of an application that uses L. In fact it is often necessary for two "critical" access paths in the source code of the methods (more precisely the instructions corresponding to them) to interleave in an execution,

to produce an atomicity violation. With this in mind, we could imagine that a user (or the developer herself) annotates blocks of code in each method as critical accesses to a particular unit of memory structures. We can now say that an execution exhibits a "high-level" race (with respect to this annotation) if it interleaves two critical accesses to the same memory structure.

Suppose we now had a way of finding the precise set R of pairs of critical accesses that could race with each other, across *all* executions in *all* applications programs that use L. We call this the problem of finding all high-level races in L. The user can now focus on the set R, which is hopefully a small fraction of the set of all possible pairs, and investigate each of them to see whether they could lead to atomicity violations. We note that the user can *soundly* disregard the pairs outside R as they can never race to begin with, and hence can never be the cause of any atomicity violation.

In this paper we are interested in the problem of finding all high-level races in a library like the Application Programmer Interface (API) of a real-time kernel. The particular system we are interested in is a real-time operating system (RTOS) called FreeRTOS [64]. FreeRTOS is one of the most popular operating systems in the embedded industry, and is widely used in real-time embedded applications that run on micro-controllers with small memory. FreeRTOS is essentially a library of API functions written in C and Assembly, that an application programmer invokes to create and manage tasks. Despite running on a *single* processor or core, the execution of tasks (and hence the kernel API functions) can interleave due to interrupts and context-switches, leading to potential races on the kernel data-structures.

The kind of control-flow and synchronization mechanisms that kernels like FreeRTOS use are non-standard from a traditional programming point of view. To begin with, the control-flow *between* threads is very non-standard. In a typical concurrent program, control could potentially switch between threads at *any* time. However in FreeRTOS, control switching is restricted and depends on whether interrupts have been disabled, the value of certain flag variables like `SchedulerSuspended`, and whether the task is running as part of an interrupt service routine (ISR). Secondly, FreeRTOS does not use standard synchronization mechanisms like locks, but relies instead on mechanisms like disabling interrupts and flag-based synchronization. This makes it difficult to use or adapt some of the existing approaches to high-level race detection like [7, 81] or classical data race detection like [29, 82], which are based on standard control-flow and lock-based synchronization.

An approach based on model-checking could potentially address some of the hurdles above: one could model the control-flow and synchronization mechanism in each API function faithfully, create a "generic" task process that non-deterministically calls each API function, create

a model (say M_n) that runs n of these processes in parallel, and finally model-check it for data-races. But this approach has some basic roadblocks: certain races need a minimum number of processes running to orchestrate it—how does one determine a sufficient number of processes n that is guaranteed to generate *all* races? Secondly, even with a small number of processes, the size of the state-space to be explored by the model-checker could be prohibitively large.

The approach we propose and carry out in this paper is based on the model-checking approach above, but finds a way around the hurdles mentioned. The key idea is to create a set of *reduced* models, say $\mathcal{M}_{red}$, in which each model essentially runs only three API functions at a time. We then argue that a race that shows up in M_n, for *any* n, must also be a race in one of the reduced models in $\mathcal{M}_{red}$. Model-checking each of these reduced models is easy, and gives us a way of finding *all* data-races that may ever arise due to use of the FreeRTOS API. We note that the number of API functions to run in each reduced model (three in this case), and th argument of sufficiency, is specific to FreeRTOS. In general, this will depend on the library under consideration.

On applying this technique to FreeRTOS (with our own annotation of critical accesses) we found a total of 48 pairs of critical accesses that could race. Of these 10 were found to be false positives (i.e. they could not happen in an actual execution of a FreeRTOS application). Of the remaining, 16 were classified as harmful, in that they could be seen to lead to atomicity violations. The bottom-line is that the user was able to disregard 99.8% of an estimated 41,000 potential high-level races.

1.3 Structure of this Thesis

The rest of this thesis is organized as follows. In Part I of this thesis (between Chapter 2 and Chapter 7) we devise efficient static analyses for the class of race free multi-threaded programs. Chapter 2 sets up the mathematical notations, followed by the definition of the standard interleaving semantics of programs. We conclude the chapter by precisely defining the notion of data race freedom. Chapter 3 introduces a novel thread-local semantics called *L-DRF*, which we show to be equivalent to the interleaving semantics. The computable abstractions of the *L-DRF* semantics are discussed in Chapter 4. We then parameterize the *L-DRF* with region definitions, and present more precise analyses in Chapter 5. Finally, we present our experimental evaluation in Chapter 6 and conclude this part in Chapter 7.

In Part II of the thesis, we present a technique to detect *all* high-level data races in an RTOS kernel. Chapter 8 provides an overview of the architecture of FreeRTOS. Next, Chapter 9 defines what we mean by data races in executions of FreeRTOS applications. Chapter 10 describes

how we model the concurrency in FreeRTOS, as well as accesses to shared data structures. We state and prove our reduction argument in Chapter 11. Finally, we present our experimental evaluation in Chapter 12 and present related works in Chapter 13.

We conclude the thesis in Chapter 14, and provide pointers to future work.

Part I

Thread-Local Semantics, and its Efficient Abstractions, for Race Free Programs

Chapter 2

Concurrent Programs and their Interleaving Semantics

With this chapter commences the first part of this thesis, wherein our objective is to devise efficient data flow analyses for the class of data race free (DRF) programs. In this chapter, we will setup the mathematical notations we will be using throughout Part I of this thesis (till Chapter 6). We also define the standard interleaving semantics for concurrent programs, and provide the definition of data races.

2.1 Mathematical Notations

We use $\rightarrow$ and $\rightharpoonup$ to denote total and partial functions, respectively, and $\perp$ to denote a function which is not defined anywhere. We use "_" to denote an irrelevant value, which is implicitly existentially quantified. We write $\bar{S}$ to denote a (possibly empty) finite sequence of elements coming from a set S. We denote the *length* of a sequence π by $|\pi|$, and the i-th element of π, for $0 \leq i < |\pi|$, by π_i. We denote the domain of a function ϕ by $\text{dom}(\phi)$ and write $\phi[x \mapsto v]$ to denote the function $\lambda y.\texttt{if } y = x \texttt{ then } v \texttt{ else } \phi(y)$. Given a pair of functions $ve = \langle \phi, \nu \rangle$, we write $ve \cdot \phi$ and $ve \cdot \nu$ to denote the ϕ and ν components of ve (note that the we always consider $a \cdot b$ to associate from left to right), respectively.

This work was done in collaboration with Oded Padon, Sharon Shoham and Noam Rinetzky, at the Tel-Aviv University, Israel, and Deepak D'Souza, at the Indian Institute of Science.

2.1.1 Programming Language and Programs

Type	Syntax	Description
Assignment	$x := e$	Assigns the value of expression e to variable $x \in \mathcal{V}$
Assume	`assume`(b)	Blocks the computation if boolean condition b does not hold, else skip
Acquire	`acquire`(`m`)	Acquires lock `m`, provided it is not *held* by any thread
Release	`release`(`m`)	Releases lock `m`, provided the executing thread holds it

Table 2.1: Set of Program Commands, *cmd*

A multi-threaded program P consists of four finite sets: *threads* $\mathcal{T}$, *control locations* $\mathcal{L}$, program *variables* $\mathcal{V}$ and *locks* (*mutexes*) $\mathcal{M}$. We denote by $\mathbb{V}$ the set of values the program variables can assume. Without loss of generality, we assume in this work that $\mathbb{V}$ is simply the set of integers. Figure 2.1 summarizes the semantic domains we use and the meta-variables ranging over them..

t	$\in$	$\mathcal{T}$			Thread identifiers
n	$\in$	$\mathcal{L}$			Program locations
x, y	$\in$	$\mathcal{V}$			Variable identifiers
l	$\in$	$\mathcal{M}$			Lock identifiers
r	$\in$	R			Region identifiers
v	$\in$	$\mathbb{V}$			Values
pc	$\in$	PC	$\equiv$	$\mathcal{T} \to \mathcal{L}$	Program counters
μ	$\in$	LM	$\equiv$	$\mathcal{M} \rightharpoonup \mathcal{T}$	Lock map
ϕ	$\in$	Env	$\equiv$	$\mathcal{V} \to \mathbb{V}$	Environments
ν	$\in$	$\mathcal{V}V$	$\equiv$	$\mathcal{V} \to \mathbb{N}$	Variable versions
ve	$\in$	VE	$\equiv$	$Env \times \mathcal{V}V$	Versioned environments
$s = \langle pc, \mu, \phi \rangle$	$\in$	$\mathcal{S}$	$\equiv$	$PC \times LM \times Env$	Standard States
$\sigma = \langle pc, \mu, \Theta, \Lambda \rangle$	$\in$	Σ	$\equiv$	$PC \times LM \times (\mathcal{T} \to VE) \times (\mathcal{L} \to VE)$	Thread-Local States

Figure 2.1: Semantic Domains.

Every thread $t \in \mathcal{T}$ has an entry location ent_t and a set of instructions $inst_t \subseteq \mathcal{L} \times cmd \times \mathcal{L}$, which defines the *control flow graph* of t. The set of program commands, denoted by *cmd*, is

defined in Table 2.1. A `goto` instruction from program location l to l' can be simulated by the instruction $\langle l, \texttt{assume}(\texttt{true}), l' \rangle$. Commands like **fork** and **join** of a bounded number of threads and can be simulated using locks. In particular, the `fork` of a thread can be simulated by the release of a particular lock, while the `join` operation can be simulated by a lock acquisition. Figure 2.2 explains this in more detail using an example. The `join` operation can be simulated in an analogous fashion. Note that, since we assume we are dealing with programs with a fixed number of threads, any loop which creates threads must be bounded. Consequently, they may be unrolled to this depth, and the aforementioned technique, of substituting forks with lock acquisitions, can be applied. A similar argument applies to `join` statements within a loop.

```
main() {
  ...
  fork(T1);
}

T1() {
  start;
  ...
}
```

```
main() {
  // assume main initially holds l_T1
  ...
  release(l_T1);
}

T1() {
  acquire(l_T1);
  ...
}
```

Figure 2.2: Simulating the forking of a thread using `acquire` and `release` instructions. For each thread, we assume we have a unique "thread lock" associated with it (in fact, each thread object in Java does have an unique associated monitor). We can assume that the `main` thread initially holds all the thread locks. In the example above, the unique lock associated with `T1` is called `l_T1`. The left side of the code shows the actual code with `fork`, while the one on the right is an equivalent code using commands from *cmd*. The `fork` of `T1` is substituted with the `release(l_T1)`, while the `begin` in `T1` is substituted with `acquire(l_T1)`. In other words, the first command `T1` tries to execute is the acquisition of the lock `l_T1`, and `T1` can only proceed if the lock acquisition succeeds.

For generality, we refrain from defining the syntax of the expressions e and boolean conditions b. An instruction $\langle n_s, c, n_t \rangle$ comprises a *source* location n_s, a *command* $c \in cmd$, and a *target* location n_t.

We denote the set of commands appearing in program P by $cmd(P)$. We refer to an assignment $x := e$ as a *write-access* to x, and as a *read-access* to every variable that appears in the expression e. Without loss of generality, we assume variables appearing in conditions of `assume()` commands in instructions of some thread t do not appear in any instruction of any other thread $t' \neq t$. Local variables may be dealt with by appending them with their respective thread identifiers, and then treating them as global variables.

We denote by $\mathcal{L}_t$ the set of locations in instructions of thread t, and require that the sets be disjoint for different threads. For a location $n \in \mathcal{L}$ $\left(= \bigcup_{t \in \mathcal{T}} \mathcal{L}_t\right)$, we denote by $tid(n)$ the thread t which contains location n, i.e., $n \in \mathcal{L}_t$. We forbid different instructions from having the same source and target locations, and further expect instructions pertaining to assignments, `acquire`() and `release`() commands to have unique source and target locations. Let $\mathcal{L}_t^{rel}$ be the set of program locations in the body of thread t following a `release`() command. We refer to $\mathcal{L}_t^{rel}$ as t's *post-release points* and denote the set of *release points* in a program by $\mathcal{L}^{rel} = \bigcup_{t \in \mathcal{T}} \mathcal{L}_t^{rel}$. Similarly, we define t's *pre-acquire points*, denoted $\mathcal{L}_t^{acq}$, and denote a program's *acquire points* by $\mathcal{L}^{acq} = \bigcup_{t \in \mathcal{T}} \mathcal{L}_t^{acq}$. We denote the sets of post-release and pre-acquire points pertaining to operations on lock $\mathtt{m}$ by $\mathcal{L}_{\mathtt{m}}^{rel}$ and $\mathcal{L}_{\mathtt{m}}^{acq}$, respectively.

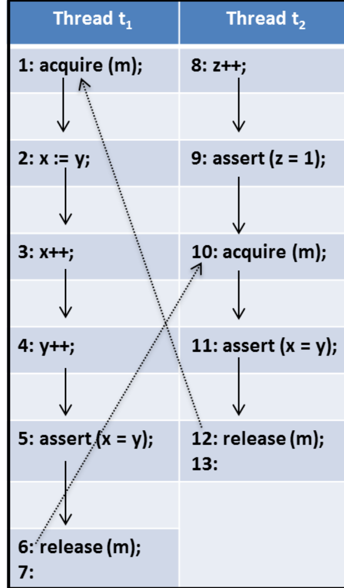

Figure 2.3: The example program from Figure 1.2.

We illustrate the definitions using an example. Consider again the program from Figure 1.2, which we present in Figure 2.3, for ease of reference. Some example instructions in this program are $\langle 2, x := y, 3\rangle$ and $\langle 10, \mathtt{acquire(m)}, 11\rangle$. The set $\mathcal{L}_{t_1}$, which is the set of program locations in thread t_1, is $\{1, 2, 3, 4, 5, 6, 7\}$, and $tid(8) = t_2$, and so forth. In this program, the set $\mathcal{L}_{t_2}^{rel} = \{13\}$, which is the set of post-release points in t_2. The set of post-release points of the whole program $\mathcal{L}^{rel} = \{7, 13\}$. The set of pre-acquire points of the whole program $\mathcal{L}^{acq} = \{1, 12\}$. Since this program has a single lock $\mathtt{m}$, $\mathcal{L}_{\mathtt{m}}^{rel} = \{7, 13\}$ and $\mathcal{L}_{\mathtt{m}}^{acq} = \{1, 12\}$.

2.2 Interleaving Semantics

Let us fix a program $P = (\mathcal{T}, \mathcal{L}, \mathcal{V}, \mathcal{M})$. We define the standard interleaving semantics of a program using a labeled transition system $\langle \mathcal{S}, s_{ent}, TR^{\mathsf{s}} \rangle$, where $\mathcal{S}$ is the set of *states*, $s_{ent} \in \mathcal{S}$ is the *initial state*, and $TR^{\mathsf{s}} \subseteq \mathcal{S} \times \mathcal{T} \times \mathcal{S}$ is a transition relation, as defined below.

States. A *state* $s \in \mathcal{S}$ is a tuple $\langle pc, \mu, \phi \rangle$, where $pc \in PC \stackrel{\text{def}}{=} \mathcal{T} \to \mathcal{L}$ records the *program counter* (or location) of every thread, $\mu \in LM \stackrel{\text{def}}{=} \mathcal{M} \rightharpoonup \mathcal{T}$ is a *lock map* which associates every lock to the thread that holds it (if such a thread exists), and $\phi \in Env \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{V}$ is an *environment*, mapping variables to their values.

Initial State. We refer to the state $s_{ent} = \langle \lambda t.\, ent_t, \bot, \lambda x.\, 0 \rangle$ as the initial state. In s_{ent} every thread is at its entry program location, no thread holds a lock, and all the variables are initialized to zero.

Transition Relation. The transition relation $TR^{\mathsf{s}}_P \subseteq \mathcal{S} \times \mathcal{T} \times \mathcal{S}$ captures the interleaving semantics of the program P. A transition $\tau = \langle s, t, s' \rangle$, also denoted by $s \Rightarrow_t s'$, says that thread t can execute a command which transforms (the *source*) state s to (the *target*) state s'. As such, the transition relation is the set of all possible transitions generated by its commands, that is, $TR^{\mathsf{s}}_P = \bigcup_{c \in cmd(P)} TR^{\mathsf{s}}_c$. In these transitions, one thread executes a command and changes its program counter accordingly, while all other threads remain stationary.

Intuitively, the semantics of the program commands are as follows. An assignment $x := e$ command updates the value of the variables according to the (possibly non-deterministic) expression e. An `assume`(b) command generates transitions only from states in which the (deterministic) boolean interpretation of the condition b is *true*. An `acquire(m)` command executed by thread t sets $\mu(\mathtt{m}) = t$, provided the lock `m` is not held by any other thread. A `release(m)` command executed by thread t sets $\mu(\mathtt{m}) = _$ provided t holds `m`. A thread attempting to release a lock that it does not own gets blocked.

For a transition $\tau = \langle pc, \mu, \phi \rangle \Rightarrow_t \langle pc', \mu', \phi' \rangle \in TR^{\mathsf{s}}_P$, we denote by $tid(()\tau) = t$ the thread that *executes* the transition, and by $c(\tau)$ the (unique) command $c \in cmd(P)$, such that $\langle pc(t), c, pc'(t) \rangle \in inst_t$, which it executes.

The formal semantic definitions of the commands are given below.

- *Assignment.* We denote the set of valuations of a (possibly non-deterministic) expression e, in an environment ϕ, by $[\![e]\!]\phi \subseteq \mathbb{V}$. We define the meaning of assignments as a function mapping an input environment to a set of environments.

$$[\![x := e]\!]_{\mathsf{s}} \colon Env \to \mathbb{P}(Env) \stackrel{\text{def}}{=} \lambda\phi.\, \{\phi[x \mapsto v] \mid v \in [\![e]\!]\phi\}$$

- *Assume.* We denote the (deterministic) boolean interpretation of a boolean condition b, in an environment ϕ, by $[\![b]\!]\phi \in \{true, false\}$. We define the meaning of assume commands as a function mapping an input environment to a set of environments.

$$[\![\texttt{assume}(b)]\!]_{\mathtt{s}} \colon Env \to \mathbb{P}(Env) \stackrel{\text{def}}{=} \lambda\phi. \begin{cases} \{\phi\} & [\![b]\!]\phi = true \\ \emptyset & \text{otherwise} \end{cases}$$

  The set of transitions generated from an assume or an assignment command c is:

$$TR^{\mathtt{s}}_c = \{\langle pc, \mu, \phi\rangle \Rightarrow_t \langle pc[t \mapsto n'], \mu, \phi'\rangle \mid \langle pc(t), c, n'\rangle \in inst_t \wedge \phi' \in [\![c]\!]_{\mathtt{s}}\phi\}$$

- *Acquire.* An $\texttt{acquire(m)}$ command executed by thread t assigns lock $\texttt{m}$ to a thread t by modifying the lock map μ, provided the lock is not held by any other thread, thus making t the *owner* of the lock. The set of transitions pertaining to an acquire command $c = \texttt{acquire(m)}$ is

$$TR^{\mathtt{s}}_c = \{\langle pc, \mu, \phi\rangle \Rightarrow_t \langle pc[t \mapsto n'], \mu[\texttt{m} \mapsto t], \phi\rangle \mid \langle pc(t), c, n'\rangle \in inst_t \wedge \mu(\texttt{m}) = _\}$$

- *Release.* A $\texttt{release(m)}$ command executed by thread t makes lock $\texttt{m}$ available by changing μ to be undefined for $\texttt{m}$, provided t owns $\texttt{m}$. A thread attempting to release a lock that it does not own gets stuck.[1] The set of transitions pertaining to a release command $c = \texttt{release(m)}$ is

$$TR^{\mathtt{s}}_c = \{\langle pc, \mu, \phi\rangle \Rightarrow_t \langle pc[t \mapsto n'], \mu[\texttt{m} \mapsto _], \phi\rangle \mid \langle pc(t), c, n'\rangle \in inst_t \wedge \mu(\texttt{m}) = t\}$$

Transition Relation. The transition relation of the program P, denoted by $TR^{\mathtt{s}}_P$, is the set of all possible transitions generated by its commands. Formally,

$$TR^{\mathtt{s}}_P = \bigcup_{c \in cmd(P)} TR^{\mathtt{s}}_c\ .$$

Executions. An execution π of the concurrent program P is a finite sequence of transitions coming from its transition relation, such that s_{ent} is the source of transition π_0 and the source

[1] The decision to block a thread releasing a lock it does not own was made to simplify the semantics. Our results hold even if this action would have aborted the program instead.

state of every transition π_i, for $0 < i < |\pi|$, is the target state of transition π_{i-1}. Where convenient, we also write executions as sequences of states interleaved with thread identifiers: $\pi = s_0 \Rightarrow_{t_1} s_1 \Rightarrow_{t_2} \ldots \Rightarrow_{t_n} s_n$. Figure 2.4 shows an execution of the program in Figure 2.3 in the standard semantics.

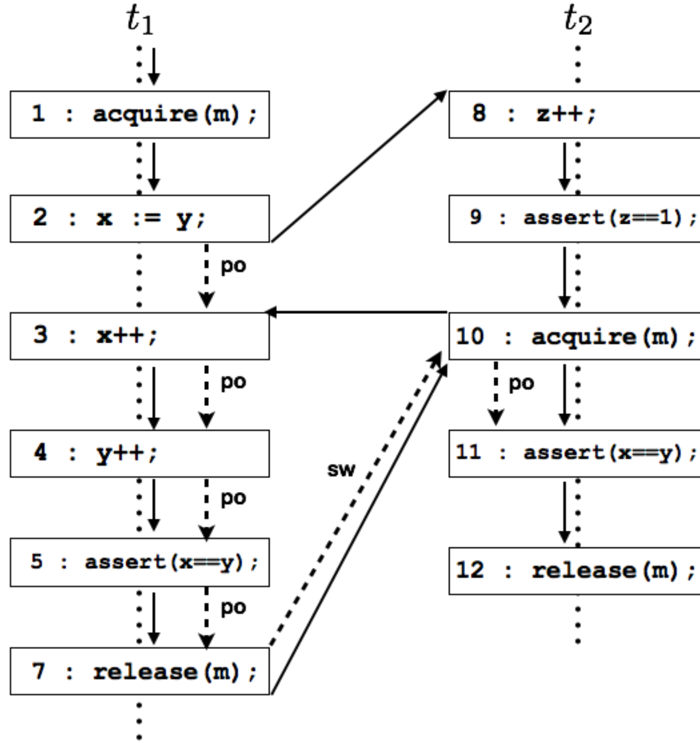

Figure 2.4: A typical execution of the program in Fig. 1.1 with two threads, in accordance to the standard interleaving semantics. The execution is an interleaving of instructions from the different threads, and is indicated by the solid arrows. The dashed arrows indicate a couple of partial-orders induced by this execution: the *program-order* (po), which relates successive instructions from the same thread, and *synchronizes-with* (sw), which relates successive synchronization operations. The reflexive and transitive closure of these partial orders forms the *happens-before* (hb) relation, induced by this particular execution. The write of `x` at location 2 in thread t_1, and its subsequent read at location 11 in thread t_2, can be seen to be related by hb.

Collecting semantics. The collecting semantics of a program P, according to the standard semantics, is the set of reachable states starting from the initial state s_{ent}. We define

$$Reachable^{\mathbf{s}}(P) = \{s \mid (s_{ent} \Rightarrow_{t_1} \ldots \Rightarrow_{t_n} s) \text{ is an execution of } P\}$$

to be the set of reachable states of the program P. Then, $Reachable^{\mathsf{s}}(P)$ can be seen to be equivalent to the least fixpoint of the functional $\mathcal{F}^{\mathsf{s}}$, where

$$\mathcal{F}^{\mathsf{s}} = \lambda X.\, \{s_{ent}\} \cup \{s' \mid \exists s \in X, t \in \mathcal{T} :\ s \Rightarrow_t s'\}$$

where $X \in \mathbb{P}(\mathcal{S})$. In other words, if $[\![P]\!]^{\mathsf{s}} = LFP\, \mathcal{F}^{\mathsf{s}}$, then

$$[\![P]\!]^{\mathsf{s}} = Reachable^{\mathsf{s}}(P) \tag{2.1}$$

2.3 Data Races and the Happens-Before Relation

Now that we have formally defined the standard interleaving semantics, we are in a position to formally define what constitutes a data race. A standard way to formalize the notion of *data race freedom* (DRF), is to use the *happens before* [50] relation induced by executions.

For a given execution of program P in the standard interleaving semantics, the happens-before relation is defined as the reflexive and transitive closure of the *program-order* and *synchronizes-with* relations, formalized below.

Definition 2.1 (Program order) *Let π be an execution of P. The transition π_i is related to the transition π_j according to the* program-order *relation in π, denoted by $\pi_i \xrightarrow{po}_\pi \pi_j$, if $j = \min\{k \mid i < k < |\pi| \wedge t(\pi_k) = t(\pi_i)\}$, i.e., π_i and π_j are successive executions of commands by the same thread.*[1]

Definition 2.2 (Synchronize-with) *Let π be an execution of P. The transition π_i is related to the transition π_j according to* synchronizes-with *relation in π, denoted by $\pi_i \xrightarrow{sw}_\pi \pi_j$, if $c(\pi_i) = \texttt{release(m)}$ for some lock $\texttt{m}$, and $j = \min\{k \mid i < k < |\pi| \wedge c(\pi_k) = \texttt{acquire(m)}\ \}$, i.e., π_i and π_j are successive release and acquire commands of the same lock in the execution.*

Definition 2.3 (Happens before) *The* happens-before *relation pertaining to an execution π of P, denoted by $\cdot \xrightarrow{hb}_\pi \cdot$, is the reflexive and transitive closure of the union of the program-order and synchronizes-with relations induced by π.*

Note that transitions executed by the same thread are always related by program-order, and are thus always related according to the happens-before relation.

[1] Strictly speaking, the various relations we define are between indices $\{0, \ldots, |\pi| - 1\}$ of an execution, and not transitions, so we should have written, e.g., $i \xrightarrow{po}_\pi j$ instead of $\pi_i \xrightarrow{po}_\pi \pi_j$. We use the informal latter notation, for readability.

Definition 2.4 (Data Race) *Let π be an execution of P. Transitions π_i and π_j constitute a racing pair, or a data-race, if the following conditions are satisfied:*

(i) $c(\pi_i)$ and $c(\pi_j)$ both access the variable x, with at least one of the accesses being a write to x, and

(ii) neither $\pi_i \xrightarrow{hb}_\pi \pi_j$ nor $\pi_j \xrightarrow{hb}_\pi \pi_i$ holds.

To illustrate these definitions, consider the execution of the program of Figure 2.3 as shown in Figure 2.4. The transitions where t_1 executes $\texttt{x} := \texttt{y}$ and the one where t_1 executes $\texttt{x}++$ are related by program-order. Likewise, some of the other program-order related transitions are labeled 'po' in the figure. The transition where t_1 releases the lock `m`, and the subsequent transition where t_2 acquires `m`, are related by the synchronizes-with relation (and is hence marked 'sw'). There is a happens-before path, denoted by the path comprising the dashed arrows in Figure 2.4, between the writes to `x` by t_1, and the subsequent read of `x` by t_2. This path is made up of pairs of transitions which are either related by program-order, or synchronizes-with. Note that even though the instruction $\texttt{x} := \texttt{y}$ is executed by t_1 before t_2 executes $\texttt{z}++$ in the execution in Figure 2.4, these two instructions are *not* related by happens-before. Consider, for a moment, if t_2 did *not* have the `acquire(m)` instruction. Then, the transitions made by t_1 could never be happens-before related to the ones in t_2 (due to the lack of sw relations). In particular, the writes to `x` by t_1 would *not* be happens-before ordered with the read of `x` in t_2, and we would have a data race in the execution.

In Part I of the thesis, we focus on efficient analyses for the class of race free programs.

Chapter 3

The Thread-Local Semantics *L-DRF*

In this chapter, we formally define a novel *thread-local* semantics for the class of data race free programs, which we refer to as the *L-DRF* semantics. The *L-DRF* semantics paves the way towards devising efficient "thread-local" data flow analyses for race free concurrent programs. Like the standard interleaving semantics in the earlier chapter, we present the *L-DRF* semantics of a program P as a labeled transition system. We then prove that the *L-DRF* semantics is sound and complete with respect to the standard semantics presented in Section 2.2, in the sense that for each trace of P in the standard semantics, there is an "equivalent" trace in the *L-DRF* semantics, and vice versa.

3.1 Overview

Our thread-local semantics, like the standard one defined in Section 2.2, is based on the interleaving of transitions made by different threads, and the use of a lock map to coordinate the use of locks. However, *unlike* the standard semantics, where the threads share access to a single *global* environment, in the *L-DRF* semantics, every thread has its own *local* environment which it uses to evaluate conditions and perform assignments.

Threads exchange information through *release buffers*: every post-release point $n \in \mathcal{L}_t^{rel}$ [1]of a thread t is associated with a *buffer*, $\Lambda(n)$, which records a snapshot of t's local environment the last time t ended up at the program point n. Recall that this happens right after t executes the instruction $\langle n_s, \texttt{release}(\texttt{m}), n\rangle \in inst_t$. When a thread t subsequently acquires the lock `m`, it updates its local environment using the snapshots stored in all the buffers pertaining to the release of `m`.

[1]Recall that $\mathcal{L}_t^{rel}$ is the set of all post-release points in the thread t.

To ensure that t updates its environment such that the value of every variable is up-to-date, every thread maintains its own *version map* $\nu : \mathcal{V} \to \mathbb{N}$, which associates a counter to each variable. A thread increments $\nu(x)$ whenever it writes to x. Along any execution, the version $\nu(x)$, for $x \in \mathcal{V}$, in the version map ν of thread t, associates a unique prior write with this particular valuation of x. It also reflects the total number of write accesses made (across threads) to x to obtain the value of x stored in the map. A thread stores both its local environment and ν in the buffer after releasing a lock $\mathtt{m}$. When a thread subsequently acquires lock $\mathtt{m}$, it copies from the the release buffers at $\mathcal{L}^{rel}_{\mathtt{m}}$, the most up-to-date value (according to the version numbers) of every variable. We prove that for data race free programs, there can be only one such value.

3.2 The thread-local *L-DRF* semantics

Let $P = (\mathcal{T}, \mathcal{L}, \mathcal{V}, \mathcal{M})$ be a race free concurrent program. As in Section 2.2, we define the *L-DRF* semantics of P in terms of a labeled transition system $(\Sigma, \sigma_{ent}, TR_P)$.

States. A *state* $\sigma \in \Sigma$ in the *L-DRF* semantics is a tuple $\langle pc, \mu, \Theta, \Lambda \rangle$. The functions pc and μ have the same role as in the standard semantics; that is, they record the program counter of every thread and the ownership of locks, respectively. A *versioned environment*

$$ve = \langle \phi, \nu \rangle \in VE = Env \times (\mathcal{V} \to \mathbb{N})$$

is a pair comprising an environment ϕ and a version map ν. The environment $ve \cdot \phi$ is a valuation of the program variables. The version map $ve \cdot \nu$ assigns a "version count" to each variable. The version count of a variable x is incremented by a thread t whenever it writes to x. The version counts, as we explain in more detail later in this chapter, are used by a thread to ensure it has the most up-to-date value and version of the variables in its local state. The local environment map $\Theta : \mathcal{T} \to VE$ maps every thread to its local versioned environment and $\Lambda : \mathcal{L}^{rel} \to VE$ records the snapshots of versioned environments stored in buffers associated with post-release points.

Initial State. The initial state is defined to be

$$\sigma_{ent} = \langle \lambda t.\, ent_t, \bot, \lambda t.\, ve_{ent}, \bot \rangle$$

where $ve_{ent} = \langle \lambda x.0, \lambda x.0 \rangle$. In σ_{ent}, every thread is at its entry program location, no thread holds a lock, all the thread-local versioned environments have all the variables and versions initialized to 0, and the contents of each release buffer is undefined.

Transition Relation. The transition relation $TR_P \subseteq \Sigma \times \mathcal{T} \times \Sigma$ captures the interleaving nature of the *L-DRF* semantics of P. A transition $\tau = \langle \sigma, t, \sigma' \rangle$, also denoted by $\sigma \Rightarrow_t \sigma'$, says that thread t can execute a command which transforms state $\sigma \in \Sigma$ to state $\sigma' \in \Sigma$. As is the case in the standard semantics, in these transitions, one thread executes a command and changes its program counter accordingly, while all other threads remain stationary.

The formal semantics definitions of each command is given below.

- *Assignment.* We define the meaning of an assignment command as a function from a versioned environment to a set of versioned environments (in order to account for the non-determinism). The function makes use of the standard interpretation of the expression on the right-hand side over the environment component in the versioned environment. In addition, assignments increment the version of a variable being assigned to. Formally,

$$[\![x := e]\!] : VE \to \mathbb{P}(VE) = \lambda \langle \phi, \nu \rangle \, . \, \{\langle \phi[x \mapsto v], \nu[x \mapsto \nu(x) + 1] \rangle \mid v \in [\![e]\!]\phi\}$$

  where $[\![e]\!]\phi$ denotes the value of the (possibly non-deterministic) expression e in ϕ.

- *Assume.* As with assignments, we define the meaning of an assignment command as a function from a versioned environment to a set of versioned environments (in order to account for the fact that an input environment may not satisfy the boolean condition). This function also makes use of the standard (deterministic) interpretation of the boolean condition `b` over the environment component in the versioned environment. Assume commands do not alter either the valuation or the versions of any variables.

$$[\![\texttt{assume}(b)]\!] : VE \to \mathbb{P}(VE) = \lambda \langle \phi, \nu \rangle \, . \begin{cases} \{\langle \phi, \nu \rangle\} & [\![b]\!]\phi = true \\ \emptyset & \text{otherwise} \end{cases}$$

  where $[\![b]\!]\phi$ denotes the value of the boolean expression b in ϕ.

  If c is either an `assume()` or an assignment command, then the set of transitions TR_c generated by c is given by:

$$TR_c = \{\langle pc, \mu, \Theta, \Lambda \rangle \Rightarrow_t \langle pc[t \mapsto n'], \mu, \Theta[t \mapsto ve'], \Lambda \rangle \mid \langle pc(t), c, n' \rangle \in inst_t \wedge ve' \in [\![c]\!](\Theta(t))\}$$

  Note that for these commands each thread t only accesses and modifies its *own* local versioned environment.

- *Acquire.* An `acquire(m)` command executed by a thread t has the same effect on the lock map component as in the standard semantics (see Section 2.2). In addition, it updates the versioned environment $\Theta(t)$ based on the contents of the *relevant* release buffers. The release buffers relevant to a thread when it acquires `m` are the ones at $\mathcal{L}^{rel}_{\mathtt{m}}$. Conversely, for any post-release point $n \in \mathcal{L}^{rel}_{\mathtt{m}}$, we use the symbol $\mathcal{G}(n)$ to denote the set of pre-acquire points which can observe the buffer $\Lambda(n)$. In other words, $\mathcal{G}(n)$ are the set of pre-acquire points for which $\Lambda(n)$ is relevant. For our purposes, for any $n \in \mathcal{L}^{rel}_{\mathtt{m}}$, $\mathcal{G}(n) = \mathcal{L}^{acq}_{\mathtt{m}}$ [1]. The auxiliary function *updEnv* is used to update the value of each $x \in \mathcal{V}$ (along with its version) in $\Theta(t)$, by taking its value from a snapshot stored at a relevant buffer which has the highest version of x, if the latter version is higher than $\Theta(t) \cdot \nu(x)$. If the version of x is highest in $\Theta(t) \cdot \nu(x)$, then t simply retains this value. Finding the most up-to-date snapshot for x (or determining that $\Theta(t) \cdot \nu(x)$ is the highest) is the role of the auxiliary function $take_x$. It takes as input $\Theta(t)$, as well as the versioned environments in the relevant release buffers, and returns the versioned environments for which the version associated with x is the highest. We separately prove that, along any execution, if there is a state in the *L-DRF* semantics σ with two component versioned environments (in thread local states or buffers) ve_1 and ve_2 such that $ve_1 \cdot \nu(x) = ve_2 \cdot \nu(x)$, then $ve_1\phi(x) = ve_2\phi(x)$. The set of transitions pertaining to an acquire command $c = \mathtt{acquire(m)}$ is

$$TR_c = \{\langle pc, \mu, \Theta, \Lambda\rangle \Rightarrow_t \langle pc[t \mapsto n'], \mu[\mathtt{m} \mapsto t], \Theta[t \mapsto ve], \Lambda\rangle \mid \langle pc(t), c, n'\rangle \in inst_t \wedge \mu(\mathtt{m}) = _ \wedge ve \in updEnv(\Theta(t), \Lambda)\}$$

where $updEnv : (VE \times (\mathcal{L}^{rel} \to VE)) \to \mathbb{P}(VE)$ is given by

$$updEnv(ve, \Lambda) = \{ve' \mid \bigwedge_{x\in\mathcal{V}} \exists ve_x \in take_x(Y) : ve' \cdot \phi(x) = ve_x \cdot \phi(x) \bigwedge ve' \cdot \nu(x) = ve_x \cdot \nu(x)\}$$

with the set Y and the function $take_x$ being defined as

$$Y = \{ve\} \cup \{\Lambda(\bar{n}) \mid \bar{n} \in \mathcal{L}^{rel}_{\mathtt{m}} \bigwedge pc(t) \in \mathcal{G}(\bar{n})\}$$

and

$$take_x \stackrel{\text{def}}{=} \lambda Y \in \mathbb{P}(VE). \{\langle \phi, \nu\rangle \in Y \mid \nu(x) = \max\{\nu'(x) \mid \langle \phi', \nu'\rangle \in Y\}\} .$$

Given a set of versioned environments, $take_x$ returns the set of versioned environments where the version of x is the highest. The function *updEnv* is given access to the local versioned environment of t, namely $\Theta(t)$, and the buffers Λ. For each variable x, the

[1] We outline, later on, that it is possible to refine the set $\mathcal{G}_n$ such that the resulting *L-DRF* semantics is still equivalent to the interleaving semantics, but the abstract analyses derived from such an *L-DRF* semantics has increased precision

function then extracts the versioned environment containing the highest version of x, in either $\Theta(t)$ or one of the relevant buffers. Once it has this set of versioned environments, it then proceeds to "mix" them.

As an example, consider again the execution of the program in Figure 1.2, presented in Figure 3.1. When thread t_2 is executes the `acquire(m)` instruction, the condition of the relevant buffers and the thread local state of t_2 is shown in Figure 3.2. The figure also outlines the operation of the functions $take_x$, $take_y$ and $take_y$, and finally the operation of the function *updEnv*.

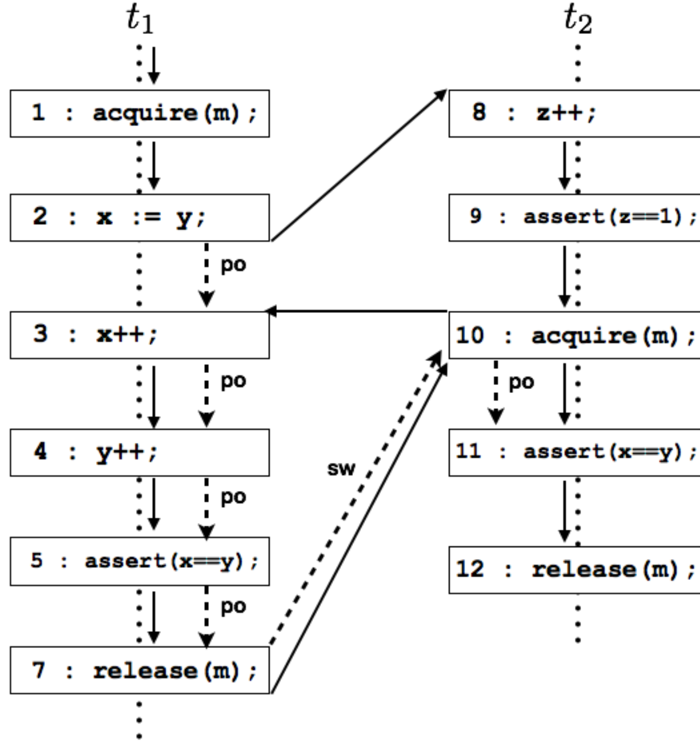

Figure 3.1: An execution of program Figure 1.2 with 2 threads.

- *Release.* A `release(m)` command executed by a thread t has the same effect on the lock map component of the state in the *L-DRF* semantics that it has in the standard semantics (See Section 2.2). In addition, it stores the local versioned environment of t, $\Theta(t)$, in the buffer associated with the post-release point of the executed `release(m)` instruction.

  The set of transitions pertaining to a release command $c = \texttt{release(m)}$ is

$$TR_c = \{\langle pc, \mu, \Theta, \Lambda\rangle \Rightarrow_t \langle pc[t \mapsto n'], \mu[\texttt{m} \mapsto _], \Theta, \Lambda[n' \mapsto \Theta(t)]\rangle \mid \langle pc(t), c, n'\rangle \in inst_t \wedge \mu(\texttt{m}) = t\}$$

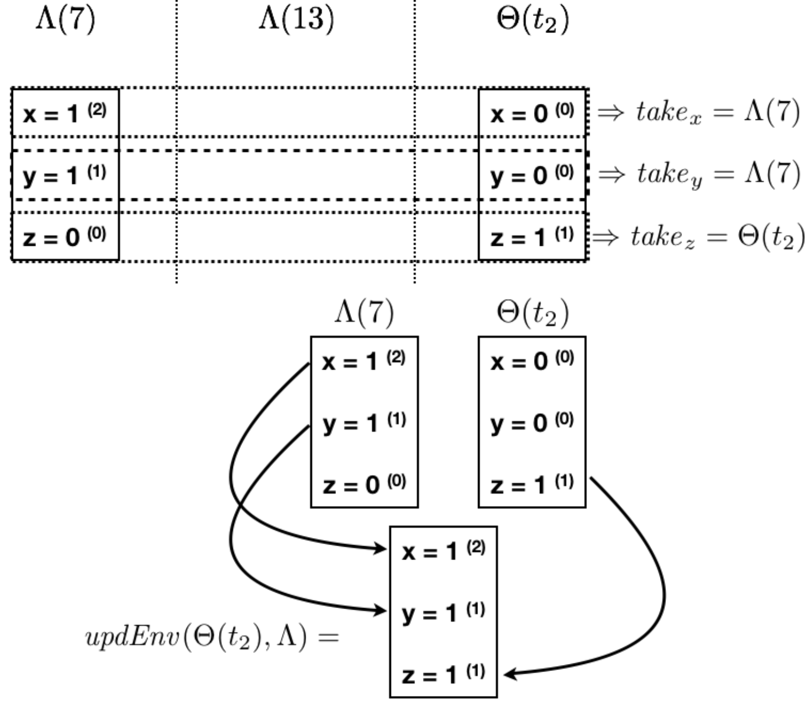

Figure 3.2: Operation of the functions $take_x$, $take_y$, $take_z$, and *updEnv* when t_2 acquires `m`. The superscripts indicate the versions.

Transition Relation. The transition relation TR_P of program P according to the *L-DRF* semantics, is the set of all possible transitions generated by its commands. Formally,

$$TR_P = \bigcup_{c \in cmd(P)} TR_c$$

Executions. An execution $\hat{\pi}$ of the concurrent program P in the *L-DRF* semantics is a finite sequence of transitions coming from its transition relation TR_P, such that σ_{ent} is the source of transition $\hat{\pi}_0$ and the source state of every transition $\hat{\pi}_i$, for $0 < i < |\hat{\pi}|$, is the target state of transition $\hat{\pi}_{i-1}$. Where convenient, we also write executions as sequences of states interleaved with thread identifiers: $\hat{\pi} = \sigma_0 \Rightarrow_{t_1} \sigma_1 \Rightarrow_{t_2} \ldots \Rightarrow_{t_n} \sigma_n$. where $\sigma_i = \langle pc_i, \mu_i, \Theta_i, \Lambda_i \rangle$.

Collecting Semantics. The collecting semantics of a program P, according to the *L-DRF* semantics, is the set of reachable states starting from the initial state σ_{ent}, which we denote as *Reachable*(P).

$$Reachable(P) = \{\sigma \mid (\sigma_{ent} \Rightarrow_{t_1} \ldots \Rightarrow_{t_n} \sigma) \text{ is a valid trace of } P \text{ in L-DRF}\}$$

Then, *Reachable*(P) can be seen to be the least fixpoint of the functional $\mathcal{F}$, where

$$\mathcal{F} = \lambda X. \{\sigma_{ent}\} \cup \{\sigma' \mid \exists \sigma \in X, \exists t \in \mathcal{T} : \sigma \Rightarrow_t \sigma' \in TR_P\}$$

In other words, if $[\![P]\!] = LFP\, \mathcal{F}$, then

$$Reachable(P) = [\![P]\!] \tag{3.1}$$

3.3 Soundness and Completeness of *L-DRF*

In this section, we show that for the class of data race free programs, the thread local semantics *L-DRF* is sound and complete with respect to the standard interleaving semantics. Intuitively, the *L-DRF* and the standard semantics are "equivalent" since for each execution of a program P in the standard semantics, one can find a corresponding execution in the *L-DRF* semantics which coincides with the values read from the variables. Likewise, every execution of program P in the *L-DRF* semantics has a corresponding execution in the standard semantics. As in earlier chapters, we fix a program $P = (\mathcal{T}, \mathcal{L}, \mathcal{V}, \mathcal{M})$.

To formalize the above claim, we define a function which extracts a state in the interleaving semantics from a state in the *L-DRF* semantics.

Definition 3.1 (Extraction Function χ)

$$\chi : \Sigma \to \mathcal{S} = \lambda \langle pc, \mu, \Theta, \Lambda \rangle \,.\, \left\langle pc, \mu, \lambda x. \Theta \left(\operatorname*{argmax}_{t \in \mathcal{T}} \Theta\,(t)\, \nu(x) \right) \phi(x) \right\rangle$$

The function χ preserves the values of the program counters and the lock map, while it takes the value of every variable x from the thread which has the maximal version count for x in its local environment. χ is well-defined for *admissible* states where if $\Theta(t) \cdot \nu(x) = \Theta(t') \cdot \nu(x)$, then $\Theta(t) \cdot \phi(x) = \Theta(t') \cdot \phi(x)$. We denote the set of admissible states by $\tilde{\Sigma}$. As we prove later in Lemma 3.4, the *L-DRF* semantics only produces admissible states. The function χ can be extended to executions in the *L-DRF* semantics by applying it to each state in the execution. Formally, for any execution $\hat{\pi} = \sigma_{ent} \Rightarrow_{t_1} \ldots \Rightarrow_{t_n} \sigma_n$ of program P in the *L-DRF* semantics:

$$\chi\,(\hat{\pi}) \stackrel{\text{def}}{=} \chi\,(\sigma_{ent}) \Rightarrow_{t_1} \ldots \Rightarrow_{t_n} \chi\,(\sigma_n)$$

We prove in Theorem 3.2 that the sequence $\chi\,(\hat{\pi})$ is indeed a valid trace of P in the interleaving semantics. The following theorems state our soundness and completeness results.

Theorem 3.1 ***Soundness.*** *For any execution π of P in the interleaving semantics, there exists a trace $\hat{\pi}$ of P in the L-DRF semantics such that $\chi(\hat{\pi}) = \pi$.*

Theorem 3.2 ***Completeness.*** *For any trace $\hat{\pi}$ of P in the L-DRF semantics, $\chi(\hat{\pi})$ is a trace in the interleaving semantics of P.*

In order to prove Theorem 3.1 and Theorem 3.2, we need to establish a few intermediate results.

Lemma 3.1 *In any execution $\hat{\pi}$ in the L-DRF semantics of P, the version of any variable $x \in \mathcal{V}$ in any component versioned environment of any state σ in $\hat{\pi}$ is bounded by the total number of writes to x preceeding it.*

Proof: The only command which can increment the version of variable x in a versioned environment is a write to x, of the form $x := e$. The other commands (`assume`, `acquire` and `release`) only make copies of existing version counts. If there are n such write commands in $\hat{\pi}$, and the initial version count of x is 0 in all the component versioned environments of the initial state σ_{ent}, the version of x, in any component versioned environment of any state σ in $\hat{\pi}$ can be at most n. □

Lemma 3.2 *Let $\hat{\pi} = \langle pc_0, \mu_0, \Theta_0, \Lambda_0 \rangle \Rightarrow_{t_1} \ldots \Rightarrow_{t_N} \langle pc_N, \mu_N, \Theta_N, \Lambda_N \rangle$ be a trace in the L-DRF semantics of program P. Let $\tau_j = \left\langle pc_{j-1}, \mu_{j-1}, \Theta_{j-1}, \Lambda_{j-1} \right\rangle \Rightarrow_{t_j} \left\langle pc_j, \mu_j, \Theta_j, \Lambda_j \right\rangle$ be a transition in $\hat{\pi}$ which contains an access (read or write) to the variable x. Let $\tau_i = \left\langle pc_{i-1}, \mu_{i-1}, \Theta_{i-1}, \Lambda_{i-1} \right\rangle \Rightarrow_{t_i} \langle pc_i, \mu_i, \Theta_i, \Lambda_i \rangle$ be the last transition, prior to τ_j, which contains an assignment to x. Then,*

$$\Theta_{j-1}(t_j) \cdot \nu(x) \geq \Theta_i(t_i) \cdot \nu(x)$$

In other words, the version of x in $\Theta(t_j)$ is no less than the version of x in the local state of t_i post the write at τ_i.

Proof: Figure 3.3 provides a pictorial description of the situation we are considering. Since τ_i and τ_j both contain accesses to the variable x, and since the program P is assumed to be free from races, we have $\tau_i \xrightarrow{hb}_{\hat{\pi}} \tau_j$ (indicated by the dotted arrows in Figure 3.3). Note that the notion of a happens-before path, which we defined for the interleaving semantics, can be naturally extended to *L-DRF* traces. The sequence of transitions in $\hat{\pi}$ can also be viewed as an interleaved execution, and the resulting happens-before path in $\hat{\pi}$ contains the same sequence of transitions as the happens-before path in the interleaved execution.

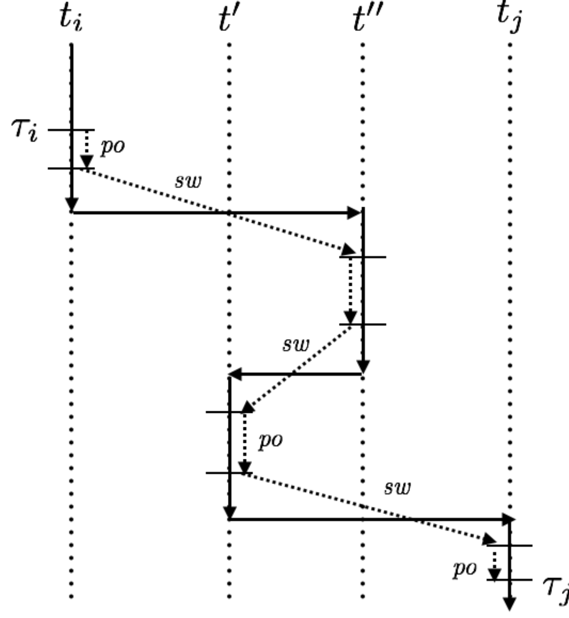

Figure 3.3: A typical execution of a program P in the *L-DRF* semantics. The solid arrows represent the interleaved execution of the instructions from different threads. The dotted arrows denote the happens-before path induced by this execution. The figure marks the sections of the happens-before path which are program-order related (po), and the transitions related by synchronizes-with(sw).

<u>Claim</u>: For each post-state σ_k of a transition τ_k, other than τ_j, in some happens-before path ρ between τ_i and τ_j, we must have $\Theta_k(t_k) \cdot \nu(x) \geq \Theta_i(t_i) \cdot \nu(x)$. We prove the claim using induction on the position n of a transition in ρ.

Base Case. If $n = 1$, then τ_i and τ_j must be related by program order, which implies $t_i = t_j$ and $j = i + 1$. This implies the absence of any intervening transitions which can alter the version of x in the local state of t_i, post τ_i, until τ_j. Thus, the result holds for $n = 1$.

Inductive Case. Assume that the hypothesis holds for all transitions at positions $k \leq n$ in ρ. We now consider a transition at index $n + 1$. We denote this transition as

$$\tau_v = \left\langle pc_{v-1}, \mu_{v-1}, \Theta_{v-1}, \Lambda_{v-1} \right\rangle \Rightarrow_{t_v} \langle pc_v, \mu_v, \Theta_v, \Lambda_v \rangle$$

and the n-th transition as

$$\tau_u = \left\langle pc_{u-1}, \mu_{u-1}, \Theta_{u-1}, \Lambda_{u-1} \right\rangle \Rightarrow_{t_u} \langle pc_u, \mu_u, \Theta_u, \Lambda_u \rangle$$

There are two possible cases here. Either $\tau_u \xrightarrow{po}_{\hat{\pi}} \tau_v$, and consequently $t_u = t_v$. This implies

the absence of any intervening transitions which can alter the version of x in the local state of t_u, post τ_u, until τ_v. Thus, the result holds. On the other hand, if $\tau_u \xrightarrow{sw}_{\hat{\pi}} \tau_v$, then $c(\tau_u)$ must be the `release` of some lock `m`, and $c(\tau_v)$ must be the `acquire` of `m`. By the *L-DRF* semantics of `acquire`, thread t_v will observe the buffer associated with the `release` command of τ_u. Consequently,

$$\begin{aligned} &\Theta_v(t_v) \cdot \nu(x) \geq \Theta_u(t_u) \cdot \nu(x) && \text{by the semantics of the acquire command} \\ \text{and } &\Theta_u(t_u) \cdot \nu(x) \geq \Theta_i(t_i) \cdot \nu(x) && \text{by the inductive hypothesis} \\ \implies &\Theta_v(t_v) \cdot \nu(x) \geq \Theta_i(t_i) \cdot \nu(x) \end{aligned}$$

Thus, the hypothesis holds in this case as well. This proves the claim.
Returning to the proof of the lemma, let $\tau' = \langle pc', \mu', \Theta', \Lambda' \rangle$ be the last transition, before τ_j, in the happens-before path between τ_i and τ_j. Since τ_j is not a synchronization operation, it must be the case that $\tau' \xrightarrow{po}_{\hat{\pi}} \tau_j$, which implies $t' = t_j$. By the earlier claim, $\Theta'(t_j) \cdot \nu(x) \geq \Theta_i(t_i) \cdot \nu(x)$, and the absence of any intervening instructions, between τ' and τ_j, which can alter the version of x, we infer that $\Theta_{j-1}(t_j) \cdot \nu(x) \geq \Theta_i(t_i) \cdot \nu(x)$. □

Lemma 3.3 *Let* $\hat{\pi} = \langle pc_0, \mu_0, \Theta_0, \Lambda_0 \rangle \Rightarrow_{t_1} \ldots \Rightarrow_{t_N} \langle pc_N, \mu_N, \Theta_N, \Lambda_N \rangle$ *be an execution in the L-DRF semantics of a program* P*, and let* $\tau_i = \langle pc_{i-1}, \mu_{i-1}, \Theta_{i-1}, \Lambda_{i-1} \rangle \Rightarrow_{t_i} \langle pc_i, \mu_i, \Theta_i, \Lambda_i \rangle$ *denote the* i*-th transition in the trace. If* $c(\tau_i)$ *is an assignment to* x*, then*

$$\Theta_i(t_i) \cdot \nu(x) = |\{j \mid j \leq i \wedge c(\tau_j) \textit{ is an assignment to } x\}|$$

That is, at the post-state of an assignment to a variable x *by thread* t*, the version of* x *in the local versioned environment of* t *equals the total number of writes made to* x *till that point.*

Proof: We prove this lemma using the following claim. Let $\hat{\pi}$ be an execution of length N in the *L-DRF* semantics of P. If the N-th transition of $\hat{\pi}$ is of the form $x := e$, then

$$\Theta_N(t_N) \cdot \nu(x) = |\{j \mid j \leq N \wedge c(\tau_j) \text{ is an assignment to } x\}|$$

Note that this claim implies the original statement of the lemma, since any prefix of length i of an *L-DRF* execution is also a valid execution of length i in the *L-DRF* semantics. We prove our claim using induction on N.
Base Case. The claim vacuously holds for $N = 0$, since the 0 length *L-DRF* trace does not contain any write to x.

Inductive Case. Assume the claim holds for all $0 \leq N \leq n$. Consider the case when $N = n+1$, and the command associated with $c(\tau_{n+1}$ involves a write to x. The $n+1$ execution $\hat{\pi}$ is of the form

$$\langle pc_0, \mu_0, \Theta_0, \Lambda_0 \rangle \Rightarrow_{t_1} \ldots \Rightarrow_{t_{n+1}} \langle pc_{n+1}, \mu_{n+1}, \Theta_{n+1}, \Lambda_{n+1} \rangle$$

Let the last write to x, prior to $c(\tau_{n+1})$, be in the transition τ_i. Since the program is data race free, $\tau_i \xrightarrow{hb}_{\hat{\pi}} \tau_{n+1}$. By inductive hypothesis,

$$\begin{aligned}\Theta_i(t_i) \cdot \nu(x) &= |\{j \mid j \leq i \wedge c(\tau_j) \text{ is an assignment to } x\}| \\ &= m \text{ (say)}\end{aligned}$$

We now infer the following:

$$\begin{aligned}\Theta_n(t_{n+1}) \cdot \nu(x) &\geq m \quad \text{from Lemma 3.2 and} \\ \Theta_n(t_{n+1}) \cdot \nu(x) &\leq m \quad \text{from Lemma 3.1} \\ \text{Therefore, } \Theta_n(t_{n+1}) \cdot \nu(x) &= m\end{aligned}$$

Since τ_{n+1} increments the version of x in $\Theta_{n+1}(t_{n+1}) \cdot \nu$, we have

$$\begin{aligned}\Theta_{n+1}(t_{n+1}) \cdot \nu(x) &= \Theta_n(t_{n+1}) \cdot \nu(x) + 1 \\ &= m + 1 \\ &= |\{j \mid j \leq i \wedge c(\tau_j) \text{ is an assignment to } x\}| + 1 \\ &= |\{j \mid j \leq n+1 \wedge c(\tau_j) \text{ is an assignment to } x\}|\end{aligned}$$

This completes the proof of the claim, and therefore the lemma. □

Corollary 3.1 *Let $\hat{\pi}$ be a trace in the L-DRF semantics of P, and let $\tau_i = \sigma_{i-1} \Rightarrow_{t_i} \sigma_i$ (the i-th transition in $\hat{\pi}$) contain an access (read or write) to the variable x. Let m be the highest version count of x among all component versioned environments in σ_{i-1}. Then, $\sigma_{i-1}\Theta(t_i) \cdot \nu(x) = m$. In other words, whenever a thread accesses a variable x, the version of x is the highest in its local versioned environment.*

Proof: We prove the result via induction on the length N of the trace $\hat{\pi}$.

Base Case. The result vacuously holds for $N = 0$, since the 0 length trace does not contain any transitions.
Inductive Case. Assume that the result holds for all traces of length $0 \leq N$. Consider a trace $\hat{\pi}'$ of length $n+1$. The $(n+1)$'th transition in this trace is $\tau_{n+1} = \sigma_n \Rightarrow_{t_{n+1}} \sigma_{n+1}$. We case split on $c(\tau_{n+1})$. The result continues to holds if $c(\tau_{n+1})$ is either an `acquire` or a `release` command, since such commands do not involve any variable accesses. If, instead, $c(\tau_{n+1})$ is either an `asssume` or an assignment command, let x be any variable accessed (either read-from, or written-to) in the command. Let τ_i be the last write to x in $\hat{\pi}'$, prior to τ_{n+1}. Since the program is free from races, $\tau_i \xrightarrow{hb}_{\hat{\pi}'} \tau_{n+1}$. Let the total number of writes to x in $\hat{\pi}'$, including τ_i, be w. By Lemma 3.1, for any component versioned environment ve in σ_n,

$$ve \cdot \nu(x) \leq w$$

Thus, w is the highest version count of x in any component versioned environment in σ_n. By Lemma 3.2,

$$\sigma_n \cdot \Theta(t_{n+1}) \cdot \nu(x) \geq w$$

Thus, $\sigma_n \cdot \Theta(t_{n+1}) \cdot \nu(x) = w$, and the result holds for the trace of length $(n+1)$. This proves the lemma. □

Lemma 3.4 *Let $\sigma_{ent} \Rightarrow_{t_1} \ldots \Rightarrow_{t_N} \sigma_N$ be an execution of P in the L-DRF semantics. Then, for any σ_i, with two component versioned environments (in thread local states or buffers) ve_1 and ve_2, if $ve_1\nu(x) = ve_2\nu(x)$, then $ve_1\phi(x) = ve_2\phi(x)$.*

Proof: [Lemma 3.4] We prove the lemma using induction on the length of the trace. Let $\mathbf{P}(N)$ denote the following hypothesis. Let $\sigma_{ent} \Rightarrow_{t_1} \ldots \Rightarrow_{t_N} \sigma_N$ be an execution of P in the *L-DRF* semantics of length $N \geq 0$. Then, for any σ_i, with two component versioned environments (in thread local states or buffers) ve_1 and ve_2 such that $ve_1 \cdot \nu(x) = ve_2 \cdot \nu(x)$, we must have $ve_1 \cdot \phi(x) = ve_2 \cdot \phi(x)$. We outline the inductive arguments.
Base Case. For $N = 0$, the trace contains the single state σ_{ent}. By the definition of σ_{ent}, for any two component thread local states (the buffers are initially empty) ve_1 and ve_2 such that $ve_1 \cdot \nu(x) = ve_2 \cdot \nu(x)$, we have $ve_1 \cdot \phi(x) = ve_2 \cdot \phi(x)$. Thus $\mathbf{P}(0)$ holds.
Inductive Case. Assume $\mathbf{P}$ holds for all executions of length k, where $0 \leq k \leq n$. We show that $\mathbf{P}(n+1)$ holds. Let $\hat{\pi}$ be an execution of P in the *L-DRF* semantics of length $n+1$, and let $\tau_{n+1} = \sigma_n \Rightarrow_{t_{n+1}} \sigma_{n+1}$ be the last transition in $\hat{\pi}'$. We case-split on $c(\tau_{n+1})$.
$\mathbf{P}(n+1)$ trivially holds if $c(\tau_{n+1})$ is either an `assume` or a `release`, since these commands do

not alter any versions or values.

If $c(\tau_{n+1}) = \texttt{acquire(m)}$, then t_{n+1} updates its local versioned environment based on its local versioned environment, and the versioned environments at relevant buffers. By the inductive hypothesis, for two component versioned environments ve_1 and ve_2 of σ_n, such that $ve_1 \cdot \nu(x) = ve_2 \cdot \nu(x)$, we have $ve_1 \cdot \phi(x) = ve_2 \cdot \phi(x)$. By the semantics of the `acquire`, t_{n+1} copies over the version *and* the valuation of x from one such ve (including, possibly, t_{n+1}'s local versioned environment) in σ_n. Thus, $\mathbf{P}(n+1)$ holds.

If $c(\tau_{n+1}) = x := e$, then t_{n+1} updates the version and valuation of x in its local versioned environment. By Lemma 3.1, in any component versioned environment ve in σ_n, we must have

$$ve \cdot \nu(x) \leq |\{j \mid j \leq n \wedge c(\tau_j) \text{ is an assignment to } x\}|$$

where the right-hand side denotes the total number of writes to x, excluding the write in τ_{n+1}. By Lemma 3.3,

$$\sigma_{n+1} \cdot \Theta(t_{n+1}) \cdot \nu(x) = |\{j \mid j \leq n+1 \wedge c(\tau_j) \text{ is an assignment to } x\}|$$

This implies, for any component versioned environment ve' in σ_{n+1}, other than the local versioned environment of t_{n+1},

$$ve' \cdot \nu(x) < \Theta_{n+1}(t_{n+1}) \cdot \nu(x)$$

Since none of the other versioned environments are modified, $\mathbf{P}(n+1)$ continues to hold for such pairs of ve_1 and ve_2. Thus, $\mathbf{P}(N)$ holds for all $N \geq 0$. □

This proves that the *L-DRF* semantics only generates admissible states. Thus, χ is well-defined on any state which arises in the *L-DRF* execution of a program P.

Corollary 3.2 ($\chi(\hat{\pi})$ is well-defined) *For any execution $\hat{\pi}$ in the L-DRF semantics of P, $\chi(\hat{\pi})$ is well-defined.*

Proof: The function χ is only defined for admissible states, and by Lemma 3.4, the *L-DRF* semantics only produces executions containing admissible states. Thus, for any trace $\hat{\pi}$, $\chi(\hat{\pi})$ is well-defined. □

We now proceed to prove the soundness and completeness results (Theorem 3.1 and Theorem 3.2).

Proof: [Soundness, Theorem 3.1] We first outline the idea behind the proof, using Figure 3.4. For any trace π of P in the interleaving semantics, we obtain a corresponding trace $\hat{\pi}$ in the *L-*

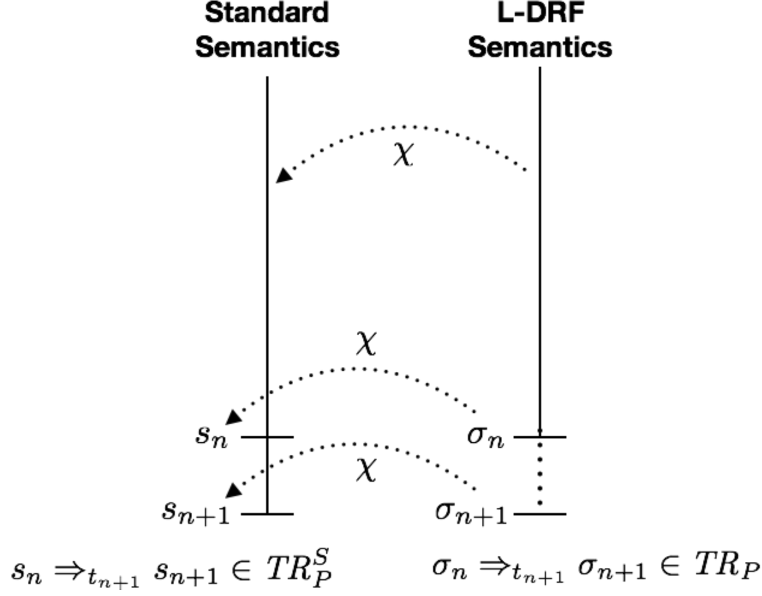

Figure 3.4: The proof obligation for Soundness. For any $n+1$ length trace π of program P in the standard semantics, we show that there exists a $n+1$ length trace $\hat{\pi}$ in the *L-DRF* semantics, such that $\chi(\hat{\pi}) = \pi$.

DRF semantics by taking the same interleaving of instructions from the threads. Our inductive hypothesis is that every N length interleaving execution has a corresponding N length *L-DRF* execution. We now consider a $N+1$ length execution π in the interleaving semantics, and we show that there exists a state σ_{n+1}, using which we can extend the N length *L-DRF* trace to create a $N+1$ length trace which is χ equivalent to π.

We prove the result using induction on the length of the traces. Let $\mathbf{P}(N)$ denote the following hypothesis. For any trace

$$\pi = \langle pc_0, \phi_0, \mu_0 \rangle \Rightarrow_{t_1} \ldots \Rightarrow_{t_N} \langle pc_N, \phi_N, \mu_N \rangle$$

of program P in the standard semantics, there exists a trace

$$\hat{\pi} = \langle pc_0, \mu_0, \Theta_0, \Lambda_0 \rangle \Rightarrow_{t_1} \ldots \Rightarrow_{t_N} \langle pc_N, \mu_N, \Theta_N, \Lambda_N \rangle$$

in the *L-DRF* semantics such that $\chi(\hat{\pi}) = \pi$. We outline the inductive arguments.
Base Case. For $N = 0$, the execution π contains the single state s_{ent}. The length 0 *L-DRF* execution contains the single state σ_{ent}. Since $\chi(\sigma_{ent}) = s_{ent}$, $\mathbf{P}(0)$ holds.
Inductive Case. Assume that $\mathbf{P}(k)$ holds for all executions of length k, where $0 \leq k \leq n$. We

prove that $\mathbf{P}(n+1)$ holds. Consider a $n+1$ length execution

$$\pi = \langle pc_0, \phi_0, \mu_0 \rangle \Rightarrow_{t_1} \ldots \Rightarrow_{t_{n+1}} \left\langle pc_{n+1}, \phi_{n+1}, \mu_{n+1} \right\rangle$$

of program P in the interleaving semantics. We denote by $\pi[1 \ldots n]$ the n-length prefix of π. By the inductive hypothesis, there exists a trace

$$\hat{\pi}' = \langle pc_0, \mu_0, \Theta_0, \Lambda_0 \rangle \Rightarrow_{t_1} \ldots \Rightarrow_{t_N} \langle pc_n, \mu_n, \Theta_n, \Lambda_n \rangle$$

of length n in the *L-DRF* semantics, such that $\pi[1 \ldots n] = \chi\left(\hat{\pi}'\right)$. Note that this implies that

$$\chi\left(\sigma_n\right) = s_n \tag{3.2}$$

where $\sigma_n = \langle pc_n, \mu_n, \Theta_n, \Lambda_n \rangle$. We show that there exists a state $\sigma_{n+1} = \left\langle pc_{n+1}, \mu_{n+1}, \Theta_{n+1}, \Lambda_{n+1} \right\rangle$ in the *L-DRF* semantics, such that $\chi\left(\sigma_{n+1}\right) = s_{n+1}$ and $\tau = \sigma_n \Rightarrow_{t_{n+1}} \sigma_{n+1} \in TR_P$, such that $c(\tau) = c(\pi_{n+1})$ (where π_{n+1} is the $n+1$-th transition in π). This would prove, in turn, that the *L-DRF* trace $\hat{\pi}'.\tau$ satisfies the property $\chi\left(\hat{\pi}'.\tau\right) = \pi$. We show this proof obligation diagrammatically in Figure 3.4. Note that, by Corollary 3.2, the function χ is well-defined for all the *L-DRF* traces we deal with. Let the last instruction in π_{n+1} be $\langle l, c, l' \rangle$, where

$$\begin{aligned} l &= pc_n \\ c &= c(\pi_{n+1}) \\ l' &= pc_{n+1} \end{aligned} \tag{3.3}$$

Note that, by construction, the pc and μ components of σ_{n+1} and s_{n+1} are made equal. We now case split on c.

- `acquire(m)`: Consider the state σ_{n+1} as follows:

$$\begin{aligned} \Theta_{n+1} &= \Theta_n[t_{n+1} \mapsto \mathit{updEnv}(\Theta_n(t_{n+1}), \Lambda_n)] \\ \Lambda_{n+1} &= \Lambda_n \end{aligned}$$

  Since $\sigma_n \cdot \mu = s_n \cdot \mu$, the lock acquisition succeeds from both s_n and σ_n. By the *L-DRF* semantics of `acquire`, $\sigma_n \Rightarrow_t \sigma_{n+1} \in TR_P$. Since the `acquire` does not change the maximum version, and the corresponding value, of each $x \in \mathcal{V}$ between σ_n and σ_{n+1}, we have $\chi\left(\sigma_{n+1}\right) = s_{n+1}$. Thus $\mathbf{P}(n+1)$ holds.

- `release(m)`: Consider the state σ_{n+1} as follows:

$$\Theta_{n+1} = \Theta_n$$
$$\Lambda_{n+1} = \Lambda_n[l' \mapsto \Theta_n(t_{n+1})]$$

Since $\sigma_n \cdot \mu = s_n \cdot \mu$, the lock release succeeds from both s_n and σ_n. By the *L-DRF* semantics of `release`, $\sigma_n \Rightarrow_t \sigma_{n+1} \in TR_P$. Since the `release` does not change the maximum version, and the corresponding value, of each $x \in \mathcal{V}$ between σ_n and σ_{n+1}, we have $\chi(\sigma_{n+1}) = s_{n+1}$. Thus $\mathbf{P}(n+1)$ holds.

- `assume(b)`: Consider the state σ_{n+1} as follows:

$$\Theta_{n+1} = \Theta_n \tag{3.4}$$
$$\Lambda_{n+1} = \Lambda_n \tag{3.5}$$

Consider an arbitrary variable x that is read in the condition `b`. By Corollary 3.1, in σ_n, the version of x is highest in $\Theta_n(t_{n+1})$. This implies, for any such variable x (since by inductive hypothesis, $\chi(\sigma_n) = s_n$),

$$\phi_n(x) = \Theta_n(t_{n+1}) \cdot \phi(x)$$
$$\implies [\![e]\!]\phi_n = [\![e]\!]\Theta_n(t_{n+1}) \cdot \phi \tag{3.6}$$

Since, by assumption, $s_n \Rightarrow_t s_{n+1} \in TR_P^{\mathbf{s}}$, and by Equation (3.6), we conclude that $\sigma_n \Rightarrow_t \sigma_{n+1} \in TR_P$. Since the `assume` does not alter the maximum version, and the corresponding value, of each $x \in \mathcal{V}$ between σ_n and σ_{n+1}, we have $\chi(\sigma_{n+1}) = s_{n+1}$. Thus $\mathbf{P}(n+1)$ holds.

- $x := e$: Consider the state σ_{n+1} as follows:

$$\Theta_{n+1} = \Theta_n[t_{n+1} \mapsto ve']$$
$$\Lambda_{n+1} = \Lambda_n$$

where $ve' = \langle \phi', \nu' \rangle$ such that

$$\phi' = \Theta_n(t_{n+1}) \cdot \phi[x \mapsto \phi_{n+1}(x)]$$
$$\nu' = \nu''[x \mapsto \nu''(x) + 1]$$

where $\nu'' = \Theta_n(t_{n+1}) \cdot \nu$. Consider an arbitrary variable y that is read in the expression e. By Corollary 3.1, in σ_n, the version of y is highest in $\sigma_n \cdot \Theta(t_{n+1})$. This implies, for any such variable $y \in \mathcal{V}$,

$$\begin{aligned} \phi_n(y) &= \Theta_n(t_{n+1}) \cdot \phi(y) \\ \implies [\![e]\!]\phi_n &= [\![e]\!]\Theta_n(t_{n+1}) \cdot \phi \\ \implies \phi_{n+1} &\in [\![e]\!]\Theta_n(t_{n+1}) \cdot \phi \end{aligned} \tag{3.7}$$

Coupled with the definition of ν', this proves that

$$ve' \in [\![x := e]\!]\Theta_n(t_{n+1}) \tag{3.8}$$

which allows us to conclude that $\sigma_n \Rightarrow_t \sigma_{n+1} \in TR_P$. By Lemma 3.3 and the construction of σ_{n+1}, the version of x is highest in $\Theta_{n+1}(t_{n+1})$, among all other component versioned environments of σ_{n+1}. This, coupled with the fact that no other versions are modified, we conclude that $\chi(\sigma_{n+1}) = s_{n+1}$. Consequently, $\mathbf{P}(n+1)$ holds.

Thus, $\mathbf{P}(N)$ holds for all $N \geq 0$. □

Proof: [Completeness, Theorem 3.2] We outline the proof idea, using Figure 3.5. Here, the situation is inverse of that in Figure 3.4. Here, we consider any trace $\hat{\pi}$ in the *L-DRF* semantics of P, and we show that the sequence $\chi(\hat{\pi})$ is a valid execution of P in the interleaving semantics. Our inductive hypothesis is on the length N of the *L-DRF* trace. When we consider a length $N+1$ length *L-DRF* execution $\hat{\pi}'$, we know there exists an execution π in the interleaving semantics corresponding to the N length prefix of $\hat{\pi}'$. We show that we can *extend* π by using $\chi(\sigma_{n+1})$ in order to obtain an $N+1$ length execution in the interleaving semantics, which is χ related to $\hat{\pi}'$.

We prove the result using induction on the length of the *L-DRF* execution. Let $\mathbf{P}(N)$ denote the following hypothesis. For any trace $\hat{\pi}$ of program P in the *L-DRF* semantics, $\chi(\hat{\pi})$ is a valid trace of P in the standard semantics. We prove the result using induction on the

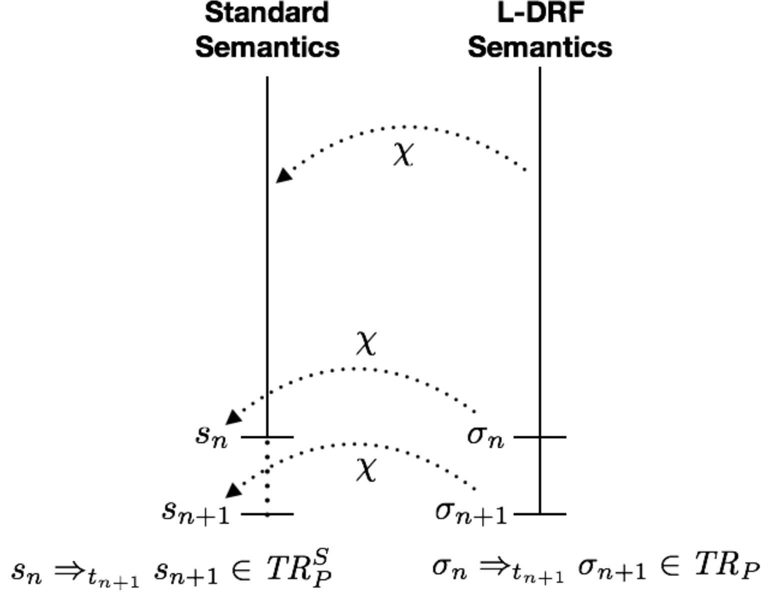

Figure 3.5: The proof obligation for Completeness. For any $n+1$ length trace $\hat{\pi}$ of program P in the *L-DRF* semantics, we show that $\chi(\hat{\pi})$ is a valid a $n+1$ length trace of P in the standard semantics.

length N of the trace $\hat{\pi}$.

Base Case. For $N = 0$, the trace $\hat{\pi}$ contains the single state σ_{ent}. Since $\chi(\sigma_{ent}) = s_{ent}$, which is a valid length 0 trace of P in the standard semantics, $\mathbf{P}(0)$ holds.

Inductive Case. Assume that $\mathbf{P}$ holds for all *L-DRF* traces of length $n \geq 0$. We show that $\mathbf{P}(n+1)$ holds too. Consider an arbitrary $(n+1)$ length trace

$$\hat{\pi} = \sigma_{ent} \Rightarrow_{t_1} \ldots \Rightarrow_{t_{n+1}} \sigma_{n+1}$$

of P in the *L-DRF* semantics, where each $\sigma_i = \langle pc_i, \mu_i, \Theta_i, \Lambda_i \rangle$. We show that $\chi(\hat{\pi})$ is a valid trace of P in the standard semantics. We show the proof obligation diagrammatically in Figure 3.5. If $\hat{\pi}[1 \ldots n]$ denotes the n length prefix of the execution $\hat{\pi}$, then by the induction hypothesis,

$$\chi(\hat{\pi}[1 \ldots n]) = (s_0 \Rightarrow_{t_1} \ldots \Rightarrow_{t_n} s_n)$$

is a valid execution of P in the interleaving semantics. We let $s_{n+1} = \chi(\sigma_{n+1})$ and show that $s_n \Rightarrow_{t_{n+1}} s_{n+1} \in TR_P^{\mathbf{s}}$.

We case split on $c(\tau_{n+1})$, where τ_{n+1} is the last transition in $\hat{\pi}$. We denote $\chi(\sigma_{n+1})$ as s_{n+1}.

If $c(\tau_{n+1})$ is either an `acquire` or a `release`, then since the lock maps are the same in both s_n and σ_n, the lock acquisition (or release) succeeds from s_n. Moreover, since neither of the commands alter the versions between σ_n and σ_{n+1}, we have $s_n \cdot \phi = s_{n+1} \cdot \phi$. Thus, $s_n \Rightarrow_{t_{n+1}} s_{n+1} \in TR^{\mathbf{s}}_P$, and $\mathbf{P}(n+1)$ holds.
If $c(\tau_{n+1})$ is `assume(b)`, then, by Corollary 3.1, the version of any variable x read in the condition `b` is highest in $\Theta_n(t_{n+1})$. Moreover, since $\chi(\sigma_n) = s_n$ (by induction hypothesis), for any variable x accessed in the condition `b`, we must have

$$env_n(x) = \Theta_n(t_{n+1}) \cdot \phi(x)$$

This implies, $[\![b]\!]\phi_n = [\![b]\!]\Theta_n(t_{n+1})\phi$. Thus, $s_n \Rightarrow_{t_{n+1}} s_{n+1} \in TR^{\mathbf{s}}_P$, and $\mathfrak{P}(n+1)$ holds.
Finally, we consider the case when $c(\tau_{n+1})$ is an assignment statement of the form $x := e$. In a manner analogous to the case of the `assume` earlier, we can prove that $[\![e]\!]\phi_n = [\![e]\!]\Theta_n(t_{n+1}) \cdot \phi$. By, Lemma 3.3, the version of x in σ_{n+1} is highest in $\Theta_{n+1}(t_{n+1})$. Thus,

$$\phi_{n+1}(x) = \Theta_{n+1}(t_{n+1}) \cdot \phi(x) \tag{3.9}$$

Since the assignment command is always enabled, and by Equation (3.9), we obtain $s_n \Rightarrow_{t_{n+1}} s_{n+1} \in TR^{\mathbf{s}}_P$, and $\mathbf{P}(n+1)$ holds.
Thus, $\mathbf{P}$ holds for all *L-DRF* traces of length $N \geq 0$. □

Corollary 3.3 *For any race free concurrent program P, the L-DRF analysis is precise. In other words, consider an arbitrary state $\sigma = \langle pc, \mu, \Theta, \Lambda \rangle$ in $[\![P]\!]$. Let S be the set of variables which are relevant [1] at $pc(t)$, for some thread t. Then there exists a trace $\pi = s_{ent} \Rightarrow_{t_1} \ldots \Rightarrow_{t_n} s_n$ of P in the standard semantics, such that for some $s_i = \langle pc_i, \mu_i, \phi_i \rangle$ with $pc_i(t) = pc(t)$, we have*

$$\forall x \in S : \phi_i(x) = \Theta(t) \cdot \phi(x)$$

Proof: If $\sigma \in [\![P]\!]$, then there must exist an execution $\hat{\pi} = \sigma_e nt \Rightarrow_{t_1} \ldots \Rightarrow_t \sigma$ of P in the *L-DRF* semantics. By Theorem 3.2, $\chi(\hat{\pi}) = s_{ent} \Rightarrow_{t_1} \ldots \Rightarrow_t s$ is a valid trace of P in the standard semantics, where $\chi(\sigma) = s$. By Corollary 3.1, the version of each $x \in S$ is highest in $\Theta(t)$, among all component versioned environments in σ. By the construction of the function χ, we have, for each variable $x \in S$, $\phi_i(x) = \Theta(t) \cdot \phi(x)$. □

Remark.ill now we assumed that buffers associated with every post-release point in $\mathcal{L}^{rel}_{\mathtt{m}}$

[1] Recall that a set of variables are *relevant* at a program point if they are accessed (either read from or written to) in a command at that point.

are relevant to each pre-acquire point in $\mathcal{L}_{\mathtt{m}}^{acq}$. That is, $\forall n \in \mathcal{L}_{\mathtt{m}}^{rel} : \mathcal{G}(n) = \mathcal{L}_{\mathtt{m}}^{acq}$. However, if no (standard) execution of the program P contains a transition τ_i (with the target location being n) which synchronizes-with a transition τ_j (with source location $n' \in \mathcal{L}_{\mathtt{m}}^{acq}$), then Theorem 3.1 (as well as Theorem 3.2) holds even if we remove n' from $\mathcal{G}(n)$. This is true because in race-free programs, conflicting accesses are ordered by the happens-before relation. Thus, if the most up-to-date value of a variable accessed by t was written by another thread t', then in between these accesses there must be a (sequence of) synchronization operations starting at a lock released by t' and ending at a lock acquired by t. This refinement of the set $\mathcal{G}$ based on the above observation can be used to improve the precision of the analyses derived from *L-DRF*, as it reduces the set of possible release points an acquire can observe.

Chapter 4

Sequential Abstractions of the *L-DRF* semantics

In this chapter, we show how to employ standard *sequential* analyses to compute sound approximations of the *L-DRF* semantics. We call this class of sequential analyses "*sync*-CFG based analyses", since these can be thought of as analyzing concurrent programs represented as the *sync*-CFG control flow structure, with each thread operating on local copies of the data states, and communication between the threads limited to synchronization points alone. The *sync*-CFG representation of a concurrent program P (which was first introduced in [22]) comprises the control flow graphs of each static thread code, augmented with *synchronizes-with* edges between synchronization operations (like releases and acquires of the same lock).

A *sync*-CFG differs from the standard "product-graph" representation of concurrent programs in two important ways:

1. The *sync*-CFG contains nodes corresponding to each control location in the concurrent program P. In contrast, the product graph contains nodes corresponding to every possible *combination* of control locations in P.

2. Each execution of P corresponds to some path in its product graph representation. A *sync*-CFG does *not* maintain such a correlation. In fact, a *sync*-CFG ensures that for each execution of P, every *happens-before* path induced by the execution corresponds to some path in the *sync*-CFG. Since the *sync*-CFG does not soundly approximate the set of possible executions of P, this has some interesting implications regarding the soundness of the analyses which use it. In particular, data flow facts computed for a variable are guaranteed to be sound only at *relevant* points, where the variable is accessed. The facts

may be unsound elsewhere.

As an example, consider again the program in Figure 1.2. The *sync*-CFG representation of the program is given on the left in Figure 1.3. On the other hand, an excerpt of the product-graph of this program is shown on the right of the same figure. As one may expect, any analysis based on the product graph would be infeasible for large programs.

Thanks to Theorem 3.1 and Theorem 3.2, we can now devise computable and efficient abstract analyses for data race free concurrent programs using the *sync*-CFG. In particular, this also allows us to establish the soundness of the value-set based analysis [22] by casting it as an abstract interpretation of the *L-DRF* semantics.

Technically, the class of *sync*-CFG analyses are derived by two (successive) abstraction steps: First, we abstract the *L-DRF* semantics using a thread-local cartesian abstraction which ignores version numbers and forgets the correlation between the local states of the different threads. This results in cartesian states where every program point is associated with a set of (thread-local) environments. The form of these cartesian states is precisely the one obtained when computing the collecting semantics of *sequential* programs. Thus, they can be further abstracted using any sequential abstraction, including relational ones. This allows maintaining correlations between variables at all points except synchronization points (acquires and releases of locks).

We make the initial decision to abstract away the versions for simplicity. We will refine this abstraction later on in the chapter.

4.1 Theory of Consistent Abstractions

In this section, we recall the theory of consistent abstractions from [19]. An *abstract interpretation*, or simply an *analysis*, of a program P is a structure $\mathcal{A} = (D, \leq, f)$, where

- D is the domain and $\leq$ is a partial ordering on D such that $(D, \leq)$ forms a complete lattice
- $f : D \to D$ is a monotone transfer function

By the Tarski fixpoint theorem [77], f has a least fixpoint in D, which we denote as $[\![f]\!]_{\mathcal{A}}$.

Given analyses $\mathcal{C} = (D, \leq, f)$ and $\mathcal{A} = (D', \leq', f')$, we say $\mathcal{A}$ is a consistent abstraction of $\mathcal{C}$, if there exists functions $\alpha : D \to D'$ (called the abstraction function), and $\gamma : D' \to D$ (called the concretization function), such that

1. α and γ form a Galois connection, which entails
   (a) α and γ are monotonic
   (b) α and γ satisfy

$$\forall d \in D : \gamma(\alpha(d)) \geq d$$
$$\forall d' \in D' : \alpha(\gamma(d')) = d'$$

2. $[\![f]\!]_{\mathcal{C}} \leq \gamma([\![f']\!]_{\mathcal{A}})$

Theorem 4.1 (Sufficient Condition for Consistent Abstraction, [19]) *Given analyses* $\mathcal{C} = (D, \leq, f)$ *and* $\mathcal{A} = (D', \leq', f')$*, sufficient conditions for* $\mathcal{A}$ *to be a consistent abstraction of* $\mathcal{C}$ *are the following:*

1. *there exists functions* $\alpha : D \to D'$*, and* $\gamma : D' \to D$*, such that* α *and* γ *form a Galois connection.*
2. f' *safely approximates* f*, in that*

$$\forall d \in D : \alpha(f(d)) \leq' f'(\alpha(d))$$

4.2 *A-DRF*: A Canonical *sync*-CFG Analysis based on *L-DRF*

In this section, we define a canonical abstract analysis, operating on the *sync*-CFG representation of an input data race free program P, derived from the *L-DRF* semantics. We then prove the soundness of *A-DRF* by showing that it satisfies the sufficient conditions outlined in Section 4.1, which imply that the least fixpoint solution of *A-DRF* is a sound approximation of the least fixpoint solution of *L-DRF*. As before, we fix a data race free concurrent program $P = (\mathcal{T}, \mathcal{L}, \mathcal{V}, \mathcal{M})$.

4.2.1 Thread-Local Cartesian Abstract Domain

The abstract domain is a complete lattice over *cartesian states*, functions mapping program locations to sets of environments, ordered by point-wise inclusions. We denote the set of cartesian states of P in the *A-DRF* analysis by $\mathcal{A}_\times$, and range over it using $a_\times$.

$$\mathcal{D}_\times \equiv \langle \mathcal{A}_\times, \sqsubseteq_\times \rangle \quad \text{where } \mathcal{A}_\times \equiv \mathcal{L} \to \mathbb{P}(Env) \quad \text{and} \quad a_\times \sqsubseteq_\times a'_\times \iff \forall n \in \mathcal{L}.\, a_\times(n) \subseteq a'_\times(n)$$

The abstraction function $\alpha_\times$ maps a set of *L-DRF* states $C \subseteq \Sigma$ to a *cartesian state* $a_\times \in \mathcal{A}_\times$. The abstract value $\alpha_\times(C)(n)$ contains the collection of t's environments (where $t = tid(n)$) coming from any state $\sigma \in C$ where t is at location n. In addition, if n is a post-release point, $\alpha_\times(C)(n)$ also contains the contents of the buffer $\Lambda(n)$ for each state $\sigma \in C$. As a first cut, we abstract away the versions entirely. In a later section, we outline how to recover the versions in order to improve the precision of the abstract analyses.

The concretization function $\gamma_\times$ maps a cartesian state $a_\times$ to a set of (admissible) *L-DRF* states C in which the local state of a thread t, when t is at program point $n \in \mathcal{L}_t$, comes from $a_\times(n)$ and the contents of the release buffer pertaining to the post-release location $n \in \mathcal{L}^{rel}$ also comes from $a_\times(n)$.

$$
\begin{aligned}
&\alpha_\times : \mathbb{P}(\Sigma) \to \mathcal{A}_\times,\\
&\quad \text{where } \alpha_\times(C) = \lambda n \in \mathcal{L}.\, \{\phi \mid \langle pc, \mu, \Theta, \Lambda\rangle \in C \wedge pc(tid(n)) = n \wedge \langle \phi, \nu\rangle = \Theta(tid(n))\} \cup \\
&\qquad\qquad\qquad\qquad\qquad\qquad \{\phi \mid \langle pc, \mu, \Theta, \Lambda\rangle \in C \wedge n \in \mathcal{L}^{rel} \wedge \langle \phi, \nu\rangle = \Lambda(n)\}\\
&\gamma_\times : \mathcal{A}_\times \to \mathbb{P}(\Sigma),\\
&\quad \text{where } \gamma_\times(a_\times) = \left\{ \langle pc, \mu, \Theta, \Lambda\rangle \in \Sigma \;\middle|\; \begin{array}{l} pc \in PC \wedge \mu \in LM \wedge \\ \forall t \in \mathcal{T} :\; \Theta(t) = \langle \phi, \lambda x._\rangle \wedge \phi \in a_\times(pc(t)) \wedge \\ \forall n \in \mathcal{L}^{rel} :\; \Lambda(n) = \langle \phi, \lambda x._\rangle \wedge \phi \in a_\times(n)\} \end{array} \right\}
\end{aligned}
$$

4.2.2 Abstract Transitions

The abstract cartesian semantics is defined using a transition relation, $TR^\times \subseteq \mathcal{A}_\times \times \mathcal{T} \times \mathcal{A}_\times$.

- *Assignment.* Since we have already abstracted away the version numbers, we define the meaning of assignments commands c using their interpretation according to the standard semantics, denoted by $[\![c]\!]_{\mathbf{s}}$. The set of transitions generated by an assignment command c is:

$$
TR_c^\times = \left\{ a_\times \Rightarrow_t^\times a_\times \left[n' \mapsto a_\times(n') \cup \bigcup_{\phi \in a_\times(n)} [\![c]\!]_{\mathbf{s}}(\phi) \right] \;\middle|\; \langle n, c, n'\rangle \in inst_t \right\}
$$

- *Assume.* Similar to the assignment commands, the semantics of the assume commands c make use of their interpretation according to the standard semantics. The set of transitions generated by an assume command c is:

$$
TR_c^\times = \left\{ a_\times \Rightarrow_t^\times a_\times \left[n' \mapsto a_\times(n') \cup \bigcup_{\phi \in a_\times(n)} [\![c]\!]_{\mathbf{s}}(\phi) \right] \;\middle|\; \langle n, c, n'\rangle \in inst_t \right\}
$$

- *Acquire.* With the omission of any information pertaining to ownership of locks, an acquire command executed at program location n is only required to over-approximate the effect of updating the environment of a thread based on the contents of all buffers relevant to n. To do so, we define an abstract *mix* operation which mixes together different environments at the granularity of single variables. The set of transitions pertaining to an acquire command $c = \texttt{acquire(m)}$ is

$$TR_c^{\times} = \{a_\times \Rightarrow_t^\times a_\times[n' \mapsto E_{mix}] \mid \langle n, c, n' \rangle \in inst_t\} \text{ , where}$$
$$E_{mix} = mix(a_\times(n') \cup \bigcup\{a_\times(\bar{n}) \mid \bar{n} \in \mathcal{L}_{\mathtt{m}}^{rel} \wedge n \in \mathcal{G}(\bar{n})\}) \quad \text{, and}$$
$$mix : \mathbb{P}(Env) \to \mathbb{P}(Env) \equiv \lambda B_\times.\{\phi' \mid \forall x \in \mathcal{V}, \exists \phi \in B_\times : \phi'(x) = \phi(x)\}$$

  In other words, the *mix* takes a cartesian product of the input states. Note that as a result of abstracting away the version numbers, a thread cannot determine the most up-to-date value of a variable, and thus conservatively picks any possible value found either in its own local environment or in a relevant release buffer. Figure 4.1 illustrates the operation of the *mix* function.

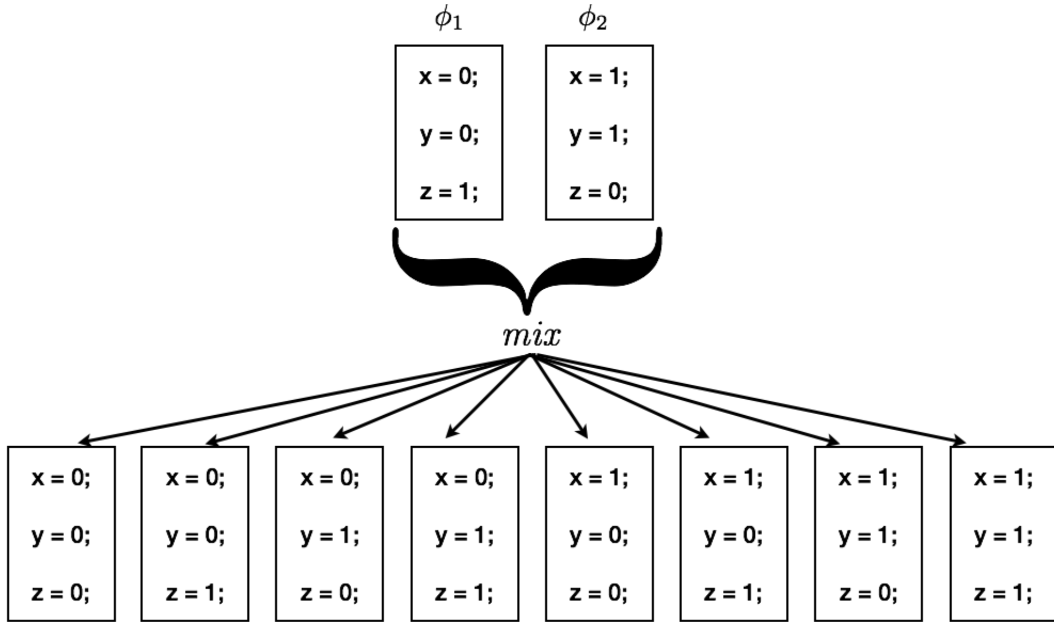

Figure 4.1: Illustrating the *mix* on a set of containing two environments ϕ_1 and ϕ_2. Observe that the invariant $x = y$ holds in the input environments. However, since this *mix* operates at the granularity of single variables, the correlation is lost in the output states.

- *Release.* Interestingly, the effect of release commands in the cartesian semantics is the

same as `skip`: This is because the abstraction neither tracks ownership of locks nor explicitly manipulates the contents of buffers. Hence, the set of transitions pertaining to a release command $c = \texttt{release(m)}$ is

$$TR_c^\times = \{a_\times \Rightarrow_t^\times a_\times[n' \mapsto a_\times(n') \cup a_\times(n)] \mid \langle n, c, n' \rangle\rangle \in inst_t\}$$

4.3 The LFP solution of *A-DRF*

The set of abstract transitions generated by the program P is

$$TR^\times = \bigcup_{c \in cmd(P)} TR_c^\times$$

The least fixpoint formulation of the *A-DRF* analysis is given by

$$\begin{aligned}
&[\![P]\!]_\times = LFP\ \mathcal{F}_\times \quad \text{where} \\
&\mathcal{F}_\times = \lambda a_\times .\, a_\times^{ent} \textstyle\bigsqcup_\times \left(\bigsqcup_\times \{a'_\times \mid \exists t \in \mathcal{T} :\ \left(a_\times \Rightarrow_t^\times a'_\times\right) \in TR^\times\}\right) \ \text{, where} \\
&\qquad a_\times^{ent} = \alpha_\times(\{\sigma_{ent}\})
\end{aligned}$$

The initial state $a_\times^{ent}$ maps the entry location of every thread to the set containing the single environment, where all the variables are initialized to 0. Every other program location is mapped to the empty set.

Note that the least fixpoint formulation of P in the *A-DRF* analysis can be viewed as the collecting semantics of a *sequential* program P' obtained by augmented the control-flow graphs of the threads in P with edges from post-release points n' to pre-acquire points n in $n \in \mathcal{G}(n')$, and where a special *mix* operator is used to combine information at the acquire points. Further, note that we abstract the environment of buffers and their corresponding release location into a single entity, which is the standard over-approximation of the set of environments at a given program location. Hence, the concurrent analysis of P can be reduced to the *sequential analysis* of P', provided a sound over-approximation of the *mix* operator is given. This is why our technique allows one to quickly port an existing sequential analysis to an analysis for race free concurrent programs. Lastly, the thread-local cartesian abstract analysis is expressed as the LFP of a first-order equation, where the only unknown is the set of reachable states. This is unlike the analysis formulated in [59], which is an abstract interpretation formulation of the rely-guarantee paradigm [88]. The latter analysis has *two* unknowns: the set of reachable states *and* the set of "interferences". Consequently, the analysis in [59] involves a *nested* fixpoint

computation.

The analysis in [22] is obtained by abstracting the thread-local cartesian states using the value set abstraction on the environments domain. Note that in the value set domain, where every variable is associated with (an over approximation of) the set of its possible values, the mix operator reduces to a join operator.

4.4 Soundness of the Sequential Abstractions

The soundness of the sequential abstractions is expressed by the following theorem.

Theorem 4.2 (Soundness of Sequential Abstractions) $\gamma_\times([\![P]\!]_\times) \supseteq [\![P]\!]$.

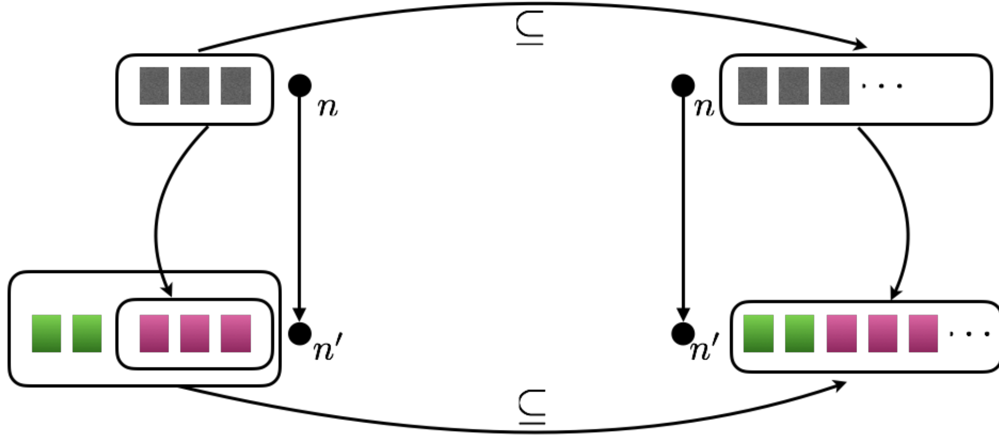

Figure 4.2: The proof obligation for proving that the analysis $\mathcal{F}_\times$ is a consistent abstraction of the *L-DRF* analysis $\mathcal{F}$.

Proof: In order to prove this theorem, we show that the analysis $\mathcal{F}_\times$ is a *consistent abstraction* of the *L-DRF* analysis $\mathcal{F}$ (presented in Section 3.2). Recall that the *L-DRF* analysis is

$$\begin{aligned}&\mathcal{F} = \lambda X.\, \{\sigma_{ent}\} \cup post(X) \text{ where}\\ &post(X) = \{\sigma' \mid \exists \sigma \in X, \exists t \in \mathcal{T} :\, \sigma \Rightarrow_t \sigma' \in TR_P\}\end{aligned}$$

For ease of reference, we rewrite the analysis $\mathcal{F}_\times$ as

$$\begin{aligned}&\mathcal{F}_\times = \lambda X.\, \{a_\times^{ent}\} \bigsqcup\nolimits_\times post_\times(X) \text{ where}\\ &post_\times(X) = \bigsqcup\nolimits_\times \{a'_\times \mid \exists t \in \mathcal{T} :\, a_\times \Rightarrow_t^\times a'_\times \in TR^\times\}\end{aligned}$$

We already defined an abstraction $\alpha_\times$ and concretization $\gamma_\times$ function between the domain $\mathbb{P}(\Sigma)$, of $\mathcal{F}$, and the domain $\mathcal{A}_\times$, of $\mathcal{F}_\times$, earlier in this chapter. It is fairly straightforward to

show that $\alpha_\times$ and $\gamma_\times$ satisfy the conditions for forming a Galois connection. We now prove that the $post_\times$ function is a sound abstraction of *post*. Consider an arbitrary set of states

$$X = \{\sigma_1, \ldots, \sigma, \ldots \sigma_m\}$$

The *post* operator chooses some $\sigma \in X$, and makes a transition $\sigma \Rightarrow_t \sigma'$, which is made possible by some instruction $\langle n, c, n' \rangle \in inst_t$, where $n = \sigma \cdot pc(t)$. This results in

$$X' = \{\sigma_1, \ldots, \sigma', \ldots \sigma_m\}$$

We let $a_\times = \alpha_\times(X)$, and denote the resulting abstract state after t executes the instruction $\langle n, c, n' \rangle$ as $a'_\times$.

We define the set S as

$$S = \{\phi \mid \exists \sigma \in X : \sigma \cdot \Theta(t) = \langle \phi, \nu \rangle \wedge \sigma \cdot pc(t) = n\}$$

In other words, S is set of all thread-local environments of t, in states in X where t is at control location n. By the construction of $\alpha_\times$,

$$S \subseteq \alpha_\times(X)(n)$$

In an analogous fashion, we define the set S' as

$$S' = \{\phi \mid \exists \sigma \in X' : \sigma \cdot \Theta(t) = \langle \phi, \nu \rangle \wedge \sigma \cdot pc(t) = n'\}$$

We need to prove that

$$S' \subseteq a'_\times(n')$$

This proof obligation is pictorially depicted in Figure 4.2. We case split on c, the command in the instruction $\langle n, c, n' \rangle$.

- *Assignments and Assume.* By the construction of $\alpha_\times$, $(\sigma \cdot \Theta(t) \cdot \phi) \in a_\times(n)$. Thus, by the *L-DRF* semantics and the *A-DRF* analysis, $(\sigma' \cdot \Theta(t) \cdot \phi) \in a'_\times(n')$. Again, by the construction of $\alpha_\times$, we know that

$$\{\phi \mid \exists \sigma \in X : \sigma \cdot \Theta(t) = \langle \phi, \nu \rangle \wedge \sigma \cdot pc(t) = n'\} \subseteq a_\times(n') \tag{4.1}$$

Since none of the other states $\sigma' \in X$ are altered by this transition, and since the *A-DRF* analysis takes a union of the existing elements at $a_\times(n')$ in the computation of $a'_\times(n')$, we have

$$S' \subseteq a'_\times(n')$$

- *Release.* By the construction of $\alpha_\times$,

  $$(\sigma \cdot \Theta(t) \cdot \phi) \in a_\times(n)$$

  By the semantics of the the release in the *A-DRF* analysis, and since $\sigma' \cdot \Theta(t) = \sigma \cdot \Theta(t)$, we have

  $$(\sigma' \cdot \Theta(t) \cdot \phi) \in a'_\times(n')$$

  Coupled with Equation (4.1) and the fact that the *A-DRF* analysis takes a union of the existing elements at $a_\times(n')$ in the computation of $a'_\times(n')$, we have

  $$S' \subseteq a'_\times(n')$$

- *Acquire.* In this case, n chooses to take the value of a variable x in the thread-local environment of t, from the versioned environment ve in some relevant buffer, or the existing thread-local environment of t. By the construction of $\alpha_\times$, if ve was chosen from some post-release point $\bar{n}$, then this environment is guaranteed to exist in $a_\times(\bar{n})$. Likewise, if ve is simply the thread-local versioned environment of t, then the environment would be in $a_\times(n)$. Since, by the semantics of the `acquire` in the *A-DRF* analysis, all the environments at all such $\bar{n}$, and the environment at n, is taken into account in the *mix*, and since this operation is performed for each variable $x \in \mathcal{V}$,

  $$(\sigma' \cdot \Theta(t) \cdot \phi) \in a_\times(n')$$

  Coupled with Equation (4.1) and the fact that the *A-DRF* analysis takes a union of the existing elements at $a_\times(n')$ in the computation of $a'_\times(n')$, we have

  $$S' \subseteq a'_\times(n')$$

This completes the proof. □

Remark.t is worth noting at this point that the main cause behind the loss in precision in the *A-DRF* analysis is the *mix* operator, applied at the inter-thread join points. Since the function operates at the granularity of individual variables, it essentially takes a cartesian product of the input environments, resulting in quite severe loss of precision.

4.5 Other abstractions of *L-DRF*

The analysis in [22] can be obtained, with an additional abstraction, from *A-DRF*. A value-set domain VS maps each program variable to a *set* of values, that is, $VS : \mathcal{V} \rightarrow \mathbb{P}(\mathbb{V})$. Thus, we define the value-set abstraction function $\alpha_{vs} : \mathcal{A}_\times \rightarrow VS$ as

$$\alpha_{vs}(a_\times) = \lambda n.\, (\lambda x \in \mathcal{V}.\{v \mid \exists \phi \in a_\times(n) : \phi(x) = v\})$$

With this join of the abstract domain, the abstract *mix* operator reduces to the standard value-set join operation (which takes a component wise union of the value-sets).

We can improve upon *A-DRF* by not forgetting the versions entirely. We augment $\mathcal{A}_\times$ with a set S of "recency" information based on the versions as follows:

$$S = \lambda C.\{\bar{t} \mid \exists \sigma \in C, x \in \mathcal{V} : \left(\operatorname*{argmax}_{t \in \mathcal{T}} \sigma\Theta(t)\nu(x)\right) = \bar{t}\}$$

In other words, S soundly approximates the set of threads which contain the most up-to-date value of some variable $x \in \mathcal{V}$. This additional information can now be used to improve the precision of *mix*.

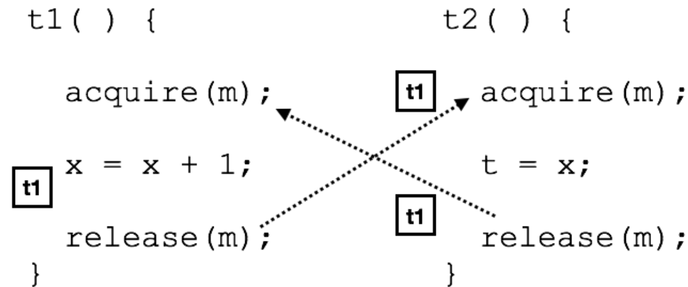

Figure 4.3: A simple program demonstrated the benefit of using thread-identifiers in the abstract state. In the normal setting, the synchronizes-with edges creates a cycle in the program, and it is not possible to derive an upper bound on the value of `x`. However, if we track thread-identifiers in the state, thread `t1` observes that any state it receives from `t2` is tagged with the {`t1`}, and thus `t1` can safely drop the data flow facts.

In the program shown in Figure 4.3, thread `t1` writes to `x`, while holding the lock `m`, whereas thread `t2` reads from `x` while holding `m`. In the usual *sync*-CFG setting, the synchronizes-with edges creates a cycle in the program graph. Thus, the data flow facts propagate back and forth between the threads, and the analysis, in this example, fails to derive an upper bound for the value of `x`. In the recency based analysis, the data flow fact comprises elements from $\mathcal{A}_\times$, as well as a set S of thread-identifiers. Whenever a thread writes to a variable, it adds its identifier to S. Other commands do not affect S. In the example, `t1` adds its identifier to S, and this is propagated to `t2`. However, since `t2` does *not* write to `x`, the set S is propagated back, unaltered, to `t1`. The thread `t1` now finds that the incoming data flow fact contains a singleton S, with its thread-identifier, which indicates it is receiving a stale fact. Had any other thread written to `x`, it would contain at least two thread-identifiers. This allows the thread to safely drop the data flow fact, thereby breaking the cycle. An abstract analysis based on thread-identifiers can, in fact, prove an upper bound for `x`.

In our experiments, we further abstract *A-DRF* using numerical domains like octagons and intervals.

Chapter 5

A Region-Parameterized version of *L-DRF*

In this chapter, we introduce a refined notion of data race freedom, based on *data regions*, and derive from it a more precise abstract analysis capable of transferring *some* relational information between threads at synchronization points. The objective is to modify the *L-DRF* semantics such that the abstract mix operates at a granularity higher than individual variables.

5.1 Why do we need another semantics?

Figure 4.1 outlines the key issue with the *L-DRF* semantics: any abstract analysis derived from the *L-DRF* semantics must make use of an abstract *mix* which operates at the granularity of individual variables. Thus, even though two variables may be related in the input environments to *mix* (like $x = y$ in Figure 4.1), the function must necessarily forget their correlation after the mixing. This is essential for soundness. This is the reason that prevents us from proving the assertion $x = y$ at line 11 in the motivating example in Figure 1.2. Even though the `acquire(m)` in t_2 obtains the fact $x = y$ from both its input edges, it fails to maintain this correlation post the mix.

Essentially, regions are a user-defined partitioning of the set of program variables. We call each partition a *region* r, and denote the set of regions as R and the region of a variable x by $rg(x)$.

The semantics precisely tracks correlations between variables *within* regions *across* inter-thread communication, while abstracting away the correlations between variables across regions. This partitioning is based on the semantics of the program: developers often write code where a group of variables form a logical cluster. Typically, some invariant holds on the variables within

a region, though there may not necessarily be such a correlation between variables across regions. The semantics operates accordingly: it precisely tracks correlations between variables *within* regions *across* inter-thread communication, while abstracting away the correlations between variables across regions. With suitable abstractions, the tracked correlations can improve the precision of the analysis for programs which conform to the notion of race freedom defined below.

5.2 Region Race Freedom

We present a refinement of the standard notion of data race freedom by ensuring that variables residing in the same region are manipulated atomically across threads. A *region-level data race* occurs when two concurrent threads access variables from the same region r (not necessarily the same variable), with at least one access being a write, and the accesses are devoid of any ordering constraints.

A command $x := e$ constitutes a *write access* to the region $rg(x)$, and a *read access* of every region $rg(y)$, for each variable y appearing in the expression e. Similarly, a command `assume(b)` constitutes a read access of every region $rg(y)$, for each variable y appearing in the condition `b`. We are now in a position to introduce our notion of region level races.

Definition 5.1 (Region-level races) *Let P be a program and let R be a region partitioning of P. An execution π of P, in the standard interleaving semantics, has a* region-level race *if there exists $0 \leq i < j < |\pi|$, such that $c(\pi_i)$ and $c(\pi_j)$ both access variables in region $r \in R$, at least one access is a write, and it is not the case that $\pi_i \xrightarrow{hb}_{\pi} \pi_j$.*

The problem of checking for region races can be reduced to the problem of checking for data races as follows. We introduce a fresh variable $\mathtt{X}_r$ for each region $r \in R$. We now transform the input program P to a program P' with the following additions.

- We precede every assignment statement `x` := `e`, where r_w is the region which is written to, and $r_1, \ldots, r_n$ are the regions read, with a sequence of instructions $\mathtt{X}_{r_w} := \mathtt{X}_{r_1}; \ldots \mathtt{X}_{r_w} := \mathtt{X}_{r_n};$.
- Statements of the form $\mathtt{assume}(b)$ do not need to be changed because b may refer only to thread-private variables.
- The `acquire` and `release` statements do not involve the access of any variable. Thus, they remain unmodified.

Note that these modifications do not alter the semantics of the original program (for each trace of P there is a corresponding trace in P', and vice versa). We now check for *data races* on the variables $\mathtt{X}_r$'s.

5.3 The *R-DRF* semantics

The *R-DRF* semantics is obtained via a simple change to the *L-DRF* semantics, a write-access to a variable x leads to incrementing the version of every variable that resides in x's region. In other words, the semantics of the assignment command is as follows:

$$[\![x := e]\!] :\ VE \to \mathbb{P}(VE) = \lambda\, \langle \phi, \nu \rangle\, .\, \{ \langle \phi[x \mapsto v], \nu[y \mapsto \nu(y) + 1] \rangle \mid v \in [\![e]\!]\phi \wedge y \in \mathcal{V} \wedge rg(x) = rg(y) \}$$

It is easy to see that Theorems 3.1 and 3.2 hold if we consider the *R-DRF* semantics instead of the *L-DRF* semantics, provided the program is region race free with respect to the given region specification. Hence, we can analyze such programs using abstractions of *R-DRF* and obtain sound results with respect to the interleaving semantics (Section 2.2).

5.3.1 Thread-Local Abstractions of the *R-DRF* Semantics

The cartesian abstractions defined in Section 4 can be extended to accommodate regions in a natural way. The only difference lies in the definition of the *mix* operation, which now operates over *regions*, rather than variables:

$$mix : \mathbb{P}(Env) \to \mathbb{P}(Env) \stackrel{\text{def}}{=} \lambda B_\times . \{ \phi' \mid \forall r \in R, \exists \phi \in B_\times : \forall x \in \mathcal{V}.\, rg(x) = r \implies \phi'(x) = \phi(x) \}$$

where the function rg maps a variable to its region. Mixing environments at the granularity of regions is permitted because the *R-DRF* semantics ensures that all the variables in the same region have the same version. Thus, their most up-to-date values reside in either the thread's local environment or in one of the release buffers. As before, we can obtain an effective analysis using any sequential abstraction, provided that the abstract domain supports the (more precise) region based mix operator.

5.4 Illustrative Example

We illustrate the effect of the regions using some small examples. Consider again the situation in Figure 4.1. Recall that even though the input environments maintained $x = y$, the *mix* was unable to preserve this correlation because it operated at the granularity of individual variables. However, when *mix* is made aware of the region definitions, it maintains the correlation between

variables *within* a region. Thus, in Figure 5.1, the invariant $x = y$ continues to hold in the output state.

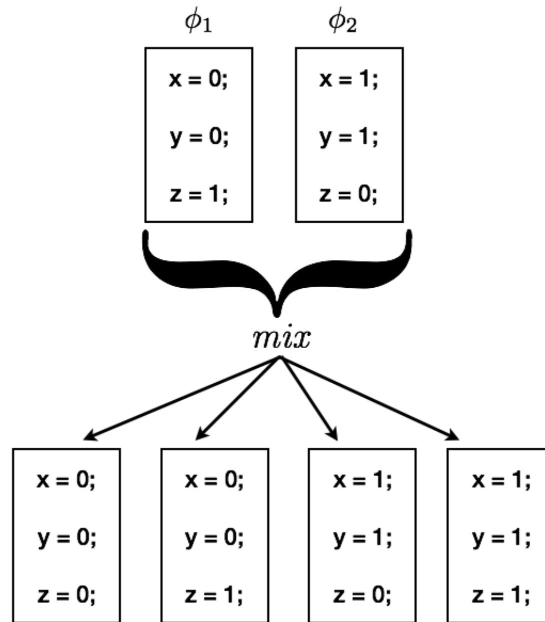

Figure 5.1: Illustrating the operation of *mix* when it is aware of regions. In this example, with the regions being $\langle\{x, y\}, \{z\}\rangle$, the function maintains the correlation between x and y in the output.

Returning to the program in Figure 1.2, consider the situation at the `acquire` at line 10 (illustrated in Figure 5.2). It receives the invariant $x = y$ from both its input branches. The *mix* in the polyhedral abstraction of *L-DRF* only outputs the correct bounds for the variables, and forgets the correlation between x and y. However, the region aware *mix* preserves this invariant, which enables the analysis derived from *R-DRF* to prove the assertion at line 11.

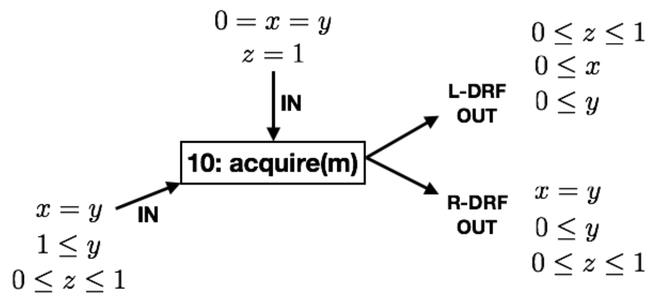

Figure 5.2: The improved precision of the region aware *mix* derived from the *R-DRF* semantics allows it to prove the additional assertion at line 11 in Figure 1.2.

Chapter 6

Implementation and Experiments

6.1 RATCOP: Relational Analysis Tool for COncurrent Programs

In this section, we perform a thorough empirical evaluation of our analyses using a prototype analyzer which we have developed, called RATCOP[1], for the analysis of race-free concurrent Java programs. RATCOP comprises around 4000 lines of Java code, and implements a variety of relational analyses based on the theoretical underpinnings described in earlier sections of this paper. Through command line arguments, each analysis can be made to use any one of the following three numerical abstract domains provided by the Apron library [46]: Convex Polyhedra (with support for strict inequalities), Octagons and Intervals. RATCOP also makes use of the Soot [79] analysis framework. The tool reuses the code for fixed point computation and the graph data structures in the implementation of [22].

The tool takes as input a Java program with assertions marked at appropriate program points. We first checked all the programs in our benchmarks for dataraces and region races using Chord [62]. For detecting region races, we have implemented the translation scheme outlined in Section 5.2. RATCOP then performs the necessary static analysis on the program until a fixpoint is reached. Subsequently, the tool automatically tries to prove the assertions using the inferred facts (which translates to checking whether the inferred fact at a program point implies the assertion condition): if it fails to prove an assertion, it dumps the corresponding inferred fact in a log file for manual inspection. Figure 6.1 summarizes the set of operations in RATCOP.

[1] The project artifacts are available at `https://bitbucket.org/suvam/ratcop`

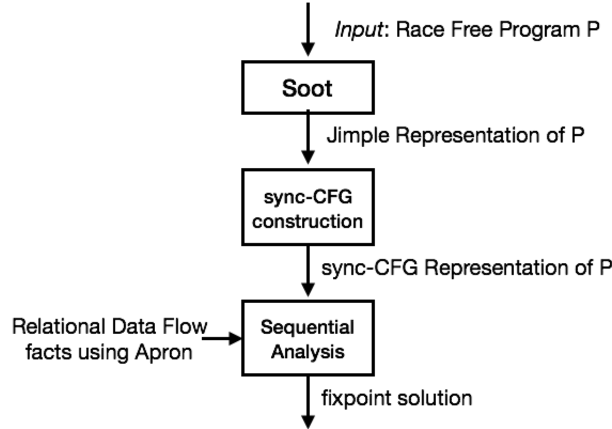

Figure 6.1: Overview of RATCOP.

As benchmarks, we use a subset of concurrent programs from the SV-COMP 2015 suite [9]. We ported the programs to Java and introduced locks appropriately to remove races. As we mentioned earlier, we use [62] to ensure the absence of races. We also use a program from [58], which is abstraction of a producer-consumer scenario. While these programs are not too large, they have challenging invariants to prove, and provide a good test for the precision of the various analyses. We ran the tool in a virtual machine with 16GB RAM and 4 cores. The virtual machine, in turn, ran on a machine with 32GB RAM and a quad-core Intel i7 processor. We evaluate 5 analyses on the benchmarks, with the following abstract domains:

1. **A1:** Without regions and thread identifiers [1].

2. **A2:** With regions, but with no thread identifiers.

3. **A3:** Without regions, but with thread identifiers.

4. **A4:** With regions and thread identifiers.
   The analyses **A1** - **A4** all employ the Octagon numerical abstract domain. And finally,

5. **A5:** The value-set analysis of [22], which uses the Interval domain.

In terms of the precision of the abstract domains, the analyses form the following partial order: **A5** $\prec$ **A1** $\prec$ **A3** $\prec$ **A4** and **A5** $\prec$ **A1** $\prec$ **A2** $\prec$ **A4**. We use **A5** as the baseline.

[1]By thread-identifiers we are referring to the abstraction of the versions outlined in Remark Section 4.5

6.2 Evaluation

6.2.1 Porting Sequential Analyses to Concurrent Analyses.

For the sequential commands, we perform a lightweight parsing of statements and simply re-use the built-in transformers of Apron. The only operator we need to define afresh is the abstract *mix*. Since Apron exposes functions to perform each of the constituent steps, implementing the abstract *mix* is straight forward as well.

6.2.2 Precision and Efficiency.

Fig. 6.1 summarizes the results of the experiments.

Program	LOC	Threads	Asserts	A1		A2		A3		A4		A5	
				✓	Time (ms)	✓	Time (ms)	✓	Time (ms)	✓	Time (ms)	✓	Time (ms)
reorder_2	106	5	2	0(C)	77	2(C)	43	0(C)	71	2(C)	37	0	25
sigma [B]*	118	5	5	0	132	0	138	4	48	4	50	0	506
sssc12	98	3	4	4	76	4	90	4	82	4	86	2	28
unverif	82	3	2	0	115	0	121	0	84	0	86	0	46
spin2003	65	3	2	2	6	2	9	2	10	2	10	2	8
simpleLoop	74	3	2	2	56	2	61	2	57	2	64	0	27
simpleLoop5	84	4	1	0	40	0	50	0	31	0	37	0	20
doubleLock_p3	64	3	1	1	11	1	24	1	16	1	19	1	9
fib_Bench	82	3	2	0	138	0	118	0	129	0	102	0	56
fib_Bench_Longer	82	3	2	0	95	0	103	0	123	0	91	0	35
indexer	119	2	2	2	1522	2	1637	2	1750	2	1733	2	719
twostage_3 [B]	93	2	2	0	61	0	48	0	57	0	28	0	59
singleton_with_uninit	59	2	1	1	31	1	29	1	14	1	10	1	28
stack	85	2	2	0	151	0	175	0	127	0	129	0	71
stack_longer	85	1	2	0	1163	0	669	0	1082	0	1186	0	597
stack_longest	85	2	2	0	1732	0	1679	0	1873	0	2068	0	920
sync01 *	65	2	2	2	7	2	25	2	37	2	33	2	10
qw2004 *	90	2	4	0	1401	4	1890	0	1478	4	1913	0	698
[58] Fig. 3.11	89	2	2	0	49	2	46	0	54	2	36	0	19
Total	**1625**	**3 (Avg)**	**42**	**14**	**361 (Avg)**	**22**	**366 (Avg)**	**18**	**374 (Avg)**	**26**	**406 (Avg)**	**10**	**204 (Avg)**

Table 6.1: Summary of the experiments. Superscript [B] indicates that the program has an actual bug. (C) indicates the use of Convex Polyhedra as abstract data domain. "*" indicates a program where we have altered/weakened the original assertion. The ✓ column indicates the number of assertions the tool was able to prove.

While all the analyses failed to prove the assertions in `reorder_2`, **A2** and **A4** were able to prove them when they used convex polyhedra instead of octagons. Since none of the analyses track arrays precisely, all of them failed to prove the original assertion in `sigma` (which involves checking a property involving the sum of the array elements). However, **A3** and **A4** correctly detect a potential array out-of-bounds violation in the program, by indicating that the loop

index can, in fact, equal the length of the array. The improved precision is due to the fact that **A3** and **A4** track thread identifiers in the abstract state, which avoids spurious read-write cycles in the analysis of `sigma`. The program `twostage_3` has an actual bug, and the assertions are expected to fail. This program provides a "sanity check" of the soundness of the analyses. Programs marked with "*"" contain assertions which we have altered completely and/or weakened. In these cases, the original assertion was either expected to fail or was too precise (possibly requiring a disjunctive domain in order to prove it). In `qw2004`, for example, we prove assertions of the form $x = y$. **A2** and **A4** perform well in this case, since we can specify a region containing x and y, which precisely track their correlation across threads. The imprecision in the remaining cases are mostly due to the program requiring *disjunctive* domains to discharge the assertions, or the presence of spurious write-write cycles which weakens the inferred facts.

Of the total 40 "valid" assertions (excluding the two in `twostage_3`), **A4** is the most precise, being able to prove 65% of them. It is followed by **A2** (55%), **A3** (45%), **A1** (35%) and, lastly, **A5** (25%). Thus, the new analyses derived from *L-DRF* and *R-DRF* perform significantly better than the value-set analysis of [22]. Moreover, this total order respects the partial ordering between the analyses defined earlier.

With respect to the running times, the maximum time taken, across all the programs, is around 2 seconds, by **A4**. **A5** turns out to be the fastest in general, due to its lightweight abstract domain. **A2** and **A4** are typically slower that **A1** and **A3** respectively. The slowdown can be attributed to the additional tracking of regions by the former analyses.

6.3 Comparing with a current abstract interpretation based tool.

We also compared the efficiency of RATCOP with that of Batman, a tool implementing the previous state-of-the-art analyses based on abstract interpretation [59, 60] (a discussion on the precision of our analyses against those in [59] is presented in Chapter 7). The basic structure of the benchmark programs for this experiment is as follows: each program defines a set of shared variables. A `main` thread then partitions the set of shared variables, and creates threads which access and modify variables in a unique partition. Thus, the set of memory locations accessed by any two threads is disjoint. In our experiments, each thread simply performed a sequence of writes to a specific set of shared variables. In some sense, these programs represent a "best-case" scenario because there are no interferences between threads. Unlike RATCOP, the Batman tool, in its current form, only supports a small toy language and does not provide

the means to automatically check assertions. Thus, for the purposes of this experiment, we only compare the time required to reach a fixpoint in the two tools. We compare **A3** against Batman running with the Octagon domain and the BddApron library [45] (Bm-oct).

#Threads	A3 Time (ms)	Bm-oct Time (ms)
2	61	7706
3	86	82545
4	138	507663
5	194	2906585
6	261	13095977
7	368	53239574

Table 6.2: Running times of RATCOP (**A3**) and Batman (Bm-oct) on loosely coupled threads. The number of shared variables is fixed at 6.

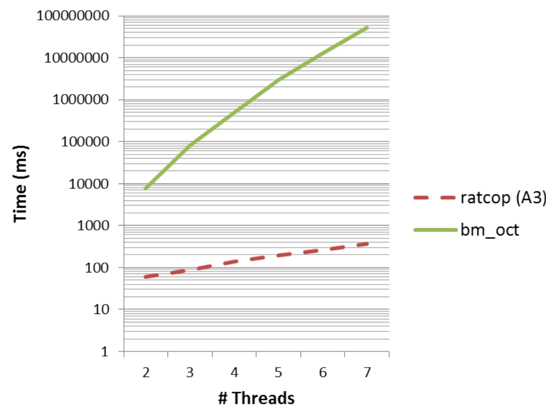

Figure 6.2: Running times, on a log scale, of RATCOP (**A3**) and Batman (Bm-oct) on loosely coupled threads. The number of shared variables is fixed at 6.

The running times of the two analyses are given in Fig. 6.2. The graph in 6.2 plots the running times of the two analyses on a logarithmic scale.

In the benchmarks, with increasing number of threads, RATCOP was upto 5 orders of magnitude faster than Bm-oct. The rate of increase in running time was almost linear for RATCOP, while it was almost exponential for Bm-oct. Unlike RATCOP, the analyses in [59, 60] compute sound facts at *every* program point, which contributes to the slowdown.

Chapter 7

Related Work and Discussion

In the first part of the thesis, we presented a framework for developing data-flow analyses for data race free shared-memory concurrent programs, with a statically fixed number of threads, and with variables having primitive data types.

There is a rich literature on concurrent data flow analyses and Rinard [65] provides a detailed survey of some of them. We compare some of the relevant ones in this section.

Degree of Inter-thread Communication. Chugh et al [16] automatically lifts a given sequential analysis to a sound analysis for concurrent programs, using a datarace detector. However, data-flow facts are not communicated across threads, and this can lose a lot of precision. The work by Mine [57] allows a greater degree of inter-thread communication. Here, the overall analysis can be thought of to proceed in rounds of thread-modular analyses. At the end of each round, every thread generates a set of per-thread "interferences"- for each variable x, a thread t stores the set of values it writes to x when t was analyzed modularly. In the next iteration, each thread $t' \neq t$ takes into account this interference information from t, whenever it reads x. This, in turn, generates more interferences for t', and the process continues till fixpoint. Thus, the inter-thread communication is flow insensitive. Unlike our semantics, this analysis is unable to infer relational properties between variables. Mine [59] presents an abstract interpretation formulation of the rely-guarantee proof paradigm [48, 88], and allows one to derive analyses with varying degrees of inter-thread flow sensitivity. In particular, the work in [57] is shown to be an abstraction of the semantics in [59]. The semantics in [59] involves a nested fixed-point computation, compared to our single fixed-point formulation. The resulting analysis aims to be sound at *all* program points (e.g, in Fig. 1.2 the value of y at line 9 in t_2), due to which many more interferences will have to be propagated than we do, leading to a

less efficient analysis. The times clocked by Batman, in comparison to RATCOP, is testament to this. [59] attempts to retrieve some degree of efficiency by computing “lock invariants”, which are essentially summaries of each critical section. However, to make use of this, the program must be well-synchronized- every access of a shared variable must be protected by a lock, which is a stronger requirement than data race freedom. Moreover, for certain programs, our abstract analyses are more precise. Fig. 7.1 shows a program which is race free, even though the conflicting accesses to x in lines 2 and 12 are not protected by a common lock. The “lock invariants” in [59] would consider these accesses as potentially racy, and would allow the read at line 12 to observe the write at line 2, thereby being unable to prove the assertion. However, our analyses would ensure that the read only observes the write at line 11, and is able to prove the assertion. [33] presents an operational semantics for concurrent programs, parameterized by a relation. It makes additional assumptions about code regions which are unsynchronized (allowing only read-only shared variables and local variables in such regions). Moreover, it too computes sound facts at every point, resulting in less efficient abstractions. In this sense, De et al [22] strikes a sweet spot: by leveraging the race freedom assumption, the analysis restricts data flow facts to synchronization points alone, thereby gaining efficiency. However, this work cannot compute relational information either, being based on a cartesian value-set domain.

Data Structure of Program Representation. The methods described in [25, 39, 22] present concurrent data flow algorithms by building specialized concurrent flow graphs. However, the class of analyses they address are restricted – [25] handles properties expressible as Quantified Regular Expressions, [39] handles reaching definitions, while [22] only handles value-set analyses. While our analyses also makes use of the *sync*-CFG data structure of [22], the *L-DRF* and *R-DRF* semantics allows us to use it in conjunction with much more expressive abstract domains. In contrast to our approach, the techniques in [30, 32] provide an approach to verifying properties of concurrent programs using *data flow* graphs, rather than use control flow graphs like we do.

Resource Invariants vs. Regions. A traditional approach to analyzing concurrent programs involves *resource invariants* associated with every lock (e.g. Gotsman et al [38]). This approach depends on a *locking policy* where a thread only accesses global data if it holds a protecting lock. In contrast, our approach does not require a particular locking policy (e.g., see Fig. 7.1), and is based on a parameterized notion of data-race-freedom, which allows to encode locking policies as a particular case. Thus, our new semantics provides greater flexibility to analysis writers, at the cost of assuming data race freedom. The analysis in [38] also works in similar spirit as the *sync*-CFG a selected part of the heap protected by a lock is made

```
    Thread1() {
1:  acquire(m);
2:  x := 1;
3:  y := 1;
4:  release(m);
5:  }
```

```
     Thread2() {
 6:  while( p != 1 ) {
 7:    acquire(m);
 8:    p := y;
 9:    release(m);
10:  }
11:  x := 2;
12:  p := x;
13:  assert(p != 1);
14:  }
```

Figure 7.1: Example demonstrating that a program can be DRF, when when the accesses of a global variable (in this case, the write and read of x at lines 11 and 12 respectively) are not directly guarded by any lock.

accessible to a thread only when it acquires the lock. In contrast, the synchronization edges in a *sync*-CFG propagates the *entire* data flow fact. The locking policy employed by [38] is stronger than the notion of race freedom, and the class of programs the analysis can handle is a subset of what we handle in this work.

Region Races. Our notion of region races is inspired by the notion of high-level data races [8]. The concept of splitting the state space into regions was earlier used in [55], which used these regions to perform shape analysis for concurrent programs. However, that algorithm still performs a full interleaving analysis which results in poor scalability. The notion of variable packing [10] is similar to our notion of data regions. However, variable packs constitute a purely *syntactic* grouping of variables, while regions are semantic in nature. A syntactic block may not access all variables in a semantic region, which would result in a region partitioning more refined than what the programmer has in mind, which would result in decreased precision.

As future work, we would like to evaluate the performance of our tool when equipped with disjunctive relational domains. In this work, we do not consider dynamically allocated memory, and extending the *L-DRF* semantics to account for the heap memory is interesting future work. Abstractions of such a semantics could potentially yield efficient shape analyses for race free concurrent programs.

Part II

Detecting all High-Level Data Races in an RTOS Kernel

Chapter 8

The architecture of FreeRTOS

In this part of the thesis, we address the problem of detecting atomicity violations in the library functions of an RTOS kernel. We introduce the notion of a *high-level* data race, which occurs when the execution of an application interleaves instructions corresponding to user-annotated critical accesses of shared-memory data structures. Such races are a necessary condition for atomicity violations. We propose a technique for detecting *all* high-level data races in a system library like the kernel API of a real-time operating system (RTOS) that relies on flag-based scheduling and synchronization. Our methodology is based on model-checking, but relies on a meta-argument to bound the number of concurrent tasks needed to orchestrate a race.

We describe our approach in the context of FreeRTOS, a popular real-time operating system in the embedded systems domain. FreeRTOS is representative of other operating systems in its class. Thus, techniques we outline here generalize to other concurrent libraries which permit interrupts and make use of flags to control synchronization and scheduling.

In this chapter, we provide an overview of the architecture of FreeRTOS. We highlight the key data structures and API functions using a small application, and snippets from the actual library functions.

8.1 Overview of FreeRTOS

FreeRTOS [64] is a real-time kernel meant for use in embedded applications that run on microcontrollers with small to mid-sized memory.

It allows an application to organise itself into multiple independent tasks (or threads) that will be executed according to a priority-based preemptive scheduling policy. It is implemented

This work was done in collaboration with Arun Kumar and Deepak D'Souza, at the Indian Institute of Science.

```
int main(void) {
  QueueHandle q;
  q = QueueCreate(1, sizeof(int));
  TaskCreate(prod, "Prod", 2, ...);
  TaskCreate(cons, "Cons", 1, ...);
  StartScheduler();
}

void prod(void* params) {
  for(;;) {
    QueueSend(q,...);
    TaskDelay(2);
  }
}

void cons(void* params) {
  for(;;) {
    QueueReceive(q,...);
  }
}
```

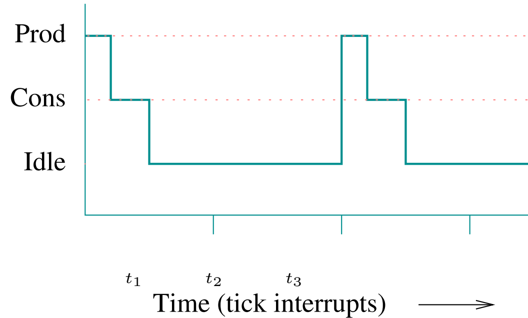

Figure 8.1: An example FreeRTOS application and its execution

as a library of functions (or an API) written mostly in C (some parts of the OS are written at assembly level), that an application programmer can include with their code and invoke as functions. The API provides the programmer the means to create and schedule tasks, communicate between tasks (via message queues, semaphores, etc), and carry out time-constrained blocking of tasks.

Fig. 8.1 shows a simple FreeRTOS application. In `main` the application first creates a queue with the capacity to hold a single message of type `int`. It then creates two tasks called "`Prod`" and "`Cons`" of priority 2 and 1, respectively, using the `TaskCreate` API function, which adds these two tasks to the "Ready" list. The FreeRTOS scheduler is then started by the call to `StartScheduler`. The scheduler schedules the `Prod` task first, it being the highest priority ready task. `Prod` sends a message to the queue, and then asks to be delayed for two time units. This results in `Prod` being put into the "Delayed" list. The next available task, `Cons`, is run next. It dequeues the message from the queue, but is blocked when it tries to dequeue again. The scheduler now makes the `Idle` task run. A timer interrupt now occurs, causing an Interrupt Service Routine (ISR) called `IncrementTick` to be run. This routine increments the current tick count, and checks the delayed list to see if any tasks need to be woken up. There are none, so the `Idle` task resumes execution. However when the second tick interrupt occurs, the ISR finds that the `Prod` task needs to be woken up, and moves it to the ready

list. As Prod is now the highest priority ready task, it executes next. This cycle repeats, ad infinitum.

The FreeRTOS kernel maintains a bunch of data-structures, variables and flags, some of which are depicted in Fig. 8.2. Tasks that are ready to run are kept in the ReadyTasksList, an array which maintains—for each priority—a pointer to a linked list of tasks of that priority that are ready to run. When a running task delays itself, it is moved from the ReadyTasksList to the DelayedTaskList, with an appropriate time-to-awake value. User-defined queues, like q in the example application, are maintained by the kernel as a chunk of memory to store the data (shown as QueueData in the figure), along with an integer variable MessagesWaiting that records the number of messages in the queue, and two associated lists WaitingToSend and WaitingToReceive that, respectively, contain the tasks that are blocked on sending to and receiving from the queue.

Even though FreeRTOS applications typically run on a *single* processor (or a *single core* of a multi-core processor), the kernel API functions can interact with each other in an interleaved manner. While a function invoked by the current task is running, there could be an interrupt due to which an ISR runs, which in turn may either invoke another API function, or unblock a higher priority task which goes on to execute another API function. The FreeRTOS API functions thus need to use some kind of synchronization mechanism to ensure "exclusive" access to the kernel data-structures. They do so in a variety of ways, to balance the trade-off between securing fully exclusive access and not losing interrupts. The strongest exclusion is achieved in a "critical section," where an API function disables interrupts to the processor, completes its critical accesses, and then re-enables interrupts. During such a critical section no preemption (and hence no interleaving) is possible. The second kind of exclusion is achieved by "suspending" the scheduler. This is done by setting the kernel flag SchedulerSuspended to 1. While the scheduler is suspended (i.e. this flag is set), no other task will be scheduled to run; however, unlike in a critical section, *interrupts* can still occur and an ISR can execute some designated API functions (called "fromISR" functions which are distinguished from the other "task" functions). The implicit protocol is that these functions will check whether the SchedulerSuspended flag is set, and if so they will not access certain data-structures like the ReadyTasksList, but move tasks when required to the PendingReadyList instead. Fig. 8.2 shows some of the structures protected by the SchedulerSuspended flag.

The final synchronization mechanism used in FreeRTOS is a pair of per-user-queue "locks" (actually *flags* which also serve as *counters*) called RxLock and TxLock, that protect the WaitingToReceive and WaitingToSend lists associated with the queue. When a task

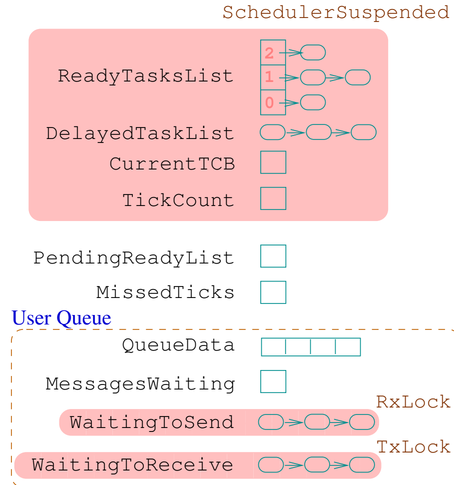

Figure 8.2: Kernel data-structures in FreeRTOS. The data structures within the upper rectangle are protected by the `SchedulerSuspended` flag. `RxLock` and `TxLock` protects the `WaitingToSend` and `WaitingToReceive` data structures, respectively.

executes an API function that accesses a user-queue, the function sets these locks (increments them from their initial value of -1 to 0). Any fromISR function that now runs will now avoid accessing the waiting lists associated with this queue, and instead increment the corresponding lock associated with the queue to record the fact that data has been added or removed from the queue. We would like to iterate once again that `RxLock` and `TxLock` are merely counters, and are not actual locks. When the interrupted function resumes, it will move a task from the waiting list back to the ready list, for each increment of a lock done by an ISR. These locks and the lists they protect are also depicted in Fig. 8.2.

Fig. 8.3 shows parts of the implementation of two library functions in the FreeRTOS API. The `QueueSend` function is used by a task to enqueue an item in a user-defined queue `pxQ`. Lines 3–9 are done with interrupts disabled, and corresponds to the case when there is available space in the queue: the item is enqueued, a task at the head of `WaitingToReceive` is moved to the `ReadyTasksList`, and the function returns successfully. In lines 14–26, which corresponds to the case when the queue is full, the function enables interrupts, checks again that the queue is still full (since after enabling interrupts, an ISR could have removed something from the queue), and goes on to move itself from the Ready queue to the `WaitingToSend` list of `pxQ`. This whole part is done by first suspending the scheduler and locking `pxQ`, and finally unlocking the queue and resuming the scheduler. The call to `LockQueue` in line 16 increments both `RxLock` and `TxLock`. The call to `UnlockQueue` in line 21 decrements `RxLock` as many times as its value exceeds 0, each time moving a task (if present) from `WaitingToSend` to Ready. It does a similar sequence of steps with `TxLock`. Both these functions first disable

```
int QueueSend(QHandle pxQ, void *ItemToQueue) {
 1  // Repeat till successful send
 2  DISABLE__INTERRUPTS();
 3  if(!QueueFull(pxQ)) { // Queue is not full
 4    // Copy data to queue
 5    CopyDataToQueue(pxQ, ItemToQueue);
 6    if(!empty(pxQ->WaitingToReceive)) {
 7      ... // Move task from WaitingToReceive
 8      ... // to ReadyTasksList
 9    }
10    ENABLE__INTERRUPTS();
11    return PASS;
12  }
13  // Reach here when queue is full
14  ENABLE__INTERRUPTS();
15  ++SchedulerSuspended; // Suspend scheduler
16  LockQueue(pxQ);//Inc Tx(Rx)Lock with ints disabled
17  if(QueueFull(pxQ)) { // Check if queue still full
18    ... // Move current task from ReadyTasksList
19    ... // to WaitingToSend
20  }
21  UnlockQueue(pxQ);//Move tasks from waiting lists
22    // and unlock, with ints disabled
23  --SchedulerSuspended; // Resume scheduler
24  if (...) { // higher priority task woken
25    YIELD();
26  }
}

void IncrementTick() {
 1  if(SchedulerSuspended == 0) {
 2    ++TickCount;
 3    if(TickCount == 0) {
 4      ... // swap delayed lists
 5      DelayedTaskList = OverflowDelayedTaskList;
 6    }
 7    ... // Move tasks whose time-to-awake is now,
 8    ... // from DelayedTaskList to ReadyTasksList.
 9  }
10  else {
11   ++MissedTicks;
12  }
}
```

Figure 8.3: Excerpts from FreeRTOS functions

interrupts and re-enable them once their job is done. Finally, in lines 24–26, the function checks to see if it has unblocked a higher priority task, and if so “yields” control to the scheduler.

The second API function in Fig. 8.3 is the `IncrementTick` function that is called by the timer interrupt, and which we consider to be in the fromISR category of API functions. If the scheduler is *not* suspended, it increments the `TickCount` counter, and moves tasks in the `DelayedTaskList` whose time-to-awake equals the current tick count, to the Ready list. If the scheduler *is* suspended, it simply increments the `MissedTicks` counter.

Chapter 9

Atomicity Violations, and High-Level Data Races

9.1 $(\mathcal{S}, \mathcal{C})$ Races

In this chapter we describe our notion of a high-level race in a system library like FreeRTOS. Essentially a race occurs when two "critical" access paths in two API functions interleave. We make this notion more precise below.

Consider a system library L. Our notion of a race in L is parameterized by a set $\mathcal{S}$ of shared memory structures maintained by the library, and a set $\mathcal{C}$ of "critical accesses" of structures in $\mathcal{S}$. The set of structures in $\mathcal{S}$ is largely determined by the developer's design for thread-safe access. We can imagine that the developer has in mind a partitioning of the shared memory structures into "units" which can be independently accessed: thus, it is safe for two threads to simultaneously access two *distinct* units, while it is potentially unsafe for two threads to access the *same* unit simultaneously. For instance, in FreeRTOS, the set $\mathcal{S}$ could contain shared variables like `SchedulerSuspended`, or shared data-structures like `ReadyTasksList`, or an entire user-queue. The set of critical accesses $\mathcal{C}$ would comprise contiguous blocks of code in the API functions of L, each of which corresponds to an access of one of these units in $\mathcal{S}$. The accesses are "critical", in the sense that they are *not* meant to interleave with other accesses to the same structural unit. Each critical access comes with a classification of being a *write* or *read* access to a particular shared structure v in $\mathcal{S}$. For example, we could have the block of code in lines 17–20 of the `QueueSend` function in Figure 8.3, reproduced in Figure 9.1, as a critical write to the user-queue structure, in $\mathcal{C}$.

Finally, we say that a pair of accesses in $\mathcal{C}$ are *conflicting* if they both access the same

```
17  if(QueueFull(pxQ)) { // Check if queue still full
18     ... // Move current task from ReadyTasksList
19     ... // to WaitingToSend
20  }
```

Figure 9.1: Extract from the `QueueSend` function in Figure 8.3. These instructions constitute a critical write to the user-queue data structure.

structure v in $\mathcal{S}$ and at least one is a write access.

An execution of an application program A that uses L—an *L-execution* for short—is an interleaving of the execution of the tasks (or threads) it creates. An execution of a task in turn is a (feasible) sequence of instructions that follows the control-flow graph of its compiled version. Since these tasks may periodically invoke the functions in L, portions of their execution will correspond to the critical paths in these functions. We say that an L-execution exhibits an $(\mathcal{S}, \mathcal{C})$ *high-level race* (or just $(\mathcal{S}, \mathcal{C})$-*race* for short) on a structure v in $\mathcal{S}$, if it *interleaves* the execution paths corresponding to two conflicting critical accesses to v (i.e. the second critical access begins before the first ends).

When do we say an $(\mathcal{S}, \mathcal{C})$-race is "harmful"? We can use the notion of atomicity violation from [36] (see also [31]) to capture this notion. Consider an L-execution ρ. Each task in the application may invoke functions in L along ρ, and some of these invocations may overlap (or interleave) with invocations of functions of L in other tasks. A *linearized* version of ρ follows the same sequence of invocations of the functions in L along ρ, except that *overlapping* invocations are re-ordered so that they no longer overlap. We refer the reader to [43] for a more formal definition of linearizability. We can now say that an L-execution ρ exhibits an *atomicity violation* if there is *no* linearized version of the execution that leaves the shared memory structures in the same state as ρ. This definition differs slightly from [36] in that we prefer to use the notion of linearizability rather than serializability.

For a given $(\mathcal{S}, \mathcal{C})$, we say that an $(\mathcal{S}, \mathcal{C})$-race is *harmful* if there is an L-execution that contains this race, exhibits an atomicity violation, and this race plays a role (possibly along with other threads) in producing this atomicity violation. Otherwise we say the race is *benign*. Finally, we say that a given $(\mathcal{S}, \mathcal{C})$ pair is *safe* for L, if every L-execution that exhibits an atomicity violation also exhibits an $(\mathcal{S}, \mathcal{C})$-race. We note that we can always obtain a safe $(\mathcal{S}, \mathcal{C})$ by putting all memory structures into a single unit in $\mathcal{S}$ and entire method bodies into $\mathcal{C}$. However this would lead to lots of false positives, and it is thus preferable to have as finely-granular an $(\mathcal{S}, \mathcal{C})$ as possible.

We now proceed to describe our choice of what we believe to be safe choice of $\mathcal{S}$ and $\mathcal{C}$ for

FreeRTOS. Some natural candidates for units in $\mathcal{S}$ are the various task lists like ReadyTasksList and DelayedTaskList. For a user-defined queue, one could treat the entire queue—comprising QueueData, MessagesWaiting, and the WaitingToSend and WaitingToReceive lists—as a single unit. However, this view would go against the fact that, by design, a task could be accessing the WaitingToSend component, while an ISR accesses the QueueData component. Hence, we keep each component of a user-defined queue as a separate unit in $\mathcal{S}$. Finally, we include all shared flags like SchedulerSuspended, pointer variables like CurrentTCB, and counters and locks like TickCount and xRxLock, in $\mathcal{S}$. Corresponding to this choice of units in $\mathcal{S}$, we classify, for example, the following blocks of code as critical accesses in $\mathcal{C}$: line 3 of the QueueSend function (Figure 9.2) as a read access of MessagesWaiting, line 5 as a write to QueueData, line 6 as a read of WaitingToReceive, and lines 7–8 as a write to both WaitingToReceive and ReadyTasksList.

9.2 Examples of $(\mathcal{S}, \mathcal{C})$ races in FreeRTOS

We now give a couple of examples of races with respect to the set of structures $\mathcal{S}$ and accesses $\mathcal{C}$ described above. Assume for the sake of illustration, that the QueueSend function did *not* disable interrupts in line 2. The modified QueueSend function is shown in Figure 9.2. Consider an execution of the example application in Figure 8.1, reproduced here in Figure 9.2, in which the Prod task calls the QueueSend function, and begins the critical write to ReadyTasksList. At this point a timer interrupt comes and causes the IncrementTick ISR to run and execute the critical write to ReadyTasksList in lines 7–8. This execution would constitute a race on ReadyTasksList.

As a second example, consider the write access to SchedulerSuspended in the equivalent of line 15 of the QueueSend function (see Figure 9.2) in QueueReceive, and the read access of the same variable in line 1 of IncrementTick (an excerpt of IncrementTick is shown in Figure 9.3).

Then an execution of the example application in Figure 9.2 in which Cons calls the QueueReceive function when the queue is empty and executes the equivalent of line 15 to suspend the scheduler, during which it is interrupted by the IncrementTick ISR which goes on to execute line 1. This execution constitutes a race between the QueueReceive and IncrementTick API functions on the SchedulerSuspended variable.

The race on ReadyTasksList above is an example of a harmful race since it could lead to the linked list being in an inconsistent state that cannot be produced by any linearization of the execution. The race on SchedulerSuspended turns out to be benign, essentially due to

```
int QueueSend(QHandle pxQ, void *ItemToQueue) {
 1  // Repeat till successful send
 2  // DISABLE__INTERRUPTS();
 3  if(!QueueFull(pxQ)) { // Queue is not full
 4    // Copy data to queue
 5    CopyDataToQueue(pxQ, ItemToQueue);
 6    if(!empty(pxQ->WaitingToReceive)) {
 7      ... // Move task from WaitingToReceive
 8      ... // to ReadyTasksList
 9    }
10    // ENABLE__INTERRUPTS();
11    return PASS;
12  }
13  // Reach here when queue is full
14  // ENABLE__INTERRUPTS();
15  ++SchedulerSuspended; // Suspend scheduler
16  LockQueue(pxQ);//Inc Tx(Rx)Lock with ints
   disabled
17  if(QueueFull(pxQ)) { // Check if queue still full
18    ... // Move current task from ReadyTasksList
19    ... // to WaitingToSend
20  }
21  UnlockQueue(pxQ);//Move tasks from waiting lists
22    // and unlock, with ints disabled
23  --SchedulerSuspended; // Resume scheduler
24  if (...) { // higher priority task woken
25    YIELD();
26  }
}
```

```
int main(void) {
  QueueHandle q;
  q = QueueCreate(1, sizeof
      (int));
  TaskCreate(prod, "Prod",
      2, ...);
  TaskCreate(cons, "Cons",
      1, ...);
  StartScheduler();
}

void prod(void* params) {
  for(;;) {
    QueueSend(q,...);
    TaskDelay(2);
  }
}

void cons(void* params) {
  for(;;) {
    QueueReceive(q,...);
  }
}
```

Figure 9.2: The code on the left is a version of the `QueueSend` library function, with the interrupts *disabled* at line 2. The code on the right is the example application presented in Chapter 8.

```
void IncrementTick() {
 1  if(SchedulerSuspended == 0) {
      ...
 6    }
 7    ... // Move tasks whose time-to-awake is now,
 8    ... // from DelayedTaskList to ReadyTasksList.
 9  }
10  else {
11   ++MissedTicks;
12  }
}
```

Figure 9.3: Excerpt of the `IncrementTick` function outlined in Figure 8.3.

the variable being declared to be *volatile* (so reads/writes to it are done directly from memory), and fact that an ISR runs to *completion* before we can switch back to `QueueReceive`.

Chapter 10

Modelling FreeRTOS in Spin

In this chapter we describe how we model the FreeRTOS API and check for $(\mathcal{S}, \mathcal{C})$-races using the model-checking tool Spin [44]. Spin's modeling language Promela can be used to model finite-state concurrent systems with standard communication and synchronization mechanisms (like message channels, semaphores, and locks). One can then model-check the system model to see if it satisfies a given state assertion or LTL property. For more details on Spin we refer the reader to [44].

10.1 Modeling M_n

Our first aim is to generate a Promela model M_n which captures the possible interleavings of critical accesses in any FreeRTOS application with at most n tasks. To make this more precise, consider a FreeRTOS application that—along any execution—creates at most n tasks. We denote such an application by A_n. We now define a Promela model M_n that has the following property (**P**):

> For every execution of A_n, which exercises the critical accesses within the FreeRTOS API functions in a certain interleaved manner, there is a corresponding execution in M_n with a similar manner of interleaving.

For a given n, the Promela model M_n is built as follows. We introduce four semaphores called `task`, `sch`, `isr`, and `schsus` to model the possible control switches between processes. Recall that a (binary) semaphore has two possible states 0 and 1, and blocking operations `up` and `down` which respectively change the state from 0 to 1 and 1 to 0. Initially all the semaphores are down (i.e. 0) except `sch` which is 1 to begin with. The semaphores are used to indicate when a particular API function is enabled. For example when the `sch` semaphore is

up, the scheduler process—which first tries to down the `sch` semaphore—is enabled. Similarly, the `task` semaphore controls when a task function is enabled, and the `isr` semaphore controls when a fromISR function is enabled. The `schsus` semaphore is used to ensure that whenever a task function is interrupted while the scheduler is *suspended*, control returns to the interrupted task function only. Figure 10.1 provides a graphical representation of our model of the control flow between the various components of FreeRTOS: the scheduler, an arbitrary library function, and an ISR.

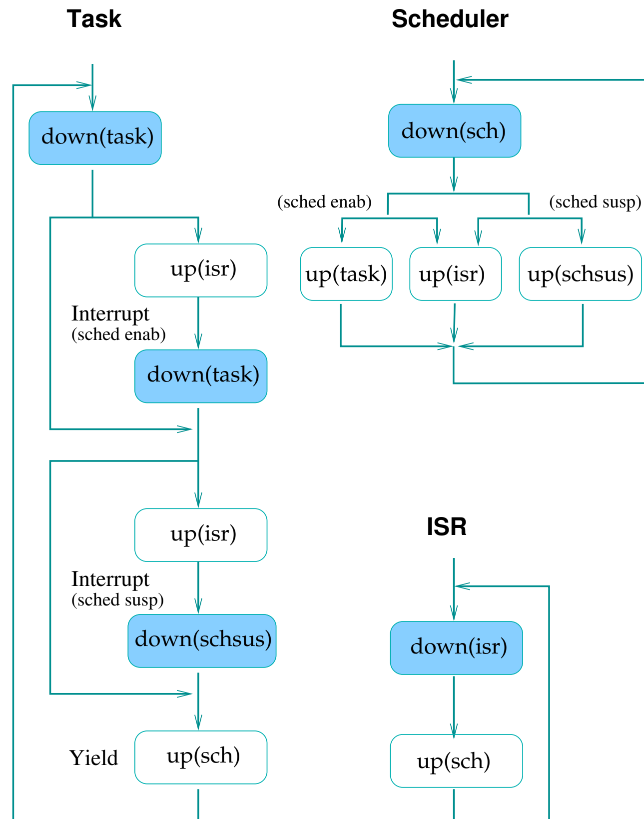

Figure 10.1: Model of the flow of control between the scheduler, a library function (task), and an ISR in FreeRTOS.

Each API function is modeled as a Promela function with the same name. We model variables of FreeRTOS that are critical to maintaining mutual exclusion, like `SchedulerSuspended`, `RxLock` and `TxLock`. We capture conditionals involving these variables and updates to these variables faithfully, and abstract the remaining conditionals conservatively to allow control-flow (non-deterministically) through both true and false branches of the conditional.

For each structure $\mathtt{v} \in \mathcal{S}$, we introduce a numeric variable called "`_v`", which is initialized to 0. For each critical write access to a structure `v` in an API function F, we add a statement `_v`

`+= 2` (short for `_v = _v +2`) at the beginning of the block, and the statement `_v -= 2` at the end of the block, in the Promela version of F. Similarly, for a read access of `v` we add the statements `_v++` and `_v--` at appropriate points in the function. The possible context-switches due to an interrupt or yield to the scheduler are captured by up-ing the `isr` or `sch` semaphore. In particular, at any point in a function where an interrupt can occur (i.e. whenever interrupts are *not* disabled or an ISR itself is running), we add a call to `interrupt()` which essentially up's the `isr` semaphore and waits till a down is enabled on the `task` semaphore. Figure 10.2 is our Promela model for the interrupt invocation.

```
inline interrupt() {
  if
  :: SchedulerSuspended==0 ->
        up(isr); down(task)
  :: SchedulerSuspended==1 ->
         up(isr); down(schsus)
   :: skip;
   fi
 }
```

Figure 10.2: Promela model of an interrupt. If the `SchedulerSuspended` flag is set, we ensure that the control returns to the preempted task. Otherwise, after the execution of the ISR, context may switch to an arbitrary ready task.

The Promela function corresponding to the `QueueSend` function is shown in Figure 10.3.

Each task in A_n is abstracted and conservatively modeled by a single process called `taskproc` in M_n, which repeatedly chooses a task API function non-deterministically and calls it. In a similar way, we model the fromISR API functions, and the `isrproc` process repeatedly invokes one of these non-deterministically. The Promela code of model M_n is depicted in Figure 10.6. Thus, M_n runs one `scheduler` process, one `isrproc` process, and n `taskproc` processes.

Let us define what we consider to be a race in M_n. Let the statements in M_n be $s_1, \ldots, s_m$. If statement s_i is part of the definition of API function F, we write $\Gamma(s_i) = F$. An execution of M_n is a sequence of these statements that follows the control-flow of the model, and is *feasible* in that each statement is *enabled* in the state in which it is executed. We say an execution ρ of M_n exhibits a data race on a structure `v`, involving statements s_i and s_j if (a) s_i and s_j are both increments of `_v`, (b) at least one increments `_v` by 2, and (c) ρ is of the form $\pi_1 \cdot s_i \cdot \pi_2 \cdot s_j$ with the segment π_2 not containing the decrement of `_v` corresponding to s_i. Note that the value of `_v` along ρ will exceed 2 after s_j.

It is not difficult to see that M_n satisfies the property (**P**) above. Consequently, any race on

```
inline QueueSend() {
  // do in a loop
  interrupt();
  // atomically, so no interrupts
  _MessagesWaiting++;
  _MessagesWaiting--;
  if
  :: skip ->
        // Copy data to queue
                _queueData += 2; _queueData -= 2;
                // Check if WaitingToReceive is non-empty
                _WaitingToReceive++; _WaitingToReceive--;
                // Move task from WaitingToReceive to Ready
                _WaitingToReceive += 2; _WaitingToReceive -= 2;
                _ReadyTasksList += 2; _ReadyTasksList -= 2;
        :: skip;
        fi
        // end of atomic, so interrupts enabled
        interrupt();
        if
        :: ++SchedulerSuspended; interrupt();
                LockQueue(); interrupt();
                // Move current task from Ready to WaitingToSend
                _ReadyTasksList += 2; interrupt(); _ReadyTasksList -= 2;
                _WaitingToSend += 2; interrupt(); _WaitingToSend -= 2;
                UnlockQueue(); interrupt();
                --SchedulerSuspended;  // Resume scheduler
                if
                :: up(sch); down(task); // Yield
                :: skip;
                fi
        :: skip;
        fi
}
```

Figure 10.3: The abstraction of the `QueueSend` library function in the Promela modeling language. For each shared data structure `x` we are interested in modeling, we introduce an integer variable _x. For example, we use `_queueData` above as an abstraction of the data component of a queue. Reads to `x` are modeled by an increment of _x by 1, followed by a decrement. Similarly, writes are modeled by an increase of _x by 2, followed by a decrease by 2. If, in addition, the accesses are made in a non-atomic section of code, the increment and decrement operations are interspersed by a call to the `interrupt` function, to model the fact that an interrupt invocation may occur while `x` is being accessed. The `Lock` and `UnLock` functions are described in Figure 10.4 and Figure 10.5 respectively.

a structure `v` $\in \mathcal{S}$ in application A_n will have a corresponding execution in M_n which exhibits a data race on _v. Thus, it follows that by model-checking M_n for the invariant

```
inline LockQueue() {
  //Inc Tx(Rx)Lock with ints disabled
  _TxLock +=2; TxLock++;
  _TxLock -= 2;
  _RxLock +=2; RxLock++;
  _RxLock -= 2;
}
```

Figure 10.4: The `LockQueue` function used in Figure 10.3.

```
inline UnlockQueue() {
  // atomic
  do
  :: TxLock > 0 -> ... --TxLock;
        //Move tasks from
        //WaitingToReceive to Ready
  :: TxLock = 0 -> break;
  od
  TxLock = -1; // unlock queue
  //end atomic
  interrupt();
  // atomic
  do
  :: RxLock > 0 -> ... --RxLock;
        //Move tasks from
        //WaitingToReceive to Ready
  :: RxLock = 0 -> break;
  od
  RxLock = -1; // unlock queue
  //end atomic
  interrupt();
}
```

Figure 10.5: The `UnlockQueue` function used in Figure 10.3.

```
((_ReadyTasksList < 3) && (_DelayedTaskList < 3) && ...)
```

we will find all races that may arise in an n-task application A_n. We note that there may be some false positives, due to conservative modeling of conditionals in the API functions, or because of 3 consecutive read accesses.

10.2 Strategy to identify *all* high-level races

There are now two hurdles in our path. The first is that we need to model-check M_n for *each* n, as it is possible that some races manifest only for certain values of n. Secondly, model-checking even a single M_n may be prohibitively time-consuming due to the large state-space of these

```
proctype scheduler() {
  do
  :: down(sch);
     if
     :: SchedulerSuspended==0 -> up(task)
     :: SchedulerSuspended==1 -> up(schsus)
     :: up(isr)
     fi
  od
}

proctype taskproc() {
  do
  :: down(task); QueueSend(); up(sch);
  :: ...
  :: down(task); TaskDelay(); up(sch);
  od
}

proctype isrproc() {
  do
  :: down(isr); IncrementTick(); up(sch);
  :: ...
  :: down(isr); QueueSendFromISR(); up(sch);
  od
}

init {
  run scheduler();
  // start n task and 1 ISR process
  run taskproc();
  ...;
  run taskproc();
  run isrproc();
}
```

Figure 10.6: Promela model M_n.

models. In fact, as we report in Sec. 12, Spin times out even on M_2, after running for several hours. We propose a way out of this problem, by first proving a meta-claim that any race between API functions F and G in M_n, will also manifest in a *reduced* model, $M_{F,G,I}$, in which we have a process that runs *only* F, one that runs *only* G, another that runs a fromISR function I, along with the `scheduler` process, and an ISR process that runs only the `IncrementTick` function. We denote this set of reduced models by $\mathcal{M}_{red}$. We then go on to model-check each of these reduced models for data races. Though there are now thousands of models to check, each one model-checks in a few seconds, leading to tractable overall running time.

In the next chapter we justify our meta-claim.

Chapter 11

Reduction to $\mathcal{M}_{red}$

Before we proceed with our reduction claim, we note that this claim may not hold for a general library. Consider, for example, the library L with three API functions F, G, and H shown in Fig. 11.1. Suppose the variable x belongs to the set of structures $\mathcal{S}$ and the lines 2 and 6 constitute a critical read and write access, respectively, to x. Then the $(\mathcal{S}, \mathcal{C})$-race on x involving these accesses will never show up in any reduced model in $\mathcal{M}_{red}$, since we need all three functions to execute in order to produce this race. Thus, as we do for FreeRTOS below, any choice regarding the structure of models in $\mathcal{M}_{red}$, and the argument for its sufficiency, must be tailored for a given library and the way it has been modeled.

```
F() {                     G() {                     H() {
1:  ...                   4:  ...                   7:  ...
2:  read(x);              5:  if(flag)              8:  flag = true;
3:  ...                   6:      write(x);         9:  ...
}                         }                         }
```

Figure 11.1: An example library where there is a potential data race between library functions F and G. However, any execution which causes this data race also needs to execute the library function H. Thus, a "reduction" to $\mathcal{M}_{red}$ does not suffice for this library.

We now describe the reduction claim for our FreeRTOS model:

Theorem 11.1 *Let $n \geq 1$, and let ρ be an execution of M_n exhibiting a race involving statements s_i and s_j of M_n. Then there exists a model $M \in \mathcal{M}_{red}$, and an execution ρ_{red} of M, which also exhibits a race on s_i and s_j.*

We denote the set of "fromISR" functions as L_{isr}. We make use of the following properties of the concurrency model for our proof.

- **P1.** The control flow path taken by any fromISR function $f \in L_{isr}$ is only governed by the value of the flags `SchedulerSuspended`, `TxLock`, and `RxLock`.
- **P2.** No $f \in L_{isr}$ modifies the value of `SchedulerSuspended`.
- **P3.** For the fromISR functions which do read the flags `TxLock`, the control flow path depends on whether `TxLock` $= -1$ or `TxLock` ≥ 0. In particular, while the functions do distinguish between the cases when `TxLock` is -1 and non-negative, they do not distinguish between whether `TxLock` $= 0$ or `TxLock` > 0. A similar property holds for `RxLock`.
- **P4.** If $f \in L_{isr}$ chooses the path guarded by the condition `TxLock` ≥ 0, then it exits in a state with `TxLock` > 0. In other words, the operations within this path strictly increase `TxLock`. A similar property holds for `RxLock`.
- **P5.** In an execution, if a task function F suspends the scheduler, then it is F which resumes it. No other task function can execute while the scheduler is suspended.
- **P6.** `LockQueue` and `UnlockQueue` operations are always bundled within a scheduler-disabled block of code (that is, with `SchedulerSuspended` $= 1$). Thus, in an execution, if a function F invokes the `LockQueue` operation, then it is F which invokes the corresponding `UnlockQueue`.
- **P7.** All operations *within* an `UnlockQueue` are within interrupt-disabled sections, and hence no preemption is possible. Consequently, no operation within an `UnlockQueue` operation can participate in a data-race.

In the proof below, we do away with the scheduler component and allow the ISRs to directly "up" the `api` and `schsus` semaphores. This is just to simplify the argument.

Proof: By the construction of M_n, the execution ρ must be of the form

$$\pi_1 \cdot \texttt{up(api)} \cdot \texttt{down(api)} \cdot \pi_2 \cdot s_i \cdot \texttt{up(isr)} \cdot \pi_3 \cdot s_k$$

where s_i is a statement in an API task function F, and $\texttt{down(api)} \cdot \pi_2 \cdot s_i \cdot \texttt{up(isr)}$ is the portion of ρ corresponding to the racy invocation of F. Figure 11.2 provides a diagrammatic representation of the situation under consideration. We note that s_i must be part of a task function, while s_k could be part of either a task or fromISR function.

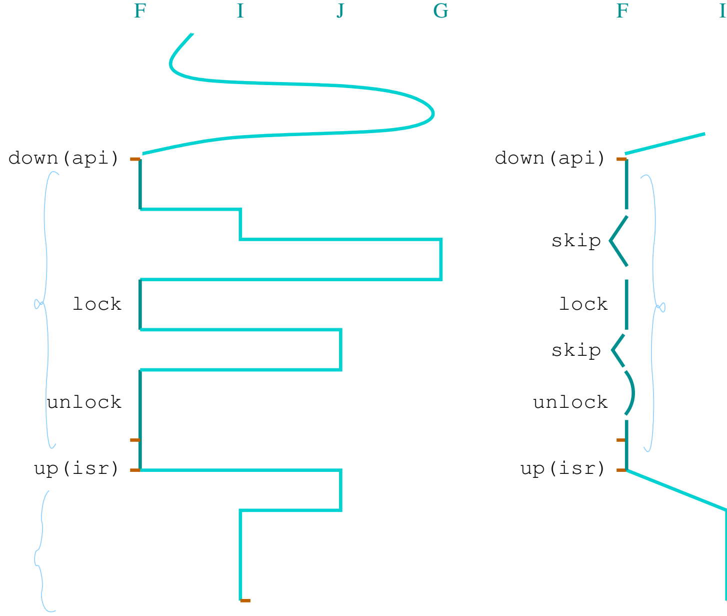

Figure 11.2: The execution ρ and its reduction ρ_{red}.

We consider two top-level cases: the first when s_k is in a fromISR function, and the second when it is in a task function.

Case 1. s_k is in a fromISR function I. In this case, the path π_3 must be of the form $\pi_3' \cdot$ `up(isr)` $\cdot\, \pi_4 \cdot s_k$, where π_3' comprises an interleaved sequence of fromISR functions and task functions, and $\pi_4 \cdot s_k$ is an initial path in I, beginning with a `down(isr)`.

We consider two further sub-cases corresponding to whether the `SchedulerSuspended` flag is 1 or 0 after the statement s_i in ρ.

Case 1(a). `SchedulerSuspended` is 1 after s_i. In this case, the path π_3' must comprise a sequence of fromISR functions. This is because the scheduler remains suspended after s_i in ρ (only F can resume it, and F never executes after $s_i \cdot$ `up(isr)` in ρ), and hence no task API function can run in this suffix of ρ.

Consider the path π_2 that begins in F, contains some interrupting paths that visit other task or fromISR functions, and ends at s_i in F. We define an "uninterrupted" version of π_2, denoted *unint*(π_2), to be the path that replaces each interrupt path by a `skip` statement (note that this non-deterministic branch exists in each interrupt call). In addition, the portion of π_2 that goes through an `UnlockQueue` call may have to change: the path through an `UnlockQueue` depends on the values of `TxLock` and `RxLock`, and these values may have changed by eliding

the interrupt paths from π_2. Nevertheless, there is a path through `UnlockQueue` enabled for these new values, and we use these paths to obtain a feasible path $unint(\pi_2)$ through F.

We can now define the reduced path ρ_{red} we need as follows (see Fig. 11.2):

$$\rho_{red} = \texttt{up(api)} \cdot \texttt{down(api)} \cdot unint(\pi_2) \cdot s_i \cdot \texttt{up(isr)} \cdot \pi_4 \cdot s_k.$$

We need to argue that ρ_{red} is a valid execution of $M_{F,*,I}$ (here "$*$" stands for an arbitrary task function). We have already argued that

$$\texttt{up(api)} \cdot \texttt{down(api)} \cdot unint(\pi_2) \cdot s_i \cdot \texttt{up(isr)}$$

is a valid execution of the model. Let the resulting state after this path be u'. It remains to be shown that the path $\pi_4 \cdot s_k$ is a feasible initial path in I, beginning in state u'.

Let us call two states v and w *equivalent* if they satisfy the following conditions: (a) the value of the control semaphores are the same (i.e. $v(\texttt{isr}) = w(\texttt{isr})$, etc), (b) the value of `SchedulerSuspended` is the same, (c) $v(\texttt{TxLock}) = -1$ iff $w(\texttt{TxLock}) = -1$ and $v(\texttt{TxLock}) \geq 0$ iff $w(\texttt{TxLock}) \geq 0$, and (d) similarly for `RxLock`. By inspection of the conditionals in any fromISR function J in the model, we observe that the set of feasible initial paths through J, beginning from *equivalent* states is exactly the same. Let u be the resulting state after the prefix $\delta = \pi_1 \cdot \texttt{up(api)} \cdot \texttt{down(api)} \cdot \pi_2 \cdot s_i \cdot \texttt{up(isr)}$ of ρ, and let v be the resulting state after $\delta \cdot \pi_3' \cdot \texttt{up(isr)}$. To argue that $\pi_4 \cdot s_k$ is a feasible initial path in I it is thus sufficient to argue that the states u' and v are equivalent.

To do this we argue that (a) u' and u are equivalent, and (b) that u and v are equivalent. To see (a), clearly the value of the control semaphores are identical in u and u'. Further, the value of `SchedulerSuspended` continues to be 1 in u' as well, as we are only excising paths from π_2 that are "balanced" in terms of setting and unsetting this flag. Finally, if the value of `TxLock` was -1 in u, it continues to be -1 in u' as well, since an `UnlockQueue` always resets the flag to -1. If the value of `TxLock` was 0 or more in u, then we must be between a `LockQueue` and its corresponding `UnlockQueue`. In this case the value of `TxLock` would have been set to 0 in u'. A similar argument holds for `RxLock` as well, and we are done. To see that the claim (b) holds, we note that only fromISR functions can execute between u and v, and they always either leave the value of `TxLock` and `RxLock` intact, or increment them if their value was already ≥ 0.

Thus, ρ_{red} is a valid execution of $M_{F,*,I}$, and it clearly contains a race on s_i and s_k. This completes the proof of the first case we were considering.

Case 1(b). We now consider the case when `SchedulerSuspended` is 0 after s_i in ρ. We have two further possibilities to consider: whether `SchedulerSuspended` is 0 at the start of π_4 or whether it is 1.

Case 1(b)(i). `SchedulerSuspended` $= 0$ after s_i and at the start of π_4. The reduced path we need is this case is

$$\rho_{red} = \texttt{up(api)} \cdot \texttt{down(api)} \cdot \mathit{unint}(\pi_2) \cdot s_i \cdot \texttt{up(isr)} \cdot \pi_4 \cdot s_k.$$

We need to prove that this is a valid execution of $M_{F,*,I}$. As argued in Case 1, `up(api)` $\cdot$ `down(api)` $\cdot\, \mathit{unint}(\pi_2) \cdot s_i$ is a valid execution of this model. Let u' be the resulting state after executing `up(api)` $\cdot$ `down(api)` $\cdot\, \mathit{unint}(\pi_2) \cdot s_i \cdot$ `up(isr)`, u be the state after executing $\delta = \pi_1 \cdot$ `up(api)` $\cdot$ `down(api)` $\cdot\, \pi_2 \cdot s_i \cdot$ `up(isr)` and v be the state after executing $\delta \cdot \pi_3' \cdot$ `up(isr)`. Using arguments similar to Case 1, we can conclude that states u and u' are equivalent. We now show that u and v are equivalent. Between states u and v, the control semaphores clearly have the same values. By assumption, the value of `SchedulerSuspended` is 0 at u and v. Since `LockQueue` operations are necessarily performed with `SchedulerSuspended` $= 1$, `TxLock` $= -1$ at u. Since `SchedulerSuspended` $= 0$ at v (by assumption), we must have `TxLock` $= -1$ at v. A similar set of arguments hold for `RxLock`. These conditions imply that u and v are equivalent.

Thus ρ_{red} is a valid execution of $M_{F,*,I}$ and contains a race between s_i and s_k.

Case 1(b)(ii). `SchedulerSuspended` $= 0$ after s_i and at the start of π_4. Clearly, this happens if some task function G executes in π_3 which suspends the scheduler, following which I is invoked.

We sub-divide this case into two further cases, depending on whether `TxLock` $= -1$ or `TxLock` ≥ 0 prior to π_4.

Case 1(b)(ii)(A). If `TxLock` $= -1$ just prior to π_4, then we consider the reduced path

$$\rho_{red} = \texttt{down(api)} \cdot \mathit{unint}(\pi_2) \cdot s_i \cdot \langle \texttt{tick} \rangle \cdot \texttt{up(api)} \cdot \mathit{unint}(\pi_G) \cdot s_j \cdot \texttt{up(isr)} \cdot \pi_4 \cdot s_k.$$

where $\mathit{unint}(\pi_G)$ corresponds to the interrupt-free version of the path taken by the invocation of G in ρ prior to π_4. Here, s_j is the last suspend scheduler operation (`SchedulerSuspended++`) prior to π_4. We want to prove that ρ_{red} is a valid execution of the model $M_{F,G,I}$. By the properties of the *unint* operator and using arguments similar to Case 1, we can show that the subpath `down(api)` $\cdot\, \mathit{unint}(\pi_2) \cdot \ldots s_j \cdot$ `up(isr)` is a valid execution of the model. Let u' be the resulting state after executing `down(api)` $\cdot\, \mathit{unint}(\pi_2) \cdot \cdots \cdot s_j \cdot$ `up(isr)`. Let u be the

resulting state in ρ after s_j, while v is the state in ρ prior to π_4. As before, we attempt to prove that states u' and u are equivalent, while u and v are equivalent. Consider first the states u' and u. Clearly, the value of the control semaphores are the same. By construction, the value of `SchedulerSuspended` is 1 after s_j in both ρ and ρ_{red}. Since `LockQueue` operations are performed *after* suspending the scheduler, `TxLock` $= -1$ after s_j in both ρ and ρ_{red}. A similar argument holds for `RxLock`. Thus, u' and u are equivalent.

Now consider the states u and v. The control semaphores clearly have the same value. From the structure of ρ, it is clear that `SchedulerSuspended` $= 1$ at u and v. By assumption, `TxLock` $= -1$ at v. Since `LockQueue` operations always come after a suspend scheduler operation, `TxLock` $= -1$ in u. We make similar arguments for `RxLock`. Thus, states u and v are equivalent. Consequently, ρ_{red} is a valid execution of $M_{F,G,I}$, and contains a race between s_i and s_k.

Case 1(b)(ii)(B). If `TxLock` ≥ 0 prior to π_4, then we consider the reduced path

$$\rho_{red} = \texttt{down(api)} \cdot \mathit{unint}(\pi_2) \cdot s_i \cdot \langle \texttt{tick} \rangle \cdot \texttt{up(api)} \cdot \mathit{unint}(\pi_G) \cdot s_j \cdot \texttt{up(isr)} \cdot \pi_4 \cdot s_k.$$

where $\mathit{unint}(\pi_G)$ corresponds to the interrupt-free version of the path taken by the invocation of G in ρ prior to π_4. Here, s_j is the last `LockQueue` invocation before π_4. Note that since `LockQueue-UnlockQueue` operations are only performed with the scheduler suspended, and since G is the last task function which suspends the scheduler, the last task function which can execute `LockQueue` prior to I can only be G. We wish to show that ρ_{red} is a valid execution of $M_{F,G,I}$.

In a manner similar to the previous case, we can show that the sub-path of ρ_{red} till π_4 is a valid execution of the model. We now need to show that $\pi_4 \cdot s_k$ is also a valid execution. Let u' be the resulting state after executing `down(api)` $\cdot\, \mathit{unint}(\pi_2) \cdot \cdots \cdot s_j \cdot$ `up(isr)`. Let u be the resulting state in ρ after s_j, while v be the state in ρ prior to π_4. We will show that u' and u are equivalent, as are u and v. Consider first the states u and u'. Clearly, the value of the control semaphores are the same. Since `LockQueue` operations are always performed with the scheduler suspended, `SchedulerSuspended` $= 1$ in both u and u'. Since s_j is a `LockQueue` operation, `TxLock` $= 0$ in both u and u'. A similar argument holds for `RxLock`. Thus, u and u' are equivalent. We now consider u and v. The value of the control semaphores are clearly the same. Since s_j is a `LockQueue` operation, which is necessarily performed with the scheduler suspended, no other task function can executed between s_j and π_4. Thus, `SchedulerSuspended` $= 1$ at u and v. Moreover, `TxLock` $= 0$ immediately after s_j. Since

there could be fromISR functions which execute in π_3' which increment `TxLock`, the value of `TxLock` ≥ 0 in v. A similar set of arguments hold for `RxLock`. Thus, u and v are equivalent. Consequently, ρ_{red} is a valid execution of $M_{F,G,I}$ and contains a race between s_i and s_k.

Case 2. We now consider the case when s_k belongs to a task function in ρ. Note that, in this case, `SchedulerSuspended` $= 0$ after s_i, otherwise a control switch to a different task function would not be possible. We sub-divide this case into two. One case considers the situation when s_k is *not* a statement within `UnlockQueue` (thereby it is always enabled), while the other considers the situation when s_k is a statement within `UnlockQueue` (and is thereby guarded by either `TxLock` > 0 or `RxLock` > 0.

Case 2(a). s_k is *not* a statement within `UnlockQueue`. In this case, we consider the reduced path

$$\rho_{red} = \texttt{down(api)} \cdot unint(\pi_2) \cdot s_i \cdot \langle \texttt{tick} \rangle \cdot \texttt{down(api)} \cdot unint(\pi_G) \cdot s_k.$$

where $unint(\pi_G)$ is the uninterrupted version of the path taken by the racy invocation of the task function G in ρ. The symbol $\langle \texttt{tick} \rangle$ denotes an invocation of the tick interrupt. We prove that ρ_{red} is a valid execution of some $M_{F,G,*}$, where $*$ denotes any fromISR function.

By the property of the *unint* operator, the sub-path `down(api)` $\cdot\, unint(\pi_2) \cdot s_i$ is a valid execution of the model. Further, since `SchedulerSuspended` $= 0$ after s_i, a `tick` interrupt is permitted at this point, and thus a control switch to the task function G is permitted. Again, by the property of *unint*, the sub-path `down(api)` $\cdot\, unint(\pi_G) \cdot s_k$ is a valid execution of the model. Thus, the concatenation of the sub-paths is a valid execution of any model $M_{F,G,*}$ and contains a race between s_i and s_k.

Case 2(b). s_k is a statement within `UnlockQueue` (thereby guarded by either `TxLock` > 0 or `RxLock` > 0). In this case, we consider the reduced path

$$\rho_{red} = \texttt{down(api)} \cdot unint(\pi_2) \cdot s_i \cdot \langle \texttt{tick} \rangle \cdot \texttt{down(api)} \cdot \pi_5 \cdot s_k.$$

We wish to prove that this is a valid execution of $M_{F,G,I}$, where I is any fromISR function which increments `TxLock`. Here, $\pi_5 = \pi_6 \cdot s_j \cdot \pi_7 \cdot \pi_8$, such that $unint(\pi_G) = \pi_6 \cdot s_j \cdot \pi_8$ and π_7 is the path taken by I. The `LockQueue` operation, corresponding to the `UnlockQueue` to which s_k belongs, must be executed by G itself (since `LockQueue`$-$`UnlockQueue` operations are always performed with the scheduler suspended). We let s_j be this `LockQueue` operation. Since interrupts are enabled at s_j, I can execute. Moreover, by inspection of conditionals in any fromISR function J in the model, we observe if J starts in a state with `TxLock` $= 0$, it

exits in a state with `TxLock` > 0. Thus, the resulting state after π_7 has `TxLock` > 0, and none of the operations in π_8 touch this variable, which implies that the guard condition of s_k is satisfied. Thus, ρ_{red} is a valid execution of $M_{F,G,I}$ and contains a race between s_i and s_k. $\square$

Chapter 12

Experimental Evaluation

12.1 Experimental Setup

Of the 69 API functions in FreeRTOS v6.1.1, we model 17 task and 8 fromISR functions. These 25 library functions form the "core" of the FreeRTOS API. The remaining 44 functions are either defined in terms of these core functions, or they simply invoke the core functions with specific arguments, or are synchronization constructs. For example, the functions `xQueuePeek` and `xSemaphoreTake` are listed as library functions. However, they are defined in terms of the core function `xQueueGenericReceive`, which we do model. Thus, modeling these additional functions would be redundant: the races would still be in the core library functions which they invoke.

Our tool-chain is as follows: the user provides a Promela file which models each library function, as well as a template for the reduced models. Next, a Java program creates the "reduced" models (2023 of them in this case) from this Promela template. We then verify these reduced models using Spin. The output of the verification phase is a set of error trails, one corresponding to each interleaving which results in the violation of an assertion. The trails are not in a human readable format, so we need to perform a *simulation* run in Spin using these trails. The output of the simulation run is a set of human readable error traces. However, the number of such traces can be large (around 70,870 were generated during our experiments) and it is infeasible to manually parse them to find the list of races. Instead, we have yet another Java program which scans through these traces and reports the list of unique racing pairs. By a racing pair, we mean statements (s_i, s_k) constituting the race, along with the data-structure `v` involved (we also indicate a trace exhibiting the race).

While the model and the reduction argument need to be tailor-made for different kernel

APIs, the software component of the tool-chain is fairly straightforward to reuse. Given a Promela model of some kernel API other than FreeRTOS, where the modeling follows the rules outlined in Chapter 10 (and the model is shown to be reducible), the tool-chain can be used to detect races with minimal changes.

An important point to consider here is the guarantees provided by the Spin tool itself. Spin does *not* exhaustively search for *all* possible violations of an assertion [44]. Instead, it is guaranteed to report *at least* one counter-example if the property is not satisfied. Hence, we make use of an iterative strategy. After each iteration, we change the assertion statement to suppress reporting the detected races again. We do this by associating a counter value with each critical access path. The tool reports pairs of counter values, which indicate a specific interleaving of critical access paths. In the next iteration, we enrich the assertion to suppress error reports involving the set of pairs reported in the earlier iteration, to ensure that fresh racing pairs are reported. We continue this process until no further assertion violations are detected by Spin. Thus, by the final iteration, we are *guaranteed* to have flagged every high-level data race.

All our experiments were performed on a quad-core Intel Core i7 machine with 32 GB RAM, running Ubuntu 14.04. We use Spin version 6.4.5 for our experiments.

12.2 Evaluating M_2

The verification of M_2 on our machine took up memory in excess of 32 GB. As a result, we had to kill the verification run prematurely. Even on on a more powerful machine with 4 quad-Xeon processors (16 cores), 128 GB of RAM, running Ubuntu 14.04, the verification run took 39 GB of RAM, while executing for more than 3 hours, before timing out. The total number of tracked states was 4.43×10^8. Using rough calculations, we estimated that the total amount of memory needed to store the full state space of this model (assuming that the size of a single state is 100 bytes) is around 1 TB. On the contrary, while model-checking the 2023 reduced models, the RAM usage never exceeded 9 GB.

12.3 Evaluating $\mathcal{M}_{red}$

Recall that each $M_{F,G,I} \in \mathcal{M}_{red}$ comprises 5 processes: the first process runs the task function F, the second runs the task function G, the third runs the ISR I (excluding the `tick` interrupt), the fourth runs the `tick` interrupt in a loop, while the fifth process runs the scheduler. Since there are 17 task functions and 7 fromISR functions (excluding the `tick`), we generate $17 \times 17 \times 7 = 2023$ models. We model check these reduced models in iterations, altering the assertions

at the end of each iteration to ensure that the reported races are not flagged again in the next iteration.

We suppress reporting races on the `SchedulerSuspended` flag, which by design are aplenty. We have manually verified (along with discussions with the FreeRTOS developers) that these races are benign.

In the first iteration, the verification of $\mathcal{M}_{red}$ generated 38 assertion violations. Of these, 10 were false positives, owing to three consecutive read accesses or the conservative modeling of the conditionals. Among the rest, 16 can be definitely classified as harmful.

In the second iteration, the tool reported 10 assertion violations, all of them being potentially benign races involving the variable `pxCurrentTCB`. The cause was an unprotected read of the variable in the function `vTaskResume`. As there were several races involving this statement (it would race with every access, protected or otherwise, of `pxCurrentTCB` in almost all other functions), we suppressed races involving this statement. With this change, we performed a third iteration of the verification process, which resulted in no further assertion violations.

Iteration	#Violations	F.P.	Harmful	Benign?	Time
1	38	10	16	12	1.5 hr
2	10	-	-	10	2.4 hr
3	-	-	-	-	1.84 hr

Figure 12.1: Experimental Evaluation of $\mathcal{M}_{red}$

The FreeRTOS API is quite carefully written. Despite the complexity of the possible task interactions, there are not many harmful races. Among the 16 harmful races detected after the first iteration, most involved the function `vQueueDelete`, which deletes the queue passed to it as argument. Several operations are involved as part of the deletion (removal of the queue from the registry, deallocating the memory assigned to the queue, etc.). Surprisingly, the set of operations, which forms a critical access path for the queue data-structure, is devoid of any synchronization. This causes critical access paths of the queue in other functions, for example `xQueueReceiveFromISR` (which reads the contents of the queue), to interleave with the path in `vQueueDelete`. The race is harmful because functions can potentially observe an inconsistent (partially deleted) state of the queue, which it would not otherwise observe along any linearized execution. We reported this bug to the FreeRTOS developers, and they argue that this is not serious since queue delete operations are rare and are usually performed at the end of the application's lifetime.

The other harmful races involve the `QueueRegistry` data-structure (which is essentially

Table 12.1: Some Sample Detected Races. "H" indicates Harmful races and "PB" indicates Possibly Benign.

Structure	Library Functions Involved	Scenario	H/PB
`xQueueRegistry`	P1: `vQueueAddToRegistry` P2: `vQueueUnregisterQueue`	While P1 reads the registry, P2 modifies it. Read-write race.	H
`userQueue`	P1: `vQueueDelete` P2: `xQueueSend`	P2 sends data to a queue while it is being deleted by P1. Write-write race.	H
`uxPriority`	P1: `xTaskCreate` P2: `vTaskPrioritySet`	P1 checks the value of uxPriority while it is being set by P2. The result is never an inconsistent state.	PB

an array), where addition and deletion of items in the `QueueRegistry` can interleave, thereby causing a function to observe an inconsistent state of the registry. Some of the sample races are given in Tab. 12.1.

A summary of the various statistics of the experiments is given in Fig. 12.1. The running times are reported in hours. All artifacts of this work are available online at

https : //bitbucket.org/suvam/freertos

12.3.1 List of Detected Races

We report the list of all detected races, along with their classification.

Table 12.2: List of Harmful Races

	Variable	Functions	Comments
1	xQueueRegistry	vQueueUnregisterQueue (w) vQueueAddToRegistry (r)	Simultaneous removal and addition to the registry.
2	xQueueRegistry	vQueueAddToRegistry (w) vQueueAddToRegistry (r)	Reading the registry while it is being updated simultaneously.
3	xQueueRegistry	vQueueUnregisterQueue (w) vQueueAddToRegistry (w)	Simultaneous updates to the registry.
4	xQueueRegistry	vQueueUnregisterQueue (w) vQueueUnregisterQueue (w)	Simultaneous updates to the registry.
5	userQueue	xQueueReceive (r) vQueueDelete (w)	Retrieval from a queue while it is being deleted simultaneously.
6	userQueue	xQueueReceiveFromISR (r) vQueueDelete (w)	Retrieval from a queue (in ISR) while it is being deleted simultaneously.
7	userQueue	uxQueueMessagesWaitingFromISR (r) vQueueDelete (w)	ISR attempts to read the queue while it is being deleted simultaneously.
8	userQueue	vQueueDelete (w) vQueueAddToRegistry (r)	Reading components of a queue while the queue is being deleted simultaneously.
9	userQueue	vQueueDelete (w) vQueueUnregisterQueue (r)	Reading components of a queue while the queue is being deleted simultaneously.
10	userQueue	xQueueIsQueueFullFromISR (r) vQueueDelete (w)	Reading components of a queue while the queue is being deleted simultaneously.
11	userQueue	xQueueSendFromISR (r) vQueueDelete (w)	Reading components of a queue while the queue is being deleted simultaneously.
12	userQueue	xQueueIsQueueEmptyFromISR (r) vQueueDelete (w)	Reading components of a queue while the queue is being deleted simultaneously.
13	userQueue	xQueueSend (r) vQueueDelete (w)	Reading components of a queue while the queue is being deleted simultaneously.
14	userQueue	uxQueueMessagesWaiting (r) vQueueDelete (w)	Reading components of a queue while the queue is being deleted simultaneously.
15	userQueue	xQueueIsQueueFullFromISR (r) vQueueDelete (w)	Reading components of a queue while the queue is being deleted simultaneously.
16	pxCurrentTCB	xTaskCreate (w) vTaskResume (r)	Simultaneous access to pxCurrentTCB.

Table 12.3: List of Potentially Benign Races

	Variable	Functions	Comments
1	userQueue	xQueueReceiveFromISR (w) vQueueAddToRegistry (r)	The shared queue component is not modified in either function.
2	userQueue	xQueueSendFromISR (w) vQueueAddToRegistry (r)	The shared queue component is not modified in either function.
3	userQueue	xQueueSend (w) vQueueAddToRegistry (r)	The shared queue component is not modified in either function.
4	userQueue	xQueueSend (w) vQueueUnregisterQueue (r)	Does not lead to an inconsistent state.
5	userQueue	xQueueReceiveFromISR (w) vQueueUnregisterQueue (r)	Does not lead to an inconsistent state.
6	userQueue	xQueueReceive (w) vQueueUnregisterQueue (r)	Does not lead to an inconsistent state.
7	userQueue	xQueueReceive (w) vQueueAddToRegistry (r)	Does not lead to an inconsistent state.
8	pxCurrentTCB	vTaskSwitchContextFromISR (w) vTaskResume (r)	Does not lead to an inconsistent state.
9	pxCurrentTCB	vTaskSwitchContextFromISR (w) xTaskCreate (r)	Does not lead to an inconsistent state.
10	pxCurrentTCB	xTaskCreate (w) vTaskResume (r)	Does not lead to an inconsistent state.
11	uxCurrent-NumberOfTasks	xTaskCreate (w) uxTaskGetNumberOfTasks (r)	Does not lead to an inconsistent state.
12	uxPriority	vTaskPrioritySet (w) xTaskCreate (r)	Causes an extra yield.
13	pxCurrentTCB	xPendingReadyList (r) vTaskDelay (r) vTaskDelay (r)	False Positive
14	pxCurrentTCB	xQueueReceiveFromISR (r) vTaskDelayUntil (r) vTaskResume (r)	False Positive
15	pxCurrentTCB	vTaskSwitchContextFromISR (w) vTaskResume (r)	Does not lead to an inconsistent state.

Chapter 13

Related Work and Discussion

Along with work on detecting high-level races and atomicity violations, we also consider work on detecting classical (location-based) races as some of these techniques could be adapted for high-level races as well. We group the work into three categories below and discuss them in relation to our work. The table alongside summarizes the applicability of earlier approaches to our problem setting.

Dynamic analysis based approaches. Artho et al [7] coined the term "high-level data race" and gave an informal definition of it in terms of accessing a set of shared variables (what we call a unit in $\mathcal{S}$) "atomically." They define a notion of a thread's "view" of the set of shared variables, and flag potential races whenever two threads have inconsistent views. They then provide a lock-set based algorithm, for detecting view inconsistencies dynamically along an execution. Among the techniques for detecting atomicity violations, Atomizer [34] uses the notion of left/right moving actions, SVD [87] uses atomic regions as subgraphs of the dynamic PDG, AVIO [54] uses interleaving accesses, and [85] uses a notion of trace-equivalence; to check if a given execution exhibits an atomicity violation. Techniques for dynamically detecting classical data races use locksets computed along an execution (for example [72, 15]), or use the happens-before ordering (for example [24, 35]), to detect races. None of these techniques apply directly to the kind of concurrency and synchronization model of FreeRTOS (there are no explicit locks, and no immediate analogue of the happens-before relation). Most importantly, by design these techniques explore only a part of the execution space and hence cannot detect *all* races.

Static analysis based approaches. von Praun and Gross [81] and Pessanha et al [23] extend the view-based approach of [7] to carry out a static analysis to detect high-level races. The notion of views could be used to obtain an annotation of critical accesses (an $\mathcal{S}$ and $\mathcal{C}$

in our setting) for methods in a library. However definition of views are lock-based and it is not clear what is the corresponding notion in our setting, and whether it would correspond intuitively to what we need.

Flannegan and Qadeer [36] give a type system based static analysis for proving atomicity of methods (i.e. the method's actions can be serialized with respect to interleavings with actions of another thread). The actions of a method are typed as left/right-movers and the analysis soundly infers methods to be atomic. Wang and Stoller [85] extend this type system for lock-free synchronization. In our setting, the notion of left/right-movers is not immediate, and such an approach will likely have a large number of false positives. Static approaches for classical race detection (e.g. [76, 29, 82]) are typically based on a lockset-based data-flow analysis, where the analysis keeps track of the set of locks that are "must" held at each access to a location and reports a race if two conflicting accesses hold disjoint locks. In [1] the locksets are built into a type system which associates a lock with each field declaration. All the approaches above can handle libraries and can detect all races in principle, but in practice are too imprecise (lots of false positives) and often use unsound filters (for example [23, 82]) that improve precision at the expense of missing real races.

Schwarz et al [74, 75] provide a precise data-flow analysis for checking races in FreeRTOS-like applications that handles flag-based synchronization and interrupt-driven scheduling. The technique is capable of detecting all races, but is applicable only to a given application rather than a library.

Model-Checking approaches. In [4] Alur et al study the problem of deciding whether a finite-state model satisfies properties like serializability and linearizability. This approach is attractive as in principle it could be used to verify freedom from atomicity violations. However, the number of threads need to be bounded (hence they cannot handle libraries) and the running time is prohibitive (exponential in number of transitions for serializability, and doubly-exponential in number of threads for linearizability). Farzan and Madhusudan [31] consider a stronger notion of serializability called "conflict serializability" and give a monitoring algorithm to detect whether a given execution is conflict-serializable or not. This also leads to a model-checking algorithm for conflict-serializability based atomicity violations. Again, this is applicable only to applications rather than libraries.

Several researchers have used model-checking tools like Slam, Blast, and Spin to precisely model various kinds of control-flow and synchronization mechanisms and detect errors exhaustively [42, 26, 40, 41, 90]. All these approaches are for specific application programs rather than libraries. Chandrasekharan et al [14] follow a similar approach to ours for verifying thread-safety

Earlier Work	FreeRTOS-like Concurrency	Handles Libraries	Can Detect all races
[7], [34], [87], [54], [85], [63], [15], [72], [24], [35], [90]	No	No	No
[76], [29], [82], [81], [23], [36], [85] [1], [14]	No	Yes	Yes
[4], [31], [42], [49]	No	No	Yes
[74],[75]	Yes	No	Yes

Figure 13.1: Applicability of earlier work to our setting.

of a multi-core version of FreeRTOS. However, the library there uses explicit locks and a standard notion of control-flow between threads. Further, they use a model which is the equivalent of M_2, which does not scale in our setting.

Chapter 14

Concluding Remarks

In this thesis, we investigate two different problems in two different types of concurrent systems.

The first problem we address is that of devising efficient static analyses for the class of multi-threaded shared-variable programs. This part of the thesis leverages the abstract interpretation framework [19]. Our starting point is a novel thread-local semantics for race free programs, which we call *L-DRF*. Abstract analyses derived from *L-DRF* make use of the *sync*-CFG representation of the program, which was first introduced in [22]. The resulting abstractions are able to maintain relational information everywhere, except at synchronization points. To further improve precision, we introduce the notion of user-defined regions, which is a semantic partitioning of the set of program variables. We introduce the notion of region races, which is stronger than the standard notion of data races. Given a program P which additionally satisfies region race freedom, we parameterize the *L-DRF* semantics with the region definitions, which yields abstract analyses with greater precision. The precision is due to the fact that the inter-thread join is additionally able to preserve invariants that hold *within* a region. We instantiate the abstractions to devise efficient relational analyses for race free concurrent programs, in a tool called RATCOP. In our experiments, RATCOP was fairly precise while being fast.

Currently, the *L-DRF* semantics does not handle dynamically allocated memory. If the *L-DRF* semantics is equipped with a heap component, then it may be abstracted to efficient analyses for concurrent programs which manipulate the heap. This would yield efficient pointer analyses and shape analyses for concurrent programs.

The second problem we address is that of detecting all high-level data races in an real-time operating system (RTOS) kernel that relies on flag-based scheduling and synchronization. Such races are good indicators of possible atomicity violations. We describe our methodology with respect to FreeRTOS, a popular operating system in the embedded systems domain.

14. CONCLUDING REMARKS

Since FreeRTOS is quite representative of the class of micro kernel libraries, we expect our techniques to generalize to other such libraries. The specific modeling and reduction arguments would of course have to be tailor-made for the given library. Our methodology here is based on model-checking. We avoid the standard issues related to model-checking (incompleteness and scalability) by making use of a meta-argument which bounds the concurrent tasks needed to orchestrate a race. We implemented our techniques in a tool which we used to check for high-level races in FreeRTOS, and uncovered several harmful defects.

While our arguments, in this work, are tailor-made for the library under consideration (in this case, FreeRTOS), the assumptions we make are purely syntactic in nature. It is, perhaps, not unreasonable to assume that such syntactic features exist in libraries in other domains. Thus, as future work, it would be interesting to apply our methodology to concurrent libraries like the `java.util.concurrent`. Moreover, we only handle high-level races in this work. An interesting extension would be to check for other concurrency defects, like deadlocks.

Bibliography

[1] Martin Abadi, Cormac Flanagan, and Stephen N Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):207–255, 2006. 112, 113

[2] Sarita V Adve and Hans-J Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, 2010. 4

[3] Sarita V Adve and Mark D Hill. Weak orderinga new definition. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 2–14. ACM, 1990. 4

[4] Rajeev Alur, Kenneth L. McMillan, and Doron A. Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000. doi: 10.1006/inco.1999.2847. URL http://dx.doi.org/10.1006/inco.1999.2847. 112, 113

[5] Paul Anderson. Fun with Concurrency Problems. http://blogs.grammatech.com/fun-with-concurrency-problems. [Online; accessed 12-July-2017]. 4

[6] Apple Inc. 3

[7] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *J. Software Testing, Verification & Reliability*, page 2003, 2003. 13, 111, 113

[8] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *New Technologies for Information Systems, Proceedings of the 3rd International Workshop on New Developments in Digital Libraries, NDDL 2003*, pages 82–93, 2003. 73

[9] Dirk Beyer. Software verification and verifiable witnesses. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–416. Springer, 2015. 66

[10] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. *CoRR*, abs/cs/0701193, 2007. URL `http://arxiv.org/abs/cs/0701193`. 73

[11] Hans-J Boehm and Sarita V Adve. You don't know jack about shared variables or memory models. *Communications of the ACM*, 55:48–54, 2012. 4, 5

[12] Hans-Juergen Boehm. How to miscompile programs with "benign" data races. In *3rd USENIX Workshop on Hot Topics in Parallelism, HotPar'11, Berkeley, CA, USA, May 26-27, 2011*, 2011. URL `https://www.usenix.org/conference/hotpar-11/how-miscompile-programs-benign-data-races`. 4

[13] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. *NFM*, 15:3–11, 2015. 3

[14] Prakash Chandrasekaran, K. B. Shibu Kumar, Remish L. Minz, Deepak D'Souza, and Lomesh Meshram. A multi-core version of FreeRTOS verified for datarace and deadlock freedom. In *Proc. ACM/IEEE Formal Methods and Models for Codesign (MEMOCODE)*, pages 62–71, 2014. doi: 10.1109/MEMCOD.2014.6961844. URL `http://dx.doi.org/10.1109/MEMCOD.2014.6961844`. 112, 113

[15] 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 *Proc. ACM SIGPLAN Programming Languages Design and Implementation (PLDI)*, pages 258–269, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: 10.1145/512529.512560. URL `http://doi.acm.org/10.1145/512529.512560`. 111, 113

[16] Ravi Chugh, Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 316–326, 2008. doi: 10.1145/1375581.1375620. URL `http://doi.acm.org/10.1145/1375581.1375620`. 71

[17] Ravi Chugh, Jan Wen Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the ACM SIGPLAN*

2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008, pages 316–326, 2008. doi: 10.1145/1375581.1375620. URL http://doi.acm.org/10.1145/1375581.1375620. 4

[18] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming, Paris, France*. Dunod, 1976. 10

[19] 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. 3, 5, 50, 51, 115

[20] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages*, pages 84–96. ACM, 1978. 10

[21] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 21–30, 2005. doi: 10.1007/978-3-540-31987-0_3. URL https://doi.org/10.1007/978-3-540-31987-0_3. 3

[22] Arnab De, Deepak D'Souza, and Rupesh Nasre. Dataflow analysis for datarace-free programs. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011*, pages 196–215, 2011. iii, xi, 4, 5, 7, 8, 10, 49, 50, 55, 58, 65, 66, 68, 72, 115

[23] Ricardo J. Dias, Vasco Pessanha, and João Lourenço. Precise detection of atomicity violations. In *Hardware and Software: Verification and Testing - 8th International Haifa Verification Conference, (HVC)*, pages 8–23, 2012. doi: 10.1007/978-3-642-39611-3_8. URL http://dx.doi.org/10.1007/978-3-642-39611-3_8. 111, 112, 113

[24] Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. In *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD, pages 85–96. ACM, 1991. ISBN 0-89791-457-0. doi: 10.1145/122759.122767. URL http://doi.acm.org/10.1145/122759.122767. 111, 113

[25] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *SIGSOFT '94, Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering, New Orleans, Louisiana, USA, December 6-9, 1994*, pages 62–75, 1994. doi: 10.1145/193173.195295. URL http://doi.acm.org/10.1145/193173.195295. 72

[26] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Precise race detection and efficient model checking using locksets. Technical Report MSR-TR-2005-118, Microsoft Research, 2005. 112

[27] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *ACM SIGPLAN Notices*, volume 42, pages 245–255. ACM, 2007. 3

[28] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. Delay-bounded scheduling. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 411–422, 2011. doi: 10.1145/1926385.1926432. URL http://doi.acm.org/10.1145/1926385.1926432. 1, 2

[29] Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, October 2003. ISSN 0163-5980. doi: 10.1145/1165389.945468. URL http://doi.acm.org/10.1145/1165389.945468. 13, 112, 113

[30] Azadeh Farzan and Zachary Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–308, 2012. doi: 10.1145/2103656.2103693. URL http://doi.acm.org/10.1145/2103656.2103693. 72

[31] Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Proc. Computer Aided Verification (CAV)*, pages 52–65, 2008. doi: 10.1007/978-3-540-70545-1_8. URL http://dx.doi.org/10.1007/978-3-540-70545-1_8. 84, 112, 113

[32] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 129–142, 2013. doi: 10.1145/2429069.2429086. URL http://doi.acm.org/10.1145/2429069.2429086. 72

[33] Rodrigo Ferreira, Xinyu Feng, and Zhong Shao. Parameterized memory models and concurrent separation logic. In *European Symposium on Programming*, pages 267–286. Springer, 2010. 72

[34] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. *Sci. Comput. Program.*, 71(2):89–109, 2008. doi: 10.1016/j.scico.2007.12.001. URL `http://dx.doi.org/10.1016/j.scico.2007.12.001`. 3, 111, 113

[35] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proc. ACM SIGPLAN PLDI*, pages 121–133. ACM, 2009. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542490. URL `http://doi.acm.org/10.1145/1542476.1542490`. 3, 111, 113

[36] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. ACM SIGPLAN Programming Language Design and Implementation (PLDI)*, pages 338–349, 2003. doi: 10.1145/781131.781169. URL `http://doi.acm.org/10.1145/781131.781169`. 12, 84, 112, 113

[37] Patrice Godefroid. Model checking for programming languages using verisoft. In *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 174–186, 1997. doi: 10.1145/263699.263717. URL `http://doi.acm.org/10.1145/263699.263717`. 1

[38] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, pages 266–277, 2007. doi: 10.1145/1250734.1250765. URL `http://doi.acm.org/10.1145/1250734.1250765`. 72, 73

[39] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 159–168, 1993. doi: 10.1145/155332.155349. URL `http://doi.acm.org/10.1145/155332.155349`. 72

[40] Klaus Havelund and Jens U. Skakkebæk. Applying Model Checking in Java Verification. In *Proc. Theoretical and Practical Aspects of SPIN Model Checking*, volume 1680, pages 216–231. Springer, 1999. 112

[41] Klaus Havelund, Michael R. Lowry, and John Penix. Formal analysis of a space-craft controller using SPIN. *IEEE Trans. Software Eng.*, 27(8):749–765, 2001. doi: 10.1109/32.940728. URL `http://dx.doi.org/10.1109/32.940728`. 112

[42] Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Race checking by context inference. In *Proc. ACM SIGPLAN Programming Language Design and Implementation (PLDI)*, pages 1–13, 2004. doi: 10.1145/996841.996844. URL `http://doi.acm.org/10.1145/996841.996844`. 112, 113

[43] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. 84

[44] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Engineering*, 23: 279–295, 1997. 2, 89, 106

[45] Bertrand Jeannet. Some experience on the software engineering of abstract interpretation tools. *Electronic Notes in Theoretical Computer Science*, 267(2):29–42, 2010. 69

[46] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*, pages 661–667. Springer, 2009. 65

[47] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41 (4):21:1–21:54, 2009. doi: 10.1145/1592434.1592438. URL `http://doi.acm.org/10.1145/1592434.1592438`. 2

[48] Cliff B. Jones. *Developing methods for computer programs including a notion of interference*. PhD thesis, University of Oxford, UK, 1981. URL `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.259064`. 71

[49] Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *ACM SIGSOFT FSE*, pages 13–22. ACM, 2009. 113

[50] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL `http://doi.acm.org/10.1145/359545.359563`. 26

[51] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9):690–691, 1979. 6

[52] Tong Li, Carla Schlatter Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, pages 31–44, 2005. URL `http://www.usenix.org/events/usenix05/tech/general/li.html`. 3

[53] LLVM Project. 3

[54] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: detecting atomicity violations via access interleaving invariants. In *Proc. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 37–48, 2006. doi: 10.1145/1168857.1168864. URL `http://doi.acm.org/10.1145/1168857.1168864`. 111, 113

[55] Roman Manevich, Tal Lev-Ami, Mooly Sagiv, Ganesan Ramalingam, and Josh Berdine. Heap decomposition for concurrent shape analysis. In *Static Analysis, 15th International Symposium, SAS*, pages 363–377, 2008. doi: 10.1007/978-3-540-69166-2_24. URL `https://doi.org/10.1007/978-3-540-69166-2_24`. 73

[56] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040336. URL `http://doi.acm.org/10.1145/1040305.1040336`. 4

[57] Antoine Miné. Static analysis of run-time errors in embedded real-time parallel C programs. *Logical Methods in Computer Science*, 8(1), 2012. doi: 10.2168/LMCS-8(1:26)2012. URL `https://doi.org/10.2168/LMCS-8(1:26)2012`. 71

[58] Antoine Miné. *Static analysis by abstract interpretation of concurrent programs*. PhD thesis, Ecole Normale Supérieure de Paris-ENS Paris, 2013. 66, 67

[59] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation*, pages 39–58. Springer, 2014. 54, 68, 69, 71, 72

[60] Raphaël Monat and Antoine Miné. Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 386–404. Springer, 2017. 7, 68, 69

[61] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 446–455, 2007. doi: 10.1145/1250734.1250785. URL `http://doi.acm.org/10.1145/1250734.1250785`. 2

[62] Mayur Naik. Chord: A Program Analysis Platform for Java. `http://www.cis.upenn.edu/~mhnaik/chord.html`. Accessed: 2017-03-27. 65, 66

[63] Shaz Qadeer and Dingha Wu. KISS: keep it simple and sequential. In *Proc. ACM SIGPLAN Programming Languages Design and Implementaion (PLDI)*, pages 14–24, 2004. doi: 10.1145/996841.996845. URL `http://doi.acm.org/10.1145/996841.996845`. 113

[64] Real Time Engineers Ltd. The FreeRTOS Real Time Operating System, 2014. URL `www.freertos.org`. 13, 77

[65] Martin C. Rinard. Analysis of multithreaded programs. In *Static Analysis, 8th International Symposium, SAS*, pages 1–19, 2001. doi: 10.1007/3-540-47764-0_1. URL `https://doi.org/10.1007/3-540-47764-0_1`. 71

[66] Malavika Samak and Murali Krishna Ramanathan. Trace driven dynamic deadlock detection and reproduction. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 29–42, 2014. doi: 10.1145/2555243.2555262. URL `http://doi.acm.org/10.1145/2555243.2555262`. 3

[67] Malavika Samak and Murali Krishna Ramanathan. Omen+: a precise dynamic deadlock detector for multithreaded java libraries. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 735–738, 2014. doi: 10.1145/2635868.2661670. URL `http://doi.acm.org/10.1145/2635868.2661670`. 3

[68] Malavika Samak and Murali Krishna Ramanathan. Omen: a tool for synthesizing tests for deadlock detection. In *Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '14, Portland, OR, USA, October 20-24, 2014 - Companion Volume*, pages 37–38, 2014. doi: 10.1145/2660252.2664663. URL `http://doi.acm.org/10.1145/2660252.2664663`. 3

[69] Malavika Samak and Murali Krishna Ramanathan. Multithreaded test synthesis for deadlock detection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 473–489, 2014. doi: 10.1145/2660193.2660238. URL `http://doi.acm.org/10.1145/2660193.2660238`. 3

[70] Malavika Samak and Murali Krishna Ramanathan. Synthesizing tests for detecting atomicity violations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 131–142, 2015. doi: 10.1145/2786805.2786874. URL `http://doi.acm.org/10.1145/2786805.2786874`. 3

[71] Malavika Samak, Murali Krishna Ramanathan, and Suresh Jagannathan. Synthesizing racy tests. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 175–185, 2015. doi: 10.1145/2737924.2737998. URL `http://doi.acm.org/10.1145/2737924.2737998`. 3

[72] 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 (TOCS)*, 15(4):391–411, 1997. 111, 113

[73] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP*, pages 27–37, 1997. doi: 10.1145/268998.266641. URL `http://doi.acm.org/10.1145/268998.266641`. 3

[74] Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, Peter Lammich, and Markus Müller-Olm. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In *Proc. ACM SIGPLAN-SIGACT Principles of Programming Languages (POPL)*, pages